**AppleIIAsm Library**

**Reference Manual**

Version 0.5.0

Nathan Riggs

**TABLE OF CONTENTS**

Preface

Introduction

**PART I**

Package Overview

Standard Practices / Procedures

Naming Conventions

Commenting Conventions

Parameter Passing

Main Source Sequencing

Quick Reference: Macros

Quick Reference: Subroutines

**PART II**

Detailed Descriptions and Listings

Library Disks

Disk 1: REQCOM (Required & Common Libraries)

Required Library

HEAD.REQUIRED

MAC.REQUIRED

LIB.REQUIRED

Common Library

HOOKS.COMMON

MAC.COMMON

DEMO.COMMON

Utilities

Integrated Libraries

Demo Disks

Appendix A: Companion Books

The New Merlin 8 Pro User Guide

Learning 6502 Assembly with AppleIIAsmLib and AppleChop

Forever Machine: The Past and Future on the Apple II Platform

**Preface**

This is the first complete reference manual for the AppleIIAsm macro and subroutine library. Currently, this library is in the alpha stages of development: not all disks are complete, there may be some bugs here and there, and major workflow decisions may still be in flux. However, this version, 0.5.0, represents a major step forward in functionality, optimization and standardization, and at least for what is complete—the first eleven disks as well as some demo disks—the library can be reasonably considered to be stable. That does not, of course, mean that there are any guarantees.

I started this project as research into how the Apple II works as a platform for another book I am writing, and eventually became interested in the cohesive technical documentation (or sometimes lack thereof) that was available to beginning coders in the heyday of the Apple II as well as those looking to learn Apple II (6502) Assembly today. Having no prior experience with Assembly language, I began coding the library itself as part of my own learning process while trying to write subroutines that provided much of the functionality afforded by Applesoft BASIC. Eventually, this became a beast of its own, and what you’re reading here is (part) of the result.

As the library grows and morphs, so will this document. If nothing else, I hope that the library and its accompanying documentation helps hobbyists, researchers, and otherwise self-hating hopeless nerds learn and accomplish what they want or need—at least as much as it has helped, and harmed, me.

Nathan Riggs

**Introduction**

The AppleIIAsm Library is a collection of subroutines and macros for the Apple II line of computers, aimed at providing a stable set of assembly routines for most common tasks. Additionally, this library is meant to ease the transition between programming in Applesoft BASIC and 6502 Assembly by not only providing the basic data structures and functions found in higher-level languages but also by providing a set a macros—currently dubbed AppleChop—that simulates the design and workflow of BASIC. A companion booklet to this library, *From Applesoft to AppleChop and Assembly,* provides a framework for making that transition.

These subroutines and macros are written for the Merlin Pro 8 assembler, which should run on any Apple II with 64k of memory (programs assembled with Merlin Pro 8 will run on machines with less than 64k, however). Since we are using 6502 Assembly here, however, it should not be too difficult to port the subroutines to other assemblers and even other systems like the Commodore 64, Nintendo Entertainment System, BBC Micro, and more. For a guide on using the Merlin Pro 8 Assembler, see the other companion booklet, *The New Merlin Pro 8 User Guide*.

**Who is this manual for?**

The primary audience for this manual is someone who is already familiar with 6502 Assembly, or who is working their way through *From Applesoft to AppleChop and Assembly*. Like all manuals, this is primarily a reference: beyond this introduction and early sections of Part I, this manual is not meant to be read straight through. Feel free to flip back and forth as you wish!

**Who is this manual NOT for?**

This manual is definitely not for beginners, but nor is it really aimed at 6502 experts. The library itself can be used by beginner and expert alike, but whereas this manual would likely confuse the absolute beginner, an expert interested in optimizing their work (and these subroutines) will not find much help here.

As someone who spends a *lot* of time thinking about, writing about, and teaching different facets of technical writing (in its broadest sense), I can confirm the following: there are thousands of books written about the 6502 architecture and Assembly programming. I can also confirm that these books--as well as most websites--tend to approach the subject from a "writerly" position rather than a reader-centered one; that is, it's written for engineers and computer scientists who have already spent a lot of time and money understanding the theory, learning the jargon, and training themselves to be able to do things by muscle memory. That's great for established engineers, mathematicians, computer scientists and the like, as well as those who can afford to dedicate years of their lives (and again, gobs of $$$) to obtain a degree that qualifies them as entry level in the field. It is not so great, however, for beginners, hobbyists, or those trying to study it from a non-engineering theoretical perspective. That is, at least, part of the gap I am hoping to fill.

That said, I myself would have failed quite readily without at least a few key texts and websites, and it would be remiss to not list them here. And if you're committed to learning this, know that there is no good replacement to sitting down, typing out a listing from a book, assembling it and then trying to figure out what the hell you just did—or what you did wrong! There is no doing without learning, and there is no learning without doing.

**Why Merlin Pro 8? Why not something...modern?**

Understanding how coding for a specific platform and a specific era works is not merely a matter of knowledge, but a matter of practice. Much of the way development happens, in computer software or not, is predicated on the apparatus in place that allows for it. Changing that apparatus, whether it be adding modern components like multiple open files, faster assembly, easier access and legibility and so on changes your understanding of how everything worked (and works). Especially with an ancient (and largely obsolete) language like 6502 assembly, few people are learning it to accomplish a practical task. Instead, we are approaching the topic more like an archaeologist or historical reenactor: going through the same motions to understand the topic cohesively.

That said, there is nothing inherently wrong with using modern tools—it just does not fit the goals for writing this library. Brutal Deluxe software has rewritten a more modern version of Merlin 16, and the CC65 compiler/assembler makes contemporary 6502 development far more efficient and less frustrating overall. If Merlin 8 Pro feels too dated—and to many, it will feel hopelessly so—by all means use these modern software packages. Just be aware that some substantial effort may be involved in rewriting the code here for different assemblers.

**Further Resources**

While beginners are welcome to use this library, and it is partially aimed at those who are trying to learn 6502 Assembly on the Apple II, a cohesive and thorough guide to 6502 programming is beyond the scope of this manual. For a better understanding of the hardware, programming, and culture surrounding the Apple II, I would suggest consulting the following sources.

*6502 Programming Books*

* Roger Wagner, Chris Torrence. [*Assembly Lines: The Complete Book*](https://www.amazon.com/Assembly-Lines-Complete-Roger-Wagner/dp/1312089407/). May 10, 2017.
* Lance A. Leventhal, Winthrop Saville. *6502 Assembly Language Subroutines*. 1982.
* Don Lancaster. *Assembly Cookbook for the Apple II, IIe*. 1984, 2011.
* Mark Andrews. \_Apple Roots: Assembly Language Programming for the Apple IIe and IIc. 1986.
* CW Finley, Jr., Roy E. Meyers. *Assembly Language for the Applesoft Programmer*. 1984.
* Randy Hyde. *Using 6502 Assembly Language*. 1981.
* Glen Bredon. *Merlin Pro Instruction Manual*. 1984.
* JS Anderson. *Microprocessor Technology*. 1994. (also covers z80 architecture)

*6502 Programming Websites*

* [CodeBase64](http://codebase64.org/doku.php)
* [6502.org](http://www.6502.org/)
* [Easy6502](http://skilldrick.github.io/easy6502/)

*Apple II Books*

* Bill Martens, Brian Wiser, William F. Luebbert, Phil Daley. [*What's Where in the Apple, Enhanced Edition: A Complete Guide to the Apple ][ Computer*.](https://www.amazon.com/gp/product/136517364X/) October 11, 2016.
* David Flannigan. [*The New Apple II Users' Guide*](https://www.amazon.com/gp/product/0615639879/). June 6, 2012.
* David L. Craddock. [*Break Out: How the Apple II Launched the PC Gaming Revolution*](https://www.amazon.com/gp/product/0764353225/). September 28, 2017.
* Steven Weyhrich. [*Sophistication & Simplicity: The Life and Times of the Apple II Computer*](https://www.amazon.com/gp/product/0986832278/). December 1, 2013.
* Ken Williams, Bob Kernagham, Lisa Kernagham. [*Apple II Computer Graphics*](https://www.amazon.com/gp/product/B01FIXG7ZK/). November 3, 1983.
* Lon Poole. [*Apple II Users' Guide*.](https://www.amazon.com/gp/product/0931988462/). 1981.
* Jeffrey Stanton. *Apple Graphics and Arcade Game Design*. 1982.
* Apple. *Apple Monitors Peeled*. 1981.
* Apple. \_Apple II/IIe/IIc/IIgs Technical Reference Manual.

*Apple II Websites*

* [Apple II Text Files](http://textfiles.com/apple/.windex.html)
* [Apple II Programming](http://apple2.org.za/gswv/a2zine/faqs/csa2pfaq.html)
* [The Asimov Software Archive](https://ftp.apple.asimov.net/)
* [Apple II Online](https://apple2online.com/index.php)
* [Juiced.GS: A Quarterly Apple II Journal](https://juiced.gs/)

*Related GitHub Projects*

A number of folk are doing work on 6502 or the Apple II on GitHub. While I cannot possibly list each and every one (that's what the search function is for!), these are projects I have found particularly useful, informative, entertaining, or inspiring.

* [Prince of Persia Apple II Source Code](https://github.com/fabiensanglard/Prince-of-Persia-Apple-II), by Jordan Mechner
* [WeeGUI, a small gui for the Apple II](https://github.com/blondie7575/WeeGUI)
* [Two-lines or less Applesoft programs](https://github.com/thelbane/Apple-II-Programs) -- a lot can be accomplished!
* [Doss33FSProgs](https://github.com/deater/dos33fsprogs), programs for manipulating the DOS 3.3 filesystem
* [ADTPro](https://github.com/ADTPro/adtpro), a requirement for anyone working with real Apple II hardware today.
* [CC65](https://github.com/cc65), a modern cross-compiling C compiler and assembler for 6502 systems.
* [PLASMA: The Proto-Language Assembler for All](https://github.com/dschmenk/PLASMA) -- this was originally written for the Apple II alone, but has recently expanded to other systems.

**Part I**

The AppleIIAsm Library

**Library Overview**

The AppleIIAsm library consists of 25 disks that contain thematically related subroutines, demos and utilities, as well as two extra disks that hold minified versions of every subroutine for convenience. The contents of each disk and library are covered in Part II: Detailed Descriptions and Listings. The disks are ordered as follows:

* Disk 1 – REQCOM (Required and Common Libraries)
* Disk 2 – STDIO (Standard Input and Output Library)
* Disk 3 – ARRAYS (Array Library)
* Disk 4 – MATH (Math Library)
* Disk 5 – STRINGS (String Library)
* Disk 6 – FILEIO (File Input and Output Library)
* Disk 7 – CONVERT (Data Type Conversion Library)
* Disk 8 – LORES (Low Resolution Graphics Library)
* Disk 9 – SPEAKER (Mono Speaker Library)
* Disk 10 – HIRES (High Resolution Graphics Library)
* Disk 11 – APPLECHOP (AppleChop High-Level Library)
* Disk 12 – SERIALPRN (Serial and Printer Libraries)
* Disk 13 – 80COL (80-Column Text Library)
* Disk 14 – MOCKINGBOARD (Mockingboard Sound Card Library)
* Disk 15 – DBLLORES (Double Low Resolution Graphics Library)
* Disk 16 – DBLHIRES (Double High Resolution Graphics Lib)
* Disk 17 – DETECT (Hardware Detection Library)
* Disk 18 – SORTSEARCH (Sort & Search Libraries)
* Disk 19 – TMENWIN (Text Menu and Text Window Libraries)
* Disk 20 – MISC (Miscellaneous Libraries)
* Disk 21 – MINIDISKA (Minified Libraries Disk A)
* Disk 22 – MINIDISKB (Minified Libraries Disk B)
* Disk 23 – UTILS (Utilities Disk)
* Disk 24 – DEMOSA (Demo Disk A)
* Disk 25 – DEMOSB (Demo Disk B)

**Standard Practices / Procedures**

AppleIIAsmLib follows certain conventions due to hardware limitations, operating system requirements, ease of reading, program flow and just plain old personal preference. While there might be times when these conventions are eschewed or changed entirely, you can reasonably expect, and be expected to follow, adherence to the following standards.

**Naming Conventions**

#### Filenames

Given the lack of directory structures in DOS 3.3, we are using a filename prefixes to indicate file types rather than suffixes. The extensions should be applied to a filename in this order:

* MIN: signifies that the code has been stripped of comments
* HEAD: indicates that this should be the first file included in the main source listing.
* HOOKS: indicates hooks related to the specific library’s macros and subroutines.
* SUB: signifies that the file holds a subroutine
* MAC: signifies a collection of macros
* LIB: signifies a collection of subroutines
* DEMO: signifies that the program is a sub-library demo
* <FILENAME>: the actual name of the subroutine, macro, our other file.

Additionally, Merlin Appends a ".S" to the end of a filename if it is saved as a source, and prepends the file with "T." to signify it being a text file. This prepended T. overrides our own naming conventions.

*Sample Filenames*

* T.MIN.MAC.STDIO
* T.SUB.TFILLA
* T.MIN.LIB.REQUIRED
* T.DEMO.STDIO

#### Variables

In Merlin Pro 8, assembler variables are preceded by a ] sign. These variables are temporarily assigned, and can be overwritten further down in the code. Unless highly impractical, constant hooks should use native assembly's system of assigning labels (just the label), as should hook entry points. The exception to this is within macro files, as these could easily lead to label conflicts.

#### Local Hooks

Local labels are preceded by a **:** sign (colon) in Merlin Pro 8. When at all possible, local subroutines should have local labels. This does not apply to Merlin variables.

#### Macros

Macros should be named with regard to mneumonic function, when possible, and should not exceed five characters unless absolutely necessary. Additionally, macros may use the following prefixes to signify their classification:

* @: signifies a higher-level control structure, such as @IF,@ELSE,@IFX.
* \_: signifies a macro mostly meant to be used internally, though it may have limited use outside of that context.

**Commenting Conventions**

***Inline Comments***

For the sake of beginners, at least every other directive should have an inline comment that describes what that line, or two lines, is accomplishing. Inline comments are added at the end of a line with a semicolon to denote the comment. Note that the audience for these comments are readers who may not have a good grasp of 6502 Assembly, so they should be as descriptive as possible.

*File Headers*

If the file does not hold a single subroutine, every file should include a header with the following information:

* A brief description of the file
* Any subroutines or macros that are included in the file, along with brief descriptions of each.
* Operating System, Main Author, Contact Information, Date of Last Revision, and intended Assembler.
* If the file contains a collection of macros, the subroutines used by the macros should be listed as well.

***Subroutine Headers***

All subroutines require headers that document its input, output, register, flag and memory destructions, minimum number of cycles used, and the size of the subroutine in bytes. Headers should all follow the same basic format, and a single space should be used to denote section inclusion.

*Macro Headers*

Macro headers should include a brief description of the macro, a listing of the parameters with short descriptions thereof, and a sample usage section.

***Other Comments***

If a section of code needs more explanation than can be explained at the end of a line (a common issue, since there is limited space on the Apple II screen), these should be placed just above the code in question using asterisks to denote the line is a comment. Have a blank comment line before and after the comment with only one asterisk, while using two asterisks for the lines with actual comments.

**Parameter Passing**

*Macro Parameters*

In general, macro parameters follow a specific hierarchy of order, with the exception of rare cases where another order makes more sense. The hierarchy is as follows:

Source > Destination > Index > Value > Other

Additionally, parameters passed to macros, when addresses are concerned, follow a strict distinction between literal addresses and indirect addresses. If the address passed is a literal value (preceded by # in Merlin Pro 8), then that is the actual address of the data in question. If, however, the address passed is non-literal, then the two-byte value at that address is used as the intended address to be used.

*Subroutine Parameters*

Subroutines are passed parameters by way of the registers, zero-page location values, or via the stack. Which one of these are used depends on the number of bytes being passed; different methods are used in order to maximize speed based on the needs of a subroutine.

If there are less than four bytes of data being passed, the registers are used; when a 16-bit address is being passed, it is convention to pass the low byte in **.A** and the high byte in **.X**.

If there are between four and ten total bytes in need of passing, the zero page is used. The locations used are defined in HEAD.REQUIRED, and specify three areas for 16-bit (two-byte word) values and four areas for 8-bit (single-byte) values. These are labeled as **WPAR1**, **WPAR2**, **WPAR3**, **BPAR1**, **BPAR2**, **BPAR3**, and **BPAR4**, respectively.

As a last resort, parameters are passed via that stack. This should, however, be a rare occurrence, as it is the slowest method available of passing parameters. Thankfully, since most of the subroutines in the library are meant to provide basic higher-level functionality, there is little need for recourse to this option.

By and large, all parameters should be one or two-byte values; if a string, array or other data type is being passed, its address is passed rather than the data itself.

Since the method of passing parameters can change from subroutine to subroutine, it is highly suggested to use the macros that call the subroutines when possible.

**Main Source Sequencing**

After necessary assembler directives, files should be loaded in the following order:

* HEAD.REQUIRED is **always** loaded first (PUT).
* MAC.REQUIRED **always** follows second (USE).
* Any HOOKS files should be loaded afterwards (PUT).
* Any MAC files being utilized should be loaded next.
* Now comes the source of the main listing that the programmer will write.
* After the main source, LIB.REQUIRED should be included (PUT).
* Then, any needed subroutine (SUB) files should be included (PUT).
* Any user-created PUT or USE files should be placed at the very end.

**Miscellaneous Standards**

### *Subroutine Independence*

Beyond needing the core required library files as well as the hook files for the library category in question, a subroutine should be able to operate independently of other subroutines in the library. This will generally mean some wasted bytes here and there, but this can be remedied by the end programmer if code size is a major concern.

### *Control Structures*

While a number of helpful, higher-level control structures are included as part of the core required library, subroutines in the library itself should refrain from using this shorthand. Control Structure Macros are preceded with a '@' sign to signify their classification as such. Exceptions can be given to control structures that merely extend existing directives for better use, such as BEQW being used to branch beyond the normal byte limit; such macros forego the preceding @-sign.

**Quick Reference: Macros**

**Disk 1: MAC.REQUIRED**

|  |  |  |  |
| --- | --- | --- | --- |
| **MACRO** | **DEPEND** | **PARAMETERS** | **RETURNS** |
| \_**AXLIT** | none | ]1 = memory address | **.A** = address low byte  **.X** = address high byte |
| Loads the **.A** and **.X** registers with appropriate values based on the status of the parameter as a literal. | | | |
| \_**AXSTR** | **\_AXLIT** | ]1 = memory address | **.A** = address low byte  **.X** = address high byte |
| Loads the **.A** and **.X** registers with appropriate address based on whether the parameter is a string or an address. | | | |
| **CLRHI** | **\_\_CLRHI** | ]1 = byte to clear the high nibble of | **.A** = cleared byte |
| Clears the high nibble of a byte and then returns new byte. | | | |
| **DUMP** | **\_AXLIT**; **\_\_DUMP** | ]1 = memory address  ]2 = number of bytes to dump | **.Y** = number of bytes displayed |
| Dumps the hex values at a given address for a given range. | | | |
| **ERRH** | **\_AXLIT**; **\_\_ERRH** | ]1 = memory address | none |
| Sets the Applesoft error handling routine address. | | | |
| **GRET** | **\_AXLIT**; **\_\_GETRET** | ]1 = destination address | **.Y** = return value length |
| Copies the data held into return to the given address. | | | |
| \_**ISLIT** | None | ]1 = memory address | See description |
| Pushes the appropriate values (two bytes) to the stack based on the status of the parameter as a literal. | | | |
| \_**ISSTR** | **\_ISLIT** | ]1 = memory address | See description |
| Pushes the appropriate address to the stack based on whether the parameter is a string or an address. | | | |
| \_**MLIT** | None | ]1 = memory address ]2 = destination zero-page address | See description |
| Loads the zero-page address with appropriate values based on the status of the parameter as a literal. | | | |
| \_**PRN** | **\_\_P** | ]1 = string | None |
| Sends the given ASCII string to COUT1 (the screen). | | | |
| \_**WAIT** | **\_\_W** | None | **.A** = keypress value |
| Waits until a key is pressed. | | | |

**Disk 1: MAC.COMMON**

|  |  |  |  |
| --- | --- | --- | --- |
| **MACRO** | **DEPEND** | **PARAMETERS** | **RETURNS** |
| **BEEP** | none | ]1 = number of rings | None |
| Ring the system bell. | | | |
| **DELAY** | **DELAYMS** | ]1 = number of milliseconds | None |
| Delay execution for a specified number of milliseconds. | | | |
| **MFILL** | **\_MLIT**; **MEMFILL** | ]1 = starting address  ]2 = length in bytes  ]3 = fill value | None |
| Fill a specified range of memory with a single value. | | | |
| **MMOVE** | **\_MLIT**; **MEMMOVE** | ]1 = starting address  ]2 = destination address  ]3 = length in bytes | None |
| Copy a specified range of memory to another memory address. | | | |
| **MSWAP** | **\_MLIT**; **MEMSWAP** | ]1 = first address  ]2 = second address  ]3 = length in bytes | None |
| Swap the values stored at two different ranges of memory. | | | |
| **ZLOAD** | **\_AXLIT**; **ZMLOAD** | ]1 = address to load from | None |
| Reload the previously stored values into the zero page. | | | |
| **ZSAVE** | **\_AXLIT**; **ZMSAVE** | ]1 = address to save to | None |
| Copy the values stored on the zero page that the library uses to a backup location. | | | |

**Disk 2: MAC.STDIO**

|  |  |  |  |
| --- | --- | --- | --- |
| **MACRO** | **DEPEND** | **PARAMETERS** | **RETURNS** |
| **COL40** | None | None | None |
| Turn on 40-column text mode. | | | |
| **COL80** | None | None | None |
| Turn on 80-column text mode. | | | |
| **CURB** | None | ]1 = number of spaces to move | None |
| Move cursor backward by a number of spaces. | | | |
| **CURD** | None | ]1 = number of spaces to move | None |
| Move cursor down by a number of spaces. | | | |
| **CURF** | None | ]1 = number of spaces to move | None |
| Move cursor forward by a number of spaces. | | | |
| **CURU** | None | ]1 = number of spaces to move | None |
| Move cursor up by a number of spaces. | | | |
| **DIE80** | None | none | None |
| Kill 80-column mode. | | | |
| **GKEY** | None | none | **.A** = key code |
| Wait for a keypress from end user. | | | |
| **INP** | **SINPUT** | none | **RETURN** = string with preceding length byte |
| Prompt end user to enter a string, followed by return. | | | |
| **MTXT0** | None | none | None |
| Turn of mousetext. | | | |
| **MTXT1** | None | none | None |
| Turn on mousetext. | | | |
| **PBX** | None | ]1 = Paddle Button Number; PB0, PB1, PB2 or PB3 | **.X** = 1 if button pushed |
| Read the state of a paddle button. | | | |
| **PDL** | None | ]1 = paddle number, usually 0 | **.Y** = paddle state |
| Read the state of the specified paddle. | | | |
| **PRN** | **\_MLIT**;  **DPRINT**;  **XPRINT**; | ]1 = literal string or address of string to print | None |
| Print a literal string or a null-terminated string at a given address. | | | |
| **RCPOS** | None | ]1 = X position  ]2 = Y position | **.A** = character code |
| Read the character on the screen at position X,Y. | | | |
| **SCPOS** | None | ]1 = X position  ]2 = Y position | None |
| Set the cursor position to X,Y. | | | |
| **SETCX** | None | ]1 = X position | None |
| Set the X position of the cursor. | | | |
| **SETCY** | None | ]1 = Y position | None |
| Set the Y position of the cursor. | | | |
| **SPRN** | **\_AXLIT**;  **PRNSTR** | ]1 = address of string | None |
| Print a string with a preceding length byte. | | | |
| **TCIRC** | **TCIRCLE** | ]1 = center X position  ]2 = center Y position  ]3 = radius  ]4 = fill character | None |
| Draw a text circle with the given radius at X,Y. | | | |
| **THLIN** | **THLINE** | ]1 = starting X position  ]2 = ending X position  ]3 = Y position  ]4 = fill character | None |
| Draw a horizontal text line. | | | |
| **TLINE** | **TBLINE** | ]1 = X origin  ]2 = Y origin  ]3 = X destination  ]4 = Y destination | None |
| Draw a text line from X,Y to X2,Y2. | | | |
| **TPUT** | **TXTPUT** | ]1 = X coordinate  ]2 = Y coordinate  ]3 = fill character | None |
| Plot a single text character. | | | |
| **TRECF** | **TRECTF** | ]1 = X origin  ]2 = Y origin  ]3 = X destination  ]4 = Y destination  ]5 = fill character | None |
| Plot a filled text rectangle from X,Y to X1,Y1. | | | |
| **TVLIN** | **TVLINE** | ]1 = Y origin  ]2 = Y destination  ]3 = X coordinate  ]4 = fill character | None |
| Draw a vertical text line. | | | |
| **WAIT** | None | None | **.A** = key code |
| Wait for a keypress without using COUT; no echo of key character. | | | |

**Disk 3: MAC.ARRAYS**

|  |  |  |  |
| --- | --- | --- | --- |
| **MACRO** | **DEPEND** | **PARAMETERS** | **RETURNS** |
| **DIM81** | **\_MLIT;**  **ADIM81** | ]1 = array address  ]2 = number of indices  ]3 = element length  ]4 = fill value | **RETURN** = total bytes used |
| Initialize an 8-bit, one-dimensional array. | | | |
| **GET81** | **\_AXLIT;**  **AGET81** | ]1 = array address  ]2 = element index | **.A** = length of data  **RETURN** = element data  **RETLEN** = length of data |
| Get the data stored in an element of an 8-bit, one-dimensional array. | | | |
| **PUT81** | **\_MLIT;**  **APUT81** | ]1 = source address  ]2 = array address  ]3 = element index | **.A** = element size  **.X** = element address low byte  **.Y** = element address high byte |
| Put data into an element in an 8-bit, one-dimensional array. | | | |
| **DIM82** | **\_MLIT;**  **ADIM82** | ]1 = array address  ]2 = 1st dimension indices  ]3 = 2nd dimension indices  ]4 = element length  ]5 = fill value | **RETURN** = total bytes used |
| Initialize an 8-bit, two-dimensional array. | | | |
| **GET82** | **\_MLIT;**  **AGET82** | ]1 = array address  ]2 = 1st dimension index  ]3 = 2nd dimension index | **.A** = length of data  **RETURN** = element data  **RETLEN** = length of data |
| Get the data stored in an element of an 8-bit, two-dimensional array. | | | |
| **PUT82** | **\_MLIT;**  **APUT82** | ]1 = source address  ]2 = array address  ]3 = 1st dimension index  ]4 = 2nd dimension index | **.A** = element size  **.X** = element address low byte  **.Y** = element address high byte |
| Put data into an element in an 8-bit, two-dimensional array. | | | |
| **DIM161** | **\_MLIT;**  **ADIM161** | ]1 = array address  ]2 = number of indices  ]3 = element length  ]4 = fill value | **RETURN** = total bytes used |
| Initialize an 16-bit, one-dimensional array. | | | |
| **GET161** | **\_MLIT;**  **AGET161** | ]1 = array address  ]2 = element index | **.A** = length of data  **RETURN** = element data  **RETLEN** = length of data |
| Get the data stored in an element of a 16-bit, one-dimensional array. | | | |
| **PUT161** | **\_MLIT;**  **APUT161** | **]1** = source address  **]2** = array address  **]3** = element index | **.A** = element size  **.X** = element address low byte  **.Y** = element address high byte |
| Put data into an element in a 16-bit, one-dimensional array. | | | |
| **DIM162** | **\_MLIT;**  **ADIM162** | **]1** = array address  **]2** = 1st dimension indices  **]3** = 2nd dimension indices  **]4** = element length  **]5** = fill value | **RETURN** = total bytes used |
| Initialize an 16-bit, two-dimensional array. | | | |
| **GET162** | **\_MLIT;**  **AGET162** | **]1** = array address  **]2** = 1st dimension index  **]3** = 2nd dimension index | **.A** = length of data  **RETURN** = element data  **RETLEN** = length of data |
| Get the data stored in an element of a 16-bit, two-dimensional array. | | | |
| **PUT162** | **\_MLIT;**  **APUT162** | **]1** = source address  **]2** = array address  **]3** = 1st dimension index  **]4** = 2nd dimension index | **.A** = element size  **.X** = element address low byte  **.Y** = element address high byte |
| Put data into an element in a 16-bit, two-dimensional array. | | | |

**Disk 4: MAC.MATH**

|  |  |  |  |
| --- | --- | --- | --- |
| **MACRO** | **DEPEND** | **PARAMETERS** | **RETURNS** |
| **ADD8** | none | ]1 = first addend ]2 = second addend | **.A** = sum **RETURN** = sum **RETLEN** = 1 |
| Add two 8-bit values and return an 8-bit sum. | | | |
| **SUB8** | none | ]1 = minuend ]2 = subtrahend | **.A** = difference **RETURN** = difference **RETLEN** = 1 |
| Subtract one 8-bit value from another and return an 8bit difference. | | | |
| **ADD16** | **\_MLIT**; **ADDIT16** | ]1 = first addend ]2 = second addend | **.A** = sum low byte **.X** = sub high byte **RETURN** = sum (2b) **RETLEN** = 2 |
| Add two 16-bit values and return a 16-bit sum. | | | |
| **SUB16** | **\_MLIT; SUBT16** | ]1 = Minuend ]2 = Subtrahend | **.A** = difference low byte **.X** = difference high byte **RETURN** = difference (2b) **RETLEN** = 2 |
| Subtract a 16-bit subtrahend from a 16-bit minuend and return a 16-bit difference. | | | |
| **MUL16** | **\_MLIT**; **MULT16** | ]1 = multiplicand ]2 = multiplier | **.A** = product low byte  **.X** = product high byte (16 bit) **RETURN** = 32-bit product, unsigned **RETLEN** = 4 |
| Multiply two 16-bit values and return a 16-bit product in **.A** and **.X** (low, high), and a 32-bit product in **RETURN** if both values are unsigned. | | | |
| **DIV16** | **\_MLIT; DIVD16** | ]1 = dividend ]2 = divisor | **.A** = result low byte **.X** = result high byte **RETURN** = result (2b) **RETLEN** = 2 |
| Divide a 16-bit dividend by a 16-bit divisor and return a 16-bit result. | | | |
| **RAND** | **RANDB** | ]1 = low boundary ]2 = high boundary | **.A** = pseudorandom value **RETURN** = value (1b) **RETLEN** = 1 |
| Return an 8-bit pseudo-random value between a low bound and a high bound. | | | |
| **CMP16** | **\_MLIT; COMP16** | ]1 = first comparison ]2 = second comparison | See detailed description |
| Compare two 16-bit values and change the status register appropriately. | | | |
| **MUL8** | **MULT8** | ]1 = multiplicand ]2 = multiplier | **.A** = product low byte **.X** = product high byte **RETURN** = product (2b) **RETLEN** = 2 |
| Multiply two 8-bit values and return a 16-bit product. | | | |
| **DIV8** | **DIVD8** | ]1 = dividend ]2 = divisor | **.A** = quotient **.X** = remainder  **RETURN** = quotient (1b) **RETLEN** = 1 |
| Divide one 8-bit value by another and return the quotient and remainder. | | | |
| **RND16** | **RAND16** | none | **.A** = pseudorandom value low byte **.X** = pseudorandom value high byte **RETURN** = pseudorandom value **RETLEN** = 2 |
| Generate a 16-bit pseudorandom value between 1 and 65536. | | | |
| **RND8** | **RAND8** | none | **.A** = pseudorandom value **RETURN** = pseudorandom value **RETLEN** = 1 |
| Generate an 8-bit pseudorandom value between 1 and 255. | | | |

**Disk 5: MAC.STRINGS**

|  |  |  |  |
| --- | --- | --- | --- |
| **MACRO** | **DEPEND** | **PARAMETERS** | **RETURNS** |
| **SCMP** | **STRCMP** | **]1** = first string to compare  **]2** = 2nd string to compare | **.Z** = 1 if strings equal  **.Z** = 0 if string !=  **.C** = 1 if 1st string < 2nd  **.C** = 0 if 2nd string >= 2nd |
| SCMP compares two strings and alters the status register accordingly. | | | |
| **SCAT** | **STRCAT** | **]1** = first string  **]2** second string | **.A** = new string length  **RETURN** = new string chars  **RETLEN** = length byte |
| Concatenates two strings. | | | |
| **SPRN** | **PRNSTR** | **]1** = string to print | **.A** = string length |
| Prints a string with a preceding length byte. | | | |
| **SPOS** | **SUBPOS** | **]1** = source string  **]2** = substring | **.A** = substring index  **RETURN** = substring index  **RETLEN** = 1 |
| Finds the index of a substring within a string. | | | |
| **SCOP** | **SUBCOPY** | **]1** = source string  **]2** = substring index  **]3** = substring length | **.A** = new string length  **RETURN** = new string chars  **RETLEN** = length byte |
| Copy a substring from a string. | | | |
| **SDEL** | **SUBDEL** | **]1** = source string  **]2** = substring index  **]3** = substring length | **.A** = new string length  **RETURN** = new string chars  **RETLEN** = length byte |
| Delete a substring from a string. | | | |
| **SINS** | **SUBINS** | **]1** = string address  **]2** = substring address  **]3** = substring index | **.A** = length byte  **RETURN** = new string chars  **RETLEN** = length byte |
| Insert a substring into a string at a given index. | | | |

**Disk 6: MAC.FILEIO**

|  |  |  |  |
| --- | --- | --- | --- |
| **MACRO** | **DEPEND** | **PARAMETERS** | **RETURNS** |
| **BSAVE** | **BINSAVE** | **]1** = string | none |
| Save memory to a binary file. | | | |
| **BLOAD** | **BINLOAD** | **]1** = string | none |
| Load memory from a binary file. | | | |
| **AMODE** | NONE | none | none |
| Feign Applesoft mode. | | | |
| **CMD** | **DOSCMD** | **]1** = string | none |
| Execute a DOS command. | | | |
| **FPRN** | **FPRINT** | **]1** = string | none |
| Output a null-terminated string to a file. | | | |
| **FINP** | **FINPUT** | none | **RETURN** = string chars  **RETLEN** = length byte  **.A** = length |
| Read a string from a text file. | | | |
| **SLOT** | NONE | **]1** = slot number | none |
| Change the RWTS slot. | | | |
| **DRIVE** | NONE | **]1** = drive number | none |
| Change the RWTS drive. | | | |
| **TRACK** | NONE | **]1** = track number | none |
| Change the RWTS track. | | | |
|  |  |  |  |
| **SECT** | NONE | **]1** = sector number | none |
| Change the RWTS sector. | | | |
| **DSKR** | NONE | none | none |
| Set RWTS to read mode. | | | |
| **DSKW** | NONE | none | none |
| Set RWTS to write mode. | | | |
| **DBUFF** | NONE | **]1** = buffer address | none |
| Set the disk buffer address. | | | |
| **DWRTS** | **DISKRW** | None | **.A** = error code  **RETURN** = byte returned or written  **RETLEN** = 1 |
| Read or write to the disk. | | | |

**Disk 7: CONVERT**

|  |  |  |  |
| --- | --- | --- | --- |
| **MACRO** | **DEPEND** | **PARAMETERS** | **RETURNS** |
| **I2STR** | **\_MLIT;**  **HEX2INTASC** | **]1** = value to convert | **.A** = string length  **RETURN** = string characters  **RETLEN** = length byte |
| Convert a 16-bit value to its string equivalent in decimal format. | | | |
| **STR2I** | **\_MSTR;**  **INTASC2HEX** | **]1** = string or address | **.A** = value low byte  **.X** = value high byte  **RETURN** = converted value  **RETLEN** = 2 |
| Convert a string containing a decimal value representation to its equivalent numerical value. | | | |
| **H2STR** | **HEX2HEXASC** | **]1** = value to convert | **RETURN** = string characters  **RETLEN** = 2 |
| Convert an 8-bit numeric value to its string equivalent in hexadecimal format. | | | |
| **STR2H** | **\_MSTR;**  **HEXASC2HEX** | **]1** = string or address | **.A** = converted value  **RETURN** = converted value  **RETLEN** = 1 |
| Convert a string containing a representation of a hexadecimal number value into its 8-bit value equivalent. | | | |
| **B2STR** | **HEX2BINASC** | **]1** = value to convert | **RETURN** = string characters  **RETLEN** = 8 |
| Convert an 8-bit numeric value into its string equivalent in binary format. | | | |
| **STR2B** | **\_MSTR;**  **BINASC2HEX** | **]1** = string or address | **.A** = converted value  **RETURN** = converted value  **RETLEN** = 1 |
| Convert a string containing the binary representation of a number and convert it to its actual value. | | | |

**Quick Reference: Subroutines**

**Disk 1: LIB.REQUIRED**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SUBROUTINE** | **FILE** | **DESTROYS** | **CYCLES** | **SIZE** |
| **\_\_CLRHI** | LIB.REQUIRED | ANZC | 16 | 6 |
| **\_\_DUMP** | LIB.REQUIRED | AXYMZCN | 184+ | 114 |
| **\_\_ERRH** | LIB.REQUIRED | AXYMZCN | 51 | 31 |
| **\_\_GETRET** | LIB.REQUIRED | AXYMZCN | 32+ | 18 |
| **\_\_P** | LIB.REQUIRED | AYNZCMS | 63+ | 33 |
| **\_\_W** | LIB.REQUIRED | ANZC | 18+ | 11 |

**Disk 1: Other Subroutines**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SUBROUTINE** | **FILE** | **DESTROYS** | **CYCLES** | **SIZE** |
| **DELAYMS** | SUB.DELAYMS | AXYNZCM | 39+ | 29 |
| **MEMFILL** | SUB.MEMFILL | AXYNZM | 117+ | 60 |
| **MEMMOVE** | SUB.MEMMOVE | AXYNZCM | 267+ | 150 |
| **MEMSWAP** | SUB.MEMSWAP | AXYNZCM | 100+ | 43 |
| **ZMLOAD** | SUB.ZMLOAD | AXYNZCM | 123+ | 71 |
| **ZMSAVE** | SUB.ZMSAVE | AXYNZCM | 138+ | 84 |

**Disk 2: STDIO**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SUBROUTINE** | **FILE** | **DESTROYS** | **CYCLES** | **SIZE** |
| **DPRINT** | SUB.DPRINT | AXYNZM | 61+ | 27 |
| **PRNSTR** | SUB.PRNSTR | AXYNVZCM | 28+ | 22 |
| **SINPUT** | SUB.SINPUT | AXYNVZC | 60+ | 45 |
| **TBLINE** | SUB.TBLINE | AXYNVZCM | 283+ | 188 |
| **TCIRCLE** | SUB.TCIRCLE | AXYNVZCM | 494+ | 420 |
| **THLINE** | SUB.THLINE | AXYNVBZCM | 90+ | 47 |
| **TRECTF** | SUB.TRECTF | AXYNVZCM | 69+ | 74 |
| **TVLINE** | SUB.TBLINE | AXYNVZCM | 33+ | 34 |
| **TXTPUT** | SUB.TXTPUT | AXYNVZCM | 29+ | 30 |
| **XPRINT** | SUB.XPRINT | AXYNVZCM | 63+ | 33 |

**Disk 3: ARRAYS**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SUBROUTINE** | **FILE** | **DESTROYS** | **CYCLES** | **SIZE** |
| **ADIM81** | SUB.ADIM81 | AXYNVZCM | 176+ | 160 |
| **AGET81** | SUB.AGET81 | AXYNVZC | 134+ | 134 |
| **APUT81** | SUB.APUT81 | AXYNVZCM | 170+ | 145 |
| **ADIM82** | SUB.ADIM82 | AXYNVZCM | 282+ | 244 |
| **AGET82** | SUB.AGET82 | AXYNVZCM | 288+ | 243 |
| **APUT82** | SUB.APUT82 | AXYNVZCM | 274+ | 239 |
| **ADIM161** | SUB.ADIM161 | AXYNVZCM | 172+ | 162 |
| **AGET161** | SUB.AGET161 | AXYNVZCM | 126+ | 135 |
| **APUT161** | SUB.APUT161 | AXYNVZCM | 181+ | 135 |
| **ADIM162** | SUB.ADIM162 | AXYNVZCM | 426+ | 312 |
| **AGET162** | SUB.AGET162 | AXYNVZCM | 410+ | 277 |
| **APUT162** | SUB.APUT162 | AXYNVZCM | 404+ | 273 |

**Disk 4: MATH**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SUBROUTINE** | **FILE** | **DESTROYS** | **CYCLES** | **SIZE** |
| **ADDIT16** | SUB.ADDIT16 | AXYNVBDIZCM | 43+ | 24 |
| **COMP16** | SUB.COMP16 | AXYNVBDIZCM | 51+ | 27 |
| **DIVD16** | SUB.DIVD16 | AXYNVBDIZCM | 92+ | 53 |
| **DIVD8** | SUB.DIVD8 | AXYNVBDIZCM | 58+ | 34 |
| **MULT16** | SUB.MULT16 | AXYNVBDIZCM | 101+ | 61 |
| **MULT8** | SUB.MULT8 | AXYNVBDIZCM | 81+ | 47 |
| **RAND16** | SUB.RAND16 | AXYNVBDIZCM | 90+ | 60 |
| **RAND8** | SUB.RAND8 | AXYNVBDIZCM | 44+ | 27 |
| **RANDB** | SUB.RANDB | AXYNVBDIZCM | 248+ | 476 |
| **SUBT16** | SUB.SUBT16 | AXYNVBDIZCM | 29+ | 13 |

**Disk 5: STRINGS**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SUBROUTINE** | **FILE** | **DESTROYS** | **CYCLES** | **SIZE** |
| **PRNSTR** | SUB.PRNSTR | AXYNVBDIZCM | 46+ | 26 |
| **STRCAT** | SUB.STRCAT | AXYNVBDIZCM | 115+ | 75 |
| **STRCMP** | SUB.STRCOMP | AXYNVBDIZCM | 61+ | 32 |
| **SUBCOPY** | SUB.SUBCOPY | AXYNVBDIZCM | 46+ | 27 |
| **SUBDEL** | SUB.SUBDEL | AXYNVBDIZCM | 79+ | 47 |
| **SUBINS** | SUB.SUBINS | AXYNVBDIZCM | 106+ | 67 |
| **SUBPOS** | SUB.SUBPOS | AXYNVBDIZCM | 150+ | 103 |

**Disk 6: FILEIO**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SUBROUTINE** | **FILE** | **DESTROYS** | **CYCLES** | **SIZE** |
| **BINLOAD** | SUB.BINLOAD | AXYNVBDIZCM | 124+ | 82 |
| **BINSAVE** | SUB.BINSAVE | AXYNVBDIZCM | 124+ | 82 |
| **DISKRW** | SUB.DISKRW | AXYNVBDIZCM | 41+ | 34 |
| **DOSCMD** | SUB.DOSCMD | AXYNVBDIZCM | 76+ | 52 |
| **FPRINT** | SUB.FPRINT | AXYNVBDIZCM | 63+ | 37 |
| **FINPUT** | SUB.FINPUT | AXYNVBDIZCM | 54+ | 41 |
| **FPSTR** | SUB.FPSTR | AXYNVBDIZCM | 38+ | 25 |

**Disk 7: Convert**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SUBROUTINE** | **FILE** | **DESTROYS** | **CYCLES** | **SIZE** |
| **BINASC2HEX** | SUB.BINASC2HEX | AXYNVBDIZCM | 400+ | 320 |
| **HEX2BINASC** | SUB.HEX2BINASC | AXYNVBDIZCM | 134+ | 159 |
| **HEX2HEXASC** | SUB.HEX2HEXASC | AXYNVBDIZCM | 80+ | 77 |
| **HEX2INTASC** | SUB.HEX2INTASC | AXYNVBDIZCM | 226+ | 352 |
| **HEXASC2HEX** | SUB.HEXASC2HEX | AXYNVBDIZCM | 82+ | 61 |
| **INTASC2HEX** | SUB.INTASC2HEX | AXYNVBDIZCM | 266+ | 196 |

**Part II**

Detailed Descriptions and Listings

**Disk 1: REQCOM**

The first disk in the collection holds all of the required files, subroutines and macros as well as the library of common macros and subroutines.

**REQUIRED LIBRARY FILES**

All AppleIIAsm macro and subroutine libraries require these core macros and routines to function properly. For the most part, the average programmer can ignore the macros and subroutines here, as they will be used rarely outside of the inner workings of the library itself. However, a working understanding of how the library works might be necessary in cases where optimizations are required that need to deconstruct the library to its barest bones (or maybe you just want to know for the sake of knowing!). Thus, these macros and subroutines are documented here.

The required library consists of:

* HEAD.REQUIRED
* MAC.REQUIRED
* LIB.REQUIRED

**HEAD.REQUIRED** is a header that must be included in a source file prior to any other file. It includes basic variable declarations and hooks needed by the rest of the library.

**MAC.REQUIRED** is a collection of macros that the rest of the library uses. It is also important to note that the macro library itself uses its own macros, primarily for parsing literal values and indirect addresses, but also for passing the appropriate values to each subroutine.

**LIB.REQUIRED** is the collection of actual subroutines used by the rest of the library. None of these subroutines call any other, but they are all included in the same file for ease of inclusion (this is impractical for other libraries, as Merlin 8 Pro breaks down when files get too large).

The individual subroutines and macros contained within each file are explained prior to the listing of each.

**HEAD.REQUIRED**

The required library header, which should be included prior to any other file, does the following:

* Establishes a 34 byte data area for a jump table starting at the second byte of the source program; this is why it must be included before any other file. The first two bytes hold the address of the start of the main program, while the following 32 bytes are available to create custom jump tables.
* Creates a 20 byte area of memory for variable declarations. These are defined at the beginning of each subroutine.
* Declares a single length byte for return values from the library subroutines, as well as another 256 bytes to hold any return values.
* Declare four two-byte addresses of the zero page for use in indirect addressing. Note that the library only uses parts of the zero pages that are not used by DOS, ProDOS, Applesoft or the Monitor.
* Declares zero-page bytes that are used as scratchpads. These values are meant to be stored temporarily, and should not be relied on outside of a given subroutine.
* Declares an additional two bytes of the zero page to hold return addresses.
* Establishes zero-page memory addresses to hold one- or two-byte values that are passed to the various subroutines in the library.
* Declares any hooks necessary for the operation of the library as a whole.

\*

\*``````````````````````````````\*

\* HEAD.REQUIRED \*

\* \*

\* THIS HEADER MUST BE THE \*

\* INCLUDED BEFORE ANY OTHER \*

\* CODE IN ORDER FOR THE PROPER \*

\* FUNCTIONING OF THE LIBRARY. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 30-JUN-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\* VARIABLE DECLARATIONS \*\*\*\*\*\*\*\*

\*

\*\* JUMP TABLE SETUP. THIS IS FOR LOADING

\*\* SUBROUTINES INTO MEMORY FOR ACCESS BY

\*\* EXTERNAL EXECUTIONS. NOTE THAT THIS

\*\* SHOULD ALWAYS START AT SECOND BYTE OF

\*\* CODE IN THE PROGRAM SO THAT ITS

\*\* LOCATION IN MEMORY IS EASILY KNOWN.

\*

JUMPTBL JMP MAIN\_START ; \*\* ALWAYS \*\* START WITH

; JUMP TO MAIN\_START

DS 32 ; 16 MORE ENTRIES

\*

\*\* 20 BYTES FOR VARIABLES

\*

VARTAB DS 20

\*

\*\* 256 BYTES DEDICATED TO RETURN

\*\* VALUES OF VARIABLE LENGTH; CAN BE

\*\* MODIFIED TO SUIT SMALLER OR LARGER

\*\* NEEDS.

\*

RETLEN DS 1 ; RETURN VALUE BYTE LENGTH

RETURN DS 256

\*

\*\* ADDRESS STORAGE LOCATIONS FOR

\*\* INDIRECT ADDRESSING.

\*

ADDR1 EQU $06 ; AND $07

ADDR2 EQU $08 ; AND $09

ADDR3 EQU $EB ; AND $EC

ADDR4 EQU $ED ; AND $EE

\*

\*\* SCRATCHPAD ZERO PAGE LOCATIONS AND

\*\* DEDICATED ZERO PAGE ADDRESS TO HOLD

\*\* A RETURN ADDRESS PASSED VIA THE STACK

\*

SCRATCH EQU $19

SCRATCH2 EQU $1E

RETADR EQU $FE ; AND $FF

\*

\*\* ZERO PAGE ADDRESSES DEDICATED TO PASSING

\*\* BACK RESULTS WHEN THERE ARE MORE THAN

\*\* THREE BYTES BEING PASSED (AXY) AND THE

\*\* USE OF THE STACK IS IMPRACTICAL OR TOO SLOW

\*

RESULT EQU $FA

RESULT2 EQU $FC

\*

\*\* WORD AND BYTE PARAMETER SPACE USED

\*\* BY APPLEIIASM MACROS

\*

WPAR1 EQU $FA

WPAR2 EQU $FC

WPAR3 EQU $FE

BPAR1 EQU $EF

BPAR2 EQU $E3

BPAR3 EQU $1E

BPAR4 EQU $19

\*

\*\* VARIOUS HOOKS USED BY ALL ROUTINES

\*

REENTRY EQU $3D0

\*

MAIN\_START

\*

**MAC.REQUIRED**

The MAC.REQUIRED file holds all of the macros used by the rest of the AppleIIAsm library. Currently, this includes:

* \_AXLIT
* \_AXSTR
* DUMP
* ERRH
* GRET
* \_ISLIT
* \_ISSTR
* \_MLIT
* \_PRN
* \_WAIT

\*

\*``````````````````````````````\*

\* MAC.REQUIRED \*

\* \*

\* MACROS USED FOR CORE UTILS \*

\* AND LIBRARY ROUTINES. NOTE \*

\* THAT THE LIBRARIES DO NOT \*

\* USE THESE MACROS, BUT MAY \*

\* USE THE ROUTINES. THESE ARE \*

\* MERELY PROVIDED FOR THE SAKE \*

\* OF CONVENIENCE. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 30-JUN-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\* \*

\* SUBROUTINE FILES NEEDED \*

\* \*

\* LIB.REQUIRED \*

\* \*

\* MACROS INCLUDED: \*

\* \*

\* \_MLIT : IS LITERAL? (ZERO) \*

\* \_ISLIT : IS LITERAL? (STACK) \*

\* \_AXLIT : IS LITERAL? (REGS) \*

\* \_ISSTR : IS STRING? (STACK) \*

\* \_AXSTR : IS STRING? (REGS) \*

\* GRET : GET RETURN \*

\* DUMP : DUMP MEMORY \*

\* \_PRN : PRINT STRING \*

\* \_WAIT : GET KEYPRESS \*

\* ERRH : SET ERROR ROUTINE \*

\* CLRHI : CLEAR HIGH NIBBLE \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

|  |
| --- |
| **\_MLIT (macro)**  **Input**:  ]1 = Memory Address  ]2 = Destination Address  **Output**:  Correct address to  destination address  **Destroys**: ANZM  **Cycles**: 20  **Size**: 24 bytes |

**MAC.REQUIRED >> \_MLIT**

The **\_MLIT** macro is used to determine if an address passed to the macro is a literal. If it is, that value is passed to the specified zero-page location for use in another macro or subroutine; if not, then the two bytes located at the specified address are copied to the zero-page address.

For the most part, **\_MLIT** is not used beyond the core library macros. However, it can be freely utilized by your own code for passing parameters as well.

\*

\*``````````````````````````````\*

\* \_MLIT \*

\* \*

\* CHECKS IF PARAMETER IS A \*

\* LITERAL OR NOT, AND SETS THE \*

\* LO AND HI IN THE SPECIFIED \*

\* MEMORY ADDRESS. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = MEMORY ADDRESS BYTE \*

\* ]2 = ZERO PAGE ADDRESS \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* \_MLIT #$6000 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\_MLIT MAC

IF #=]1 ; IF ]1 IS A LITERAL

LDA ]1/$100 ; GET HI

STA ]2+1

LDA ]1 ; GET LO

STA ]2

ELSE ; ]1 IS ADDRESS

LDA ]1+1 ; SO GET HIGH VAL FROM ADDR

STA ]2+1

LDA ]1 ; THEN LO VAL

STA ]2

FIN

<<<

|  |
| --- |
| **\_ISLIT (macro)**  **Input**:  ]1 = Memory Address  **Output**:  Correct address to  6502 stack  **Destroys**: ANZM  **Cycles**: 20  **Size**: 16 bytes |

**MAC.REQUIRED >> \_ISLIT**

The **\_ISLIT** macro is used to determine if an address passed to the macro is a literal. If it is, that value is pushed to the stack for use in another macro or subroutine; if not, then the two bytes located at the specified address are pushed.

For the most part, **\_ISLIT** is not used beyond the core library macros. However, it can be freely utilized by your own code for passing parameters as well.

\*

\*``````````````````````````````\*

\* \_ISLIT \*

\* \*

\* CHECKS IF THE PARAMETER IS \*

\* A LITERAL OR NOT, THEN \*

\* PUSHES THE LO AND HI AS \*

\* NEEDED. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = MEMORY ADDRESS BYTE \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* \_ISLIT #$6000 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\_ISLIT MAC

IF #=]1 ; IF ]1 IS A LITERAL

LDA ]1/$100 ; GET HI

PHA

LDA ]1 ; GET LO

PHA

ELSE ; ]1 IS ADDRESS

LDA ]1+1 ; SO GET HIGH VAL FROM ADDR

PHA

LDA ]1 ; THEN LO VAL

PHA

FIN

<<<

|  |
| --- |
| **\_AXLIT (macro)**  **Input**:  ]1 = Memory Address  **Output**:  Correct address to  **.A** (low) and **.X** (high)  **Destroys**: AXNZ  **Cycles**: 6  **Size**: 4 bytes |

**MAC.REQUIRED >> \_AXLIT**

The **\_AXLIT** macro is used to determine if an address passed to the macro is a literal. If it is, that address is loaded into the **.A** register (low byte) and the **.X** register (high byte) for use in another macro or subroutine; if not, then the two bytes located at the specified address are loaded into **.A** and **.X** instead.

For the most part, **\_AXLIT** is not used beyond the core library macros. However, it can be freely utilized by your own code for passing parameters as well.

\*

\*``````````````````````````````\*

\* \_AXLIT \*

\* \*

\* CHECKS IF PARAMETER IS A \*

\* LITERAL OR NOT, AND SETS THE \*

\* LO AND HI IN .A AND .X. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = MEMORY ADDRESS BYTE \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* \_AXLIT #$6000 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\_AXLIT MAC

IF #=]1 ; IF ]1 IS A LITERAL

LDX ]1/$100 ; GET HI

LDA ]1 ; GET LO

ELSE ; ]1 IS ADDRESS

LDX ]1+1 ; SO GET HIGH VAL FROM ADDR

LDA ]1 ; THEN LO VAL

FIN

<<<

|  |
| --- |
| **\_ISSTR (macro)**  **Input**:  ]1 = Memory Address  **Output**:  Correct address of  String to the stack  **Destroys**: ANZM  **Cycles**: 13+  **Size**: 9+ bytes |

**MAC.REQUIRED >> \_ISSTR**

The **\_ISSTR** macro checks to see whether the parameter passed is a string. If it is, the string is then officially coded into machine code at the current address, which is then passed to the calling macro or subroutine via the stack. If the parameter isn’t a string, then it is assumed to be a two-byte address, which is passed to **\_ISLIT** for further parsing.

\*

\*``````````````````````````````\*

\* \_ISSTR \*

\* \*

\* CHECKS IF PARAMETER IS A \*

\* STRING, AND IF SO PROVIDE IT \*

\* WITH AN ADDRESS. IF NOT, \*

\* CHECK IF IT'S A LITERAL AND \*

\* PASS ACCORDINGLY. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = MEMORY ADDRESS BYTE \*

\* OR STRING \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* \_ISSTR "TESTING" \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\_ISSTR MAC

IF "=]1 ; IF ]1 IS A STRING

JMP \_\_STRCONT

]STRTMP STR ]1

\_\_STRCONT

\*

LDA #>]STRTMP ; GET HI

PHA

LDA #<]STRTMP ; GET LO

PHA

ELSE ; ]1 IS ADDRESS

\_ISLIT ]1

FIN

<<<

|  |
| --- |
| **\_AXSTR (macro)**  **Input**:  ]1 = Memory Address  **Output**:  Correct address of string  To **.A** (low) and **.X** (high)  **Destroys**: ANZM  **Cycles**: 7  **Size**: 7+ bytes |

**MAC.REQUIRED >> \_AXSTR**

The **\_AXSTR** macro checks to see whether the parameter passed is a string. If it is, the string is then officially coded into machine code at the current address, which is then passed to the calling macro or subroutine via **.A** register (low byte) and the **.X** register (high byte). If the parameter isn’t a string, then it is assumed to be a two-byte address, which is passed to **\_AXLIT** for further parsing.

\*

\*``````````````````````````````\*

\* \_AXSTR \*

\* \*

\* CHECKS IF PARAMETER IS A \*

\* STRING, AND IF SO PROVIDES \*

\* AN ADDRESS FOR IT. IF NOT, \*

\* CHECK IF IT'S A LITERAL, AND \*

\* STORE THE HI A LO BYTES IN \*

\* .A AND .X. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = MEMORY ADDRESS BYTE \*

\* OR STRING \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* \_AXSTR "TESTING" \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\_AXSTR MAC

IF "=]1 ; IF ]1 IS A STRING

JMP \_\_STRCNT2

]STRTMP STR ]1

\_\_STRCNT2

\*

LDX #>]STRTMP ; GET HI

LDA #<]STRTMP ; GET LO

ELSE ; ]1 IS ADDRESS

\_AXLIT ]1

FIN

<<<

|  |
| --- |
| **GRET (macro)**  **Input**:  ]1 = Memory Address  **Output**:  **RETURN** data copied to  new address.  **Destroys**: AXYNZCM  **Cycles**: 44+  **Size**: 25 bytes |

**MAC.REQUIRED >> GRET**

The **GRET** macro first sends its only parameter to **\_AXLIT** for parsing, then calls the **\_\_GETRET** subroutine, which copies the data in **RETURN** to the passed address.

\*

\*``````````````````````````````\*

\* GRET \*

\* \*

\* COPY THE VALUE IN RETURN AND \*

\* PLACE IT IN GIVEN ADDRESS. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = MEMORY ADDRESS BYTE \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* GRET #$6000 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

GRET MAC

\_AXLIT ]1

JSR \_\_GETRET

<<<

|  |
| --- |
| **DUMP (macro)**  **Input**:  ]1 = Memory Address  ]2 = Byte Length  **Output**:  Memory contents to  The screen  **Destroys**: AXYNCZM  **Cycles**: 198  **Size**: 14 bytes |

**MAC.REQUIRED >> DUMP**

The **DUMP** macro dumps the values at the specified memory address to the screen (**COUT1**). The Hexadecimal values are converted to their textual equivalents.

The first parameter, the starting address, is first sent to **\_AXLIT** for parsing as a literal or indirect address.

\*

\*``````````````````````````````\*

\* DUMP \*

\* \*

\* DUMP THE HEX AT A GIVEN \*

\* ADDRESS. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = MEMORY ADDRESS BYTE \*

\* ]2 = LENGTH IN BYTES \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* DUMP #$6000;#10 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

DUMP MAC

\_AXLIT ]1

LDY ]2

JSR \_\_DUMP

<<<

|  |
| --- |
| **\_PRN (macro)**  **Input**:  ]1 = Literal String  **Output**:  String to the screen  **Destroys**: AYNZCMS  **Cycles**: 69+  **Size**: 9 bytes |

**MAC.REQUIRED >> \_PRN**

The **\_PRN** macro is simply a quick literal string printing function for mostly debugging purposes. Unlike more versatile macros in STDIO, this macro only accepts a string as its sole parameter.

\*

\*``````````````````````````````\*

\* \_PRN \*

\* \*

\* PRINT A STRING OR ADDRESS. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = MEMORY ADDRESS BYTE \*

\* OR STRING \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* \_PRN "TESTING" \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\_PRN MAC

JSR \_\_P

ASC ]1

HEX 00

<<<

|  |
| --- |
| **\_WAIT (macro)**  **Input**:  none  **Output**:  **.A** = key value  **Destroys**: ANCZ  **Cycles**: 24+  **Size**: 3 bytes |

**MAC.REQUIRED >> \_WAIT**

The **\_WAIT** macro simply waits for a keypress, and returns the associated value in **.A** after a key is pressed. This is nearly a carbon-copy of the equivalent macro in STDIO, but is also included in the required library for debugging purposes. If memory use is an extreme concern, a negligible 11 bytes can be saved by removing the **\_\_W** from LIB.REQUIRED.

\*

\*``````````````````````````````\*

\* \_WAIT \*

\* \*

\* WAIT FOR A KEYPRESS. \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\_WAIT MAC

JSR \_\_W

<<<

|  |
| --- |
| **ERRH (macro)**  **Input**:  ]1 = memory address  **Output**:  none  **Destroys**: AXYCZNM  **Cycles**: 63  **Size**: 9 bytes |

**MAC.REQUIRED >> ERRH**

The **ERRH** macro parses the address parameter into **.A** and **.X**, then calls the **\_\_ERRH** subroutine. This simply sets the error-handling address for Applesoft. This is particularly important when file operations are concerned.

\*

\*``````````````````````````````\*

\* ERRH \*

\* \*

\* SET THE ERROR HANDLING HOOK \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = MEMORY ADDRESS BYTE \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* ERRH #$6000 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

ERRH MAC

\_AXLIT

JSR \_\_ERRH

<<<

**MAC.REQUIRED >> CLRHI**

|  |
| --- |
| **CLRHI (macro)**  **Input**:  **.A** = byte  **Output**:  **.A** = byte  **Destroys**: ANZC  **Cycles**: 22  **Size**: 5 bytes |

The **CLRHI** macro clears the high nibble of the byte held in the **.A** register. This is often used for data type conversions.

\*

\*``````````````````````````````\*

\* CLRHI \*

\* \*

\* CLEAR HI NIBBLE OF A BYTE \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = BYTE TO CLEAR \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* CLRHI #$FF \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

CLRHI MAC

LDA ]1

JSR \_\_CLRHI

<<<

**LIB.REQUIRED**

LIB.REQUIRED contains all of the subroutines that all other libraries in the collection need to operate. This includes:

* \_\_CLRHI
* \_\_DUMP
* \_\_GETRET
* \_\_ERRH
* \_\_P
* \_\_W

\*

\*``````````````````````````````\*

\* LIB.REQUIRED \*

\* \*

\* LIBRARY OF REQUIRED ROUTINES \*

\* AS PART OF THE APPLEIIASM \*

\* MACRO AND SUBROUTINE LIBRARY \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 30-JUN-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* LICENSE: APACHE 2.0 \*

\* OS: DOS 3.3 \*

\* \*

\* SUBROUTINES: \*

\* \*

\* \_\_GETRET : GET RETURN VAL \*

\* \_\_CLRHI : CLEAR HI NIBBLE \*

\* \_\_DUMP : DUMP MEMORY \*

\* \_\_P : PRINT \*

\* \_\_W : WAIT \*

\* \_\_ERRH : HANDLE ERRORS \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\*\* LIBRARY-SPECIFIC VARIABLES

\*

]RIGHT DS 1

]LEFT DS 1

]LENGTH DS 1

]A DS 1 ; REGISTER .A BACKUP

]X DS 1 ; REGISTER .X BACKUP

]Y DS 1 ; REGISTER .Y BACKUP

]C DS 1 ; CARRY FLAG BACKUP

]Z DS 1 ; ZERO FLAG BACKUP

]N DS 1 ; NEGATIVE FLAG BACKUP

]O DS 1 ; OVERFLOW FLAG BACKUP

]HEXTAB ASC "0123456789ABCDEF"

\*

\*\* LIBRARY-SPECIFIC HOOKS

\*

]COUT EQU $FDF0 ; SCREEN OUTPUT ROUTINE

]KYBD EQU $C000 ; KEYBOARD INPUT

]STROBE EQU $C010 ; KEYBOARD STROBE

|  |
| --- |
| **\_\_GETRET (sub)**  **Input**:  **.A** = address low byte  **.X** = address high byte  **RETURN** = data string  **RETLEN** = string length  **Output**:  .**Y** = data length  **RETURN** is copied to  Given address.  **Destroys**: AXYNZCM  **Cycles**: 32+  **Size**: 18 bytes |

**LIB.REQUIRED >> \_\_GETRET**

The **\_\_GETRET** subroutine copies the data in **RETURN**, which often holds the results of another subroutine’s actions, to another memory address for more permanent storage. The length of the data is returned in the **.Y** register. Note that **RETLEN** is not explicitly copied as part of the data; this must be done manually.

\*

\*``````````````````````````````\*

\* \_\_GETRET (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* .A = ADDRESS LOBYTE \*

\* .X = ADDRESS HIBYTE \*

\* RETURN = DATA STRING \*

\* RETLEN = DATA STRING LENGTH \*

\* \*

\* OUTPUT: \*

\* \*

\* COPIES CONTENT OF RETURN \*

\* TO SPECIFIED ADDRESS. \*

\* \*

\* .Y = RETURN LENGTH \*

\* \*

\* DESTROYS: AXYNVBDIZCMS \*

\* ^^^^ ^^^ \*

\* \*

\* CYCLES: 32+ \*

\* SIZE: 18 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\_\_GETRET

STA ADDR1 ; LOBYTE PASSED IN .A

STX ADDR1+1 ; HIBYTE PASSED IN .X

LDY #255 ; RESET COUNTER

:LP

INY ; INCREASE COUNTER

LDA RETURN,Y ; LOAD BYTE IN RETURN AT

STA (ADDR1),Y ; COUNTER OFFSET; STORE AT

CPY RETLEN ; NEW LOCATION

BNE :LP ; IF COUNTER < RETLEN, LOOP

RTS

|  |
| --- |
| **\_\_CLRHI (sub)**  **Input**:  **.A** = byte to clear high  nibble  **Output**:  .**A** = cleared byte  **Destroys**: ANZC  **Cycles**: 16  **Size**: 6 bytes |

**LIB.REQUIRED >> \_\_CLRHI**

The **\_\_CLRHI** subroutine takes a single byte passed in the accumulator and clears the high nibble to zero. The new value is then returned in the accumulator as well.

\*

\*``````````````````````````````\*

\* \_\_CLRHI (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* .A = BYTE TO CLEAR HIBITS \*

\* \*

\* OUTPUT: \*

\* \*

\* CLEARS 4 HIBITS FROM BYTE \*

\* \*

\* .A = CLEARED BYTE \*

\* \*

\* DESTROYS: AXYNVBDIZCMS \*

\* ^ ^ ^^ \*

\* \*

\* CYCLES: 16 \*

\* SIZE: 6 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\_\_CLRHI

\*

AND #$F0 ; CLEAR 4 RIGHT BITS

LSR ; MOVE BITS RIGHT

LSR ; MOVE BITS RIGHT

LSR ; MOVE BITS RIGHT

LSR ; MOVE BITS RIGHT

RTS

|  |
| --- |
| **\_\_DUMP (sub)**  **Input**:  **.A** = address low byte  **.X** = address high byte  **.Y** = range length  **Output**:  Outputs values stored at  Address range to screen  **Destroys**: AXYNZCM  **Cycles**: 184+  **Size**: 114 bytes |

**LIB.REQUIRED >> \_\_DUMP**

The **\_\_DUMP** subroutine outputs the values stored at a given address range. The values are first converted from hexadecimal to a string equivalent, then sent to **COUT**. This is primarily used for debugging purposes, as there are not too many cases where the end user would need to see the actual values stored at a given address.

\*

\*``````````````````````````````\*

\* \_\_DUMP: (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* .A = ADDRESS LOBYTE \*

\* .X = ADDRESS HIBYTE \*

\* .Y = NUMBER OF BYTES \*

\* \*

\* OUTPUT: \*

\* \*

\* OUTPUTS DATA LOCATED AT THE \*

\* SPECIFIED ADDRESS IN HEX \*

\* FORMAT FOR SPECIFIED NUMBER \*

\* OF BYTES. \*

\* \*

\* DESTROYS: AXYNVBDIZCMS \*

\* ^^^^ ^^^ \*

\* \*

\* CYCLES: 184+ \*

\* SIZE: 114 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\_\_DUMP

STY ]LENGTH ; LENGTH PASSED IN .Y

STA ADDR1 ; ADDRESS LOBYTE IN .A

STX ADDR1+1 ; ADDRESS HIBYTE IN .X

LDA #$8D ; LOAD CARRIAGE RETURN

JSR ]COUT ; SEND TO COUT

LDA ADDR1+1 ; GET ADDRESS HIBYTE

JSR \_\_CLRHI ; CLEAR HIBITS

TAX ; TRANSFER T .X

LDA ]HEXTAB,X ; LOAD HEX CHAR FROM TABLE AT .X

JSR ]COUT ; SEND TO COUT

LDA ADDR1+1 ; LOAD ADDRESS HIBYTE AGAIN

AND #$0F ; CLEAR LOBITS

TAX ; TRANSER TO .X

LDA ]HEXTAB,X ; LOAD HEX CHAR FROM TABLE AT .X

JSR ]COUT ; SENT TO COUT

LDA ADDR1 ; LOAD LOBYTE

JSR \_\_CLRHI ; CLEAR HIBITS

TAX ; TRANSFER TO .X

LDA ]HEXTAB,X ; LOAD HEXCHAR AT .X

JSR ]COUT ; SEND TO COUT

LDA ADDR1 ; LOAD LOBYTE AGAIN

AND #$0F ; CLEAR LOBITS

TAX ; TRANSFER T .X

LDA ]HEXTAB,X ; LOAD HEXCHAR AT .X

JSR ]COUT ; SEND TO COUT

LDA #":" ;

JSR ]COUT ; SEND COLON TO COUT

LDA #" "

JSR ]COUT ; SEND SPACE TO COUT

LDY #0 ; RESET COUNTER

:LP

LDA (ADDR1),Y ; LOAD BYTE FROM ADDRESS

JSR \_\_CLRHI ; AT COUNTER OFFSET; CLEAR HIBITS

STA ]LEFT ; SAVE LEFT INDEX

LDA (ADDR1),Y ; RELOAD

AND #$0F ; CLEAR LOBITS

STA ]RIGHT ; SAVE RIGHT INDEX

LDX ]LEFT ; LOAD LEFT INDEX

LDA ]HEXTAB,X ; GET NIBBLE CHAR

JSR ]COUT ; SEND TO COUT

LDX ]RIGHT ; LOAD RIGHT INDEX

LDA ]HEXTAB,X ; GET NIBBLE CHAR

JSR ]COUT ; SEND TO COUT

LDA #160 ; LOAD SPACE

JSR ]COUT ; SEND TO COUT

INY ; INCREASE COUNTER

CPY ]LENGTH ; IF COUNTER < LENGTH

BNE :LP ; CONTINUE LOOP

RTS ; ELSE, EXIT

|  |
| --- |
| **\_\_P (sub)**  **Input**:  ASCII input is placed  After call to subroutine  **Output**:  ASCII string to screen  **Destroys**: AYNZCMS  **Cycles**: 63+  **Size**: 33 bytes |

**LIB.REQUIRED >> \_\_P**

The **\_\_P** subroutine simply outputs a given literal string to the screen. This is primarily for debugging purposes; you should use the subroutines in the **STDIO** package for more robust and flexible screen output. The subroutine prints each character in the string consecutively until a null character is encountered, at which point control is returned to the calling routine.

Note that a **JSR** to this subroutine should be followed by the string of characters you wish to print. In Merlin, this would be accomplished by using the **ASC** instruction, followed by a **HEX 00**.

\*

\*``````````````````````````````\*

\* \_\_P: (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* ASC STRING FOLLOWING CALL \*

\* TERMINATED WITH A 00 BYTE \*

\* \*

\* OUTPUT: \*

\* \*

\* CONTENTS OF STRING. \*

\* \*

\* DESTROYS: AXYNVBDIZCMS \*

\* ^ ^^ ^^^^ \*

\* \*

\* CYCLES: 63+ \*

\* SIZE: 33 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\_\_P

PLA ; PULL RETURN LOBYTE

STA ADDR1 ; STORE TO ZERO PAGE

PLA ; PULL RETURN HIBYTE

STA ADDR1+1 ; STORE TO ZERO PAGE

LDY #1 ; SET OFFSET TO PLUS ONE

:LP LDA (ADDR1),Y ; LOAD BYTE AT OFFSET .Y

BEQ :DONE ; IF BYTE = 0, QUIT

JSR ]COUT ; OTHERWISE, PRINT BYTE

INY ; INCREASE OFFSET

BNE :LP ; IF .Y <> 0, CONTINUE LOOP

:DONE CLC ; CLEAR CARRY FLAG

TYA ; TRANSFER OFFSET TO .A

ADC ADDR1 ; ADD OFFSET TO RETURN ADDRESS

STA ADDR1 ; STORE TO RETURN ADDRESS LOBYTE

LDA ADDR1+1 ; DO THE SAME WITH THE HIBYTE

ADC #0 ; CARRY NOT RESET, SO INC HIBYTE

PHA ; IF NEEDED; THEN, PUSH HIBYTE

LDA ADDR1 ; LOAD LOBYTE

PHA ; PUSH LOBYTE

RTS ; EXIT

|  |
| --- |
| **\_\_W (sub)**  **Input**:  none  **Output**:  **.A** = key code  **Destroys**: ANZC  **Cycles**: 18+  **Size**: 11 bytes |

**LIB.REQUIRED >> \_\_W**

The **\_\_W** subroutine simply loops until a keypress is detected, then returns control back to the calling routine. The code for the key pressed is stored in the accumulator, if needed.

\*

\*``````````````````````````````\*

\* \_\_W: (NATHAN RIGGS) \*

\* \*

\* INPUT: NONE \*

\* OUTPUT: .A HOLDS KEY VALUE \*

\* \*

\* DESTROYS: AXYNVBDIZCMS \*

\* ^ ^ ^^ \*

\* \*

\* CYCLES: 18+ \*

\* SIZE: 11 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\_\_W

:LP LDA ]KYBD ; CHECK IF KEY PRESSED

BPL :LP ; IF NOT, KEEP CHECKING

AND #$7F ; SET HI BIT

STA ]STROBE ; RESET KEYBOARD STROBE

RTS ; EXIT

|  |
| --- |
| **\_\_ERRH (sub)**  **Input**:  **.A** = address low byte  **.X** = address high byte  **Output**:  New error-handling  address is set.  **Destroys**: AYNZCM  **Cycles**: 51  **Size**: 31 bytes |

**LIB.REQUIRED >> \_\_ERRH**

The **\_\_ERRH** subroutine is used to define the address that is jumped to in the case of an Applesoft error. Note that there is some trickery here in order to get the machine to think it is in Applesoft mode prior to actually assigning the address.

For the most part, this is used in conjunction with file handling subroutines, but it is common enough to be included in the required library.

\*

\*``````````````````````````````\*

\* \_\_ERRH (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* .A = ADDRESS LOBYTE \*

\* .X = ADDRESS HIBYTE \*

\* \*

\* OUTPUT: \*

\* \*

\* SETS NEW ADDRESS FOR THE \*

\* APPLSOFT ERROR HANDLING \*

\* ROUTINE. \*

\* \*

\* DESTROYS: AXYNVBDIZCMS \*

\* ^^^^ ^^^ \*

\* \*

\* CYCLES: 51 \*

\* SIZE: 31 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\_\_ERRH

LDA #1 ; TRICK DOS INTO THINKING

STA $AAB6 ; IT'S IN APPLESOFT MODE

STA $75+1 ; APPLESOFT LINE NUMBER POINTER

STA $33 ; APLESOFT PROMPT CHARACTER

STA ADDR1 ; ADDRESS LOBYTE IN .A

STX ADDR1+1 ; ADDRESS HIBYTE IN .X

LDA #$FF ; TURN ON ERROR HANDLING

STA $D8 ; BYTE HERE

LDY #0 ; CLEAR OFFSET

LDA (ADDR1),Y ; LOAD ADDRESS LOBYTE

STA $9D5A ; SET AS ERROR HANDLING LO

INY ; INCREASE OFFSET

LDA (ADDR1),Y ; LOAD ADDRESS HIBYTE

STA $9D5B ; SET AS ERROR HANDLING HI

RTS ; EXIT SUBROUTINE

**COMMON LIBRARY**

The common library includes macros and subroutines that might be commonly used in assembly programs that are not specific to a cohesive classification (with, possibly, the exception of memory management). Additionally, like most disks for AppleIIAsm, this also includes a demo of all the macros (and thus subroutines, in a roundabout way) in the library. Unlike other demos, however, the common library also illustrates uses of the common library as well as those in the required library.

The common library includes the following:

* HOOKS.COMMON
* MAC.COMMON
* SUB.DELAYMS
* SUB.MEMFILL
* SUB.MEMMOVE
* SUB.MEMSWAP
* SUB.ZMLOAD
* SUB.ZMSAVE

**HOOKS.COMMON** includes various system hooks that are related to the use of common subroutines and macros. Note that this file, like other hooks files, may also include hooks that are commented out because they currently go unused by the library, but may be helpful for specific applications.

**MAC.COMMON** contains the macros used as part of the common library.

**SUB.DELAYMS** holds the DELAYMS subroutine, which delays the microprocessor for a given number of milliseconds. This is achieved by a precise counting of CPU cycles.

**SUB.MEMFILL** contains the MEMFILL subroutine, which fills a given range of memory with a given value.

**SUB.MEMMOVE** contains the MEMMOVE subroutine, which copies a given memory range to another address range.

**SUB.MEMSWAP** contains the MEMSWAP subroutine, which swaps the values in a given address range with those values in another address range.

**SUB.ZMLOAD** contains the ZMLOAD subroutine, which loads a previously saved set of values (from ZMSAVE) that populate the portions of the zero page that the main AppleIIAsm library uses.

**SUB.ZMSAVE** holds the ZMSAVE subroutine, which saves the values stored on the zero page that are immediately relevant to the main AppleIIAsm library.

The individual subroutines and macros will be explained prior to the listing of the file in which they are included.

**HOOKS.COMMON**

Since the Common library holds a lot of unrelated but useful subroutines and macros, the hooks file does not necessarily contain thematically related entries. Those here, however, are either highly common themselves, but aren’t part of any other library, or are used by the subroutines included in the library.

\*``````````````````````````````\*

\* HOOKS.COMMON \*

\* \*

\* HOOKS TO MONITOR AND TO THE \*

\* APPLESOFT ROUTINES THAT ARE \*

\* RELATED TO COMMON TASKS. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 30-JUN-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* LICENSE: APACHE 2.0 \*

\* OS: DOS 3.3 \*

\* \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PROMPT EQU $33 ; DOS PROMPT CHARACTER

COLDENT EQU $03D3 ; COLD ENTRY TO DOS

SRESET EQU $03F2 ; SOFT RESET

PRNTAX EQU $F941 ; PRINT HEX VALS OF A,X REGISTERS

BELL EQU $FBE4 ; RING MY BELL

IOSAVE EQU $FF4A ; SAVE CURRENT STATE OF REGISTERS

IOREST EQU $FF3F ; RESTORE OLD STATE OF REGISTERS

\*

**MAC.COMMON**

MAC.COMMON contains a variety of different macros that may not be thematically cohesive, but are common enough to merit inclusion into the library. Currently, this includes the following macros:

* MFILL
* MMOVE
* BEEP
* DELAY
* ZSAVE
* ZLOAD
* MSWAP

\*``````````````````````````````\*

\* MAC.COMMON \*

\* \*

\* THIS IS A MACRO LIBRARY FOR \*

\* COMMON.LIB, AND CAN BE USED \*

\* REGARDLESS OF WHETHER A \*

\* SPECIFIC FUNCTION IS \*

\* INCLUDED AS A PUT IN THE \*

\* MAIN SOURCE. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 30-JUN-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\* \*

\* SUBROUTINE FILES NEEDED \*

\* \*

\* SUB.MEMFILL \*

\* SUB.MEMMOVE \*

\* SUB.DELAYMS \*

\* SUB.ZMSAVE \*

\* SUB.ZMLOAD \*

\* SUB.MEMSWAP \*

\* \*

\* LIST OF MACROS \*

\* \*

\* MFILL FILL MEMORY BLOCK \*

\* MMOVE MOVE MEMORY BLOCK \*

\* BEEP RING MY BELL \*

\* DELAY DELAY IN MILLISECS \*

\* ZSAVE SAVE FREE ZERO PAGE \*

\* ZLOAD LOAD SAVE ZERO PAGE \*

\* MSWAP SWAP MEM RANGES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

|  |
| --- |
| **MFILL (macro)**  **Input**:  ]1 = memory address  ]2 = number of bytes  ]3 = fill value  **Output**:  Memory range filled with  Specified fill value  **Destroys**: AXYNZCM  **Cycles**: 39+  **Size**: 29 bytes |

**MAC.COMMON >> MFILL**

The **MFILL** macro is used to fill a specified range of memory with a given value. The parameters are first parsed into the appropriate zero-page locations, with the fill value passed via the accumulator. Afterwards, the **MEMFILL** subroutine is called.

\*

\*``````````````````````````````\*

\* MFILL \*

\* \*

\* FILL BLOCK OF MEMORY WITH \*

\* SPECIFIED VALUE. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = STARTING ADDRESS \*

\* ]2 = LENGTH IN BYTES \*

\* ]3 = FILL VALUE \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* MFILL $300;#256;#0 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

MFILL MAC

\_MLIT ]1;WPAR1

\_MLIT ]2;WPAR2

LDA ]3 ; FILL VALUE

STA BPAR1

JSR MEMFILL

<<<

|  |
| --- |
| **BEEP (macro)**  **Input**:  none  **Output**:  Beep from system speaker  **Destroys**: AXYNC  **Cycles**: 86+  **Size**: 10 bytes |

**MAC.COMMON >> BEEP**

The **BEEP** macro simply loops the standard **BELL** routine for the specified number of times.

\*

\*``````````````````````````````\*

\* BEEP \*

\* \*

\* RING THE STANDARD BELL. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = NUMBER OF RINGS \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* BEEP #10 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

BEEP MAC

LDX ]1

]LP1

JSR BELL

DEX

CPX #0

BNE ]LP1

<<<

|  |
| --- |
| **MMOVE (macro)**  **Input**:  ]1 = source address  ]2 = destination address  ]3 = byte length  **Output**:  none  **Destroys**: AXYNZCM  **Cycles**: 327+  **Size**: 6 bytes |

**MAC.COMMON >> MMOVE**

The **MMOVE** macro copies a source address range to a destination address range. The parameters are first parsed to be passed via the zero page, then the **MEMMOVE** subroutine is called.

\*

\*``````````````````````````````\*

\* MMOVE \*

\* \*

\* MOVE A BLOCK OF MEMORY FROM \*

\* A SOURCE TO DESTINATION. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = SOURCE ADDRESS \*

\* ]2 = DESTINATION ADDRESS \*

\* ]3 = NUMBER OF BYTES \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* MMOVE $6A00;$7B00;#1024 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

MMOVE MAC

\_MLIT ]1;WPAR1

\_MLIT ]2;WPAR2

\_MLIT ]3;WPAR3

JSR MEMMOVE

<<<

|  |
| --- |
| **DELAY (macro)**  **Input**:  ]1 = number of  milliseconds  **Output**:  None; delayed execution  **Destroys**: AXYNZCM  **Cycles**: 158+  **Size**: 5 bytes |

**MAC.COMMON >> DELAY**

The **DELAY** macro uses a precise number of cycles to delay the calling routine’s execution for a specified number of milliseconds. The maximum number of milliseconds, given that the parameter is a byte, is 255. Therefore, for delays greater than that, it is easiest to call the macro a consecutive number of times with a value of 250 (1/4 of a second).

\*

\*``````````````````````````````\*

\* DELAY \*

\* \*

\* DELAY FOR PASSED MILLISECS \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = NUM OF MILLISECONDS \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* DELAY #250 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

DELAY MAC

LDY ]1

JSR DELAYMS

<<<

|  |
| --- |
| **ZSAVE (macro)**  **Input**:  ]1 = destination address  **Output**:  None  **Destroys**: AXYNZCM  **Cycles**: 138+  **Size**: 3 bytes |

**MAC.COMMON >> ZSAVE**

The **ZSAVE** macro backs up the zero-page locations used by the library as a whole to another non-zero-page location specified in the parameter. The parameter is parsed into the **.A** and **.X** registers (low byte, high byte), then the **ZMSAVE** subroutine is called.

\*

\*``````````````````````````````\*

\* ZSAVE \*

\* \*

\* SAVE ZERO PAGE FREE AREAS \*

\* FOR LATER RESTORE. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = ADDRESS TO STORE AT \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* ZSAVE $300 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

ZSAVE MAC

\_AXLIT ]1

JSR ZMSAVE

<<<

|  |
| --- |
| **ZLOAD (macro)**  **Input**:  ]1 = source address  **Output**:  None  **Destroys**: AXYNZCM  **Cycles**: 123+  **Size**: 3 bytes |

**MAC.COMMON >> ZLOAD**

The **ZLOAD** macro restores the zero-page addresses used by the library that were previously backed up using **ZSAVE**. Parameters are parsed in **.A** and **.X** before calling **ZMLOAD**.

\*

\*``````````````````````````````\*

\* ZLOAD \*

\* \*

\* RESTORE PREVIOUSLY SAVED \*

\* FREE ZERO PAGE VALUES. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = ADDR TO LOAD FROM \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* ZLOAD $300 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

ZLOAD MAC

\_AXLIT ]1

JSR ZMLOAD

<<<

|  |
| --- |
| **MSWAP (macro)**  **Input**:  ]1 = first address  ]2 = second address  ]3 = length in bytes  **Output**:  none  **Destroys**: AXYNZCM  **Cycles**: 100+  **Size**: 50 bytes |

**MAC.COMMON >> MSWAP**

The **MSWAP** macro swaps the values held in a given address range with those in another. Parameters are parsed into the zero-page locations first, then the **MEMSWAP** subroutine is called.

\*

\*``````````````````````````````\*

\* MSWAP \*

\* \*

\* SWAPS THE VALUES STORED IN \*

\* ONE LOCATION WITH ANOTHER \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = FIRST ADDRESS \*

\* ]2 = SECOND ADDRESS \*

\* ]3 = LENGTH IN BYTES (BYTE) \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* MSWAP $300;$400;#$90 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

MSWAP MAC

\_MLIT ]2;WPAR2

\_MLIT ]1;WPAR1

LDA ]3

STA BPAR1

JSR MEMSWAP

<<<

|  |
| --- |
| **DELAYMS (sub)**  **Input**:  **.Y** = number of  milliseconds  **Output**:  none  **Destroys**: AXYNZCM  **Cycles**: 39+  **Size**: 29 bytes |

**SUB.DELAYMS >> DELAYMS**

The **DELAYMS** subroutine halts execution of the calling routine for a specified number of milliseconds by looping through a precise number of cycles. Of all subroutines, this is probably the least transferable to systems other than the Apple II, as processor speed, etc. determines timing.

\*

\*``````````````````````````````\*

\* DELAYMS (LEVENTHAL/SEVILLE) \*

\* \*

\* ADAPTED FROM LEVANTHAL AND \*

\* SEVILLE'S /6502 ASSEMBLY \*

\* LANGUAGE ROUTINES/. \*

\* \*

\* INPUT: \*

\* \*

\* .Y = NUMBER OF MILLISECS \*

\* \*

\* OUTPUT: \*

\* \*

\* DELAYS FOR X NUMBER OF \*

\* MILLISECONDS BY LOOPING \*

\* THROUGH A PRECISE NUMBER \*

\* OF CYCLES. \*

\* \*

\* DESTROYS: AXYNVBDIZCMS \*

\* ^^^^ ^^^ \*

\* \*

\* CYCLES: 39+ \*

\* SIZE: 29 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

DELAYMS

\*

]MSCNT EQU $0CA ; LOOP 202 TIMES THROUGH DELAY1

; SPECIFIC TO 1.23 MHZ

; SPEED OF APPLE II

:DELAY

CPY #0 ; IF Y = 0, THEN EXIT

BEQ :EXIT

NOP ; 2 CYCLES (MAKE OVERHEAD=25C)

\*

\*\* IF DELAY IS 1MS THEN GOTO LAST1

\*\* THIS LOGIC IS DESIGNED TO BE

\*\* 5 CYCLES THROUGH EITHER ATH

\*

CPY #1 ; 2 CYCLES

BNE :DELAYA ; 3C IF TAKEN, ELSE 2C

JMP :LAST1 ; 3C

\*

\*\* DELAY 1 MILLISENCOND TIMES (Y-1)

\*

:DELAYA

DEY ; 2C (PREDEC Y)

:DELAY0

LDX #]MSCNT ; 2C

:DELAY1

DEX ; 2C

BNE :DELAY1 ; 3C

NOP ; 2C

NOP ; 2C

DEY ; 2C

BNE :DELAY0 ; 3C

:LAST1

\*

\*\* DELAY THE LAST TIME 25 CYCLES

\*\* LESS TO TAKE THE CALL, RETURN,

\*\* AND ROUTINE OVERHEAD INTO

\*\* ACCOUNT.

\*

LDX #]MSCNT-3 ; 2C

:DELAY2

DEX ; 2C

BNE :DELAY2 ; 3C

:EXIT

RTS ; 6C

|  |
| --- |
| **MEMFILL (sub)**  **Input**:  **BPAR1** = fill value  **WPAR2** = length (2 bytes)  **WPAR3** = address (2 bytes)  **Output**:  none  **Destroys**: AXYNZM  **Cycles**: 117+  **Size**: 60 bytes |

**SUB.MEMFILL >> MEMFILL**

The **MEMFILL** subroutine fills a given range of memory addresses with a given value. Whole pages are filled first, with the remaining partial page filled afterward.

\*

\*``````````````````````````````\*

\* MEMFILL (LEVENTHAL/SAVILLE) \*

\* \*

\* ADAPTED FROM LEVANTHAL AND \*

\* SAVILLE'S /6502 ASSEMBLY \*

\* LANGUAGE ROUTINES/. \*

\* \*

\* INPUT: \*

\* \*

\* ]FILL IN BPAR1 \*

\* ]SIZE IN WPAR2 \*

\* ]ADDR IN WPAR3 \*

\* \*

\* OUTPUT: \*

\* \*

\* FILLS THE GIVEN MEM RANGE \*

\* \*

\* DESTROYS: AXYNVBDIZCMS \*

\* ^^^^ ^ ^ \*

\* \*

\* CYCLES: 117+ \*

\* SIZE: 60 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]FILL EQU BPAR1 ; FILL VALUE

]SIZE EQU WPAR2 ; RANGE LENGTH IN BYTES

]ADDR EQU WPAR1 ; RANGE STARTING ADDRESS

\*

MEMFILL

\*

\*\* FILL WHOLE PAGES FIRST

\*

LDA ]FILL ; GET VAL FOR FILL

LDX ]SIZE+1 ; X=# OF PAGES TO DO

BEQ :PARTPG ; BRANCH IF HIGHBYTE OF SZ = 0

LDY #0 ; RESET INDEX

:FULLPG

STA (]ADDR),Y ; FILL CURRENT BYTE

INY ; INCREMENT INDEX

BNE :FULLPG ; BRANCH IF NOT DONE W/ PAGE

INC ]ADDR+1 ; ADVANCE TO NEXT PAGE

DEX ; DECREMENT COUNTER

BNE :FULLPG ; BRANCH IF NOT DONE W/ PAGES

\*

\*\* DO THE REMAINING PARTIAL PAGE

\*\* REGISTER A STILL CONTAINS VALUE

\*

:PARTPG

LDX ]SIZE ; GET # OF BYTES IN FINAL PAGE

BEQ :EXIT ; BRANCH IF LOW BYTE = 0

LDY #0 ; RESET INDEX

:PARTLP

STA (]ADDR),Y ; STORE VAL

INY ; INCREMENT INDEX

DEX ; DECREMENT COUNTER

BNE :PARTLP ; BRANCH IF NOT DONE

:EXIT

RTS

|  |
| --- |
| **MEMMOVE (sub)**  **Input**:  **WPAR3** = length (2 bytes)  **WPAR1** = source address  (2 bytes)  **WPAR2** = destination  address (2 bytes)  **Output**:  none  **Destroys**: AXYM  **Cycles**: 267+  **Size**: 150 bytes |

**SUB.MEMMOVE >> MEMMOVE**

The **MEMMOVE** subroutine copies the values held at a source address range to a destination address range. If there is an overlap, the subroutine adjusts accordingly so that the copied data overwrites the source data, thus keeping its integrity. This is, in short, why the subroutine is called **MEMMOVE** instead of MEMCOPY.

\*

\*``````````````````````````````\*

\* MEMMOVE (LEVENTHAL/SEVILLE) \*

\* \*

\* ADAPTED FROM LEVANTHAL AND \*

\* SEVILLE'S /6502 ASSEMBLY \*

\* LANGUAGE ROUTINES/. \*

\* \*

\* INPUT: \*

\* \*

\* ]SIZE AT WPAR3 \*

\* ]ADDR1 AT WPAR1 \*

\* ]ADDR2 AT WPAR2 \*

\* \*

\* OUTPUT: \*

\* \*

\* BYTES FROM SOURCE ARE \*

\* COPIED IN ORDER TO THE \*

\* DESTINATION ADDRESS FOR \*

\* AS LONG AS LENGTH. \*

\* \*

\* DESTROY: .AXY,MEMORY \*

\* CYCLES: 267+ \*

\* SIZE: 150 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]SIZE EQU WPAR3 ; LENGTH TO COPY (BYTES)

]ADDR1 EQU WPAR1 ; SOURCE ADDRESS

]ADDR2 EQU WPAR2 ; DESTINATION ADDRESS

\*

MEMMOVE

\*

\*\* DETERMINE IF DEST AREA IS

\*\* ABOVE SRC AREA BUT OVERLAPS

\*\* IT. REMEMBER, OVERLAP CAN BE

\*\* MOD 64K. OVERLAP OCCURS IF

\*\* STARTING DEST ADDRESS MINUS

\*\* STARTING SRC ADDRESS (MOD

\*\* 64K) IS LESS THAN NUMBER

\*\* OF BYTES TO MOVE.

\*

LDA ]ADDR2 ; CALC DEST-SRC

SEC ; SET CARRY

SBC ]ADDR1 ; SUBTRACT SOURCE ADDRESS

TAX ; HOLD VAL IN .X

LDA ]ADDR2+1

SBC ]ADDR1+1 ; MOD 64K AUTOMATIC

; -- DISCARD CARRY

TAY ; HOLD HIBYTE IN .Y

TXA ; CMP LOBYTE WITH # TO MOVE

CMP ]SIZE

TYA

SBC ]SIZE+1 ; SUBTRACT SIZE+1 FROM HIBYTE

BCS :DOLEFT ; BRANCH IF NO OVERLAP

\*

\*\* DEST AREA IS ABOVE SRC AREA

\*\* BUT OVERLAPS IT.

\*\* MOVE FROM HIGHEST ADDR TO

\*\* AVOID DESTROYING DATA

\*

JSR :MVERHT

JMP :MREXIT

\*

\*\* NO PROB DOING ORDINARY MOVE

\*\* STARTING AT LOWEST ADDR

\*

:DOLEFT

JSR :MVELEFT

:EXIT

JMP :MREXIT

:MVELEFT

LDY #0 ; ZERO INDEX

LDX ]SIZE+1 ; X=# OF FULL PP TO MOVE

BEQ :MLPART ; IF X=0, DO PARTIAL PAGE

:MLPAGE

LDA (]ADDR1),Y ; LOAD BYTE FROM SOURCE

STA (]ADDR2),Y ; MOVE BYTE TO DESTINATION

INY ; NEXT BYTE

BNE :MLPAGE ; CONT UNTIL 256B MOVED

INC ]ADDR1+1 ; ADV TO NEXT SRC PAGE

INC ]ADDR2+1 ; ADV NEXT DEST PAGE

DEX ; DEC PAGE COUNT

BNE :MLPAGE ; CONT UNTIL ALL FULL

; PAGES ARE MOVED

:MLPART

LDX ]SIZE ; GET LENGTH OF LAST PAGE

BEQ :MLEXIT ; BR IF LENGTH OF LAST

; PAGE = 0

; REG Y IS 0

:MLLAST

LDA (]ADDR1),Y ; LOAD BYTE FROM SOURCE

STA (]ADDR2),Y ; MOVE BYTE TO DESTINATION

INY ; NEXT BYTE

DEX ; DEC COUNTER

BNE :MLLAST ; CONT UNTIL LAST P DONE

:MLEXIT

JMP :MREXIT

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

:MVERHT

\*

\*\* MOVE THE PARTIAL PAGE FIRST

\*

LDA ]SIZE+1 ; GET SIZE HIBYTE

CLC ; CLEAR CARRY

ADC ]ADDR1+1 ; ADD SOURCE ADDRESS HIBYTE

STA ]ADDR1+1 ; POINT TO LAST PAGE OF SRC

LDA ]SIZE+1 ; GET SIZE HIBYTE

CLC ; CLEAR CARRY

ADC ]ADDR2+1 ; ADD DESTINATION HIBYTE

STA ]ADDR2+1 ; POINT TO LAST P OF DEST

\*

\*\* MOVE THE LAST PARTIAL PAGE FIRST

\*

LDY ]SIZE ; GET LENGTH OF LAST PAGE

BEQ :MRPAGE ; IF Y=0 DO THE FULL PAGES

:MR0

DEY ; BACK UP Y TO NEXT BYTE

LDA (]ADDR1),Y ; LOAD CURRENT SOURCE BYTE

STA (]ADDR2),Y ; STORE IN CURRENT DESTINATION

CPY #0 ; BRANCH IF NOT DONE

BNE :MR0 ; WITH THE LAST PAGE

:MRPAGE

LDX ]SIZE+1 ; GET SIZE HIBYTE

BEQ :MREXIT ; BR IF HYBYTE = 0 (NO FULL P)

:MR1

DEC ]ADDR1+1 ; BACK UP TO PREV SRC PAGE

DEC ]ADDR2+1 ; AND DEST

:MR2

DEY ; BACK UP Y TO NEXT BYTE

LDA (]ADDR1),Y ; LOAD SOURCE CURRENT BYTE

STA (]ADDR2),Y ; STORE BYTE IN DESTINATION

CPY #0 ; IF NOT DONE WITH PAGE

BNE :MR2 ; THEN BRANCH OUT

DEX ; DECREASE BYTE COUNTER

BNE :MR1 ; BR IF NOT ALL PAGES MOVED

:MREXIT

RTS

|  |
| --- |
| **MEMSWAP (sub)**  **Input**:  **BPAR1** = length  **WPAR1** = first address  (2 bytes)  **WPAR2** = second address  (2 bytes)  **Output**:  none  **Destroys**: AXYNZCM  **Cycles**: 100+  **Size**: 43 bytes |

**SUB.MEMSWAP >> MEMSWAP**

The **MEMSWAP** routine swaps the values stored in one address range with another. Note that this currently has no protections against an overlap in range.

\*

\*``````````````````````````````\*

\* MEMSWAP (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* ]SIZE = BPAR1 \*

\* ]ADDR1 = WPAR1 \*

\* ]ADDR2 = WPAR2 \*

\* \*

\* OUTPUT: \*

\* \*

\* SWAPS THE VALUES IN THE \*

\* MEMORY LOCATIONS GIVEN \*

\* FOR THE SPECIFIED LENGTH. \*

\* \*

\* DESTROYS: AXYNVBDIZCMS \*

\* ^^^^ ^^^ \*

\* \*

\* CYCLES: 100+ \*

\* SIZE: 43 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]SIZE EQU BPAR1 ; SIZE OF RANGE TO SWAP

]ADDR1 EQU WPAR1 ; SOURCE ADDRESS 1

]ADDR2 EQU WPAR2 ; SOURCE ADDRESS 2

\*

MEMSWAP

LDY #255 ; RESET BYTE INDEX

:LP

INY ; INCREASE BYTE INDEX

LDA (]ADDR1),Y ; LOAD BYTE FROM FIRST ADDRESS

TAX ; TRANSFER TO .X

LDA (]ADDR2),Y ; LOAD BYTE FROM SECOND ADDRESS

STA (]ADDR1),Y ; STORE IN FIRST ADDRESS

TXA ; TRANSFER FIRST BYTE VAL TO .A

STA (]ADDR2),Y ; NOW STORE THAT IN SECOND ADDRESS

CPY ]SIZE ; IF BYTE INDEX < LENGTH,

BNE :LP ; CONTINUE LOOPING

RTS ; OTHERWISE, EXIT

|  |
| --- |
| **ZMLOAD (sub)**  **Input**:  **.A** = low byte of address  **.X** = high byte of address  **Output**:  none  **Destroys**: AXYNZCM  **Cycles**: 123+  **Size**: 71 bytes |

**SUB.ZMLOAD >> ZMLOAD**

The **ZMLOAD** subroutine loads the values stored by **ZMSAVE** back into the zero page at the locations used by the library. Note that these locations go unused by the monitor, DOS or Applesoft; those locations are unaffected.

The memory addresses affected are:

19 1E E3 EB EC ED EE

EF FA FB FC FD FE FF

\*

\*``````````````````````````````\*

\* ZMLOAD (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* .A = LOBYTE OF SRC ADDR \*

\* .X = HIBYTE OF SRC ADDR \*

\* \*

\* OUTPUT: \*

\* \*

\* RESTORES PREVIOUSLY SAVED \*

\* ZERO PAGE VALUES FROM \*

\* HIGHER MEMORY LOCATION. \*

\* \*

\* DESTROYS: AXYNVBDIZCMS \*

\* ^^^^ ^^^ \*

\* \*

\* CYCLES: 123+ \*

\* SIZE: 71 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]ADR1 EQU VARTAB ; 2 BYTES

]ADR2 EQU VARTAB+2 ; 2 BYTES

]Z HEX 191EE3EBECED

HEX EEEFFAFBFCFDFEFF

HEX 00

\*

ZMLOAD

\*

STA ADDR1 ; BACKUP SOURCE ADDR LOBYTE

STX ADDR1+1 ; BACKUP HIBYTE

LDY #255 ; RESET INDEX

LDA (ADDR1),Y

STA ]ADR1 ; BACKUP $06

INY

LDA (ADDR1),Y ; BACKUP $07

STA ]ADR1+1

INY ; INCREASE INDEX

LDA (ADDR1),Y ; BACKUP $07

STA ]ADR2

INY

LDA (ADDR1),Y ; BACKUP $08

STA ]ADR2+1

:LP

INY

LDA ]Z,Y

BEQ :EXIT ; IF NULL, EXIT

STA ADDR2

LDA #0

STA ADDR2+1

LDA (ADDR1),Y

STA (ADDR2),Y

JMP :LP

:EXIT

LDY #0

LDA (ADDR1),Y+3 ; NOW RESTORE FIRST

STA $09 ; FOUR BYTES

LDA (ADDR1),Y+2

STA $08

LDA (ADDR1),Y+1

TAX

LDA (ADDR1),Y

TAY

TXA

STA ADDR1+1

TYA

STA ADDR1

RTS

|  |
| --- |
| **ZMSAVE (sub)**  **Input**:  **.A** = address low byte  **.X** = address high byte  **Output**:  none  **Destroys**: AXYNZCM  **Cycles**: 138+  **Size**: 84 bytes |

**SUB.ZMSAVE >> ZMSAVE**

The **ZMSAVE** subroutine backs up select addresses on the zero page to be later restored via the **ZMLOAD** subroutine. The addresses used by the library are unused by the monitor, Applesoft or DOS. They are as follows:

19 1E E3 EB EC ED EE

EF FA FB FC FD FE FF

\*

\*``````````````````````````````\*

\* ZMSAVE :: SAVE 0-PAGE FREE \*

\* \*

\* INPUT: \*

\* \*

\* .A = DESTINATION LOBYTE \*

\* .Y = DESTINATION HIBYTE \*

\* \*

\* OUTPUT: \*

\* \*

\* THE FREE AREAS OF THE \*

\* ZERO PAGE ARE COPIED TO \*

\* THE DESTINATION ADDRESS. \*

\* \*

\* DESTROYS: AXYNVBDIZCMS \*

\* ^^^^ ^^^ \*

\* \*

\* CYCLES: 138+ \*

\* SIZE: 84 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]ADR1 EQU VARTAB ; 2 BYTES--DEST ADDRESS

]ADR2 EQU VARTAB+2 ; 2 BYTES--SOURCE ADDRESS

]Z HEX 191EE3BECEDEEF ; ZERO PAGE LOCATIONS

HEX FAFBFCFDFEFF ; TO BE BACKED UP

HEX 00

ZMSAVE

\*

STA ]ADR1 ; BACKUP DESTINATION ADDRESS LO

STX ]ADR1+1 ; BACKUP HIBYTE

LDA ADDR2 ; BACKUP CONTENTS OF ADDR2 LOBYTE

STA ]ADR2

LDA ADDR2+1 ; BACKUP HIBYTE

STA ]ADR2+1

LDA ]ADR1 ; PUT DESTINATION ADDRESS

STA ADDR2 ; INTO ZERO-PAGE ADDR2

LDA ]ADR1 ; FOR INDIRECT ACCESS

STA ADDR2+1

LDY #0 ; CLEAR INDEX

LDA ADDR1 ; LOAD ADDR1 LOBYTE

STA (ADDR2),Y ; STORE IT IN DESTINATION

INY ; INCREASE INDEX

LDA ADDR1+1 ; GET ADDR1 HIBYTE

STA (ADDR2),Y ; STORE IN DESTINATION

INY ; INCREMENT INDEX

LDA ]ADR2 ; LOAD OLD ADDR2 LOBYTE

STA (ADDR2),Y ; COPY TO DESTINATION

INY ; INCREMENT INDEX

LDA ]ADR2+1 ; LOAD OLD ADDR2 HIBYTE

STA (ADDR2),Y ; STORE IN DESTINATION

LDX #255 ; RESET INDEX2 COUNTER

STY ]SIZE ; STORE INDEX1 IN ]SIZE

LDY #0 ; RESET Y-INDEX

:LP

INC ]SIZE ; INCREMENT SOURCE INDEX

INX ; INCREMENT TABLE INDEX

LDA ]Z,X ; GET NEXT BYTE FROM TABLE

BEQ :EXIT ; IF ZERO, QUIT

STA ADDR1 ; STORE BYTE FROM TABLE AS LOBYTE

LDA #0 ; CLEAR THE HIBYTE

STA ADDR1+1

LDA (ADDR1),Y ; INDIRECTLY LOAD ZERO-PAGE CONTENT

LDY ]SIZE ; PULL INDEX BACK INTO Y

STA (ADDR2),Y ; STORE BYTE TO DESTINATION

LDY #0 ; RESET Y

JMP :LP ; REPEAT UNTIL FINISHED

:EXIT

RTS

**DEMO.COMMON**

The **DEMO.COMMON** file contains quick demonstrations of the macros found in **MAC.REQUIRED** and **MAC.COMMON**. These are not meant to be exhaustive demos, but rather serve to quickly show how (and sometimes why) the macros work. For more complicated usage, the integrated demos should be consulted.

Note that this DEMO routine, along with all of the DEMO routines on each library disk, is impractical: using the **\_PRN** macro dedicates a byte of memory to each and every character in a string, creating unnecessarily large executables. This method of text display is discouraged in other programs; reading strings from a file and using a small piece of memory is a much more memory-efficient solution. **\_PRN** is used here only for convenience and ease of reading.

\*

\*``````````````````````````````\*

\* DEMO.COMMON \*

\* \*

\* A DEMO OF THE MACROS AND \*

\* SUBROUTINES IN THE COMMON \*

\* APPLEIIASM LIBRARY. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 30-JUN-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\*\* ASSEMBLER DIRECTIVES

\*

CYC AVE

EXP ONLY

TR ON

DSK DEMO.COMMON

OBJ $BFE0

ORG $6000

\*

\*``````````````````````````````\*

\* TOP INCLUDES (HOOKS,MACROS) \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PUT MIN.HEAD.REQUIRED

USE MIN.MAC.REQUIRED

USE MIN.HOOKS.COMMON

USE MIN.MAC.COMMON

]HOME EQU $FC58

\*

\*``````````````````````````````\*

\* PROGRAM MAIN BODY \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

JSR ]HOME

\_PRN "COMMON SUBROUTINE LIBRARY",8D

\_PRN "=========================",8D8D

\_PRN "THIS LIBRARY CONTAINES MACROS AND",8D

\_PRN "SUBROUTINES THAT MIGHT BE COMMONLY",8D

\_PRN "USED BY A BROAD RANGE OF PROGRAMS.",8D8D

\_PRN "THIS DEMO WILL ALSO ILLUSTRATE THE",8D

\_PRN "USE OF SOME MACROS IN THE REQUIRED",8D

\_PRN "LIBRARY FOR THE FIRST TIME. WE WILL",8D

\_PRN "MAKE IT CLEAR WHEN WE SWITCH LIBRARIES,",8D

\_PRN "BUT FOR QUICK REFERENCE THE MACROS",8D

\_PRN "IN EACH LIBRARY ARE:",8D8D

\_WAIT

\_PRN "REQUIRED MACROS: \_ISLIT,\_AXLIT,",8D

\_PRN "\_ISSTR,\_AXSTR,GRET,SPAR,DUMP,\_PRN,",8D

\_PRN "\_WAIT,ERRH,CLRHI",8D8D

\_WAIT

\_PRN "COMMON: MFILL,MMOVE,MSWAP,BEEP,DELAY,",8D

\_PRN "ZSAVE,ZLOAD",8D8D

\_WAIT

\_PRN "LET'S START WITH THE MOST USED REQUIRED MACROS."

\_WAIT

JSR ]HOME

\_PRN "REQUIRED LIBRARY: MOST USED",8D

\_PRN "===========================",8D8D

\_PRN "BY 'MOST USED' HERE, WE MEAN MOST",8D

\_PRN "USED BY THESE SHORT DEMOS. IN",8D

\_PRN "REALITY, OTHER MACROS ARE PROBABLY",8D

\_PRN "UTILIZED MUCH MORE OFTEN, BUT IT",8D

\_PRN "HAPPENS BEHIND THE SCENES.",8D8D

\_WAIT

\_PRN "THE TWO MOST APPARENT MACROS ",8D

\_PRN "SHOULD BE FAMILIAR IF YOU HAVE",8D

\_PRN "ALREADY EXPLORED THE STDIO LIBRARY:",8D

\_PRN "\_PRN AND \_WAIT. THESE ARE NEAR",8D

\_PRN "CARBON COPIES OF THEIR EQUIVALENT",8D

\_PRN "ROUTINES IN STDIO, AND ARE HERE FOR",8D

\_PRN "THE MOSTLY RARE CASES WHEN SOME",8D

\_PRN "MINOR INPUT AND OUTPUT ARE NECESSARY",8D

\_PRN "BUT WITHOUT THE NEED FOR USING THE",8D

\_PRN "STDIO LIBRARY. SINCE THESE EXIST",8D

\_PRN "AS PART OF THE ERQUIRED LIBRARY, YOU",8D

\_PRN "CAN USE THESE IN PLACE OF STDIO IF",8D

\_PRN "YOUR PROGRAM REQUIRES NO MORE THAN THIS",8D

\_PRN "BASIC FUNCTIONALITY."

\_WAIT

JSR ]HOME

\_PRN "THE \_PRN MACRO PRINTS A STRING THAT",8D

\_PRN "IS EITHER GIVEN AS A PARAMETER OR",8D

\_PRN "RESIDES AT A GIVEN ADDRESS AND IS",8D

\_PRN "TERMINATED BY A NULL BYTE ($00). THUS:",8D8D

\_WAIT

\_PRN " \_PRN 'HELLO, WORLD!'",8D

\_PRN " \_PRN #STRING1",8D

\_PRN " \_PRN INDIRECT",8D8D

\_WAIT

\_PRN "ARE ALL VALID USES OF \_PRN. THE FIRST",8D

\_PRN "PRINTS THE GIVEN STRING, THE SECOND",8D

\_PRN "PRINTS NULL-TERMINATED STRING AT THE",8D

\_PRN "STRING1 ADDRESS, AND THE THIRD PRINTS",8D

\_PRN "A NULL-TERMINATED STRING AT THE",8D

\_PRN "ADDRESS POINTED TO IN THE ADDRESS HELD",8D

\_PRN "IN INDIRECT.",8D8D

\_WAIT

\_PRN "THE WAIT MACRO DOES EXACTLY WHAT ",8D

\_PRN "IT SAYS: IT WAITS FOR A KEYPRESS. THE",8D

\_PRN "KEY PRESSED IS PASSED BACK IN .A"

\_WAIT

JSR ]HOME

\_PRN "MEMORY DUMPS",8D

\_PRN "============",8D8D

\_PRN "THE OTHER MACRO MOST USED IN",8D

\_PRN "THESE DEMOS IS THE DUMP MACRO, WHICH",8D

\_PRN "OUTPUTS THE HEX VALUES AT A GIVEN",8D

\_PRN "ADDRESS RANGE. THEREFORE:",8D8D

\_WAIT

\_PRN " LDA #$33",8D

\_PRN " STA $300",8D

\_PRN " STA $301",8D

\_PRN " STA $302",8D

\_PRN " DUMP #$300;#10",8D8D

\_PRN "WILL OUTPUT",8D8D

\_WAIT

LDA #$33

STA $300

STA $301

STA $302

DUMP #$300;#10

\_WAIT

JSR ]HOME

\_PRN "PARAMETERS AND RETURNS",8D

\_PRN "======================",8D8D

\_PRN "NEARLY EVERY SUBROUTINE IN THIS",8D

\_PRN "SET OF LIBRARIES UTILIZES THE",8D

\_PRN "SAME MEMORY LOCATION FOR RETURNING",8D

\_PRN "RESULTS, SAVE FOR THOSE THAT RETURN",8D

\_PRN "NOTHING. THIS LOCATION IS REFERENCED",8D

\_PRN "IN THE CODE AS THE 'RETURN' HOOK.",8D8D

\_WAIT

\_PRN "THE GRET MACRO CAN BE USED TO COPY",8D

\_PRN "THE RETURNED DATA TO A MORE PERMANENT",8D

\_PRN "LOCATION FOR RETRIEVAL LATER ON. SO:",8D8D

\_PRN " GRET #$300",8D8D

\_WAIT

\_PRN "COPIES THE DATA FROM RETURN INTO THE",8D

\_PRN "SPECIFIED LOCATION ($300). NOTE THAT",8D

\_PRN "THE LENGTH OF THE RETURN VALUE IS",8D

\_PRN "KNOWN VIA THE 'RETLEN' HOOK, WHICH",8D

\_PRN "POINTS TO A LENGTH BYTE PRECEDING RETURN"

\_WAIT

JSR ]HOME

\_PRN "INTERNAL MACROS",8D

\_PRN "===============",8D8D

\_PRN "THE MACROS \_ISLIT, \_AXLIT,",8D

\_PRN "\_ISSTR AND \_AXSTR ARE ALL MACROS USED",8D

\_PRN "BY OTHER MACROS TO DETERMINE WHAT",8D

\_PRN "KIND OF DATA IS BEING MASSED, THEN",8D

\_PRN "TRANSLATING THAT TO A MACHINE-FRIENDLY",8D

\_PRN "FORM. THESE MACROS ARE RESPONSIBLE",8D

\_PRN "FOR A MACRO'S ABILITY TO ACCEPT",8D

\_PRN "DIRECT OR INDIRECT ADDRESSING, AS",8D

\_PRN "WELL AS LITERAL STRINGS.",8D8D

\_WAIT

\_PRN "THIS CAN BE EASILY SEEN IN",8D

\_PRN "MANY MACROS THAT ACCEPT EITHER ",8D

\_PRN "STRINGS OR ADDRESSES. FIRST, THE",8D

\_PRN "PARAMETER IS PASSED TO EITHER THE",8D

\_PRN "\_ISSTR MACRO OR THE \_AXSTR MACRO;",8D

\_PRN "THESE ARE FUNCTIONALLY EQUIVALENT AND",8D

\_PRN "TEST WHETHER OR NOT THE PARAMETER",8D

\_PRN "IS A STRING OR ADDRESS, BUT DIFFER IN",8D

\_PRN "HOW THAT DATA IS THEN PASSED TO THE",8D

\_PRN "APPROPRIATE SUBROUTINE.",8D

\_WAIT

JSR ]HOME

\_PRN "\_ISSTR PASSES DATA VIA THE STACK,",8D

\_PRN "WHEREAS\_AXSTR PASSES VIA .A AND .X,"8D

\_PRN "WHICH HOLD THE LO AND HI BYTES OF THE",8D

\_PRN "ADDRESS OF THE STRING, RESPECTIVELY.",8D

\_PRN "WHICH MACRO TO USE IS PRIMARILY",8D

\_PRN "DETERMINED BY THE SUBROUTINE BEING",8D

\_PRN "CALLED, AS THEY EITHER USE ONE OR",8D

\_PRN "THE OTHER METHODS OF PASSING",8D

\_PRN "PARAMETERS. A RULE OF THUMB IS THAT",8D

\_PRN "IF THERE ARE FEWER THAN 4 BYTES",8D

\_PRN "TO BE PASSED, THEN PASSING IS DONE",8D

\_PRN "VIA REGISTERS TO SPARE A FEW CYCLES;",8D

\_PRN "OTHERWISE, THE STACK IS USED.",8D8D

\_WAIT

\_PRN "\_ISLIT AND \_AXLIT USE THE SAME LOGIC",8D

\_PRN "FOR THE PASSING OF PARAMETERS, BUT ARE",8D

\_PRN "USED TO DETERMINE WHETHER THE PARAMETER",8D

\_PRN "BEING PASSED IS A LITERAL VALUE OR A",8D

\_PRN "MEMORY LOCATION. IF THE PARAMETER IS",8D

\_PRN "A LITERAL, THEN THE MACRO SENDS IT",8D

\_PRN "AS A 2-BYTE ADDRESS THAT INDICATES",8D

\_PRN "THE DATA IS LOCATED AT THAT ADDRESS.",8D

\_PRN "IF, HOWEVER, A NON-LITERAL ADDRESS IS",8D

\_PRN "PASSED, THE LIBRARY INTERPRETS THIS AS",8D

\_PRN "AN INDIRECT REFERENCE, WHERE THE ",8D

\_PRN "ADDRESS PASSED IS A POINTER TO THE",8D

\_PRN "ACTUAL ADDRESS OF THE DATA."

\_WAIT

JSR ]HOME

\_PRN "THE REQUIRED LEFTOVERS",8D

\_PRN "======================",8D8D

\_PRN "OTHER MACROS IN THE REQUIRED LIBRARY",8D

\_PRN "ARE RARELY USED OUTSIDE OF THE",8D

\_PRN "LIBRARY ITSELF IN THE DEMOS, IF AT ALL.",8D

\_PRN "THIS INCLUDES THE ERRH AND CLRHI MACROS.",8D8D

\_WAIT

\_PRN "CLRHI TAKES ONE BYTE AND CLEARS ITS",8D

\_PRN "HIGH NIBBLE, AND IS USEFUL FOR THE",8D

\_PRN "IMPLEMENTATION OF LOOKUP TABLES, AMONG ",8D

\_PRN "OTHER USES. THE ERRH MACRO PASSES THE",8D

\_PRN "PROVIDED ADDRESS TO APPLESOFT AS A HOOK",8D

\_PRN "FOR ERROR-HANDLING, AND CAN BE THOUGHT",8D

\_PRN "OF AS A 'ONERR GOTO ###' COMMAND FOR",8D

\_PRN "ASSEMBLY. NOTE THAT THIS DOESN'T CATCH",8D

\_PRN "JUST ANY ERRORS IN YOUR CODE--YOU ",8D

\_PRN "STILL HAVE TO FIGURE THAT OUT YOURSELF.",8D

\_PRN "THE ERROR-HANDLING IS SPECIFIC TO ",8D

\_PRN "INTERFACING WITH APPLESOFT."

\_WAIT

\*

JSR ]HOME

\_PRN "COMMON MACROS, FINALLY!",8D

\_PRN "=======================",8D8D

\_PRN "WE CAN NOW MOVE ON TO THE",8D

\_PRN "MACROS IN THE COMMON LIBRARY. MOST",8D

\_PRN "OF THESE CURRENTLY FOCUS ON MEMORY",8D

\_PRN "MANAGEMENT, AND WE WILL ADDRESS THOSE",8D

\_PRN "FIRST: MFILL, MMOVE, MSWAP, ZLOAD AND",8D

\_PRN "ZSAVE."

\_WAIT

JSR ]HOME

\_PRN "MEMORY MANAGEMENT",8D

\_PRN "=================",8D8D

\_PRN "MFILL FILLS A RANGE OF MEMORY STARTING",8D

\_PRN "AT THE GIVEN ADDRESS WITH THE GIVEN",8D

\_PRN "FILL VALUE. THUS:",8D8D

\_PRN " MFILL #$300;#10;#0",8D8D

\_PRN "FILLS $300-$309 WITH ZEROS. WE CAN",8D

\_PRN "VERIFY THIS WITH A DUMP:",8D

\_WAIT

MFILL #$300;#10;#0

DUMP #$300;#10

\_WAIT

JSR ]HOME

\_PRN "MMOVE SUITABLY MOVES (OR COPIES) A",8D

\_PRN "BLOCK OF MEMORY FROM ONE ADDRESS",8D

\_PRN "RANGE TO ANOTHER. SO:",8D8D

\_WAIT

\_PRN " MMOVE #$300;#$320;#10",8D

\_PRN " DUMP #$320;#10",8D8D

\_PRN "WILL COPY THE TEN ZEROS AT $300",8D

\_PRN "TO $320-$329, THEN DUMP THE RESULTS:",8D

MMOVE #$300;#$320;#10

DUMP #$320;#10

\_WAIT

JSR ]HOME

\_PRN "SIMILARLY, MSWAP SWAPS THE DATA IN ",8D

\_PRN "THE GIVEN MEMORY RANGES. SO, TO SWAP",8D

\_PRN "$300-309 WITH $310-$319, WE'D WRITE:",8D8D

\_PRN " MSWAP #$300;#$310;#10",8D8D

\_PRN "NOW WHEN WE DUMP $300 AGAIN, IT HAS:",8D

\_WAIT

MSWAP #$300;#$310;#10

DUMP #$300;#10

DUMP #$310;#10

\_WAIT

JSR ]HOME

\_PRN "ZERO-PAGE BACKUPS",8D

\_PRN "=================",8D8D

\_PRN "THIS LIBRARY USES NEARLY EVERY",8D

\_PRN "PART OF THE ZERO PAGE THAT IS",8D

\_PRN "UNUSED BY DOS, APPLESOFT OR THE ",8D

\_PRN "MONITOR. AT TIMES, YOU MAY WANT TO",8D

\_PRN "USE THOSE LOCATIONS YOURSELF WITHOUT",8D

\_PRN "THE RISK OF THE LIBRARY WRITING OVER",8D

\_PRN "YOUR DATA. THAT'S WHERE ZSAVE AND",8D

\_PRN "ZLOAD COME INTO PLAY.",8D8D

\_WAIT

\_PRN "ZSAVE BACKUPS THE ZERO-PAGE MEMORY THAT",8D

\_PRN "IS UNUSED BY DOS/APPLESOFT/MONITOR,",8D

\_PRN "COPYING IT TO THE SPECIFIED LOCATION. ",8D

\_PRN "THEN, ZLOAD IS USED TO RESTORE THOSE",8D

\_PRN "'UNUSED' BYTES TO YOUR OWN DATA AFTER A",8D

\_PRN "LIBRARY ROUTINE IS CALLED.",8D

\_WAIT

JSR ]HOME

\_PRN "SO, WE CAN SAVE THE ZERO-PAGE AT $300",8D

\_PRN "WITH THE FOLLOWING:",8D8D

\_PRN " ZSAVE #$300",8D8D

\_PRN "AND THEN CHANGE THE ZERO PAGE SLIGHTLY:",8D8D

\_PRN " LDA #$99",8D

\_PRN " STA $06",8D

\_PRN " STA $07",8D

\_PRN " STA $08",8D

\_PRN " STA $09",8D

\_PRN " STA $19",8D8D

ZSAVE #$300

LDA #$99

STA $06

STA $07

STA $08

STA $09

STA $19

\_WAIT

JSR ]HOME

\_PRN "NOW WE'LL DUMP THE ZERO PAGE TO",8D

\_PRN "SHOW THE CHANGES:",8D

DUMP #$0;#10

DUMP #10;#10

DUMP #20;#10

\_PRN " ",8D8D

\_PRN "NOTE THAT ALREADY, THE $10 HAS BEEN",8D

\_PRN "CHANGED BY THE LIBRARY! THUS THE",8D

\_PRN "NEED FOR A BACKUP. SO, IN ORDER",8D

\_PRN "TO RECOVER OUR ZERO PAGE, USE ZLOAD:",8D8D

\_PRN " ZLOAD #$300",8D8D

\_WAIT

\_PRN "WHICH WILL THEN LEAVE US WITH:",8D

\_WAIT

ZLOAD #$300

DUMP #0;#10

DUMP #10;#10

DUMP #20;#10

\_WAIT

JSR ]HOME

\_PRN "BEEP AND DELAY",8D

\_PRN "==============",8D8D

\_PRN "LASTLY, WE HAVE THE BEEP MACRO",8D

\_PRN "AND THE DELAY MACRO FROM THE",8D

\_PRN "COMMON LIBRARY. THESE ARE PRETTY",8D

\_PRN "SELF-EXPLANATORY: 'BEEP' SENDS THE",8D

\_PRN "STANDARD TONE TO THE SPEAKER FOR ",8D

\_PRN "SPECIFIED NUMBER OF CYCLES, WHILE ",8D

\_PRN "DELAY SUSPENDS EXECUTION FOR THE",8D

\_PRN "SPECIFIED NUMBER OF MILLISECONDS. ",8D

\_PRN "SO: ",8D8D

\_PRN " BEEP #10",8D

\_PRN " DELAY #255",8D

\_PRN " BEEP #20",8D

\_PRN " DELAY #255",8D

\_PRN " BEEP #30",8D8D

\_PRN "RESULTS IN:",8D8D

\_WAIT

BEEP #10

DELAY #255

BEEP #20

DELAY #255

BEEP #30

\_WAIT

JSR ]HOME

\_PRN "WE'RE DONE HERE!",8D8D8D

\*

JMP REENTRY

\*

\*``````````````````````````````\*

\* BOTTOM INCLUDES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\*\* BOTTOM INCLUDES

\*

PUT MIN.LIB.REQUIRED

\*

\*\* INDIVIDUAL SUBROUTINE INCLUDES

\*

\* COMMON LIBRARY SUBROUTINES

\*

PUT MIN.SUB.DELAYMS

PUT MIN.SUB.MEMFILL

PUT MIN.SUB.MEMMOVE

PUT MIN.SUB.MEMSWAP

PUT MIN.SUB.ZMSAVE

PUT MIN.SUB.ZMLOAD

**Disk 2: STDIO**

The second disk in the library is dedicated to standard input and output macros and subroutines. This primarily consists of keyboard and paddle input and text screen output. More specialized input and output routines are handled in other packages. It contains the following library components:

* HOOKS.STDIO
* MAC.STDIO
* DEMO.STDIO
* SUB.DPRINT
* SUB.PRNSTR
* SUB.SINPUT
* SUB.TBLINE
* SUB.TCIRCLE
* SUB.THLINE
* SUB.TRECTF
* SUB.TVLINE
* SUB.TXTPUT
* SUB.XPRINT

**HOOKS.STDIO** contains the various hooks that are either used by the subroutines and macros on the disk or are especially relevant to standard input and output.

**MAC.STDIO** contains all of the macros dedicated to standard input and output procedures.

Each of the files with the **SUB** prefix contains the subroutine indicated in the rest of the filename.

**HOOKS.STDIO**

The hooks in this file all relate to basic input and output for text and the paddles.

\*

\*``````````````````````````````\*

\* HOOKS.STDIO \*

\* \*

\* THESE ARE HOOKS THAT ARE \*

\* USED BY THE STDIO LIBRARY. \*

\* COMMENTED HOOKS ARE RELATED \*

\* BUT CURRENTLY UNUSED. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 07-JUL-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\* OUTPUT HOOKS

\*

COUT1 EQU $FDF0 ; FASTER SCREEN OUTPUT

COUT EQU $FDED ; MONITOR STD OUTPUT

HOME EQU $FC58 ; CLEAR SCREEN, HOME CURSOR

VTAB EQU $FC22 ; MONITOR CURSOR POS ROUTINE

CURSH EQU $24 ; HPOS OF COUT CURSOR

CURSV EQU $25 ; VPOS OF COUT CURSOR

KEYBUFF EQU $0200 ; KEYBUFFER START

GSTROBE EQU $C040 ; GAME CONNECTOR STROBE

GBCALC EQU $F847 ; SCREEN CALCULATION

GBPSH EQU $26

\*

\* INPUT HOOKS

\*

KYBD EQU $C000 ; LDA SINGLE KEYPRESS

STROBE EQU $C010 ; CLEAR KYBD BUFFER

GETLN EQU $FD6F ; MONITOR GET LINE OF KB INPUT

GETKEY EQU $FD0C ; MONITOR GET SINGLE KEY INPUT

\*

\* PADDLE HOOKS

\*

PREAD EQU $FB1E ; READ STATE OF PADDLE

PB0 EQU $C061 ; PADDLE BUTTON 0

PB1 EQU $C062

PB2 EQU $C063

PB3 EQU $C060

\*

\*\* UNUSED BY LIBRARY

\*

\*WNDLEFT EQU $20 ; SCROLL WINDOW LEFT

\*WNDWIDTH EQU $21 ; SCROLL WINDOW WIDTH

\*WNDTOP EQU $22 ; SCROLL WINDOW TOP

\*WNDBOT EQU $23 ; SCROLL WINDOW BOTTOM

\*TEXTP1 EQU $0400 ; START OF TEXT PAGE 1

\*TEXTP2 EQU $0800 ; START OF TEXT PAGE 2

\*PAGE1 EQU $C054 ; SOFT SWITCH USE PAGE 1

\*PAGE2 EQU $C055 ; SOFT SWITCH USE PAGE 2

\*S80COL EQU $C01F ; READ ONLY; CHECK IF 80C

\*TXTSET EQU $C051 ; TEXT ON SOFT SWITCH

\*SETWND EQU $FB4B ; SET NORMAL WINDOW MODE

\*CURADV EQU $FBF4 ; ADVANCE CURSOR RIGHT

\*CURBS EQU $FC10 ; CURSOR LEFT

\*CURUP EQU $FC1A ; CURSOR UP

\*CR EQU $FC62 ; CARRIAGE RETURN TO SCREEN

\*LF EQU $FC66 ; LINE FEED ONLY TO SCREEN

\*CLEOL EQU $FC9C ; CLEAR TEXT TO END OF LINE

\*OPAPP EQU $C061

\*CLAPP EQU $C062

**MAC.STDIO**

MAC.STDIO contains all of the macros related to standard input and output. It contains the following macros:

* COL40
* COL80
* CURB
* CURD
* CURF
* CURU
* DIE80
* GKEY
* INP
* MTXT0
* MTXT1
* PBX
* PDL
* PRN
* RCPOS
* SPRN
* SCPOS
* SETCX
* SETCY
* TCIRC
* THLIN
* TLINE
* TPUT
* TRECF
* TVLIN
* WAIT

\*

\*``````````````````````````````\*

\* MAC.STDIO \*

\* \*

\* THIS IS A MACRO LIBRARY FOR \*

\* STANDARD INPUT AND OUTPUT. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 07-JUL-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\* \*

\* SUBROUTINES FILES USED: \*

\* \*

\* SUB.XPRINT \*

\* SUB.DPRINT \*

\* SUB.SINPUT \*

\* SUB.GPBX \*

\* SUB.TVLINE \*

\* SUB.THLINE \*

\* SUB.TRECTF \*

\* SUB.TBLINE \*

\* SUB.TCIRCLE \*

\* SUB.TXTPUT \*

\* SUB.PRNSTR \*

\* \*

\* LIST OF MACROS \*

\* \*

\* PRN : FLEXIBLE PRINT \*

\* SPRN : PRINT STRING \*

\* INP : STRING INPUT \*

\* GKEY : GET SINGLE KEY \*

\* SCPOS : SET CURS POS AT X,Y \*

\* SETCX : SET CURSOR X \*

\* SETCY : SET CURSOR Y \*

\* CURF : CURSOR FORWARD \*

\* CURB : CURSOR BACKWARD \*

\* CURU : CURSOR UP \*

\* CURD : CURSOR DOWN \*

\* RCPOS : READ CURSOR POSITION \*

\* PDL : READ PADDLE STATE \*

\* TLINE : DIAGONAL TEXT LINE \*

\* TCIRC : TEXT CIRCLE \*

\* PBX : READ PDL BTN X \*

\* TVLIN : TEXT VERTICAL LINE \*

\* THLIN : TEXT HORIZ LINE \*

\* TRECF : TEXT FILL RECTANGLE \*

\* TPUT : TEXT CHAR PLOT AT XY \*

\* COL40 : FORCE 40COL MODE \*

\* COL80 : FORCE 80COL MODE \*

\* DIE80 : KILL 80COL FIRMWARE \*

\* MTXT0 : DISABLE MOUSETEXT \*

\* MTXT1 : ENABLE MOUSETEXT \*

\* WAIT : WAIT FOR KEYPRESS \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

|  |
| --- |
| **PRN (macro)**  **Input**:  ]1 = string or address  **Output**:  Outputs the literal  String provided or the  Null-terminated string  Located at the given  Address.  **Destroys**: AXYNVZCM  **Cycles**: 94+  **Size**: 32+ bytes |

**MAC.STDIO >> PRN**

The **PRN** macro prints a string directly to the screen. First, a test is given to determine whether a literal string or an address is being passed. If the parameter is a literal string, the **XPRINT** subroutine is called. Otherwise, the parameter is parsed as an address in the zero page, and **DPRINT** is called.

\*``````````````````````````````\*

\* PRN \*

\* \*

\* PRINT A LITERAL STRING OR \*

\* A NULL-TERMINATED STRING AT \*

\* A GIVEN ADDRESS. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = STRING OR ADDRESS \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* PRN "HELLO, WORLD!" \*

\* PRN #$300 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PRN MAC

IF ",]1 ; IF PARAM=STRING

JSR XPRINT ; SPECIAL PRINT

ASC ]1 ; PUT STRING HERE

HEX 00 ; STRING TERMINATE

ELSE ; ELSE, PARAM IS

; MEMORY LOCATION

\_MLIT ]1 ; PARSE FOR LITERAL

JSR DPRINT ; OR INDIRECT

FIN

<<<

|  |
| --- |
| **SPRN (macro)**  **Input**:  ]1 = string address  **Output**:  String printed to screen  **Destroys**: AXYNVZCM  **Cycles**: 40+  **Size**: 12 bytes |

**MAC.STDIO >> SPRN**

The **SPRN** macro prints a string with a preceding length byte to the screen. Unlike the PRN macro, this does not stop printing once a null character is encountered; once the number of bytes represented by the length byte are printed, control is returned to the calling routine.

\*

\*``````````````````````````````\*

\* SPRN \*

\* \*

\* PRINTS THE STRING LOCATED AT \*

\* THE SPECIFIED ADDRESS, WHICH \*

\* HAS A PRECEDING LENGTH BYTE. \*

\* \*

\* PARAMETERS: \*

\* \*

\* ]1 = STRING ADDRESS \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* SPRN #$300 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

SPRN MAC

\_AXLIT ]1

JSR PRNSTR

<<<

|  |
| --- |
| **INP (macro)**  **Input**:  none  **Output**:  Whatever is typed  **Destroys**: AXYNVZC  **Cycles**: 60+  **Size**: 45 bytes |

**MAC.STDIO >> INP**

The **INP** macro receives a string from keyboard input (followed by return) and holds it in **RETURN**. The characters corresponding to the keypresses are displayed on the screen as they are typed. Control is returned to the calling routine once the return key is pressed.

\*

\*``````````````````````````````\*

\* INP \*

\* \*

\* INPUTS A STRING FROM KEYBRD \*

\* AND STORES IT IN [RETURN] \*

\* \*

\* PARAMETERS \*

\* \*

\* NONE \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* INP \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

INP MAC

JSR SINPUT

<<<

|  |
| --- |
| **GKEY (macro)**  **Input**:  none  **Output**:  **.A** = key code  **Destroys**: AXYNZC  **Cycles**: 12+  **Size**: 7 bytes |

**MAC.STDIO >> GKEY**

The **GKEY** macro halts execution of the calling subroutine until a key is pressed. The corresponding character to the key is not echoed to the screen. The keycode is passed back via the accumulator.

\*

\*``````````````````````````````\*

\* GKEY \*

\* \*

\* WAITS FOR USER TO PRESS A \*

\* KEY, THEN STORES THAT IN .A \*

\* \*

\* PARAMETERS \*

\* \*

\* NONE \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* GKEY \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

GKEY MAC

JSR GETKEY ; MONITOR GET SUBROUTINE

LDY #0

STY STROBE ; RESET KBD STROBE

<<<

|  |
| --- |
| **SCPOS (macro)**  **Input**:  ]1 = X position  ]2 = Y position  **Output**:  none  **Destroys**: AXYNVCM  **Cycles**: 20+  **Size**: 15 bytes |

**MAC.STDIO >> SCPOS**

The **SCPOS** macro sets the cursor position at the given X and Y coordinates.

\*

\*``````````````````````````````\*

\* SCPOS \*

\* \*

\* SETS THE CURSOR POSITION. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = X POSITION \*

\* ]2 = Y POSITION \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* SCPOS #10;#10 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

SCPOS MAC

LDX ]1

STX CURSH ; PUT X INTO HPOS

LDX ]2

STX CURSV ; PUT Y INTO VPOS

JSR VTAB ; EXECUTE VTAB MONITOR ROUTINE

<<<

|  |
| --- |
| **SETCX (macro)**  **Input**:  ]1 = X position  **Output**:  none  **Destroys**: AXZC  **Cycles**: 11+  **Size**: 8 bytes |

**MAC.STDIO >> SETCX**

The **SETCX** macro sets the horizontal (X) position of the cursor.

\*

\*``````````````````````````````\*

\* SETCX \*

\* \*

\* SETS THE CURSOR X POSITION. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = X POSITION \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* SETCX #10 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

SETCX MAC

LDX ]1

STX CURSH ; SET HORIZ POS

JSR VTAB ; CALL VTAB MONITOR ROUTINE

<<<

|  |
| --- |
| **SETCY (macro)**  **Input**:  ]1 = Y position  **Output**:  none  **Destroys**: YZC  **Cycles**: 12+  **Size**: 9 bytes |

**MAC.STDIO >> SETCY**

The **SETCY** macro sets the vertical (Y) position of the cursor.

\*

\*``````````````````````````````\*

\* SETCY \*

\* \*

\* SET THE CURSOR Y POSITION. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = Y POSITION \*

\* \*

\* SETCY #10 \*

\* \*

\* SAMPLE USAGE: SETCY #10 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

SETCY MAC

LDY ]1

STY CURSV ; SET VERTICAL POS

JSR VTAB ; CALL VTAB MONITOR ROUTINE

<<<

|  |
| --- |
| **CURF (macro)**  **Input**:  ]1 = number of spaces to  move forward.  **Output**:  none  **Destroys**: AZC  **Cycles**: 17+  **Size**: 12 bytes |

**MAC.STDIO >> CURF**

The **CURF** macro moves the cursor forward by the given number of spaces.

\*

\*``````````````````````````````\*

\* CURF \*

\* \*

\* MOVE CURSOR FORWARD A NUMBER \*

\* OF SPACES. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = # OF SPACES TO MOVE \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* CURF #10 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

CURF MAC

LDA ]1 ; GET # TO ADD TO CURRENT

CLC ; POS; CLEAR CARRY

ADC CURSH ; ADD CURSH

STA CURSH ; STORE IN CURSH

JSR VTAB ; MONITOR VTAB SUBROUTINE

<<<

|  |
| --- |
| **CURB (macro)**  **Input**:  ]1 = number of spaces to  move backward  **Output**:  none  **Destroys**: AZNC  **Cycles**: 17+  **Size**: 12 bytes |

**MAC.STDIO >> CURB**

The **CURB** macro moves the cursor backward by the specified number of spaces.

\*

\*``````````````````````````````\*

\* CURB \*

\* \*

\* MOVE THE CURSOR BACKWARD BY \*

\* A NUMBER OF SPACES. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = # OF SPACES TO MOVE \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* CURB #10 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

CURB MAC

LDA CURSH ; GET CURRENT CURSOR HORIZ

SEC ; SET CARRY

SBC ]1 ; SUBTRACT GIVEN PARAM

STA CURSH ; STORE BACK IN CURSH

JSR VTAB ; VTAB MONITOR SUBROUTINE

<<<

|  |
| --- |
| **CURU (macro)**  **Input**:  ]1 = number of spaces to  move up  **Output**:  none  **Destroys**: ANZCV  **Cycles**: 18+  **Size**: 12 bytes |

**MAC.STDIO >> CURU**

The **CURU** macro moves the cursor up vertically for the specified number of spaces.

\*

\*``````````````````````````````\*

\* CURU \*

\* \*

\* MOVE CURSOR UP BY A NUMBER \*

\* OF SPACES. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = # OF SPACES TO GO UP \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* CURU #10 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

CURU MAC

LDA CURSV ; GET CURRENT CURSOR VERT

SEC ; SET CARRY

SBC ]1 ; SUBTRACT GIVEN PARAM

STA CURSV ; STORE BACK IN CURSV

JSR VTAB ; VTAB MONITOR ROUTINE

<<<

|  |
| --- |
| **CURD (macro)**  **Input**:  ]1 = number of spaces to  move down  **Output**:  none  **Destroys**: ANZCV  **Cycles**: 18+  **Size**: 12 bytes |

**MAC.STDIO >> CURD**

The **CURD** macro moves the cursor down by a specified number of spaces.

\*

\*``````````````````````````````\*

\* CURD \*

\* \*

\* MOVE THE CURSOR DOWN BY A \*

\* NUMBER OF SPACES. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = # OF SPACES TO MOVE \*

\* \*

\* SAMPLE USAGE: CURD #10 \*

\* \*

\* CURD #10 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

CURD MAC

LDA CURSV ; GET CURRENT VERT POS

CLC ; CLEAR CARRY

ADC ]1 ; ADD GIVEN PARAMETER

STA CURSV ; STORE BACK IN CURSV

JSR VTAB ; VTAB MONITOR SUBROUTINE

<<<

|  |
| --- |
| **RCPOS (macro)**  **Input**:  ]1 = X position  ]2 = Y position  **Output**:  none  **Destroys**: AYNZCV  **Cycles**: 20+  **Size**: 12 bytes |

**MAC.STDIO >> RCPOS**

The **RCPOS** macro retrieves the character found at the given X,Y coordinates on the screen (text mode). That character is stored in the accumulator.

\*

\*``````````````````````````````\*

\* RCPOS \*

\* \*

\* READ THE CHARACTER AT POS \*

\* X,Y AND LOADS INTO ACCUM \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = X POSITION \*

\* ]2 = Y POSITION \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* RCPOS #3;#9 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

RCPOS MAC

LDY ]1 ; ROW

LDA ]2 ; COLUMN

JSR GBCALC ; GET ADDR FOR SCREEN POS

LDA (GBPSH),Y ; GET CHAR IN ADDRESS

<<<

|  |
| --- |
| **PDL (macro)**  **Input**:  ]1 = paddle number  **Output**:  **.Y** = paddle state  **Destroys**: AXNVZ  **Cycles**: 9+  **Size**: 6 bytes |

**MAC.STDIO >> PDL**

The **PDL** macro reads the state of the given paddle number (usually #0) and stores a value between 0 and 255 in the **.Y** register.

\*

\*``````````````````````````````\*

\* PDL \*

\* \*

\* SIMPLY READS STATE OF PADDLE \*

\* NUMBER [NUM] AND STORES IT \*

\* IN THE Y REGISTER. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = PADDLE # TO READ \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* PDL #0 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PDL MAC ; GET PADDLE VALUE

LDX ]1 ; READ PADDLE # ]1 (USUALLY 0)

JSR PREAD ; PADDLE READING STORED IN Y

<<<

|  |
| --- |
| **PBX (macro)**  **Input**:  ]1 = paddle button addr  **Output**:  **.X** = button state  **Destroys**: AXNZ  **Cycles**: 9  **Size**: 8 bytes |

**MAC.STDIO >> PBX**

The **PBX** macro reads the state of the specified paddle button. These can be referred to in the parameters as **PB0**, **PB1**, **PB2**, or **PB3**, which signify the different addresses to read.

\*

\*``````````````````````````````\*

\* PBX \*

\* \*

\* READ THE SPECIFIED PADDLE \*

\* BUTTON. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = PADDLE BUTTON TO READ \*

\* \*

\* PB0: $C061 PB1: $C062 \*

\* PB2: $C063 PB4: $C060 \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* PBX PB0 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PBX MAC

LDX #1

LDA ]1 ; IF BTN = PUSHED

BMI EXIT ; IF HIBYTE SET, BUTTON PUSHED

LDX #0 ; OTHERWISE, BUTTON NOT PUSHED

EXIT

<<<

|  |
| --- |
| **TVLIN (macro)**  **Input**:  ]1 = starting vertical  (Y) position  ]2 = ending vertical (Y)  position  ]3 = X position  ]4 = fill character  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 55+  **Size**: 19 bytes |

**MAC.STDIO >> TVLIN**

The **TVLIN** macro creates a vertical line in text mode with a provided character. This is printed to screen memory, and does not interfere with **COUT**, cursor position, etc.

\*

\*``````````````````````````````\*

\* TVLIN \*

\* \*

\* CREATE A VERTICAL LINE WITH \*

\* A GIVEN TEXT FILL CHARACTER \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = START OF VERT LINE \*

\* ]2 = END OF VERT LINE \*

\* ]3 = X POSITION OF LINE \*

\* ]4 = FILL CHARACTER \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* TVLIN #0;#10;#3;#$18 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

TVLIN MAC

LDA ]1 ; Y START

STA WPAR2

LDA ]2 ; Y END

STA WPAR2+1

LDA ]3 ; X POSITION

STA WPAR1

LDA ]4 ; CHARACTER

STA BPAR1

JSR TVLINE

<<<

|  |
| --- |
| **THLIN (macro)**  **Input**:  ]1 = start of horizontal  line  ]2 = end of horizontal  line  ]3 = vertical position  ]4 = fill character  **Output**:  Horizontal line to screen  **Destroys**:  **Cycles**: 112+  **Size**: 19 bytes |

**MAC.STDIO >> THLIN**

The **THLIN** macro creates a horizontal line in text mode with the specified fill character. This is blitted directly to screen memory for speed and for avoiding **COUT** interference.

\*

\*``````````````````````````````\*

\* THLIN \*

\* \*

\* CREATE A HORIZONTAL LINE \*

\* FROM A FILL CHARACTER. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = START OF HORIZ LINE \*

\* ]2 = END OF HORIZ LINE \*

\* ]3 = Y POSITION OF LINE \*

\* ]4 = FILL CHARACTER \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* THLIN #0;#10;#12;#$18 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

THLIN MAC

LDA ]1 ; X START

STA WPAR1

LDA ]2 ; X END

STA WPAR1+1

LDA ]3 ; Y POS

STA BPAR1

LDA ]4 ; FILL CHAR

STA BPAR2

JSR THLINE

<<<

|  |
| --- |
| **TRECF (macro)**  **Input**:  ]1 = X origin  ]2 = Y origin  ]3 = X destination  ]4 = Y destination  ]5 = fill character  **Output**:  none  **Destroys**:  **Cycles**: 95+  **Size**: 23 bytes |

**MAC.STDIO >> TRECF**

The **TRECF** macro draws a text rectangle to the screen at the given coordinates, filled with the specified character.

\*

\*``````````````````````````````\*

\* TRECF \*

\* \*

\* CREATE A RECTANGLE FILLED \*

\* WITH A GIVEN TEXT CHARACTER \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = HORIZ START POSITION \*

\* ]2 = VERT START POSITION \*

\* ]3 = HORIZ END POSITION \*

\* ]4 = VERT END POSITION \*

\* ]5 = FILL CHARACTER \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* TRECF #0;#10;#0;#10;#'X' \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

TRECF MAC

LDA ]1 ; LEFT BOUNDARY

STA WPAR1

LDA ]2 ; TOP BOUNDARY

STA WPAR2

LDA ]3 ; RIGHT BOUNDARY

STA WPAR1+1

LDA ]4 ; BOTTOM BOUNDARY

STA WPAR2+1

LDA ]5 ; FILL CHAR

STA BPAR1

JSR TRECTF

<<<

|  |
| --- |
| **TPUT (macro)**  **Input**:  ]1 = horizontal(X)  position  ]2 = vertical(Y)  position  ]3 = character to plot  **Output**:  Character on screen  **Destroys**: AXYNVZCM  **Cycles**: 41+  **Size**: 9 bytes |

**MAC.STDIO >> TPUT**

The **TPUT** macro displays a single character on the screen at the given X,Y coordinates. Like **TVLIN** and **THLIN**, the character is directly plotted to screen memory, bypassing **COUT**.

\*

\*``````````````````````````````\*

\* TPUT TEXT CHARACTER PLOT \*

\* \*

\* PLOT A SINGLE TEXT CHARACTER \*

\* DIRECTLY TO SCREEN MEMORY AT \*

\* A GIVEN X,Y POSITION. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = X POSITION \*

\* ]2 = Y POSITION \*

\* ]3 = CHARACTER TO PLOT \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* TPUT #10;#10;#AA \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

TPUT MAC

LDX ]1 ; XPOS INTO .X

LDY ]2 ; YPOS INTO .Y

LDA ]3 ; FILL IN .A

JSR TXTPUT

<<<

|  |
| --- |
| **DIE80 (macro)**  **Input**:  none  **Output**:  none  **Destroys**: ANVC  **Cycles**: 8  **Size**: 5 bytes |

**MAC.STDIO >> DIE80**

The **DIE80** macro kills 80-column mode, effectively forcing 40-column mode.

\*

\*``````````````````````````````\*

\* DIE80 \*

\* \*

\* SEND CTRL-U TO COUT, FORCING \*

\* 40 COLUMN MODE. \*

\* \*

\* PARAMETERS \*

\* \*

\* NONE \*

\* \*

\* USAGE \*

\* \*

\* DIE80 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

DIE80 MAC

LDA #21 ; CTRL-U CHARACTER

JSR COUT ; SEND TO SCREEN

<<<

|  |
| --- |
| **COL80 (macro)**  **Input**:  none  **Output**:  80-cloumn mode  **Destroys**: ANVC  **Cycles**: 8  **Size**: 5 bytes |

**MAC.STDIO >> COL80**

The **COL80** macro turns on 80-column mode. Note that this only works with a system capable of using 80 columns.

\*

\*``````````````````````````````\*

\* COL80 \*

\* \*

\* FORCE 80-COLUMN MODE. \*

\* \*

\* PARAMETERS \*

\* \*

\* NONE \*

\* \*

\* USAGE \*

\* \*

\* COL80 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

COL80 MAC

LDA #18 ; CTRL-R CHARACTER

JSR COUT ; SEND TO SCREEN

<<<

|  |
| --- |
| **COL40 (macro)**  **Input**:  none  **Output**:  40-column mode  **Destroys**: ANVC  **Cycles**: 8  **Size**: 5 bytes |

**MAC.STDIO >> COL40**

The **COL40** macro turns on the default 40-column mode. If this does not work on a particular system, **DIE80** may work better.

\*

\*``````````````````````````````\*

\* COL40 \*

\* \*

\* FORCE 40-COLUMN MODE \*

\* \*

\* PARAMETERS \*

\* \*

\* NONE \*

\* \*

\* USAGE \*

\* \*

\* COL40 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

COL40 MAC

LDA #17 ; CTRL-Q CHARACTER

JSR COUT ; SEND TO SCREEN

<<<

|  |
| --- |
| **MTXT0 (macro)**  **Input**:  none  **Output**:  none  **Destroys**: ANVC  **Cycles**: 8  **Size**: 5 bytes |

**MAC.STDIO >> MTXT0**

The **MTXT0** macro turns off mousetext, if it was turned on in a capable system in the first place.

\*

\*``````````````````````````````\*

\* MTXT0 \*

\* \*

\* DISABLE MOUSETEXT, IF IT IS \*

\* ENABLED. \*

\* \*

\* PARAMETERS \*

\* \*

\* NONE \*

\* \*

\* USAGE \*

\* \*

\* MTXT0 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

MTXT0 MAC

LDA #24 ; CTRL-X

JSR COUT ; SEND TO SCREEN

<<<

|  |
| --- |
| **MTXT1 (macro)**  **Input**:  none  **Output**:  none  **Destroys**: ANVC  **Cycles**: 8  **Size**: 5 bytes |

**MAC.STDIO >> MTXT1**

The **MTXT1** macro turns on mousetext, if the system is capable of using it.

\*

\*``````````````````````````````\*

\* MTXT1 \*

\* \*

\* ENABLE MOUSETEXT IF IT IS \*

\* AVAILABLE. \*

\* \*

\* PARAMETERS \*

\* \*

\* NONE \*

\* \*

\* USAGE \*

\* \*

\* MTXT1 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

MTXT1 MAC

LDA #27 ; CTRL-[

JSR COUT ; SEND TO SCREEN

<<<

|  |
| --- |
| **WAIT (macro)**  **Input**:  none  **Output**:  **.A** = key code  **Destroys**: ANV  **Cycles**: 10+  **Size**: 10 bytes |

**MAC.STDIO >> WAIT**

The **WAIT** macro halts the main subroutine’s execution until a key is pressed, then returns the key code in the accumulator. Note that this is not echoed to the screen.

\*

\*``````````````````````````````\*

\* WAIT \*

\* \*

\* WAIT FOR A KEYPRESS WITHOUT \*

\* INTERFERING WITH COUT. KEY \*

\* CODE IS STORED IN .A. \*

\* \*

\* PARAMETERS \*

\* \*

\* NONE \*

\* \*

\* USAGE \*

\* \*

\* WAIT \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

WAIT MAC

]WTLP LDA KYBD ; READ KEYBOARD BUFFER

BPL ]WTLP ; IF 0, KEEP LOOPING

AND #$7F ; OTHERWISE, SET HI BIT

STA STROBE ; CLEAR STROBE

<<<

|  |
| --- |
| **TLINE (macro)**  **Input**:  ]1 = X origin  ]2 = Y origin  ]3 = X destination  ]4 = Y destination  **Output**:  Text line to screen  **Destroys**: AXYNVZCM  **Cycles**: 309+  **Size**: bytes |

**MAC.STDIO >> TLINE**

The **TLINE** macro creates a line from the starting point X,Y to the ending point X2,Y2 in text mode with the specified fill character. This macro calls the **TBLINE** subroutine, which uses Bressenham’s line algorithm and plots the characters directly to screen memory.

\*

\*``````````````````````````````\*

\* TLINE \*

\* \*

\* USE THE BRESSENHAM LINE \*

\* ALGORITHM TO DRAW A LINE \*

\* WITH A FILL CHARACTER. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = X-ORIGIN \*

\* ]2 = Y-ORIGIN \*

\* ]3 = X-DESTINATION \*

\* ]4 = Y-DESTINATION \*

\* \*

\* USAGE \*

\* \*

\* TLINE #0;#0;#23;#39 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

TLINE MAC

LDA ]1

STA WPAR1

LDA ]2

STA WPAR1+1

LDA ]3

STA WPAR2

LDA ]4

STA WPAR2+1

LDA ]5

STA BPAR1

JSR TBLINE

<<<

|  |
| --- |
| **TCIRC (macro)**  **Input**:  ]1 = X center  ]2 = Y center  ]3 = radius  ]4 = fill character  **Output**:  Circle to text screen  **Destroys**: AXYNVZCM  **Cycles**: 516+  **Size**: 19 bytes |

**MAC.STDIO >> TCIRC**

The **TCIRC** macro draws a circle on the screen at a given radius with a specified fill character at the X,Y coordinates passed. This macro calls the **TCIRCLE** routine, which utilizes Bressenham’s circle algorithm to plot characters directly to screen memory.

\*

\*``````````````````````````````\*

\* TCIRC \*

\* \*

\* USE THE BRESSENHAM CIRCLE \*

\* ALGORITHM TO DRAW A CIRCLE \*

\* WITH A FILL CHARACTER. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = CENTER X-LOCATION \*

\* ]2 = CENTER Y-LOCATION \*

\* ]3 = RADIUS \*

\* ]4 = FILL CHARACTER \*

\* \*

\* USAGE \*

\* \*

\* TCIRC #19;#11;#10;#"\*" \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

TCIRC MAC

LDA ]1

STA WPAR1

LDA ]2

STA WPAR2

LDA ]3

STA BPAR1

LDA ]4

STA BPAR2

JSR TCIRCLE

<<<

|  |
| --- |
| **DPRINT (sub)**  **Input**:  **WPAR1** = string address,  two bytes  **Output**:  Print string to screen  **Destroys**: AXYNZM  **Cycles**: 61+  **Size**: 27 bytes |

**SUB.DPRINT >> DPRINT**

The **DPRINT** subroutine prints a null-terminated string to the screen via **COUT** from the given address. A total of only 256 characters will print at one time.

\*

\*``````````````````````````````\*

\* DPRINT (NATHAN RIGGS) \*

\* \*

\* PRINT A ZERO-TERMINATED \*

\* STRING AT A GIVEN ADDRESS. \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = STRING ADDRESS (2B) \*

\* \*

\* OUTPUT: \*

\* \*

\* PRINT STRING TO SCREEN \*

\* \*

\* DESTROYS: AXYNVBDIZCMS \*

\* ^^^^ ^ ^ \*

\* \*

\* CYCLES: 61+ \*

\* SIZE: 27 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]ADDR1 EQU WPAR1

\*

DPRINT

\*

LDY #$00 ; RESET COUNTER

:LOOP

LDA (]ADDR1),Y

BEQ :EXIT ; IF CHAR = $00 THEN EXIT

JSR COUT1 ; OTHERWISE, PRINT CHAR

INY ; INCREAS COUNTER

BNE :LOOP ; IF COUNTER < 256, LOOP

:EXIT

RTS

|  |
| --- |
| **TBLINE (sub)**  **Input**:  **WPAR1** = X origin  **WPAR2** = Y origin  **WPAR1**+1 = X destination  **WPAR2**+1 = Y destination  **Output**:  Line to screen  **Destroys**: AXYNVZCM  **Cycles**: 283+  **Size**: 188 bytes |

**SUB.TBLINE >> TBLINE**

The **TBLINE** subroutine creates a line composed of a given text character from X,Y to X2,Y2. For the sake of speed, this subroutine uses the Bressenham line algorithm to plot the line directly to screen memory.

\*

\*``````````````````````````````\*

\* TBLINE (NATHAN RIGGS) \*

\* \*

\* OUTPUTS A LINE FROM COORDS \*

\* X1,Y1 TO X2,Y2 USING THE \*

\* BRESSENHAM LINE ALOGORITHM \*

\* \*

\* INPUT: \*

\* \*

\* ]X1 STORED IN WPAR1 \*

\* ]X2 STORED IN WPAR1+1 \*

\* ]Y1 STORED IN WPAR2 \*

\* ]Y2 STORED IN WPAR2+1 \*

\* ]F STORED IN BPAR1 \*

\* \*

\* OUTPUT: \*

\* \*

\* NONE \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 283+ \*

\* SIZE: 188 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]X1 EQU WPAR1 ; PARAMETERS PASSED VIA

]X2 EQU WPAR2 ; ZERO PAGE LOCATIONS

]Y1 EQU WPAR1+1

]Y2 EQU WPAR2+1

]F EQU BPAR1

\*

]DX EQU VARTAB ; CHANGE IN X; 1 BYTE

]DY EQU VARTAB+1 ; CHANGE IN Y; 1 BYTE

]SX EQU VARTAB+2 ; X POSITION STEP; 1 BYTE

]SY EQU VARTAB+3 ; Y POSITION STEP; 1 BYTE

]ERR EQU VARTAB+4 ; SLOPE ERROR; 1 BYTE

]ERR2 EQU VARTAB+5 ; COMPARISON COPY OF ]ERR; 1 BYTE

\*

TBLINE

\*

\*\* FIRST CALCULATE INITIAL VALUES

\*

\*\* CHECK IF Y STEP IS POSITIVE OR NEGATIVE

\*

LDX #$FF ; .X = -1

LDA ]Y1 ; GET Y1 - Y2

SEC ; RESET CARRY

SBC ]Y2

BPL :YSTORE ; IF POSITIVE, SKIP TO STORE

LDX #1 ; .X = +1

EOR #$FF ; NEG ACCUMULATOR

CLC

ADC #1

:YSTORE

STA ]DY ; STORE CHANGE IN Y

STX ]SY ; STORE + OR - Y STEPPER

\*

\*\* NOW CHECK POSITIVE OR NEGATIVE X STEP

\*

LDX #$FF ; .X = -1

LDA ]X1 ; GET X1 - X2

SEC ; RESET CARRY

SBC ]X2 ; SUBTRACT X2

BPL :XSTORE ; IF POSITIVE, SKIP TO X STORE

LDX #1 ; .X = +1

EOR #$FF ; NEGATIVE ACCUMULATOR

CLC

ADC #1

:XSTORE

STA ]DX ; STORE CHANGE IN X

STX ]SX ; STORE + OR - X STEPPER

\*

\*\* IF CHANGE IN X IS GREATER THAN CHANGE IN Y,

\*\* THEN INITIAL ERROR IS THE CHANGE IN X; ELSE,

\*\* INITIAL ERROR IS THE CHANGE IN Y

\*

CMP ]DY ; DX IS ALREADY IN .A

BEQ :SKIP ; IF EQUAL, US CHANGE IN Y

BPL :SKIP2 ; IF GREATER THAN, USE CHANGE IN X

:SKIP

LDA ]DY ; GET CHANGE IN Y

EOR #$FF ; NEGATE

CLC

ADC #1

:SKIP2

STA ]ERR ; STORE EITHER DX OR DY IN ERR

ASL ]DX ; DX = DX \* 2

ASL ]DY ; DY = DY \* 2

\*

\*\* NOW LOOP THROUGH EACH POINT ON LINE

\*

:LP

\*

\*\* PRINT CHARACTER FIRST

\*

LDA ]Y1 ; .A = Y POSITION

LDY ]X1 ; .Y = X POSITION

JSR GBCALC ; FIND SCREEN MEM LOCATION

LDA ]F ; LOAD FILL INTO .A

STA (GBPSH),Y ; PUSH TO SCREEN MEMORY

\*

\*\* NOW CHECK IF X1 = X2, Y = Y2

\*

LDA ]X1 ; IF X1 != X2 THEN

CMP ]X2 ; KEEP LOOPING

BNE :KEEPGO

LDA ]Y1 ; ELSE, CHECK IF Y1 = Y2

CMP ]Y2

BEQ :EXIT ; IF EQUAL, EXIT; ELSE, LOOP

:KEEPGO

LDA ]ERR ; LOAD ERR AND BACKUP

STA ]ERR2 ; FOR LATER COMPARISON

CLC ; CLEAR CARRY

ADC ]DX ; ADD CHANGE IN X

BMI :SKIPX ; IF RESULT IS -, SKIP

BEQ :SKIPX ; TO CHANGING Y POS

LDA ]ERR ; RELOAD ERR

SEC ; SET CARRY

SBC ]DY ; SUBTRACT CHANGE IN Y

STA ]ERR ; STORE ERROR

LDA ]X1 ; LOAD CURRENT X POSITION

CLC ; CLEAR CARRY

ADC ]SX ; INCREASE OR DECREASE BY 1

STA ]X1 ; STORE NEW X POSITION

:SKIPX

LDA ]ERR2 ; LOAD EARLIER ERR

CMP ]DY ; IF ERR - CHANGE IN Y IS +

BPL :SKIPY ; SKIP CHANGING Y POS

LDA ]ERR ; RELOAD ERR

CLC ; CLEAR CARRY

ADC ]DX ; ADD CHANGE IN X

STA ]ERR ; STORE NEW ERR

LDA ]Y1 ; LOAD Y POSITION

CLC ; CLEAR CARRY

ADC ]SY ; INCREASE OR DECREASE YPOS BY 1

STA ]Y1 ; STORE NEW Y POSITION

:SKIPY

JMP :LP ; LOOP LINE DRAWING

:EXIT

RTS

|  |
| --- |
| **SINPUT (sub)**  **Input**:  None  **Output**:  **.X** = string length  **RETLEN** = string length  **RETURN** = string typed  **Destroys**: AXYNVZC  **Cycles**: 60+  **Size**: 45 bytes |

**SUB.SINPUT >> SINPUT**

The **SINPUT** subroutine halts the calling routine’s execution while it waits for input from the keyboard, echoing the keys pressed to the screen. Once the return key has been pressed, the string is then stored in **RETURN** and control is passed back to main execution.

\*

\*``````````````````````````````\*

\* SINPUT (NATHAN RIGGS) \*

\* \*

\* INPUT \*

\* \*

\* NONE \*

\* \*

\* OUTPUT: \*

\* \*

\* .X = LENGTH OF STRING \*

\* RETURN = STRING TYPED \*

\* RETLEN = LENGTH OF STRING \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^ \*

\* \*

\* CYCLES: 60+ \*

\* SIZE: 45 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]STRLEN EQU VARTAB ; 1 BYTE

\*

SINPUT

\*

LDX #$00

JSR GETLN

STX ]STRLEN ; STORE STR LENGTH

CPX #0 ; IF LEN = 0, EXIT

BNE :INP\_CLR

STX RETLEN

STX RETURN

JMP :EXIT

:INP\_CLR

LDA ]STRLEN ; LENGTH OF STRING

STA RETURN ; STRING LENGTH FIRST BYTE

STA RETLEN ; PUT LENGTH + 1 HERE

INC RETLEN

LDX #255

LDY #0

:LOOP

INX

INY

LDA KEYBUFF,X ; PUT STR INTO NEW LOC

STA RETURN,Y

CPX ]STRLEN ; IF Y < STR LENGTH

BNE :LOOP ; LOOP; ELSE, EXIT

:EXIT

RTS

|  |
| --- |
| **XPRINT (sub)**  **Input**:    ASC string following call  To the subroutine    **Output**:  String to screen  **Destroys**: AYNVZC  **Cycles**: 63+  **Size**: 33 bytes |

**SUB.XPRINT >> XPRINT**

The **XPRINT** subroutine prints a null-terminated string that follows the call to the subroutine, returning back to the program by adding the string length to the program counter. The string cannot be more than 255 characters long.

\*

\*``````````````````````````````\*

\* XPRINT (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* ASC AFTER SUBROUTINE CALL \*

\* THAT CONTAINS STRING TO PRN \*

\* \*

\* OUTPUT \*

\* \*

\* STRING TO SCREEN \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^ ^^^ ^^ \*

\* \*

\* CYCLES: 63+ \*

\* SIZE: 33 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

XPRINT

PLA ; GET CURRENT

STA ADDR1 ; EXECUTION ADDRESS

PLA

STA ADDR1+1

LDY #$01 ; POINT TO NEXT

; INSTRUCTION

:LOOP

LDA (ADDR1),Y ; GET CHARACTER

BEQ :EXIT ; IF CHAR = $00 THEN EXIT

JSR COUT1 ; OTHERWISE, PRINT CHAR

INY ; INCREASE COUNTER

BNE :LOOP ; IF COUNTER < 255, LOOP

:EXIT

CLC ; CLEAR CARRY

TYA ; MOVE .Y TO .A

ADC ADDR1 ; ADD RETURN LOBYTE

STA ADDR1 ; SAVE

LDA ADDR1+1 ; GET RETURN HIBYTE

ADC #$00 ; ADD CARRY

PHA ; PUSH TO STACK

LDA ADDR1

PHA ; PUSH TO STACK

RTS

|  |
| --- |
| **THLINE (sub)**  **Input**:  **WPAR1** = X origin  **BPAR1** = Y position  **BPAR2** = fill character  **WPAR1**+1 = X destination  **Output**:  Horizontal line to screen  **Destroys**: AXYNVBZCM  **Cycles**: 90+  **Size**: 47 bytes |

**SUB.THLINE >> THLINE**

The **THLINE** subroutine creates a horizontal line at the specified Y position, starting at a given X origin and ending at the X destination. This line is created with the specified fill character.

\*

\*``````````````````````````````\*

\* THLINE (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = X ORIGIN \*

\* WPAR1+1 = X DESTINATION \*

\* BPAR1 = Y POSITION \*

\* BPAR2 = FILL CHARACTER \*

\* \*

\* OUTPUT: HORIZONTAL LINE TO \*

\* SCREEN \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^^ ^^^ \*

\* \*

\* CYCLES: 90+ \*

\* SIZE: 47 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]X1 EQU WPAR1 ; 1 BYTE

]X2 EQU WPAR1+1 ; 1 BYTE

]Y1 EQU BPAR1 ; 1 BYTE

]F EQU BPAR2 ; 1 BYTE

\*

THLINE

LDA ]Y1 ; LOAD ROW

LDY ]X1 ; LOAD X START POS

:LOOP

JSR GBCALC ; GOSUB GBASCALC ROUTINE,

; WHICH FINDS MEMLOC FOR

; POSITION ON SCREEN

LDA ]F

STA (GBPSH),Y ; PUSH ]F TO SCREEN MEM

LDA ]Y1

INY ; INCREASE X POS

CPY ]X2 ; IF LESS THAN X END POS

BNE :LOOP ; REPEAT UNTIL DONE

:EXIT

RTS

|  |
| --- |
| **TCIRCLE (sub)**  **Input**:  **WPAR1** = center X position  **WPAR2** = center Y position  **BPAR1** = radius  **BPAR2** = fill character  **Output**:  Circle to screen  **Destroys**: AXYNVZCM  **Cycles**: 494+  **Size**: 420 bytes |

**SUB.TCIRCLE >> TCIRCLE**

The **TCIRCLE** subroutine creates a circle of text on the screen with a given radius at the specified X,Y center coordinates. The circle uses Bressenham’s circle algorithm, and plots directly to screen memory.

While this wasn’t quite copied line by line, substantial debt is owed to Marc Golombeck’s 6502 Assembly implementation of the algorithm.

\*

\*``````````````````````````````\*

\* TCIRCLE (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = X CENTER POS \*

\* WPAR2 = Y CENTER POS \*

\* BPAR1 = RADIUS \*

\* BPAR2 = FILL CHARACTER \*

\* \*

\* OUTPUT: \*

\* \*

\* USES BRESENHAM'S CIRCLE \*

\* ALGORITHM TO DRAW A CIRCLE \*

\* TO THE 40-COLUMN TEXTMODE \*

\* SCREEN. \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 494+ \*

\* SIZE: 420 BYTES \*

\* \*

\* SUBSTANTIAL DEBT IS OWED TO \*

\* MARC GOLOMBECK AND HIS GREAT \*

\* IMPLEMENTATION OF THE \*

\* BRESENHAM CIRCLE ALGORITHM \*

\* IN 6502 AND APPLESOFT, WHICH \*

\* IS BASED ON THE GERMAN LANG \*

\* VERSION OF WIKIPEDIA'S ENTRY \*

\* ON THE ALGORITHM THAT HAS A \*

\* BASIC PSEUDOCODE EXAMPLE. \*

\* THAT EXAMPLE, WITH CHANGES \*

\* VARIABLE NAMES, IS INCLUDED \*

\* BELOW. \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]XC EQU WPAR1

]YC EQU WPAR2

]R EQU BPAR1

]F EQU BPAR2

\*

]Y EQU VARTAB ; CENTER YPOS

]X EQU VARTAB+1 ; CENTER XPOS

]DY EQU VARTAB+2 ; CHANGE IN Y

]DX EQU VARTAB+4 ; CHANGE IN X

]ERR EQU VARTAB+6 ; ERROR VALUE

]DIAM EQU VARTAB+8 ; DIAMETER

]XT EQU VARTAB+10 ; INVERTED X VALUE

]YT EQU VARTAB+12 ; INVERTED Y VALUE

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* \*

\* BASIC PSEUDOCODE \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* X = R

\* Y = 0

\* ERROR = R

\* SETPIXEL XC + X, YC + Y

\* WHILE Y < X

\* DY = Y \* 2 + 1

\* Y = Y + 1

\* ERROR = ERROR - DY

\* IF ERROR < 0 THEN

\* DX = 1 - X \* 2

\* X = X - 1

\* ERROR = ERROR - DX

\* END IF

\* SETPIXEL XC + X, YC + Y

\* SETPIXEL XC - X, YC + Y

\* SETPIXEL XC - X, YC - Y

\* SETPIXEL XC + X, YC - Y

\* SETPIXEL XC + Y, YC + X

\* SETPIXEL XC - Y, YC + X

\* SETPIXEL XC - Y, YC - X

\* SETPIXEL XC + Y, YC - X

\* WEND

\*

TCIRCLE

\*

\*\* FIRST, INITIALIZE VARIABLES

\*

LDA #0 ; CLEAR YPOS

STA ]Y

LDA ]R ; LOAD RADIUS

STA ]X ; X = RADIUS

STA ]ERR ; ERROR = RADIUS

ASL ; R \* 2

STA ]DIAM ; STORE DIAMETER

\*

\*\* NOW DRAW FIRST PART OF CIRCLE

\*

\*\* CALCULATE -X AND -Y

\*

LDA ]X ; GET XPOS

EOR #$FF ; NEGATE

CLC

ADC #1

STA ]XT ; STORE NEGATED IN XT

LDA ]Y ; GET YPOS

EOR #$FF ; NEGATE

CLC

ADC #1

STA ]YT ; STORE NEGATED IN YT

\*

\*\* PLOT XC+X,YC

\*

LDA ]XC ; LOAD CIRCLE CENTER XPOS

CLC ; CLEAR CARRY

ADC ]X ; ADD CURRENT XPOS

TAY ; TRANSER TO .Y

TAX ; AND .X

LDA ]YC ; LOAD CIRCLE CENTER YPOS

JSR GBCALC ; GET X,Y SCREEN MEMORY POS

LDA ]F ; LOAD FILL CHAR

STA (GBPSH),Y ; STORE IN SCREEN MEMORY

\*

\*\* PLOT XC-X,YC

\*

LDA ]XC ; LOAD CIRCLE CENTER XPOS

CLC ; CLEAR CARRY

ADC ]XT ; ADD NEGATED CURRENT XPOS

TAX ; TRANSFER TO .X

TAY ; AND .Y

LDA ]YC ; LOAD CIRCLE CENTER YPOS

JSR GBCALC ; GET X,Y SCREEN MEMORY POS

LDA ]F ; LOAD FILL CHAR

STA (GBPSH),Y ; STORE IN SCREEN MEMORY

\*

\*\* PLOT XC,YC+X

\*

LDA ]XC ; LOAD CIRCLE CENTER XPOS

TAY ; TRANSFER TO .Y

TAX ; AND .X

LDA ]YC ; LOAD CIRCLE CENTER YPOS

CLC ; CLEAR CARRY

ADC ]X ; ADD CURRENT XPOS

JSR GBCALC ; GET X,Y SCREEN MEMORY POS

LDA ]F ; LOAD FILL CHAR

STA (GBPSH),Y ; STORE IN SCREEN MEMORY

\*

\*\* PLOT XC,YC-X

\*

LDA ]XC ; LOAD CIRCLE CENTER XPOS

TAY ; TRANSFER TO .Y

TAX ; AND .X

LDA ]YC ; LOAD CIRCLE CENTER YPOS

CLC ; CLEAR CARRY

ADC ]XT ; ADD NEGATED CURRENT XPOS

JSR GBCALC ; GET X,Y SCREEN MEMORY POS

LDA ]F ; LOAD FILL CHAR

STA (GBPSH),Y ; STORE IN SCREEN MEMORY

\*

\*\* NOW LOOP UNTIL CIRCLE IS FINISHED

\*

:LOOP

\*

\*\* CHECK IF CIRCLE FINISHED

\*

LDA ]Y ; IF Y > X

CMP ]X

BCC :LPCONT ; CONTINUE LOOPING

JMP :EXIT ; OTHERWISE, CIRCLE DONE

:LPCONT

:STEPY ; STEP THE Y POSITION

LDA ]Y ; LOAD YPOS

ASL ; MULTIPLY BY 2

\*CLC

ADC #1 ; ADD +1

STA ]DY ; STORE CHANGE OF Y

INC ]Y ; INCREASE YPOS

LDA ]DY ; NEGATE

EOR #$FF

CLC

ADC #1

ADC ]ERR ; ADD ERR

STA ]ERR ; ERR = ERR - DY

BPL :PLOT ; IF ERR IS +, SKIP TO PLOT

:STEPX

LDA ]X ; LOAD XPOS

ASL ; MULTIPLY BY 2

EOR #$FF ; NEGATE

CLC

ADC #1

ADC #1 ; (X\*2) + 1

STA ]DX ; STORE CHANGE OF X

DEC ]X ; DECREASE YPOS

LDA ]DX ; NEGATE

EOR #$FF

CLC

ADC #1

ADC ]ERR ; ADD ERR

STA ]ERR ; ERR = ERR - DX

\*

:PLOT

\*

\*\* NOW CALCULATE -X AND -Y

\*

LDA ]X

EOR #$FF ; NEGATE

CLC

ADC #1

STA ]XT

LDA ]Y

EOR #$FF ; NEGATE

CLC

ADC #1

STA ]YT

\*

\*\* NOW PLOT CIRCLE OCTANTS

\*

\*\* PLOT XC+X,YC+Y

\*

LDA ]XC ; LOAD CIRCLE CENTER XPOS

CLC ; CLEAR CARRY

ADC ]X ; ADD CURRENT XPOS

TAY ; TRANSFER TO .Y

TAX ; AND .X

LDA ]YC ; LOAD CIRCLE CENTER YPOS

CLC ; CLEAR CARRY

ADC ]Y ; ADD CURRENT YPOS

JSR GBCALC ; GET X,Y SCREEN ADDRESS

LDA ]F ; LOAD FILL CHAR

STA (GBPSH),Y ; STORE AT SCREEN ADDRESS

\*

\*\* PLOT XC-X,YC+Y

\*

LDA ]XC ; LOAD CIRCLE CENTER XPOS

CLC ; CLEAR CARRY

ADC ]XT ; ADD NEGATED CURRENT XPOS

TAY ; TRANSFER TO .Y

TAX ; AND TO .X

LDA ]YC ; LOAD CIRCLE CENTER YPOS

CLC ; CLEAR CARRY

ADC ]Y ; ADD CURRENT YPOS

JSR GBCALC ; GET X,Y SCREEN ADDRESS

LDA ]F ; LOAD FILL CHAR

STA (GBPSH),Y ; STORE AT SCREEN ADDRESS

\*

\*\* PLOT XC-X,YC-Y

\*

LDA ]XC ; LOAD CIRCLE CENTER XPOS

CLC ; CLEAR CARRY

ADC ]XT ; ADD NEGATED CURRENT XPOS

TAY ; TRANSFER TO .Y

TAX ; AND .X

LDA ]YC ; LOAD CIRCLE CENTER YPOS

CLC ; CLEAR CARRY

ADC ]YT ; ADD NEGATED CURRENT YPOS

JSR GBCALC ; GET X,Y SCREEN ADDRESS

LDA ]F ; LOAD FILL CHARACTER

STA (GBPSH),Y ; STORE AT SCREEN ADDRESS

\*

\*\* PLOT XC+X,YC-Y

\*

LDA ]XC ; LOAD CIRCLE CENTER XPOS

CLC ; CLEAR CARRY

ADC ]X ; ADD CURRENT XPOS

TAY ; TRANSFER TO .Y

TAX ; AND .X

LDA ]YC ; LOAD CIRCLE CENTER YPOS

CLC ; CLEAR CARRY

ADC ]YT ; ADD NEGATE CURRENT YPOS

JSR GBCALC ; GET X,Y SCREEN ADDRESS

LDA ]F ; LOAD FILL CHAR

STA (GBPSH),Y ; STORE AT SCREEN ADDRESS

\*

\*\* PLOT XC+Y,YC+X

\*

LDA ]XC ; LOAD CIRCLE CENTER XPOS

CLC ; CLEAR CARRY

ADC ]Y ; ADD CURRENT YPOS

TAX ; TRANSFER TO .X

TAY ; AND .Y

LDA ]YC ; LOAD CIRCLE CENTER YPOS

CLC ; CLEAR CARRY

ADC ]X ; ADD CURRENT XPOS

JSR GBCALC ; GET X,Y SCREEN ADDRESS

LDA ]F ; LOAD FILL CHAR

STA (GBPSH),Y ; STORE AT SCREEN ADDRESS

\*

\*\* PLOT XC-Y,YC+X

\*

LDA ]XC ; LOAD CIRCLE CENTER XPOS

CLC ; CLEAR CARRY

ADC ]YT ; ADD NEGATED CURRENT YPOS

TAX ; TRANSFER TO .X

TAY ; AND .Y

LDA ]YC ; LOAD CIRCLE CENTER YPOS

CLC ; CLEAR CARRY

ADC ]X ; ADD CURRENT XPOS

JSR GBCALC ; GET X,Y SCREEN ADDRESS

LDA ]F ; LOAD FILL CHAR

STA (GBPSH),Y ; STORE AT SCREEN ADDRESS

\*

\*\* PLOT XC-Y,YC-X

\*

LDA ]XC ; LOAD CIRCLE CENTER XPOS

CLC ; CLEAR CARRY

ADC ]YT ; ADD NEGATED CURRENT YPOS

TAX ; TRANSFER TO .X

TAY ; AND .Y

LDA ]YC ; LOAD CIRCLE CENTER YPOS

CLC ; CLEAR CARRY

ADC ]XT ; ADD NEGATED CURRENT XPOS

JSR GBCALC ; GET X,Y SCREEN ADDRESS

LDA ]F ; LOAD FILL CHAR

STA (GBPSH),Y ; STORE AT SCREEN ADDRESS

\*

\*\* PLOT XC+Y,YC-X

\*

LDA ]XC ; LOAD CIRCLE CENTER XPOS

CLC ; CLEAR CARRY

ADC ]Y ; ADD CURRENT YPOS

TAY ; TRANSFER TO .Y

TAX ; AND .X

LDA ]YC ; LOAD CIRCLE CENTER YPOS

CLC

ADC ]XT ; ADD NEGATED CURRENT XPOS

JSR GBCALC ; GET X,Y SCREEN ADDRESS

LDA ]F ; LOAD FILL CHAR

STA (GBPSH),Y ; STORE AT SCREEN ADDRESS

JMP :LOOP ; LOOP UNTIL FINISHED

:EXIT

RTS

|  |
| --- |
| **TVLINE (sub)**  **Input**:  **WPAR1** = X position  **WPAR2** = Y origin  **WPAR2**+1 = Y destination  **BPAR1** = fill character  **Output**:  Vertical line to screen  **Destroys**: AXYNVZCM  **Cycles**: 33+  **Size**: 34bytes |

**SUB.TVLINE >> TVLINE**

The **TVLINE** subroutine creates a text vertical line on the screen at the given row from a passed Y origin and Y destination. The line is plotted directly to screen memory.

\*

\*``````````````````````````````\*

\* TVLINE (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* ]X1 STORED AT WPAR1 \*

\* ]Y1 STORED AT WPAR2 \*

\* ]Y2 STORED AT WPAR2+1 \*

\* ]F STORED AT BPAR1 \*

\* \*

\* OUTPUT: VERT LINE TO SCREEN \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 33+ \*

\* SIZE: 34 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]X1 EQU WPAR1 ; 1 BYTE

]Y1 EQU WPAR2 ; 1 BYTE

]Y2 EQU WPAR2+1 ; 1 BYTE

]F EQU BPAR1 ; 1 BYTE

\*

TVLINE

\*

LDA ]Y1

LDY ]X1

:LOOP

JSR GBCALC ; GET POS SCREEN ADDRESS

LDA ]F

STA (GBPSH),Y ; PLOT TO SCREEN MEMORY

INC ]Y1

LDA ]Y1

CMP ]Y2 ; IF Y1 < Y2

BNE :LOOP ; LOOP; ELSE, CONTINUE

:EXIT

RTS

|  |
| --- |
| **TRECTF (sub)**  **Input**:  **WPAR1** = X origin  **WPAR2** = Y origin  **BPAR1** = fill character  **WPAR1**+1 = X destination  **WPAR2**+1 = Y destination  **Output**:  Filled rectangle to the  screen  **Destroys**: AXYNVZCM  **Cycles**: 69+  **Size**: 74 bytes |

**SUB.TRECTF >> TRECTF**

The **TRECTF** subroutine draws a rectangle filled with the given character at the provided X,Y coordinate. The rectangle is drawn directly to screen memory, bypassing **COUT**.

\*

\*``````````````````````````````\*

\* TRECTF (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = X ORIGIN \*

\* WPAR1+1 = X DESTINATION \*

\* WPAR2 = Y ORIGIN \*

\* WPAR2+1 = Y DESTINATION \*

\* BPAR1 = FILL CHARACTER \*

\* \*

\* OUTPUT \*

\* \*

\* FILLED RECTANGLE TO SCREEN \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 69+ \*

\* SIZE: 74 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]X1 EQU WPAR1 ; 1 BYTE

]X2 EQU WPAR1+1 ; 1 BYTE

]Y1 EQU WPAR2 ; 1 BYTE

]Y2 EQU WPAR2+1 ; 1 BYTE

]F EQU BPAR1 ; 1 BYTE

\*

]XC EQU VARTAB ; 1 BYTE

]YC EQU VARTAB+1 ; 1 BYTE

\*

TRECTF

LDA ]X1

STA ]XC

LDA ]Y1

STA ]YC

:LP1 ; PRINT HORIZONTAL LINE

LDA ]YC

LDY ]XC

JSR GBCALC ; GET SCREEN MEMORY ADDR

LDA ]F ; OF CURRENT POSITION

STA (GBPSH),Y ; PUT CHAR IN LOCATION

LDA ]YC

INY ; INCREASE XPOS

STY ]XC

CPY ]X2 ; IF XPOS < XMAX,

BNE :LP1 ; KEEP PRINTING LINE

\*

LDA ]X1 ; OTHERWISE, RESET XPOS

STA ]XC

INC ]YC ; AND INCREASE YPOS

LDA ]YC

CMP ]Y2 ; IF YPOS < YMAX

BNE :LP1 ; PRINT HORIZONTAL LINE

:EXIT

RTS

|  |
| --- |
| **TXTPUT (sub)**  **Input**:  **.A** = fill character  **.X** = X position  **.Y** = Y position  **Output**:  Character to screen  **Destroys**: AXYNVZC  **Cycles**: 29+  **Size**: 30 bytes |

**SUB.TXTPUT >> TXTPUT**

The **TXTPUT** subroutine plots a given character to the screen, directly placing the value in screen memory.

\*

\*``````````````````````````````\*

\* TXTPUT (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* .A = FILL CHAR \*

\* .X = X POSITION \*

\* .Y = Y POSITION \*

\* \*

\* OUTPUT \*

\* \*

\* CHAR TO SCREEN AT X,Y \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 29+ \*

\* SIZE: 30 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]Y1 EQU VARTAB ; 1 BYTE

]X1 EQU VARTAB+1 ; 1 BYTE

]F EQU VARTAB+3 ; 1 BYTE

CYC ON

\*

TXTPUT

\*

STA ]F ; GET FILL CHAR

STY ]Y1 ; GET Y POS

STX ]X1 ; GET XPOS

\*

LDA ]Y1

LDY ]X1

JSR GBCALC ; GET SCREEN ADDRESS

LDA ]F

STA (GBPSH),Y ; PUSH CHAR TO SCREEN ADDR

:EXIT

RTS

|  |
| --- |
| **PRNSTR (sub)**  **Input**:  **.A** = address lobyte  **.X** = address hibyte  **Output**:  Print string to screen  **Destroys**: AXYNVZC  **Cycles**: 28+  **Size**: 22 bytes |

**SUB.PRNSTR >> PRNSTR**

The **PRNSTR** subroutine prints a string to the screen that is preceded by a single length byte; once that length is reached in the loop, no more characters are printed.

\*``````````````````````````````\*

\* PRNSTR (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* .A = ADDRESS LOBYTE \*

\* .X = ADDRESS HIBYTE \*

\* \*

\* OUTPUT: \*

\* \*

\* PRINTS STRING TO SCREEN. \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 28+ \*

\* SIZE: 22 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]STRLEN EQU VARTAB ; 1 BYTE

\*

PRNSTR

\*

STA ADDR1

STX ADDR1+1

\*

LDY #0

LDA (ADDR1),Y ; GET STRING LENGTH

STA ]STRLEN

:LP

INY

LDA (ADDR1),Y ; GET CHARACTER

JSR COUT1 ; PRINT CHARACTER TO SCREEN

CPY ]STRLEN ; IF Y < LENGTH

BNE :LP

; LOOP; ELSE

LDY #0

LDA (ADDR1),Y

RTS

**DEMO.STDIO**

DEMO.STDIO contains brief showcases and samples of the various macros related to standard input and output. These are by no means complicated implementations; for more rigorous use, see the integrated demos.

\*

\*``````````````````````````````\*

\* DEMO.STDIO \*

\* \*

\* A DEMO OF THE MACROS AND \*

\* SUBROUTINES IN THE STDIO \*

\* APPLEIIASM LIBRARY. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 07-JUL-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\*\* ASSEMBLER DIRECTIVES

\*

CYC AVE

EXP OFF

TR ON

DSK DEMO.STDIO

OBJ $BFE0

ORG $6000

\*

\*``````````````````````````````\*

\* TOP INCLUDES (HOOKS,MACROS) \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PUT MIN.HEAD.REQUIRED

USE MIN.MAC.REQUIRED

USE MIN.MAC.STDIO

PUT MIN.HOOKS.STDIO

\*

\*``````````````````````````````\*

\* PROGRAM MAIN BODY \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

JSR HOME ; CLEAR SCREEN

\*

PRN "STDIO DEMO",8D

PRN "----------",8D8D

PRN "WELCOME! THIS IS A DEMO FOR",8D

PRN "THE STDIO LIBRARY MACROS AND ",8D

PRN "SUBROUTINES.",8D8D

WAIT

PRN "OUR FIRST OBVIOUS MACRO USED",8D

PRN "IS PRN. THIS MACRO CAN PRINT A",8D

PRN "GIVEN STRING, OR PRINT THE STRING",8D

PRN "AT A GIVEN ADDRESS THAT IS REFERENCED",8D

PRN "EITHER DIRECTLY (#) OR INDIRECTLY.",8D

PRN "THEREFORE: ",8D8D

WAIT

PRN " PRN 'HELLO!'",8D8D

PRN "PRINTS HELLO, WHEREAS",8D8D

PRN " PRN #STRING1",8D8D

PRN "PRINTS THE STRING LOCATED AT",8D

PRN "THAT EXACT ADDRESS."

WAIT

JSR HOME

PRN "MEANWHILE,",8D8D

PRN " PRN STRING2",8D8D

PRN "PRINTS THE STRING AT THE ADDRESS PASSED",8D

PRN "VIA THAT MEMORY LOCATION.",8D8D

WAIT

PRN "IT IS IMPORTANT TO NOTE THAT",8D

PRN "WHENEVER THERE IS AN OPTION FOR",8D

PRN "EITHER A STRING OR A MEMORY ADDRESS,",8D

PRN "THIS IS HOW ALL SUBROUTINES WORK IN",8D

PRN "THIS LIBRARY. IN OTHER DEMOS, IT MAY",8D

PRN "BE ASSUMED THAT THE READER KNOWS THIS."

WAIT

JSR HOME

PRN "OUR NEXT SUBROUTINE NEEDING ",8D

PRN "OUR ATTENTION IS CALLED BY THE",8D

PRN "COL40 MACRO. THIS FORCES USING",8D

PRN "40-COLUMN MODE, AND IS ESPECIALLY",8D

PRN "NECESSARY FOR ROUTINES THAT PRINT",8D

PRN "DIRECTLY TO SCREEN MEMORY INSTEAD",8D

PRN "OF USING COUT ROUTINES. SO,"8D8D

PRN " COL40",8D8D

PRN "WILL PUT US IN 40-COLUMN MODE",8D

PRN "AFTER HITTING A KEY NOW."

WAIT

COL40

JSR HOME

PRN "YOU CAN ALSO FORCE 80-COLUMN MODE",8D

PRN "WITH THE COL80 MACRO, BUT BE",8D

PRN "AWARE THAT TRECF,TPUT,THLIN",8D

PRN "AND TVLIN WILL ONLY WORK",8D

PRN "AS INTENDED IN 40 COLUMNS.",8D8D

PRN "LET'S LOOK AT THESE MACROS NOW."

WAIT

JSR HOME

PRN "ASCII DRAWING",8D

PRN "=============",8D8D

PRN "AT TIMES, YOU MAY NEED TO ",8D

PRN "PUT A BLOCK OF TEXT THAT CONSISTS",8D

PRN "OF A SINGLE CHARACTER AS QUICKLY",8D

PRN "AS POSSIBLE. CURRENTLY, THERE ARE",8D

PRN "FOUR MACROS DEDICATED TO JUST ",8D

PRN "THAT: THLIN, TVLIN, TRECF, AND TPUT.",8D8D

WAIT

PRN "THE SIMPLEST OF THESE IS TPUT:",8D

PRN "IT OUTPUTS A SINGLE CHARACTER AT",8D

PRN "THE GIVEN XY COORDINATES. SO,",8D8D

PRN " TPUT #38;#20;#'$'",8D8D

PRN "WILL PLACE THE '$' CHARACTER",8D

PRN "AT THE X-POSITION 38 AND Y-POSITION",8D

PRN "20. LET'S TRY THAT NOW...",8D8D

WAIT

TPUT #38;#20;#"$"

PRN "SEE? RIGHT OVER HERE -->"

WAIT

JSR HOME

PRN "NOT THAT THE CURSOR'S POSITION",8D

PRN "IS NOT DISTURBED BY TPUT; THIS",8D

PRN "IS DUE TO THE FACT THAT THE ROUTINE",8D

PRN "BYPASSES COUT AND INSTEAD DIRECTLY",8D

PRN "POKES THE CHARACTER INTO SCREEN MEMORY.",8D

PRN "THIS IS PRIMARILY FOR SPEED, BUT AGAIN",8D

PRN "KEEP IN MIND THAT THIS DOES NOT WORK",8D

PRN "CORRECTLY IN 80-COLUMN MODE.",8D8D

WAIT

PRN "THLIN, TVLIN, AND TRECF OPERATE IN",8D

PRN "THE SAME WAY. LET'S LOOK AT THOSE NEXT."

TPUT #38;#12;#"K"

TPUT #38;#13;#"E"

TPUT #38;#14;#"E"

TPUT #38;#15;#"P"

TPUT #38;#17;#"G"

TPUT #38;#18;#"O"

TPUT #38;#19;#"I"

TPUT #38;#20;#"N"

TPUT #38;#21;#"G"

WAIT

JSR HOME

PRN "THLIN AND TVLIN BOTH CREATE LINES",8D

PRN "FROM A SINGLE CHARACTER, HORIZONTALLY",8D

PRN "AND VERTICALLY RESPECTIVELY. THUS",8D8D

PRN " THLIN #25;#35;#20;#'X'",8D8D

WAIT

THLIN #25;#35;#20;#"X"

PRN "CREATES A HORIZONTAL LINE FROM THE",8D

PRN "X-POSITION 25 TO 35 AT THE Y-POSITION",8D

PRN "OF 20 WITH THE CHARACTER 'X'. LIKEWISE,",8D8D

PRN " TVLIN #10;#20;#35;#'Y'",8D8D

WAIT

TVLIN #10;#20;#35;#"Y"

PRN "CREATES A VERTICAL LINE FROM Y-POSITION",8D

PRN "10 TO 20 AT THE X-POSITION 35."

WAIT

JSR HOME

PRN "NOTE THAT THE LAST POSITION GIVEN",8D

PRN "IS NOT ACTUALLY FILLED. THIS IS",8D

PRN "TO KEEP PLACEMENT MORE INTUITIVE.",8D

PRN "HOWEVER, WHEN TRYING TO ARRANGE LINES",8D

PRN "CONNECTED TOGETHER, YOU WILL HAVE TO",8D

PRN "ADJUST YOUR NUMBERS ACCORDINGLY. TO",8D

PRN "CREATE A BOX, FOR INSTANCE, YOU WOULD",8D

PRN "NEED TO WRITE:",8D8D

PRN " THLIN #25;#35;#20;#'X'",8D

PRN " TVLIN #10;#20;#34;#'X'",8D

PRN " TVLIN #10;#20;#25;#'X'",8D

PRN " THLIN #25;#35;#10;#'X'",8D8D

WAIT

THLIN #25;#35;#20;#"X"

TVLIN #10;#20;#34;#"X"

TVLIN #10;#20;#25;#"X"

THLIN #25;#35;#10;#"X"

PRN "YAY!"

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

WAIT

JSR HOME

PRN "THE TLINE MACRO DRAWS A LINE FROM",8D

PRN "X1,Y1 TO X2,Y2 WITH A FILL CHARACTER.",8D

PRN "USE TVLIN OR THLINE IF YOU ARE",8D

PRN "DRAWING HORIZONTAL OR VERTICAL LINES,",8D

PRN "AS THESE USE FEWER CYCLES.",8D8D

PRN " TLINE #20;#12;#30;#22;#'\*'",8D

PRN " TLINE #30;#22;#10;#15;#'\*'",8D

PRN " TLINE #10;#15;#30;#15;'\*'",8D

PRN " TLINE #30;#15;#10;#22;#'\*'",8D

PRN " TLINE #10;#22;#20;#12;#'\*'",8D8D

PRN "WILL OUTPUT:"

WAIT

TLINE #20;#12;#30;#22;#"\*"

TLINE #30;#22;#10;#15;#"\*"

TLINE #10;#15;#30;#15;#"\*"

TLINE #30;#15;#10;#22;#"\*"

TLINE #10;#22;#20;#12;#"\*"

WAIT

JSR HOME

PRN "YOU CAN ALSO CREATE CIRCLES WITH",8D

PRN "THE TCIRC MACRO. IN THE PARAMS,",8D

PRN "YOU SPECIFY THE X POSITION OF THE",8D

PRN "CENTER, THE Y POSITION OF IT, ",8D

PRN " THE CIRCLE'S RADIUS, AND THE ",8D

PRN "FILL CHAR OF THE CIRCLE'S OUTLINE.",8D

PRN "THUS:",8D8D

PRN "TCIRC #30;#14;#7;#'\*'",8D

PRN "TCIRC #30;#14;#6;#'.'",8D

PRN "TCIRC #30;#14;#5;#'#'",8D

PRN "TCIRC #30;#14;#4;#':'",8D

PRN "TCIRC #30;#14;#3;#'@'",8D

PRN "TCIRC #30;#14;#2;#'+'",8D8D

PRN "WILL PRODUCE:"

WAIT

TCIRC #30;#14;#7;#"\*"

TCIRC #30;#14;#6;#"."

TCIRC #30;#14;#5;#"#"

TCIRC #30;#14;#4;#":"

TCIRC #30;#14;#3;#"@"

TCIRC #30;#14;#2;#"+"

WAIT

JSR HOME

PRN "THE LAST OF THESE KIND OF MACROS",8D

PRN "IS TRECF, WHICH CREATES A FILLED",8D

PRN "BOX. THIS CAN BE ESPECIALLY USEFUL",8D

PRN "FOR CREATING A SEMBLANCE OF 'WINDOWS'",8D

PRN "ON THE TEXT SCREEN. SO:",8D8D

PRN " TRECF #10;#10;#20;#20;#'#'",8D8D

PRN "WILL RESULT IN:",8D8D

WAIT

TRECF #10;#10;#20;#20;#"#"

PRN "W00T!"

WAIT

JSR HOME

PRN "CURSOR POSITIONING",8D

PRN "==================",8D8D

PRN "THE REST OF THESE ROUTINES",8D

PRN "USE COUT1 FOUR CONVENIENCE AND",8D

PRN "SAVING A FEW BYTES HERE AND THERE.",8D

PRN "THIS MEANS, AMONG OTHER THINGS, THAT",8D

PRN "THE SYSTEM MONITOR KEEPS TRACK",8D

PRN "OF OUR CURSOR POSITION, AND WE CAN",8D

PRN "CALL ITS ROUTINES TO ALTER SAID",8D

PRN "POSITION. THIS IS ACHIEVED WITH THE",8D

PRN "FOLLOWING MACROS, WHICH WE WILL EXPLORE",8D

PRN "NEXT:",8D8D

PRN " SETCX SETCY",8D

PRN " SCPOS RCPOS",8D

PRN " CURF CURB",8D

PRN " CURU CURD"

WAIT

JSR HOME

PRN "SETCX AND SETCY SIMPLY SET THE X",8D

PRN "AND Y POSITIONS OF THE CURSOR,",8D

PRN "RESPECTIVELY. SO:",8D8D

PRN " SETCX #20",8D8D

WAIT

SETCX #20

PRN "SETS THE CURSOR'S",8D

PRN "X-POSITION TO 20, WHEREAS",8D8D

PRN " SETCY #20",8D8D

WAIT

SETCY #20

PRN "SET'S THE Y-POSITION TO 20."

WAIT

JSR HOME

PRN "YOU CAN SET THESE COORDINATES",8D

PRN "AT ONCE WITH THE SCPOS MACRO. SO:",8D8D

PRN " SCPOS #8;#10"

WAIT

SCPOS #8;#10

PRN "SETS THE CURSOR AT X POSITION",8D

PRN "OF 8 AND A Y POSITION OF 10.",8D8D

WAIT

PRN "YOU CAN ALSO READ THE CHARACTER",8D

PRN "AT A GIVEN POSITION WITH THE ",8D

PRN "RCPOS MACRO. THUS,",8D8D

PRN " RCPOS #8;#10 "

WAIT

PRN "RETURNS: "

RCPOS #8;#10

JSR COUT1

WAIT

JSR HOME

PRN "THE LAST OF THE CURSOR POSITIONING",8D

PRN "MACROS ARE CURF, CURB, CURD AND CURU.",8D

PRN "THESE ALL MOVE THE CURSOR RELATIVE",8D

PRN "TO ITS CURRENT POSITION. CURF MOVES",8D

PRN "IT FORWARD BY THE SPECIFIED AMOUNT,",8D

PRN "CURB MOVES BACKWARDS, CURD MOVES",8D

PRN "DOWN AND CURU MOVES UP. THUS:",8D8D

PRN " CURF #5 ",8D8D

PRN "MOVES THE CURSOR "

WAIT

CURF #5

PRN "FORWARD BY FIVE.",8D8D

PRN "THE OTHER MACROS USE THE SAME",8D

PRN "SYNTAX."

WAIT

JSR HOME

PRN "MOUSETEXT",8D

PRN "=========",8D8D

PRN "ON CAPABLE SYSTEMS, MOUSETEXT",8D

PRN "CAN BE TURNED ON WITH THE",8D

PRN "MTXT1 MACRO AND TURNED OFF WITH",8D

PRN "THE MTXT0 MACRO. SINCE THIS",8D

PRN "WON'T HAVE A DEMO OF IT HERE."

WAIT

JSR HOME

PRN "INPUT MACROS",8D

PRN "============",8D8D

PRN "CURRENTLY, THIS STDIO LIBRARY",8D

PRN "CONTAINS FIVE MACROS FOR USER",8D

PRN "INPUT. THEY ARE AS FOLLOWS:",8D8D

PRN " INP STRING INPUT",8D

PRN " GKEY CHARACTER INPUT",8D

PRN " PDL PADDLE INPUT",8D

PRN " PBX PADDLE BUTTON INPUT",8D

PRN " WAIT CHARACTER INPUT, NO MONITOR"

WAIT

JSR HOME

PRN "WE HAVE ALREADY MADE SUBSTANTIAL",8D

PRN "USE OF THE WAIT MACRO--THAT'S ",8D

PRN "WHAT IS CALLED EVERY TIME THIS",8D

PRN "DEMO PAUSES. ONCE A KEY IS PRESSED,",8D

PRN "THE ASCII CODE FOR IT IS STORED",8D

PRN "IN THE .A REGISTER. THIS MACRO",8D

PRN "ACCEPTS NO PARAMETERS.",8D8D

PRN "A SPECIAL FEATURE OF THE WAIT",8D

PRN "MACRO IS THAT IT DOES NOT USE THE",8D

PRN "TYPICAL MONITOR ROUTINES FOR INPUT,",8D

PRN "AND READS THE KEYBOARD DIRECTLY,",8D

PRN "ALLOWING US TO NOT HAVE A CURSOR ON",8D

PRN "THE SCREEN, AMONG OTHER BENEFITS.",8D

PRN "THIS IS IN CONTRAST TO GKEY, WHICH",8D

PRN "USES THE MONITOR ROUTINE TO ACHIEVE",8D

PRN "THE SAME RESULT: "

GKEY

JSR HOME

PRN "THE INP MACRO SIMILARLY USES THE",8D

PRN "MONITOR'S INPUT ROUTINE. THIS MEANS",8D

PRN "THAT IT SUFFERS THE SAME PROBLEMS",8D

PRN "AS DOES APPLESOFT BASIC'S INPUT",8D

PRN "COMMAND: COMMAS AND SPECIAL CHARACTERS",8D

PRN "COMPLICATE MATTERS. IN FUTURE PATCHES,",8D

PRN "AN ALTERNATE NON-MONITOR ROUTINE",8D

PRN "WILL BECOME AVAILABLE.",8D8D

PRN "TYPE SOMETHING AND PRESS RETURN:",8D

INP

PRN " ",8D

PRN "YOU CAN THEN PRINT THE STRING TO ",8D

PRN "SCREEN USING THE SPRN MACRO:",8D8D

PRN "YOU TYPED:"

SPRN #RETURN

WAIT

JSR HOME

PRN "PADDLE BUTTONS CAN BE READ VIA",8D

PRN "THE PBX MACRO. THE SYNAX IS AS",8D

PRN "FOLLOWS:",8D8D

PRN " PBX [BUTTON ADDRESS]",8D8D

WAIT

PRN "THE HOOKS.STDIO FILE CONTAINS THE",8D

PRN "ADDRESSES FOR THE FOR PADDLE BUTTONS,",8D

PRN "CONVENIENTLY CALLED PB0, PB1, PB2, ",8D

PRN "AND PB3. THUS:",8D8D

WAIT

PRN " PBX #PB0",8D8D

PRN "CHECKS IF PADDLE BUTTON 0 IS PRESSED,",8D

PRN "AND RETURNS 1 IN THE .A REGISTER IF SO.",8D

PRN "OTHERWISE, A ZERO IS RETURNED.",8D8D

WAIT

PRN "SINCE THIS REQUIRES SPECIAL HARDWARE,",8D

PRN "WE WON'T BE USING THE MACRO HERE. NOTE",8D

PRN "THAT ON A ][E, //C, AND ][GS, THE OPEN",8D

PRN "APPLE KEY IS MAPPED TO BUTTON ZERO."

WAIT

JSR HOME

PRN "LASTLY, THE PREAD MACRO READS THE STATE",8D

PRN "OF THE GIVEN PADDLE'S POTENTIOMETER.",8D

PRN "A VALUE OF 0-255 IS RETURNED IN THE .Y",8D

PRN "REGISTER. SO:",8D8D

WAIT

PRN " PREAD #0",8D8D

PRN "WILL READ THE STATE OF PADDLE 0, WHICH",8D

PRN "IS THE MOST COMMON TO READ. AGAIN,",8D

PRN "DUE TO A NEED FOR SPECIAL HARDWARE, WE",8D

PRN "WON'T BE ILLUSTRATING IT HERE."

WAIT

JSR HOME

PRN " ",8D

PRN "THAT'S ALL, FOLKS!",8D8D

\*

JMP REENTRY

\*

\*``````````````````````````````\*

\* BOTTOM INCLUDES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PUT MIN.LIB.REQUIRED

\*

\*\* INDIVIDUAL SUBROUTINE INCLUDES

\*

\* STDIO SUBROUTINES

\*

PUT MIN.SUB.XPRINT

PUT MIN.SUB.DPRINT

PUT MIN.SUB.THLINE

PUT MIN.SUB.TVLINE

PUT MIN.SUB.TRECTF

PUT MIN.SUB.TXTPUT

PUT MIN.SUB.TBLINE

PUT MIN.SUB.TCIRCLE

PUT MIN.SUB.SINPUT

PUT MIN.SUB.PRNSTR

\*

**Disk 3: ARRAYS**

The third disk in the library contains macros and subroutines for handling arrays. These arrays can be either 8-bit, meaning they can hold 255 elements in a single dimension, or 16-bit, meaning they can hold 65,025 elements in a single dimension. Additionally, the arrays can come in one dimension or two dimensions. Regardless of the type, all array elements have a maximum length of 255 bytes.

It should always be remembered that the subroutines for each type of array will only work with the type of array assigned; otherwise, garbage will result. The subroutines and macros can be recognized for the array type by the ending number: 82 means an 8-bit, two-dimensional array, whereas 161 would denote a 16-bit, one-dimensional array.

Beyond the required files and some utilities, this disk contains the following components:

* HOOKS.ARRAYS
* MAC.ARRAYS
* SUB.ADIM81
* SUB.AGET81
* SUB.APUT81
* SUB.ADIM82
* SUB.AGET82
* SUB.APUT82
* SUB.ADIM161
* SUB.AGET161
* SUB.APUT161
* SUB.ADIM162
* SUB.AGET162
* SUB.APUT162

**HOOKS.ARRAYS**

The HOOKS.ARRAYS file contains dummy code at the moment, as there aren’t too many useful hooks for array manipulation. The dummy code is set so that the Merlin 8 Pro Assembler does not exit with a file not found error.

\*

\*``````````````````````````````\*

\* HOOKS.ARRAYS \*

\* \*

\* CURRENTLY, THIS HOOKS FILE \*

\* ONLY CONTAINS DUMMY CODE IN \*

\* ORDER TO PREVENT AN ERROR \*

\* DURING ASSEMBLY (EMPTY \*

\* FILE). \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 13-JUL-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

ARRMAX EQU 8192 ; MAXIMUM # OF BYTES

; AN ARRAY CAN USE

\*

**MAC.ARRAYS**

The MAC.ARRAYS file contains all macros in the library related to array functionality. This includes:

* ADIM81
* AGET81
* APUT81
* ADIM82
* AGET82
* APUT82
* ADIM161
* AGET161
* APUT161
* ADIM162
* AGET162
* APUT162

\*

\*``````````````````````````````\*

\* MAC.ARRAYS \*

\* \*

\* A MACRO LIBRARY FOR 8BIT AND \*

\* 16BIT ARRAYS, BOTH IN ONE \*

\* DIMENSION AND TWO DIMENSIONS \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 13-JUL-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\* \*

\* SUBROUTINE FILES USED \*

\* \*

\* SUB.ADIM161 \*

\* SUB.ADIM162 \*

\* SUB.ADIM81 \*

\* SUB.ADIM82 \*

\* SUB.AGET161 \*

\* SUB.AGET162 \*

\* SUB.AGET81 \*

\* SUB.AGET82 \*

\* SUB.APUT161 \*

\* SUB.APUT162 \*

\* SUB.APUT81 \*

\* SUB.APUT82 \*

\* \*

\* LIST OF MACROS \*

\* \*

\* DIM81: DIM 1D, 8BIT ARRAY \*

\* GET81: GET ELEMENT IN 8BIT, \*

\* 1D ARRAY. \*

\* PUT81: PUT VALUE INTO ARRAY \*

\* AT SPECIFIED INDEX \*

\* DIM82: DIM A 2D, 8BIT ARRAY \*

\* GET82: GET ELEMENT IN 8BIT, \*

\* 2D ARRAY \*

\* PUT82: PUT VALUE INTO ARRAY \*

\* AT SPECIFIED INDEX \*

\* DIM161: DIM 1D, 16BIT ARRAY \*

\* GET161: GET ELEMENT FROM 1D, \*

\* 16BIT ARRAY. \*

\* PUT161: PUT VALUE INTO A 1D, \*

\* 16BIT ARRAY INDEX. \*

\* DIM162: DIM 2D, 16BIT ARRAY \*

\* GET162: GET ELEMENT FROM 2D, \*

\* 16BIT ARRAY. \*

\* PUT162: PUT VALUE INTO A 2D, \*

\* 16BIT ARRAY INDEX. \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

|  |
| --- |
| **DIM81 (macro)**  **Input**:  ]1 = array address (2b)  ]2 = # of elements (1b)  ]3 = element length (1b)  ]4 = fill value (1b)  **Output**:  New 8-bit Array(]2)  **Destroys**: AXYNVZCM  **Cycles**: 214+  **Size**: 39 bytes |

**MAC.ARRAYS >> DIM81**

The **DIM81** macro initializes a new 8-bit, one-dimensional array at the given array address with the specified number of elements at the given length. Initially, all elements are filled with the value provided via ]4. Since this is an 8-bit array, it can hold no more than 255 elements, with each element capable of having a length between 1 and 255.

A one dimensional 8-bit array has a two-byte header where byte 0 of the array holds the number of elements in the array, while byte 1 contains the length of each element. Then the data held by the array follows.

\*``````````````````````````````\*

\* DIM81 (NATHAN RIGGS) \*

\* \*

\* CREATE A ONE DIMENSIONAL, \*

\* 8-BIT ARRAY AT THE GIVEN \*

\* ADDRESS. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = ARRAY ADDRESS \*

\* ]2 = ARRAY BYTE LENGTH \*

\* ]3 = ELEMENT BYTE LENGTH \*

\* ]4 = FILL VALUE \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* DIM81 #$300;#10;#2;#0 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

DIM81 MAC

\_MLIT ]1;WPAR1 ; PARSE IF LITERAL OR NOT

LDA ]2 ; ARRAY LENGTH

STA WPAR2

LDA ]3 ; ELEMENT LENGTH

STA WPAR3

LDA ]4

STA BPAR1 ; FILL VAL

JSR ADIM81

<<<

|  |
| --- |
| **GET81 (macro)**  **Input**:  ]1 = array address (2b)  ]2 = element index (1b)  **Output**:  **RETURN** = element value  **RETLEN** = element length  **Destroys**: AXYNVZCM  **Cycles**: 148+  **Size**: 11 bytes |

**MAC.ARRAYS >> GET81**

The **GET81** macro retrieves the value held in an 8-bit, one-dimensional array and copies it into **RETURN**. **RETLEN** holds the length of the element copied.

Note that trying to use **GET81** on an array initialized as a 16-bit array or a two-dimensional array will result in faulty data. Use the corresponding subroutines and macros for each type of array accordingly.

\*

\*``````````````````````````````\*

\* GET81 (NATHAN RIGGS) \*

\* \*

\* RETRIEVE A VALUE FROM THE \*

\* GIVEN ARRAY AT THE SPECIFIED \*

\* ELEMENT INDEX AND STORE THE \*

\* VALUE IN RETURN. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = ARRAY ADDRESS \*

\* ]2 = ELEMENT INDEX \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* GET81 #$300;#5 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

GET81 MAC

\_AXLIT ]1 ; PARSE ADDRESS

LDY ]2 ; ELEM INDEX

JSR AGET81

<<<

|  |
| --- |
| **PUT81 (macro)**  **Input**:  ]1 = source address (2b)  ]2 = array address (2b)  ]3 = element index (1b)  **Output**:  Array(]2) = ]1  **Destroys**: AXYNVZCM  **Cycles**: 240+  **Size**: 55 bytes |

**MAC.ARRAYS >> PUT81**

The **PUT81** macro puts a value stored in a given source address into an 8-bit, one-dimensional array. The length of the element is determined by addressing the array header, so special care should be taken to make sure that proper lengths are used; trash will be sent to the array element, if not.

\*

\*``````````````````````````````\*

\* PUT81 (NATHAN RIGGS) \*

\* \*

\* PUTS THE DATA FOUND AT THE \*

\* GIVEN ADDRESS INTO THE ARRAY \*

\* AT THE GIVEN INDEX. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = SOURCE ADDRESS \*

\* ]2 = ARRAY ADDRESS \*

\* ]3 = ELEMENT INDEX \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* PUT81 #$300;#$3A0;#5 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PUT81 MAC

\_MLIT ]1;WPAR1 ; PARSE SOURCE ADDRESS

\_MLIT ]2;WPAR2 ; PARSE DEST ADDRESS

LDA ]3 ; DEST INDEX

STA BPAR1

JSR APUT81

<<<

|  |
| --- |
| **DIM82 (macro)**  **Input**:  ]1 = array address (2b)  ]2 = first dim index (1b)  ]3 = 2nd dim index (1b)  ]4 = element length (1b)  ]5 = fill value (1b)  **Output**:  New 8-bit Array(]2,]3)  **Destroys**: AXYNVZCM  **Cycles**: 324+  **Size**: 43 bytes |

**MAC.ARRAYS >> DIM82**

The **DIM82** macro initializes a new 8-bit, two-dimensional array with the given number of elements for each dimension at the specified element length. Note that since this is an 8-bit array, it can hold up to 255 elements only, with each having a length of 1 to 255.

A two-dimensional 8-bit array has a three-byte header that contains vital information about the array. Byte 0 hold the number of elements in the first dimension, byte 1 holds the number of elements in the second dimension, and byte 3 holds the length of each element. The total number of elements can be derived by simply multiplying the number of elements in the first dimension by the number of elements in the 2nd dimension.

\*

\*``````````````````````````````\*

\* DIM82 (NATHAN RIGGS) \*

\* \*

\* INITIALIZES AN 8-BIT ARRAY \*

\* WITH TWO DIMENSIONS. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = ARRAY ADDRESS \*

\* ]2 = X DIMENSION \*

\* ]3 = Y DIMENSION \*

\* ]4 = ELEMENT SIZE \*

\* ]5 = FILL VALUE \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* DIM82 #$300;#4;#4;#1;#0 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

DIM82 MAC

\_MLIT ]1;WPAR1 ; PARSE ARRAY ADDRESS

LDA ]2 ; X DIM

STA WPAR2

LDA ]3 ; Y DIM

STA WPAR3

LDA ]4 ; ELEMENT LENGTH

STA BPAR2

LDA ]5 ; FILL VAL

STA BPAR1

JSR ADIM82

<<<

|  |
| --- |
| **GET82 (macro)**  **Input**:  ]1 = array address (2b)  ]2 = first dim index (1b)  ]3 = 2nd dim index (1b)  **Output**:  **RETURN** = element value  **RETLEN** = element length  **Destroys**: AXYNVZCM  **Cycles**: 322+  **Size**: 35 bytes |

**MAC.ARRAYS >> GET82**

The **GET82** macro retrieves the value held in an 8-bit, 2-dimensional array at the given index pair. This value is stored in **RETURN**, and the element length is stored in **RETLEN**.

Like with other GET and PUT macros, this only works properly with arrays initialized as the same array type as this subroutine expects; namely, it must be an 8-bit, two-dimensional array created by **DIM82**.

\*

\*``````````````````````````````\*

\* GET82 (NATHAN RIGGS) \*

\* \*

\* RETRIEVE VALUE FROM ELEMENT \*

\* OF 8-BIT, TWO DIMENSIONAL \*

\* ARRAY. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = ARRAY ADDRESS \*

\* ]2 = X INDEX \*

\* ]3 = Y INDEX \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* GET82 #$300;#2;#3 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

GET82 MAC

\_MLIT ]1;WPAR1

LDA ]2 ; X INDEX

STA BPAR1

LDA ]3 ; Y INDEX

STA BPAR2

JSR AGET82

<<<

|  |
| --- |
| **PUT82 (macro)**  **Input**:  ]1 = source address (2b)  ]2 = array address (2b)  ]3 = first dim index (1b)  ]4 = 2nd dim index (1b)  **Output**:  Array(]3,]4) = ]1  **Destroys**: AXYNVZCM  **Cycles**: 328+  **Size**: 59 bytes |

**MAC.ARRAYS >> PUT82**

The **PUT82** macro copies the value in a source address range to an element in a two-dimensional 8-bit array. Like with other PUT macros, the length of the value to be transferred is determined by the element length byte of the array; therefore, special attention should be given to the lengths of those values passed.

\*

\*``````````````````````````````\*

\* PUT82 (NATHAN RIGGS) \*

\* \*

\* SET VALUE OF AN ELEMENT IN \*

\* AN 8-BIT, TWO-DIMENSIONAL \*

\* ARRAY. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = SOURCE ADDRESS \*

\* ]2 = DEST ARRAY ADDRESS \*

\* ]3 = ELEMENT X INDEX \*

\* ]4 = Y INDEX \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* PUT82 #$300;$3A0;#2;#3 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PUT82 MAC

\_MLIT ]1;WPAR1 ; PARSE SOURCE ADDRESS

\_MLIT ]2;WPAR2 ; PARSE DEST ADDRESS

LDA ]3 ; X INDEX

STA BPAR1

LDA ]4 ; Y INDEX

STA BPAR2

JSR APUT82

<<<

|  |
| --- |
| **DIM161 (macro)**  **Input**:  ]1 = array address (2b)  ]2 = # of elements (2b)  ]3 = element length (1b)  ]4 = fill value (1b)  **Output**:  New 16-bit Array(]2)  **Destroys**: AXYNVZCM  **Cycles**: 226+  **Size**: 59 bytes |

**MAC.ARRAYS >> DIM161**

The **DIM161** macro initializes a 16-bit, one-dimensional array with the given number of elements that have the specified length each. Since this a 16-bit array, it can hold a total of 65,025 elements, with a maximum element length of 255.

Note that this can quickly get out of hand: 65,025 elements at a single byte each will already more than fill the total amount of RAM in most Apple II computers. Additionally, execution speed is significantly worse than using 8-bit arrays. As such, this should only be used when more than 255 elements are necessary.

16-bit two-dimensional arrays contain a three-byte header. Byte 0 holds the low byte of the number of elements, and byte 1 holds the high byte. Byte 3 holds the element length, with the array’s data following.

\*

\*``````````````````````````````\*

\* DIM161 (NATHAN RIGGS) \*

\* \*

\* INITIALIZE A 16-BIT ARRAY \*

\* WITH A SINGLE DIMENSION. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = ARRAY ADDRESS \*

\* ]2 = ARRAY BYTE LENGTH \*

\* ]3 = ELEMENT BYTE LENGTH \*

\* ]4 = ARRAY FILL VALUE \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* DIM161 #$300;#10;#2;#$00 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

DIM161 MAC

\_MLIT ]1;WPAR1 ; PARSE ARRAY ADDRESS

\_MLIT ]2;WPAR2 ; PARSE BYTE LENGTH

LDA ]3 ; ELEMENT LENGTH

STA WPAR3

LDA ]4 ; FILL VALUE

STA BPAR1

JSR ADIM161

<<<

|  |
| --- |
| **PUT161 (macro)**  **Input**:  ]1 = source address (2b)  ]2 = array address (2b)  ]3 = element index  **Output**:  16-bit Array(]3) = ]1  **Destroys**: AXYNVZCM  **Cycles**: 247+  **Size**: 75 bytes |

**MAC.ARRAYS >> PUT161**

The **PUT161** macro copies the value held in a given source address range to the specified element in a one-dimensional, 16-bit array. As with all array PUT macros and subroutines, the length of the values to be transferred is determined by the element length byte in the array header.

\*

\*``````````````````````````````\*

\* PUT161 (NATHAN RIGGS) \*

\* \*

\* SET THE VALUE OF AN INDEX \*

\* ELEMENT IN A 16-BIT, ONE- \*

\* DIMENSIONAL ARRAY. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = SOURCE ADDRESS \*

\* ]2 = ARRAY ADDRESS \*

\* ]3 = ELEMENT INDEX \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* PUT161 #$300;$3A0;#5 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PUT161 MAC

\_MLIT ]1;WPAR1 ; PARSE SOURCE ADDRESS

\_MLIT ]2;WPAR2 ; PARSE ARRAY ADDRESS

\_MLIT ]3;WPAR3 ; PARSE INDEX

JSR APUT161

<<<

|  |
| --- |
| **GET161 (macro)**  **Input**:  ]1 = source address (2b)  ]2 = element index (2b)  **Output**:  **RETURN** = element value  **RETLEN** = element length  **Destroys**: AXYNVZCM  **Cycles**: 172+  **Size**: 51 bytes |

**MAC.ARRAYS >> GET161**

The **GET161** macro retrieves the value at a given element index from a one-dimensional 16-bit array. This value is transferred to **RETURN**, with its length stored in **RETLEN**.

\*

\*``````````````````````````````\*

\* GET161 (NATHAN RIGGS) \*

\* \*

\* GET THE VALUE STORED IN THE \*

\* ELEMENT OF A 16-BIT, ONE- \*

\* DIMENSIONAL ARRAY. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = SOURCE ADDRESS \*

\* ]2 = ARRAY ADDRESS \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* GET161 #$3A0;#300 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

GET161 MAC

\_MLIT ]1;WPAR1 ; PARSE SOURCE ADDRESS

\_MLIT ]2;WPAR2 ; PARSE INDEX

JSR AGET161

<<<

|  |
| --- |
| **DIM162 (macro)**  **Input**:  ]1 = array address (2b)  ]2 = 1st dim length (2b)  ]3 = 2nd dim length (2b)  ]4 = element length (1b)  ]5 = fill value (1b)  **Output**:  New 16-bit Array(]2,]3)  **Destroys**: AXYNVZCM  **Cycles**: 500+  **Size**: 83 bytes |

**MAC.ARRAYS >> DIM162**

The **DIM162** macro initializes a 16-bit, two-dimensional array. Each dimension can theoretically hold 65,025 elements, but higher values are either impractical or impossible on most standard Apple II systems. Each element can be as high as 255 bytes long.

Two-dimensional 16-bit arrays have a five-byte header that defines the dimension lengths and element lengths. Byte 0 holds the low byte of the first dimension’s length, and byte 1 holds the high byte. Byte 2 holds the low byte of the second dimension’s length, and byte 3 holds the high byte likewise. Finally, byte 4 holds the length of each element, which is referred to by GET162 and PUT162.

For most purposes, 8-bit arrays should work fine, and are additionally much faster than 16-bit arrays. Use **DIM162** **only** if you need an array with two dimensions that hold more than 255 elements each.

\*

\*``````````````````````````````\*

\* DIM162 (NATHAN RIGGS) \*

\* \*

\* INITIALIZE A 16-BIT, TWO- \*

\* DIMENSIONAL ARRAY. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = ARRAY ADDRESS \*

\* ]2 = X DIMENSION \*

\* ]3 = Y DIMENSION \*

\* ]4 = ELEMENT SIZE \*

\* ]5 = FILL VALUE \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* DIM162 #$300;#4;#4;#1;#0 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

DIM162 MAC

\_MLIT ]1;WPAR3 ; PARSE ARRAY ADDRESS

\_MLIT ]2;WPAR1 ; PARSE X DIMENSION

\_MLIT ]3;WPAR2 ; PARSE Y DIMENSION

LDA ]4 ; ELEMENT LENGTH

STA BPAR1

LDA ]5 ; FILL VAL

STA BPAR2

JSR ADIM162

<<<

|  |
| --- |
| **PUT162 (macro)**  **Input**:  ]1 = source address (2b)  ]2 = array address (2b)  ]3 = 1st dim index (2b)  ]4 = 2nd dim index (2b)  **Output**:  16b Array(]3,]4) = ]1  **Destroys**: AXYNVZCM  **Cycles**: 490+  **Size**: 99 bytes |

**MAC.ARRAYS >> PUT162**

The **PUT162** macro sets the value at a given element in a 16-bit, two-dimensional array. Like other PUT macros, the length of the value being transferred is determined by the element length byte in the array header.

\*

\*``````````````````````````````\*

\* PUT162 (NATHAN RIGGS) \*

\* \*

\* SET VALUE OF AN ELEMENT IN \*

\* A 16-BIT, TWO-DIMENSIONAL \*

\* ARRAY. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = SOURCE ADDRESS \*

\* ]2 = DEST ARRAY ADDRESS \*

\* ]3 = ELEMENT X INDEX \*

\* ]4 = Y INDEX \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* PUT162 #$3A0;#280;#2 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PUT162 MAC

\_MLIT ]1;WPAR1 ; PARSE SOURCE ADDRESS

\_MLIT ]2;WPAR2 ; PARSE ARRAY ADDRESS

\_MLIT ]3;WPAR3 ; PARSE X INDEX

\_MLIT ]4;ADDR1 ; PARSE Y INDEX

JSR APUT162

<<<

|  |
| --- |
| **GET162 (macro)**  **Input**:  ]1 = array address (2b)  ]2 = 1st dim index (2b)  ]3 = 2nd dim index (2b)  **Output**:  **RETURN** = element value  **RETLEN** = element length  **Destroys**: AXYNVZCM  **Cycles**: 476+  **Size**: 75 bytes |

**MAC.ARRAYS >> GET162**

The **GET162** macro retrieves the value stored in a specified element of a 16-bit, two-dimensional array. This value is held in **RETURN**, whereas its length is stored in **RETLEN**.

\*

\*``````````````````````````````\*

\* GET162 (NATHAN RIGGS) \*

\* \*

\* GET THE VALUE STORED AT AN \*

\* ELEMENT OF A 16-BIT, TWO- \*

\* DIMENSIONAL ARRAY. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = ARRAY ADDRESS \*

\* ]2 = ELEMENT X INDEX \*

\* ]3 = Y INDEX \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* GET162 #$300;#1000;#10 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

GET162 MAC

\_MLIT ]1;WPAR1 ; PARSE ARAY ADDRESS

\_MLIT ]2;WPAR2 ; PARSE X INDEX

\_MLIT ]3;WPAR3 ; PARSE Y INDEX

JSR AGET162

<<<

\*

|  |
| --- |
| **ADIM81 (sub)**  **Input**:  **WPAR1** = array addr (2b)  **WPAR2** = # of elems (1b)  **WPAR3** = elem length (1b)  **BPAR1** = fill value (1b)  **Output**:  **RETURN** = total bytes  **RETLEN** = 2  **Destroys**: AXYNVZCM  **Cycles**: 176+  **Size**: 160 bytes |

**SUB.ADIM81 >> ADIM81**

The **ADIM81** subroutine initializes an 8-bit array with a single dimension. This means that it can hold a total of 255 elements, each with a possible maximum length of 255.

The 8-bit, single dimension array has a 2-byte header. Byte 0 holds the number of elements in the array, while byte 1 holds the element length.

\*

\*``````````````````````````````\*

\* ADIM81 (NATHAN RIGGS) \*

\* \*

\* INPUT \*

\* \*

\* WPAR1 = ARRAY ADDRESS (2B) \*

\* WPAR2 = # OF ELEMENTS \*

\* WPAR3 = LENGTH OF ELEMENTS \*

\* BPAR1 = FILL VALUE \*

\* \*

\* OUTPUT \*

\* \*

\* RETURN = TOTAL BYTES USED \*

\* RETLEN = 2 \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 176+ \*

\* SIZE: 160 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]ADDR EQU WPAR1

]ASIZE EQU WPAR2

]ESIZE EQU WPAR3

]FILL EQU BPAR1

\*

]MSIZE EQU VARTAB ; TOTAL BYTES OF ARRAY

]ASZBAK EQU VARTAB+4 ; ARRAY SIZE BACKUP

]ESZBAK EQU VARTAB+6 ; ELEMENT SIZE BACKUP

\*

ADIM81

LDA ]ESIZE

STA ]ESZBAK

LDA ]ASIZE

STA ]ASZBAK

LDA #0

STA ]ASIZE+1

STA ]ASZBAK+1

\*

\*\* MULTIPLY ARRAY SIZE BY ELEMENT SIZE

\*

LDY #0 ; RESET HIBYTE FOR MULTIPLY

TYA ; RESET LOBYTE FOR MULTIPLY

LDY ]ASIZE+1

STY SCRATCH ; SAVE HIBYTE IN SCRATCH

BEQ :ENTLP ; IF ZERO, SKIP TO LOOP

:DOADD

CLC ; ADD ASIZE TO LOBYTE

ADC ]ASIZE

TAX ; TEMPORARILY STORE IN .X

TYA ; TRANSFER HIBYTE TO .A

ADC SCRATCH ; ADD HIBYTE

TAY ; STORE BACK IN .Y

TXA ; LOAD LOBYTE IN .A AGAIN

:LP ; LOOP START

ASL ]ASIZE ; MULTIPLY ASIZE BY 2

ROL SCRATCH ; MULTIPLY HIBYTE BY 2

:ENTLP

LSR ]ESIZE ; DIVIDE ESIZE BY 2

BCS :DOADD ; IF >= LOBYTE IN .A, ADD AGAIN

BNE :LP ; OTHERWISE, RELOOP

\*

STX ]MSIZE ; STORE LOBYTE

STY ]MSIZE+1 ; STORE HIBYTE

LDA ]MSIZE ; NOW ADD TO BYTES

CLC ; TO MSIZE FOR ARRAY HEADER

ADC #2

STA ]MSIZE ; STORE LOBYTE

LDA ]MSIZE+1

ADC #0 ; CARRY FOR HIBYTE

STA ]MSIZE+1

\*

\*\* NOW CLEAR MEMORY BLOCKS

\*

LDA ]FILL ; GET FILL VALUE

LDX ]MSIZE+1 ; X = # O PAGES TO DO

BEQ :PART ; BRANCH IF HIBYTE = 0

LDY #0 ; RESET INDEX

:FULL

STA (]ADDR),Y ; FILL CURRENT BYTE

INY ; INCREMENT INDEX

BNE :FULL ; LOOP UNTIL PAGE DONE

INC ]ADDR+1 ; GO TO NEXT PAGE

DEX ; DECREMENT COUNTER

BNE :FULL ; LOOP IF PAGES LEFT

:PART

LDX ]MSIZE ; PARTIAL PAGE BYTES

BEQ :MFEXIT ; EXIT IF LOBYTE = 0

LDY #0 ; RESENT INDEX

:PARTLP

STA (]ADDR),Y ; STORE VAL

INY ; INCREMENT INDEX

DEX ; DECREMENT COUNTER

BNE :PARTLP ; LOOP UNTIL DONE

:MFEXIT

LDY #0 ; STORE NUMBER OF ELEMENTS

LDA ]ASZBAK ; INTO FIRST BYTE OF ARRAY

STA (]ADDR),Y

INY

LDA ]ESZBAK ; STORE ELEMENT SIZE INTO

STA (]ADDR),Y ; SECOND BYTE OF ARRAY

LDX ]ADDR ; GET LOBYTE OF ARRAY ADDRESS

LDY ]ADDR+1 ; AND HIBYTE TO RETURN IN .X, .Y

LDA ]ASZBAK ; RETURN NUMBER OF ELEMENTS IN .A

LDA ]MSIZE ; STORE TOTAL ARRAY SIZE

STA RETURN ; IN RETURN

LDA ]MSIZE+1

STA RETURN+1

LDA #2 ; SET RETURN LENGTH TO

STA RETLEN ; 2 BYTES

RTS

|  |
| --- |
| **AGET81 (sub)**  **Input**:  **.A** = array address  low byte  **.X** = array address  high byte  **.Y** = element index  **Output**:  **.A** = element length  **RETURN** = element value  **RETLEN** = element length  **Destroys**: AXYNVZCM  **Cycles**: 134+  **Size**: 134 bytes |

**SUB.AGET81 >> AGET81**

The **AGET81** subroutine retrieves a value from an 8-bit, single dimension array that has been created by the **ADIM81** subroutine. This value is stored in **RETURN**, with its length in **RETLEN**.

\*

\*``````````````````````````````\*

\* AGET81 (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* .A = ARRAY ADDRESS LOBYTE \*

\* .X = ARRAY ADDRESS HIBYTE \*

\* .Y = ARRAY ELEMENT INDEX \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = LENGTH OF ELEMENT \*

\* RETURN = ELEMENT VALUE \*

\* RETLEN = ELEMENT LENGTH \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 134 \*

\* SIZE: 134 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]RES EQU VARTAB ; MATH RESULTS

]IDX EQU VARTAB+2 ; ELEMENT INDEX

]ESIZE EQU VARTAB+4 ; ELEMENT SIZE

]ALEN EQU VARTAB+5 ; NUMBER OF ELEMENTS

\*

AGET81

STA ADDR1 ; .A HOLDS ARRAY ADDRESS LOBYTE

STX ADDR1+1 ; .X HOLDS ADDRESS HIBYTE

STY ]IDX ; .Y HOLDS THE INDEX

LDA #0 ; CLEAR INDEX HIBYTE

STA ]IDX+1

LDY #1 ; GET ELEMENT SIZE FROM ARRAY

LDA (ADDR1),Y ; HEADER

STA ]ESIZE

STA RETLEN ; STORE IN RETLEN

DEY ; MOVE TO BYTE 0 OF HEADER

LDA (ADDR1),Y ; GET NUMBER OF ELEMENTS

STA ]ALEN ; FROM THE ARRAY HEADER

\*

\*\* MULTIPLY INDEX BY ELEMENT SIZE, ADD 2

\*

TYA ; Y ALREADY HOLDS ZERO

STY SCRATCH ; RESET LO AND HI TO 0

BEQ :ENTLP ; IF ZERO, SKIP TO LOOP

:DOADD

CLC ; CLEAR CARRY FLAG

ADC ]IDX ; ADD INDEX LOBYTE

TAX ; TEMPORARILY STORE IN .X

TYA ; TRANSFER HIBYTE TO .A

ADC SCRATCH ; ADD HIBYTE

TAY ; STORE BACK INTO .Y

TXA ; RELOAD LOBYTE IN .A

:LP

ASL ]IDX ; MULTIPLY INDEX BY TWO

ROL SCRATCH ; ADJUST HIBYTE CARRY

:ENTLP

LSR ]ESIZE ; DIVIDE ELEMENT SIZE BY 2

BCS :DOADD ; IF >= LOBYTE IN .A, ADD AGAIN

BNE :LP

\*

STX ]IDX ; STORE LOBYTE

STY ]IDX+1 ; STORE HIBYTE

CLC ; CLEAR CARRY

LDA #2 ; ADD 2 BYTES TO INDEX

ADC ]IDX ; TO ACCOUNT FOR ARRAY HEADER

STA ]RES ; AND STORE IN RESULT

LDA #0 ; ACCOUNT FOR HIBYTE CARRY

ADC ]IDX+1

STA ]RES+1

\*

\*\* NOW ADD TO BASE ADDRESS TO GET ELEMENT ADDRESS

\*

CLC ; CLEAR CARRY FLAG

LDA ]RES ; LOAD RESULT FROM EARLIER

ADC ADDR1 ; ADD ARRAY ADDRESS LOBYTE

STA ]RES ; STORE BACK IN RESULT

LDA ]RES+1 ; LOAD PRIOR RESULT HIBYTE

ADC ADDR1+1 ; ADD ARRAY ADDRESS HIBYTE

STA ]RES+1 ; STORE BACK IN RESULT HIBYTE

\*

\*\* NOW MOVE ELEMENT DATA TO RETURN LOCATION

\*

LDY #0 ; RESENT INDEX

LDA ]RES ; LOAD ADDRESS LOBYTE

STA ADDR1 ; PUT IN ZERO PAGE POINTER

LDA ]RES+1 ; GET RESULT HIBYTE

STA ADDR1+1 ; PUT IN ZERO PAGE POINTER

:LDLOOP

LDA (ADDR1),Y ; LOAD BYTE FROM ELEMENT

STA RETURN,Y ; STORE IN RETURN

INY ; INCREASE BYTE INDEX

CPY RETLEN ; IF .Y <= ELEMENT SIZE

BCC :LDLOOP ; CONTINUE LOOPING

BEQ :LDLOOP ; KEEP LOOPING

\*

LDX ]RES ; RETURN ELEMENT ADDRESS

LDY ]RES+1 ; IN .X (LOBYTE) AND .Y (HI)

LDA RETLEN ; RETURN ELEMENT LENGTH IN .A

RTS

|  |
| --- |
| **APUT81 (sub)**  **Input**:  **WPAR1** = source addr (2b)  **WPAR2** = dest addr (2b)  **BPAR1** = array index (1b)  **Output**:  **.A** = element length  **.X** = element address  low byte  **.Y** = element address  high byte  **Destroys**: AXYNVZCM  **Cycles**: 170+  **Size**: 145 bytes |

**SUB.APUT81 >> APUT81**

The **APUT81** subroutine places the value at the specified address into an 8-bit, single-dimension array element. The length of the data is determined by the array’s element length byte. This only works with arrays created by the **ADIM81** subroutine.

\*

\*``````````````````````````````\*

\* APUT81 (NATHAN RIGGS) \*

\* \*

\* PUT DATA FROM SRC LOCATION \*

\* INTO 1D, 8BIT ARRAY AT THE \*

\* SPECIFIED ELEMENT. \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = SOURCE ADDRESS \*

\* WPAR2 = DESTINATION ADDRESS \*

\* BPAR1 = ARRAY INDEX \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = ELEMENT SIZE \*

\* .X = ELEMENT ADDRESS LOBYTE \*

\* .Y = ELEMENT ADDRESS HIBYTE \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 170+ \*

\* SIZE: 145 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]ADDRS EQU WPAR1 ; SOURCE ADDRESS

]ADDRD EQU WPAR2 ; DESTINATION

]AIDX EQU BPAR1 ; ARRAY INDEX

]SCRATCH EQU ADDR1 ; ZEROED HIBYTE

\*

]ESIZE EQU VARTAB ; ELEMENT SIZE

]ESIZEBK EQU VARTAB+1 ; ^BACKUP

]ASIZE EQU VARTAB+2 ; # OF ELEMENTS

]IDX EQU VARTAB+5 ; INDEX

]RES EQU VARTAB+7 ; MULTIPLICATION RESULT

\*

APUT81

LDA ]AIDX ; STORE IN 2 LOCATIONS

STA ]IDX ; FOR A BACKUP LATER

\*

\*\* MULTIPLY INDEX BY ELEM SIZE AND ADD 2

\*

LDY #1 ; GET ELEMENT LENGTH FROM

LDA (]ADDRD),Y ; BYTE 1 OF ARRAY

STA ]ESIZE

STA ]ESIZEBK

LDY #0 ; RESET INDEX

LDA (]ADDRD),Y ; GET NUMBER OF ELEMENTS

STA ]ASIZE ; FROM ARRAY

TYA ; .A = 0

STY ]SCRATCH ; LOBYTE = 0

STY ]SCRATCH+1 ; HIBYTE = 0

BEQ :ENTLPA ; IF 0, SKIP TO LOOP

:DOADD

CLC ; CLEAR CARRY FLAG

ADC ]AIDX ; ADD INDEX LOBYTE

TAX ; TEMPORARILY STORE IN .X

TYA ; TRANSFER HIBYTE TO .A

ADC ]SCRATCH ; ADD HIBYTE

TAY ; STORE BACK IN .Y

TXA ; RELOAD LOBYTE TO .A

:LPA

ASL ]AIDX ; MUL INDEX BY TWO

ROL ]SCRATCH ; ADJUST HIBYTE CARRY

:ENTLPA

LSR ]ESIZE ; DIVIDE ELEMENT SIZE BY 2

BCS :DOADD ; IF >= LOBYTE IN .A, ADD AGAIN

BNE :LPA

STX ]IDX ; STORE LOBYTE

STY ]IDX+1 ; STORE HIBYTE

CLC ; CLEAR CARRY FLAG

LDA #2 ; ADD 2 BYTES TO INDEX

ADC ]IDX ; TO ACCOUNT FOR HEADER

STA ]RES ; STORE LOBYTE

LDA #0 ; ACCOUNT FOR HIBYTE CARRY

ADC ]IDX+1

STA ]RES+1

\*

\*\* ADD RESULT TO ARRAY ADDRESS TO GET ELEMENT ADDR

\*

CLC ; CLEAR CARRY FLAG

LDA ]RES ; LOAD RESULT FROM EARLIER

ADC ]ADDRD ; ADD ARRAY ADDRESS LOBYTE

STA ]RES ; STORE BACK IN RESULT

LDA ]RES+1 ; ADD ARRAY ADDRESS HIBYTE

ADC ]ADDRD+1 ;

STA ]RES+1 ; STORE HIBYTE

\*

STA ]ADDRD+1 ; STORE IN ZERO PAGE HIBYTE

LDA ]RES ; STORE LOBYTE TO ZERO PAGE

STA ]ADDRD

\*

\*\* COPY FROM SRC ADDR3 TO ELEMENT LOCATION ADDR

\*

:LP

LDA (]ADDRS),Y ; LOAD BYTE FROM SOURCE

STA (]ADDRD),Y ; STORE IN ELEMENT ADDRESS

INY ; INCREASE BYTE INDEX

CPY ]ESIZEBK ; COMPARE TO ELEMENT SIZE

BNE :LP ; IF !=, KEEP COPYING

\*

LDY ]ADDRD+1 ; .Y = ELEMENT ADDRESS HIBYTE

LDX ]ADDRD ; .X = LOBYTE

LDA ]ESIZE ; .A = ELEMENT SIZE

RTS

|  |
| --- |
| **ADIM82 (sub)**  **Input**:  **WPAR1** = array address  (2b)  **WPAR2** = first dimension  length (1b)  **WPAR3** = second dimension  Length (1b)  **BPAR1** = fill value (1b)  **BPAR2** = element length  (2b)  **Output**:  **.A** = element size  **RETURN** = total array size  **RETLEN** = 4  **Destroys**: AXYNVZCM  **Cycles**: 282+  **Size**: 244 bytes |

**SUB.ADIM82 >> ADIM82**

The **ADIM82** subroutine initializes an 8-bit, two-dimensional array. Each dimension can carry a maximum of 255 elements, with a total of 65,025 single elements (multiplied). Each element can be a maximum of 255 bytes long.

An 8-bit, two-dimensional array has a 3-byte header. Byte 0 contains the number of indices of the first dimension, and byte 1 holds the number of indices in the second dimension. The third byte holds the element length.

\*

\*``````````````````````````````\*

\* ADIM82 (NATHAN RIGGS) \*

\* \*

\* INITIALIZE AN 8BIT, 2D ARRAY \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = ARRAY ADDRESS \*

\* WPAR2 = 1ST DIM LENGTH \*

\* WPAR3 = 2ND DIM LENGTH \*

\* BPAR1 = FILL VALUE \*

\* BPAR2 = ELEMENT LENGTH \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = ELEMENT SIZE \*

\* RETURN = TOTAL ARRAY SIZE \*

\* RETLEN = 4 \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 282+ \*

\* SIZE: 244 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]ADDR EQU WPAR1 ; ARRAY ADDRESS

]AXSIZE EQU WPAR2 ; FIRST DIM # OF ELEMENTS

]AYSIZE EQU WPAR3 ; SECOND DIM # OF ELEMENTS

]FILL EQU BPAR1 ; FILL VALUE

]ESIZE EQU BPAR2 ; ELEMENT SIZE

\*

]PROD EQU VARTAB ; PRODUCT

]AXBAK EQU VARTAB+4 ; ARRAY X SIZE BACKUP

]AYBAK EQU VARTAB+5 ; ARRAY Y SIZE BACKUP

]MLIER EQU VARTAB+6 ; MULTIPLIER

]MCAND EQU VARTAB+8 ; MULTIPLICAND, ELEMENT SIZE

\*

ADIM82

LDA ]ESIZE ; ELEMENT LENGTH

STA ]MCAND ; AND STORE AS MULTIPLICAND

LDA ]AYSIZE ; GET ARRAY Y SIZE

STA ]AYBAK ; BACK IT UP

LDA ]AXSIZE

STA ]AXBAK ; AND BACK THAT UP TOO

LDA #0 ; CLEAR MCAND HIBYTE

STA ]MCAND+1

\*

\*\* MULTIPLY X AND Y

\*

TAY ; AND LOBYTE

STY SCRATCH

BEQ :ENTLP ; IF ZERO, SKIP TO LOOP

:DOADD

CLC ; CLEAR CARRY FLAG

ADC ]AXSIZE ; ADD X LENGTH

TAX ; TEMPORARILY STORE IN .X

TYA ; TRANSFER HIBYTE TO .A

ADC SCRATCH ; ADD HIBYTE

TAY ; STORE BACK IN .Y

TXA ; RELOAD LOBYTE INTO .A

:LP

ASL ]AXSIZE ; MULTIPLY X LENGTH BY 2

ROL SCRATCH ; ADJUST HIBYTE

:ENTLP

LSR ]AYSIZE ; DIVIDE Y LENGTH BY 2

BCS :DOADD ; IF >= LOBYTE IN .A,

BNE :LP ; ADD AGAIN; OTHERWISE, LOOP

STX ]MLIER ; STORE LOBYTE IN MULTIPLIER

STY ]MLIER+1 ; STORE HIBYTE IN MULTIPLIER

\*

\*\* NOW MULTIPLY BY LENGTH OF ELEMENTS

\*

LDA #0 ; CLEAR PRODUCT LOBYTE

STA ]PROD

STA ]PROD+1 ; CLEAR NEXT BYTE

STA ]PROD+2 ; CLEAR NEXT BYTE

STA ]PROD+3 ; CLEAR HIBYTE

LDX #$10 ; LOAD $10 IN .X (#16)

:SHIFTR LSR ]MLIER+1 ; DIVIDE MLIER BY TWO

ROR ]MLIER ; ADJUST LOBYTE

BCC :ROTR ; IF LESS THAN PRODUCT, ROTATE

LDA ]PROD+2 ; LOAD PRODUCT 3RD BYTE

CLC ; CLEAR CARRY

ADC ]MCAND ; ADD MULTIPLICAND

STA ]PROD+2 ; STORE BACK INTO PRODUCT 3RD BYTE

LDA ]PROD+3 ; LOAD PRODUCT HIBYTE

ADC ]MCAND+1 ; ADD MULTIPLICAND HIBYTE

:ROTR

ROR ; ROTATE .A RIGHT

STA ]PROD+3 ; STORE IN PRODUCT HIBYTE

ROR ]PROD+2 ; ROTATE PRODUCT 3RD BYTE

ROR ]PROD+1 ; ROTATE PRODUCT 2ND BYTE

ROR ]PROD ; ROTATE PRODUCT LOBYTE

DEX ; DECREMENT COUNTER

BNE :SHIFTR ; IF NOT 0, BACK TO SHIFTER

\*

LDA ]PROD ; LOAD PRODUCT LOBYTE TO .A

CLC ; CLEAR CARRY FLAG

ADC #3 ; ADD 3

STA ]PROD ; STORE BACK INTO PRODUCT LOBYTE

LDA ]PROD+1

ADC #0 ; INITIATE CARRY FOR 2ND BYTE

STA ]PROD+1

LDA ]PROD+2

ADC #0 ; AND THIRD BYTE

STA ]PROD+2

\*

\*\* NOW CLEAR MEMORY BLOCKS, WHOLE PAGES FIRST

\*

LDA ]FILL ; GET FILL VALUE

LDX ]PROD+1 ; LOAD SECOND BYTE OF PRODUCT

BEQ :PART ; IF 0, THEN ONLY PARTIAL PAGE

LDY #0 ; CLEAR INDEX

:FULL

STA (]ADDR),Y ; COPY FILL BYTE TO ADDRESS

INY ; INCREASE INDEX

BNE :FULL ; IF NO OVERFLOW, KEEP FILL

INC ]ADDR+1 ; INCREASE ADDRESS HIBYTE

DEX ; DECREMENT COUNTER

BNE :FULL ; LOOP UNTIL PAGES DONE

:PART

LDX ]PROD ; LOAD PRODUCT LOBYTE TO X

BEQ :MFEXIT ; IF ZERO, THEN WE'RE DONE

LDY #0 ; RESET INDEX

:PARTLP

STA (]ADDR),Y ; STORE FILL BYTE

INY ; INCREASE INDEX

DEX ; DECREASE COUNTER

BNE :PARTLP ; LOOP UNTIL DONE

:MFEXIT

LDY #0 ; RESET INDEX

LDA ]AXBAK ; PUT X LENGTH INTO

STA (]ADDR),Y ; FIRST BYTE OF ARRAY

INY ; INCREMENT INDEX

LDA ]AYBAK ; PUT Y LENGTH INTO

STA (]ADDR),Y ; SECOND BYTE OF ARRAY

INY ; INCREMENT INDEX

LDA ]MCAND ; PUT ELEMENT SIZE

STA (]ADDR),Y ; INTO 3RD BYTE OF ARRAY

LDX ]ADDR ; RETURN ARRAY ADDR LOBYTE IN .X

LDY ]ADDR+1 ; RETURN ARRAY ADDR HIBYTE IN .Y

LDA ]PROD ; STORE PRODUCT LOBYTE IN RETURN

STA RETURN

LDA ]PROD+1 ; STORE NEXT BYTE

STA RETURN+1

LDA ]PROD+2 ; NEXT BYTE

STA RETURN+2

LDA ]PROD+3 ; STORE HIBYTE

STA RETURN+3

LDA #4 ; SIZE OF RETURN

STA RETLEN ; SPECIFY RETURN LENGTH

LDA ]MCAND ; RETURN ELEMENT SIZE IN .A

RTS

|  |
| --- |
| **AGET82 (sub)**  **Input**:  **WPAR1** = array address  (2b)  **BPAR1** = first dimension  index (1b)  **BPAR2** = second dimension  index (1b)  **Output**:  **.A** = element length  **RETURN** = element data  **RETLEN** = element length  **Destroys**: AXYNVZCM  **Cycles**: 288+  **Size**: 243 bytes |

**SUB.AGET82 >> AGET82**

The **AGET82** retrieves the data from an element in an 8-bit, two-dimensional array initialized by the **ADIM82** subroutine. The data is held in **RETURN**, with its length in **RETLEN**.

\*

\*``````````````````````````````\*

\* AGET82 (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = ARRAY ADDRESS \*

\* BPAR1 = 1ST DIM INDEX \*

\* BPAR2 = 2ND DIM INDEX \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = ELEMENT LENGTH \*

\* RETURN = ELEMENT DATA \*

\* RETLEN = ELEMENT LENGTH \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 288+ \*

\* SIZE: 243 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]ADDR EQU WPAR1 ; ARRAY ADDRESS

]XIDX EQU BPAR1 ; 1ST DIMENSION INDEX

]YIDX EQU BPAR2 ; 2ND DIMENSION INDEX

\*

]XLEN EQU VARTAB+0 ; X DIMENSION LENGTH

]YLEN EQU VARTAB+2 ; Y DIMENSION LENGTH

]PROD EQU VARTAB+4 ; PRODUCT

]MLIER EQU VARTAB+8 ; MULTIPLIER

]MCAND EQU VARTAB+10 ; MULTIPLICAND

]ELEN EQU VARTAB+12 ; ELEMENT LENGTH

]PBAK EQU VARTAB+14 ; PRODUCT BACKUP

\*

AGET82

LDY #0 ; RESET INDEX

LDA (]ADDR),Y ; GET X-LENGTH FROM ARRAY

STA ]XLEN

LDY #1 ; INCREMENT INDEX

LDA (]ADDR),Y ; GET Y-LENGTH FROM ARRAY

STA ]YLEN

LDY #2 ; INCREMENT INDEX

LDA (]ADDR),Y ; GET ELEMENT LENGTH FROM ARRAY

STA ]ELEN

\*

\*\* MULTIPLY Y-INDEX BY Y-LENGTH

\*

LDA #0 ; RESET LOBYTE

TAY ; RESET HIBYTE

STY SCRATCH ; SAVE HIBYTE IN SCRATCH

BEQ :ENTLP ; IF ZERO, SKIP TO LOOP

:DOADD

CLC ; CLEAR CARRY FLAG

ADC ]YIDX ; ADD Y-INDEX

TAX ; TEMPORARILY STORE IN .X

TYA ; LOAD HIBYTE TO .A

ADC SCRATCH ; ADD HIBYTE

TAY ; TRANSFER BACK INTO .Y

TXA ; RELOAD LOBYTE

:LP

ASL ]YIDX ; MULTIPLY Y-INDEX BY 2

ROL SCRATCH ; DEAL WITH HIBYTE

:ENTLP

LSR ]YLEN ; DIVIDE Y-LENGTH BY 2

BCS :DOADD ; IF >= LOBYTE IN .A, ADD AGAIN

BNE :LP ; ELSE, LOOP

STX ]PBAK ; STORE LOBYTE IN PRODUCT BACKUP

STY ]PBAK+1 ; STORE HIBYTE

\*

\*\* NOW MULTIPLY LENGTH OF ELEMENTS BY XIDX

\*

LDA ]XIDX ; PUT X-INDEX INTO

STA ]MLIER ; MULTIPLIER

LDA ]ELEN ; ELEMENT LENGTH INTO

STA ]MCAND ; MULTIPLICAND

LDA #0 ; RESET PRODUCT LOBYTE

STA ]MLIER+1 ; RESET MULTIPLIER HIBYTE

STA ]MCAND+1 ; RESET MULTIPLICAND HIBYTE

STA ]PROD

STA ]PROD+1 ; RESET PRODUCT 2ND BYTE

STA ]PROD+2 ; RESET PRODUCT 3RD BYTE

STA ]PROD+3 ; RESET PRODUCT HIBYTE

LDX #$10 ; LOAD $10 INTO .X (#16)

:SHIFTR LSR ]MLIER+1 ; DIVIDE MULTIPLIER BY 2

ROR ]MLIER ; ADJUST LOBYTE

BCC :ROTR ; IF < PRODUCT, ROTATE

LDA ]PROD+2 ; LOAD PRODUCT 3RD BYTE

CLC ; CLEAR CARRY FLAG

ADC ]MCAND ; ADD MULTIPLICAND

STA ]PROD+2 ; STORE BACK INTO 3RD

LDA ]PROD+3 ; LOAD HIBYTE

ADC ]MCAND+1 ; ADD MULTIPLICAND HIBYTE

:ROTR

ROR ; ROTATE .A RIGHT

STA ]PROD+3 ; STORE IN PRODUCT HIBYTE

ROR ]PROD+2 ; ROTATE PRODUCT 3RD BYTE

ROR ]PROD+1 ; ROTATE PRODUCT 2ND BYTE

ROR ]PROD ; ROTATE PRODUCT LOBYTE

DEX ; DECREMENT COUNTER

BNE :SHIFTR ; IF NOT 0, BACK TO SHIFTER

LDA ]PROD ; LOAD PRODUCT LOBYTE

CLC ; CLEAR CARRY FLAG

ADC #3 ; INCREASE BY 3

STA ]PROD ; STORE BACK INTO LOBYTE

LDA ]PROD+1 ; ACCOUNT FOR CARRIES

ADC #0

STA ]PROD+1

\*

\*\* NOW ADD THAT TO EARLIER CALC

\*

CLC ; CLEAR CARRY FLAG

LDA ]PROD ; LOAD PRODUCT LOBYTE

ADC ]PBAK ; ADD PREVIOUS PRODUCT

STA ]PROD ; STORE NEW PRODUCT LOBYTE

LDA ]PROD+1 ; LOAD PRODUCT HIBYTE

ADC ]PBAK+1 ; ADD PREV PRODUCT HIBYTE

STA ]PROD+1 ; STORE PRODUCT HIBYTE

\*

\*\* NOW ADD ARRAY ADDRESS TO GET INDEX ADDR

\*

CLC ; CLEAR CARRY FLAG

LDA ]PROD ; LOAD PRODUCT LOBYTE

ADC ]ADDR ; ADD ARRAY ADDRESS LOBYTE

STA ]PROD ; STORE BACK IN PRODUCT LOBYTE

LDA ]PROD+1 ; LOAD HIBYTE

ADC ]ADDR+1 ; ADD ADDRESS HIBYTE

STA ]PROD+1 ; STORE IN PRODUCT HIBYTE

\*

LDY ]PROD ; LOAD PRODUCT LOBYTE IN .Y

LDX ]PROD+1 ; LOAD HIBYTE IN .X FOR SOME REASON

STY ]ADDR ; TRANSFER TO ZERO PAGE

STX ]ADDR+1

LDY #0 ; RESET INDEX

:RLP

LDA (]ADDR),Y ; LOAD BYTE

STA RETURN,Y ; STORE IN RETURN

INY ; INCREASE INDEX

CPY ]ELEN ; IF INDEX != ELEMENT LENGTH

BNE :RLP ; THEN KEEP COPYING

LDA ]ELEN ; OTHERWISE, STORE ELEMENT LENGTH

STA RETLEN ; INTO RETURN LENGTH

LDA RETLEN ; AND IN .A

LDX ]ADDR ; RETURN ARRAY ADDRESS LOBYTE IN .X

LDY ]ADDR+1 ; RETURN HIBYTE IN .Y

RTS

|  |
| --- |
| **APUT82 (sub)**  **Input**:  **WPAR1** = source address  (2b)  **WPAR2** = array address  (2b)  **BPAR1** = first dimension  index (1b)  **BPAR2** = second dimension  index (1b)  **Output**:  **.A** = element size  **.X** = element address  low byte  **.Y** = element address  high byte  **Destroys**: AXYNVZCM  **Cycles**: 274+  **Size**: 239 bytes |

**SUB.APUT82 >> APUT82**

The **APUT82** subroutine copies the data from a source address range into an 8-bit, two dimensional array element. The length of the data copied is determined by the array’s element length byte, which is set by **ADIM82**.

\*

\*``````````````````````````````\*

\* APUT82 (NATHAN RIGGS) \*

\* \*

\* PUT DATA FROM SOURCE INTO \*

\* A 2D, 8BIT ARRAY ELEMENT. \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = SOURCE ADDRESS \*

\* WPAR2 = ARRAY ADDRESS \*

\* BPAR1 = 1ST DIM INDEX \*

\* BPAR2 = 2ND DIM INDEX \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = ELEMENT SIZE \*

\* .X = ELEMENT ADDR LOBYTE \*

\* .Y = ELEMENT ADDR HIBYTE \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 274 \*

\* SIZE: 239 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]ADDRS EQU WPAR1 ; SOURCE ADDRESS

]ADDRD EQU WPAR2 ; ARRAY ADDRESS

]XIDX EQU BPAR1 ; X INDEX

]YIDX EQU BPAR2 ; Y INDEX

\*

]ESIZE EQU VARTAB ; ELEMENT LENGTH

]MCAND EQU VARTAB+1 ; MULTIPLICAND

]MLIER EQU VARTAB+3 ; MULTIPLIER

]PROD EQU VARTAB+5 ; PRODUCT

]XLEN EQU VARTAB+9 ; ARRAY X-LENGTH

]YLEN EQU VARTAB+13 ; ARRAY Y-LENGTH

]PBAK EQU VARTAB+15 ; PRODUCT BACKUP

\*

APUT82

LDY #0 ; RESET INDEX

LDA (]ADDRD),Y ; GET ARRAY X-LENGTH

STA ]XLEN

LDY #1 ; INCREMENT INDEX

LDA (]ADDRD),Y ; GET ARRAY Y-LENGTH

STA ]YLEN

LDY #2 ; INCREMENT INDEX

LDA (]ADDRD),Y ; GET ARRAY ELEMENT LENGTH

STA ]ESIZE

\*

\*\* MULTIPLY Y-INDEX BY Y-LENGTH

\*

LDA #0 ; RESET LOBYTE

TAY ; RESET HIBYTE

STY SCRATCH ; SAVE HIBYTE IN SCRATCH

BEQ :ENTLP ; IF ZERO, SKIP TO LOOP

:DOADD

CLC ; CLEAR CARRY FLAG

ADC ]YIDX ; ADD Y-INDEX

TAX ; STORE IN .X

TYA ; LOAD HIBYTE

ADC SCRATCH ; ADD HIBYTE

TAY ; STORE IN .Y

TXA ; RELOAD LOBYTE

:LP

ASL ]YIDX ; MULTIPLY Y-INDEX BY 2

ROL SCRATCH ; DEAL WITH HIBYTE

:ENTLP

LSR ]YLEN ; DIVIDE Y-LENGTH BY 2

BCS :DOADD ; IF >= LOBYTE, ADD AGAIN

BNE :LP ; ELSE, LOOP

STX ]PBAK ; STORE LOBYTE IN PRODUCT BACKUP

STY ]PBAK+1 ; STORE HIBYTE

LDA ]XIDX ; PUT X-INDEX INTO MULTIPLIER

STA ]MLIER

LDA #0 ; RESET HIBYTE

STA ]MLIER+1 ; TRANSFER HIBYTE

LDA ]ESIZE ; PUT ELEMENT LENGTH

STA ]MCAND ; INTO MULTIPLICAND

LDA #0 ; RESET HIBYTE

STA ]MCAND+1

\*

\*\* NOW MULTIPLY XIDX BY ELEMENT LENGTH

\*

STA ]PROD ; RESET PRODUCT LOBYTE

STA ]PROD+1 ; RESET 2ND BYTE

STA ]PROD+2 ; RESET 3RD BYTE

STA ]PROD+3 ; RESET HIBYTE

LDX #$10 ; LOAD $10 INTO .X (#16)

:SHIFTR LSR ]MLIER+1 ; DIVIDE MULTIPLIER BY 2

ROR ]MLIER ; DEAL WITH HIBYTE

BCC :ROTR ; IF < RODUCT, ROTATE

LDA ]PROD+2 ; LOAD PRODUCT 3RD BYTE

CLC ; CLEAR CARRY FLAG

ADC ]MCAND ; ADD MULTIPLICAND

STA ]PROD+2 ; STORE 3RD BYTE

LDA ]PROD+3 ; LOAD HIBYTE

ADC ]MCAND+1 ; ADD MULTIPLICAND HIBYTE

:ROTR

ROR ; ROTATE .A RIGHT

STA ]PROD+3 ; STORE IN PRODUCT HIBYTE

ROR ]PROD+2 ; ROTATE PRODUCT 3RD BYTE

ROR ]PROD+1 ; ROTATE RODUCT 2ND

ROR ]PROD ; ROTATE LOBYTE

DEX ; DECREMENT COUNTER

BNE :SHIFTR ; IF NOT 0, BACK TO SHIFTER

\*

\*\* NOW ADD PRODUCT TO REST

\*

LDA ]PBAK ; LOAD FIRST PRODUCT LOBYTE

CLC ; CLEAR CARRY FLAG

ADC ]PROD ; ADD 2ND PRODUCT LOBYTE

STA ]PROD ; STORE NEW PRODUCT LOBYTE

LDA ]PBAK+1 ; LOAD FIRST PRODUCT HIBYTE

ADC ]PROD+1 ; ADD 2ND HIBYTE

STA ]PROD+1 ; STORE HIBYTE

LDA ]PROD ; LOAD NEW PRODUCT LOBYTE

CLC ; CLEAR CARRY FLAG

ADC #3 ; INCREASE BY 3

STA ]PROD ; STORE IN LOBYTE

LDA ]PROD+1 ; APPLY CARRY TO HIBYTE

ADC #0

STA ]PROD+1

\*

\*\* ADD ARRAY ADDRESS TO GET INDEX

\*

CLC ; CLEAR CARRY FLAG

LDA ]PROD ; LOAD PRODUCT LOBYTE

ADC ]ADDRD ; ADD ARRAY ADDRESS LOBYTE

STA ]PROD ; STORE IN PRODUCT

LDA ]PROD+1 ; LOAD PRODUCT HIBYTE

ADC ]ADDRD+1 ; ADD ARRAYH ADDRESS HIBYTE

STA ]PROD+1 ; STORE HIBYTE

LDX ]PROD ; PUT ELEMENT ADDRESS LOBYTE IN .X

LDY ]PROD+1 ; PUT HIBYTE IN Y

STX ADDR2 ; STORE IN ZERO PAGE

STY ADDR2+1

LDY #0 ; RESET INDEX

\*

\*\* COPY FROM SRC ADDR TO DEST ADDR

\*

:CLP

LDA (]ADDRS),Y ; GET BYTE FROM SOURCE

STA (ADDR2),Y ; STORE IN ELEMENT

INY ; INCREASE INDEX

CPY ]ESIZE ; IF < ELEMENT SIZE,

BNE :CLP ; CONTINUE COPYING

LDX ADDR2 ; PUT ELEMENT LOBYTE IN .X

LDY ADDR2+1 ; PUT HIBYTE IN .Y

LDA ]ESIZE ; PUT ELEMENT SIZE IN .A

RTS

|  |
| --- |
| **ADIM161 (sub)**  **Input**:  **WPAR1** = array address  (2b)  **WPAR2** = # of elements  (2b)  **WPAR3** = element length  (1b)  **BPAR1** = fill value (1b)  **Output**:  **.A** = element size  **RETURN** = total size  **RETLEN** = 2  **Destroys**: AXYNVZCM  **Cycles**: 172+  **Size**: 162 bytes |

**SUB.ADIM161 >> ADIM161**

The **ADIM161** subroutine initializes a 16-bit, one-dimensional array that can hold a total of 65,025 elements. This array has a three byte header: byte 0 contains the low byte of the number of elements, and byte 1 contains the high byte. Byte 3 holds the length of each element, from 0 to 255.

\*

\*``````````````````````````````\*

\* ADIM161 (NATHAN RIGGS) \*

\* \*

\* INITIALIZE A 16BIT, 2D ARRAY \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = ARRAY ADDRESS \*

\* WPAR2 = # OF ELEMENTS \*

\* WPAR3 = ELEMENT LENGTH \*

\* BPAR1 = FILL VALUE \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = ELEMENT SIZE \*

\* RETURN = TOTAL ARRAY SIZE \*

\* RETLEN = 2 \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 172+ \*

\* SIZE: 162 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]ADDRD EQU WPAR1

]ASIZE EQU WPAR2

]ESIZE EQU WPAR3

]FILL EQU BPAR1

\*

]MSIZE EQU VARTAB ; TOTAL ARRAY BYTES

]ASZBAK EQU VARTAB+4 ; BACKUP OF ELEMENT #

]ESZBAK EQU VARTAB+7 ; BACKUP

\*

ADIM161

LDA ]ESIZE ; ELEMENT SIZE

STA ]ESZBAK ; ELEMENT LENGTH BACKUP

LDA ]ASIZE

STA ]ASZBAK ; ARRAY SIZE BACKUP

LDA ]ASIZE+1

STA ]ASZBAK+1 ; BACKUP

STA SCRATCH ; HIBYTE FOR MULTIPLICATION

LDA ]ADDRD

STA ADDR2

LDA ]ADDRD+1

STA ADDR2+1

LDY #0 ; CLEAR INDEX

LDA #0 ; CLEAR ACCUMULATOR

BEQ :ENTLP ; IF 0, SKIP TO LOOP

\*

\*\* MULTIPLY ARRAY SIZE BY ELEMENT SIZE

\*

:DOADD

CLC ; CLEAR CARRY FLAG

ADC ]ASIZE ; ADD ARRAY SIZE

TAX ; HOLD IN .X

TYA ; LOAD HIBYTE

ADC SCRATCH ; ADD HIBYTE

TAY ; HOLD IN .Y

TXA ; RELOAD LOBYTE

:LP

ASL ]ASIZE ; MULTIPLY ARRAY SIZE BY 2

ROL SCRATCH ; ADJUST HIBYTE

:ENTLP

LSR ]ESIZE ; DIVIDE ELEMENT SIZE BY 2

BCS :DOADD ; IF >= LOBYTE IN .A,

BNE :LP ; ADD AGAIN--ELSE, LOOP

CLC ; CLEAR CARRY

TXA ; LOBYTE TO .A

ADC #3 ; ADD 2 FOR HEADER

STA ]MSIZE ; STORE IN TOTAL LOBYTE

TYA ; HIBYTE TO .A

ADC #0 ; DO CARRY

STA ]MSIZE+1 ; STORE IN TOTAL HIBYTE

\*

\*\* CLEAR MEMORY BLOCKS

\*

LDA ]FILL ; GET FILL VALUE

LDX ]MSIZE+1 ; LOAD TOTAL SIZE LOBYTE

BEQ :PART ; IF NO WHOLE PAGES, JUST PART

LDY #0 ; RESET INDEX

:FULL

STA (]ADDRD),Y ; COPY BYTE TO ADDRESS

INY ; NEXT BYTE

BNE :FULL ; LOOP UNTIL PAGE DONE

INC ]ADDRD+1 ; GO TO NEXT PAGE

DEX ; DECREMENT COUNTER

BNE :FULL ; LOOP IF PAGES LEFT

:PART

LDX ]MSIZE ; PARTIAL PAGE BYTES

BEQ :MFEXIT ; EXIT IF = 0

LDY #0 ; RESET INDEX

:PARTLP

STA (]ADDRD),Y ; STORE BYTE

INY ; INCREMENT INDEX

DEX ; DECREMENT COUNTER

BNE :PARTLP ; LOOP UNTIL DONE

:MFEXIT

LDY #0 ; RESET INDEX

LDA ]ASZBAK ; STORE ARRAY SIZE IN HEADER

STA (ADDR2),Y

INY ; INCREASE INDEX

LDA ]ASZBAK+1 ; STORE ARRAY SIZE HIBYTE

STA (ADDR2),Y

INY ; INCREMENT INDEX

LDA ]ESZBAK ; STORE ELEMENT SIZE

STA (ADDR2),Y ; IN HEADER

LDX ]ADDRD ; .X HOLDS ARRAY ADDRESS LOBYTE

LDY ]ADDRD+1 ; .Y HOLDS HIBYTE

LDA ]MSIZE ; STORE TOTAL ARRAY SIZE

STA RETURN ; IN RETURN

LDA ]MSIZE+1

STA RETURN+1

LDA #2

STA RETLEN ; 2 BYTE LENGTH

LDA ]ASZBAK ; .A HOLDS # OF ELEMENTS

RTS

|  |
| --- |
| **AGET161 (sub)**  **Input**:  **WPAR1** = array address  (2b)  **WPAR2** = element index  (2b)  **Output**:  **.A** = element length  **.X** = element address  low byte  **.Y** = element address  high byte  **RETURN** = element data  **RETLEN** = element length  **Destroys**: AXYNVZCM  **Cycles**: 126+  **Size**: 135 bytes |

**SUB.AGET161 >> AGET161**

The **AGET161** subroutine retrieves data from a 16-bit, one-dimensional array element created by **ADIM161** and stores the data in **RETURN**. The length of the data is stored in **RETLEN**.

\*

\*``````````````````````````````\*

\* AGET161 (NATHAN RIGGS) \*

\* \*

\* GET DATA IN ARRAY ELEMENT \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = ARRAY ADDRESS \*

\* WPAR2 = ELEMENT INDEX \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = ELEMENT LENGTH \*

\* .X = ELEMENT ADDR LOBYTE \*

\* .Y = ELEMENT ADDR HIBYTE \*

\* RETURN = ELEMENT DATA \*

\* RETLEN = ELEMENT LENGTH \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 126 \*

\* SIZE: 135 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]AIDX EQU WPAR2

]ADDR EQU WPAR1

\*

]ESIZE EQU VARTAB ; ELEMENT LENGTH

]ESIZEB EQU VARTAB+1 ; ^BACKUP

]ASIZE EQU VARTAB+2 ; NUMBER OF ELEMENTS

]IDX EQU VARTAB+6 ; INDEX BACKUP

\*

AGET161

LDA ]AIDX

STA ]IDX

LDA ]AIDX+1 ; GET INDEX HIBYTE

STA ]AIDX+1

STA SCRATCH

LDY #0 ; RESET INDEX

LDA (]ADDR),Y ; GET NUMBER OF

STA ]ASIZE ; ARRAY ELEMENTS

LDY #1 ; GET HIBYTE OF

LDA (]ADDR),Y ; # OF ARRAY ELEMENTS

STA ]ASIZE+1

INY ; INCREASE BYTE INDEX

LDA (]ADDR),Y ; GET ELEMENT LENGTH

STA ]ESIZE

STA ]ESIZEB

\*

\*\* MULTIPLY INDEX BY ELEMENT SIZE, ADD 3

\*

LDY #0 ; RESET .Y AND .A

LDA #0

BEQ :ENTLPA ; IF ZERO, SKIP TO LOOP

:DOADD

CLC ; CLEAR CARRY

ADC ]AIDX ; ADD INDEX TO .A

TAX ; HOLD IN .X

TYA ; LOAD HIBYTE

ADC SCRATCH ; ADD HIBYTE

TAY ; HOLD IN .Y

TXA ; RELOAD LOBYTE

:LPA

ASL ]AIDX ; MULTIPLY INDEX BY 2

ROL SCRATCH ; ADJUST HIBYTE

:ENTLPA

LSR ]ESIZE ; DIVIDE ELEMENT LENGTH BY 2

BCS :DOADD ; IF BIT 1 SHIFTED IN CARRY, ADD MORE

BNE :LPA ; CONTINUE LOOPING IF ZERO FLAG UNSET

STX ]IDX ; STORE LOBYTE

STY ]IDX+1 ; STORE HIBYTE

LDA #3 ; ADD 3 TO INDEX LOBYTE

CLC ; CLEAR CARRY

ADC ]IDX

STA ADDR2 ; STORE ON ZERO PAGE

LDA ]IDX+1 ; ADJUST HIBYTE

ADC #0

STA ADDR2+1

\*

LDA ADDR2 ; ADD ARRAY ADDRESS

CLC

ADC ]ADDR ; LOBYTE

STA ADDR2

LDA ADDR2+1 ; HIBYTE

ADC ]ADDR+1

STA ADDR2+1

LDY #0 ; RESET BYTE INDEX

:LP

LDA (ADDR2),Y ; GET BYTE FROM ELEMENT

STA RETURN,Y ; PUT INTO RETURN

INY ; INCREASE BYTE INDEX

CPY ]ESIZEB ; IF INDEX != ELEMENT LENGTH

BNE :LP ; CONTINUE LOOP

LDA ]ESIZEB ; .A = ELEMENT SIZE

STA RETLEN ; STORE IN RETLEN

LDY ADDR2+1 ; .Y = ELEMENT ADDRESS HIBYTE

LDX ADDR2 ; .X = ELEMENT ADDRESS LOBYTE

RTS

|  |
| --- |
| **APUT161 (sub)**  **Input**:  **WPAR1** = source address  (2b)  **WPAR2** = array address  (2b)  **WPAR3** = element index  (1b)  **Output**:  **.A** = element length  **.X** = array address  low byte  **.Y** = array address  high byte  **Destroys**: AXYNVZCM  **Cycles**: 181+  **Size**: 135 bytes |

**SUB.APUT161 >> APUT161**

The **APUT161** subroutine sets the data in a 16-bit, one-dimensional array element. The length of the data is determined by the element length byte in the array header, which is set by **ADIM161**.

\*

\*``````````````````````````````\*

\* APUT161 (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = SOURCE ADDRESS \*

\* WPAR2 = ARRAY ADDRESS \*

\* WPAR3 = ELEMENT INDEX \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = ELEMENT LENGTH \*

\* .X = ARRAY ADDRESS LOBYTE \*

\* .Y = ARRAY ADDRESS HIBYTE \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 181+ \*

\* SIZE: 135 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]ADDRS EQU WPAR1

]ADDRD EQU WPAR2

]AIDX EQU WPAR3

\*

]ESIZE EQU VARTAB ; ELEMENT SIZE

]ESIZEB EQU VARTAB+1 ; ^BACKUP

]ASIZE EQU VARTAB+2 ; NUMBER OF ELEMENTS

]IDX EQU VARTAB+6 ; ANOTHER INDEX

\*

APUT161

LDA ]AIDX

STA ]IDX

LDA ]AIDX+1

STA ]IDX+1

STA SCRATCH

LDY #0 ; RESET BYTE COUNTER

LDA (]ADDRD),Y ; GET NUMBER OF ELEMENTS

STA ]ASIZE ; LOBYTE

LDY #1 ; INCREMENT INDEX

LDA (]ADDRD),Y ; GET NUMBER OF ELEMENTS

STA ]ASIZE+1 ; HIBYTE

INY ; INCREMENT INDEX

LDA (]ADDRD),Y ; GET ELEMENT LENGTH

STA ]ESIZE

STA ]ESIZEB ; BACKUP

\*

\*\* MULTIPLY INDEX BY ELEMENT SIZE, THEN ADD 3

\*

LDY #0 ; RESET LOBYTE

LDA #0 ; AND HIBYTE

BEQ :ENTLPA ; SKIP TO LOOP

:DOADD

CLC ; CLEAR CARRY

ADC ]AIDX ; ADD INDEX LOBYTE

TAX ; HOLD IN .X

TYA ; LOAD HIBYTE

ADC SCRATCH ; ADD HIBYTE

TAY ; HOLD BACK IN .Y

TXA ; RETURN LOBYTE TO .A

:LPA

ASL ]AIDX ; MULTIPLY INDEX BY 2

ROL SCRATCH ; ADJUST HIBYTE

:ENTLPA

LSR ]ESIZE ; DIVIDE ELEM LENGTH BY 2

BCS :DOADD ; IF 1 SHIFTED TO CARRY, ADD AGAIN

BNE :LPA ; CONTINUE LOOP IF ZERO UNSET

STX ]IDX ; LOBYTE IN .X

STY ]IDX+1 ; HIBYTE IN .Y

CLC

LDA #3 ; ADD 3 TO LOBYTE

ADC ]IDX

STA ADDR2 ; STORE ON ZERO PAGE

LDA ]IDX+1 ; ADJUST HIBYTE

ADC #0

STA ADDR2+1

\*

CLC ; CLEAR CARRY

LDA ADDR2 ; ADD ARRAY ADDRESS

ADC ]ADDRD ; LOBYTE

STA ADDR2 ; ADD ARRAY ADDRESS

LDA ADDR2+1 ; HIBYTE

ADC ]ADDRD+1

STA ADDR2+1

LDY #0

:LP

\*

\*\* OOPS; NEED TO CONVERT THIS TO 16 BITS

\*

LDA (]ADDRS),Y ; GET BYTE FROM SOURCE

STA (ADDR2),Y ; STORE IN ELEMENT

INY ; INCREMENT BYTE INDEX

CPY ]ESIZEB ; IF INDEX != ELEMENT LENGTH

BNE :LP ; KEEP LOOPING

LDY ADDR2+1 ; HIBYTE OF ELEMENT ADDRESS

LDX ADDR2 ; LOBYTE

LDA ]ESIZEB ; .A = ELEMENT SIZE

RTS

|  |
| --- |
| **ADIM162 (sub)**  **Input**:  **WPAR1** = first dimension  Length (2b)  **WPAR2** = second dimension  Length (2b)  **WPAR3** = array address  (2b)  **BPAR1** = element length  (1b)  **BPAR2** = fill value (1b)  **Output**:  **.A** = element length  **RETURN** = element data  **RETLEN** = element length  **Destroys**: AXYNVZCM  **Cycles**: 426+  **Size**: 312 bytes |

**SUB.ADIM162 >> ADIM162**

The **ADIM162** subroutine initializes a two-dimensional 16-bit array. Each dimension can theoretically hold 65,025 indices each, with a total number of elements of 4,228,250,625‬ that can carry a length of 255 bytes each. Obviously, this is beyond the RAM capacity of even the most souped up Apple II, save for the GS (and even then, it would have to be heavily modified).

Two-dimensional 16-bit arrays have a 5-byte header. Byte 0 holds the low byte of the number of indices in the first dimension, with byte 1 holding the high byte. Byte 2 likewise holds the low byte of the second dimension’s number of indices, with the high in byte 3. Lastly, byte 4 holds the element length, with the data of the array following.

\*

\*``````````````````````````````\*

\* ADIM162 (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = 1ST DIM LENGTH \*

\* WPAR2 = 2ND DIM LENGTH \*

\* WPAR3 = ARRAY ADDRESS \*

\* BPAR1 = ELEMENT LENGTH \*

\* BPAR2 = FILL VALUE \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = ELEMENT LENGTH \*

\* RETURN = ELEMENT DATA \*

\* RETLEN = ELEMENT LENGTH \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 426+ \*

\* SIZE: 312 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]AXSIZE EQU WPAR1

]AYSIZE EQU WPAR2

]ELEN EQU BPAR1

]FILL EQU BPAR2

]ADDR EQU WPAR3

]ADDR2 EQU ADDR1

\*

]PROD EQU VARTAB ; PRODUCT

]AXBAK EQU VARTAB+4 ; X SIZE BACKUP

]AYBAK EQU VARTAB+6 ; Y SIZE BACKUP

]MLIER EQU VARTAB+8 ; MULTIPLIER

]MCAND EQU VARTAB+10 ; MULTIPLICAND

\*

ADIM162

LDA ]AYSIZE

STA ]AYBAK

STA ]MCAND

LDA ]AYSIZE+1

STA ]AYBAK+1

STA ]MCAND+1

LDA ]AXSIZE

STA ]AXBAK

STA ]MLIER

LDA ]AXSIZE+1

STA ]AXBAK+1

STA ]MLIER+1

LDA ]ADDR ; GET ARRAY ADDRESS

STA ]ADDR2 ; LOBYTE; PUT IN ZERO PAGE

LDA ]ADDR+1 ; GET ARRAY ADDRESS HIBYTE

STA ]ADDR2+1

\*

\*\* MULTIPLY X AND Y

\*

LDA #0 ; RESET HIBYTE,LOBYTE

STA ]PROD+2 ; CLEAR PRODUCT BYTE 3

STA ]PROD+3 ; CLEAR PRODUCT BYTE 4

LDX #$10 ; (#16)

:SHIFT\_R

LSR ]MLIER+1 ; DIVIDE MLIER BY TWO

ROR ]MLIER ; ADJUST LOBYTE

BCC :ROT\_R ; IF 0 IN CARRY, ROTATE MORE

LDA ]PROD+2 ; GET 3RD BYTE OF PRODUCT

CLC

ADC ]MCAND ; ADD MULTIPLICAND

STA ]PROD+2 ; STORE 3RD BYTE

LDA ]PROD+3 ; LOAD 4TH BYTE

ADC ]MCAND+1 ; ADD MULTIPLICAND HIBYTE

:ROT\_R

ROR ; ROTATE PARTIAL PRODUCT

STA ]PROD+3 ; STORE IN HIBYTE

ROR ]PROD+2 ; ROTATE THIRD BYTE

ROR ]PROD+1 ; ROTATE 2ND BYTE

ROR ]PROD ; ROTATE LOBYTE

DEX ; DECREASE COUNTER

BNE :SHIFT\_R ; IF NOT ZERO, BACK TO SHIFTER

\*

LDA ]ELEN ; PUT ELEMENT LENGTH

STA ]MCAND ; INTO MULTIPLICAND

LDA #0 ; CLEAR HIBYTE

STA ]MCAND+1 ;

LDA ]PROD ; LOAD EARLIER PRODUCT

STA ]MLIER ; STORE LOBYTE IN MULTIPLIER

LDA ]PROD+1 ; DO SAME FOR HIBYTE

STA ]MLIER+1

\*

\*\* NOW MULTIPLY BY LENGTH OF ELEMENTS

\*

LDA #0 ; CLEAR PRODUCT

STA ]PROD

STA ]PROD+1

STA ]PROD+2

STA ]PROD+3

LDX #$10

:SHIFTR LSR ]MLIER+1 ; SHIFT BYTES LEFT (/2)

ROR ]MLIER ; ADJUST LOBYTE

BCC :ROTR ; IF CARRY = 0, ROTATE

LDA ]PROD+2 ; LOAD 3RD BYTE OF PRODUCT

CLC

ADC ]MCAND ; ADD MULTIPLICAND

STA ]PROD+2 ; STORE IN 3RD BYTE

LDA ]PROD+3 ; LOAD HIBYTE

ADC ]MCAND+1 ; ADD MULTIPLICAND HIBYTE

:ROTR

ROR ; ROTATE .A RIGHT

STA ]PROD+3 ; ROTATE 4TH

ROR ]PROD+2 ; ROTATE 3RD

ROR ]PROD+1 ; ROTATE 2ND

ROR ]PROD ; ROTATE LOBYTE

DEX ; DECREMENT COUNTER

BNE :SHIFTR ; IF NOT 0, BACK TO SHIFTER

\*

CLC ; CLEAR CARRY

LDA ]PROD ; INCREASE BY 5

ADC #5

STA ]PROD ; SAVE LOBYTE

LDA ]PROD+1

ADC #0

STA ]PROD+1 ; SAVE HIBYTE

\*

\*\* NOW CLEAR MEMORY BLOCKS, WHOLE PAGES FIRST

\*

LDA ]FILL ; GET FILL VALUE

LDX ]PROD+1 ; LOAD PRODUCT 2ND BYTE

BEQ :PART ; IF 0, THEN PARTIAL PAGE

LDY #0 ; CLEAR INDEX

:FULL

STA (]ADDR),Y ; COPY FILL BYTE TO ADDRESS

INY ; INCREASE BYTE COUNTER

BNE :FULL ; LOOP UNTIL PAGES DONE

INC ]ADDR+1 ; INCREASE HIBYTE

DEX ; DECREASE COUNTER

BNE :FULL ; LOOP UNTIL PAGES DONE

\*

\*\* NOW DO REMAINING BYTES

\*

:PART

LDX ]PROD ; LOAD PRODUCT LOBYTE IN X

BEQ :MFEXIT ; IF 0, THEN WE'RE DONE

LDY #0 ; CLEAR BYTE INDEX

:PARTLP

STA (]ADDR),Y ; STORE FILL BYTE

INY ; INCREASE BYTE INDEX

DEX ; DECREASE COUNTER

BNE :PARTLP ; LOOP UNTIL DONE

:MFEXIT

LDY #0 ; CLEAR BYTE INDEX

LDA ]AXBAK ; LOAD ORIGINAL X LENGTH

STA (]ADDR2),Y ; STORE IN ARRAY HEADER

INY ; INCREASE BYTE COUNTER

LDA ]AXBAK+1 ; STORE HIBYTE

STA (]ADDR2),Y

INY ; INCREASE BYTE INDEX

LDA ]AYBAK ; LOAD Y LENGTH LOBYTE

STA (]ADDR2),Y ; STORE IN ARRAY HEADER

INY ; INCREMENT BYTE INDEX

LDA ]AYBAK+1 ; STORE Y HIBYTE

STA (]ADDR2),Y

INY ; INCREMENT BYTE INDEX

LDA ]ELEN ; STORE ELEMENT LENGTH

STA (]ADDR2),Y

\*

LDY ]ADDR2 ; LOBYTE OF ARRAY ADDRESS

LDX ]ADDR2+1 ; ARRAY ADDRESS HIBYTE

LDA ]PROD ; STORE TOTAL ARRAY SIZE

STA RETURN ; IN BYTES IN RETURN

LDA ]PROD+1

STA RETURN+1

LDA ]PROD+2

STA RETURN+2

LDA ]PROD+3

STA RETURN+3

LDA #4 ; SIZE OF RETURN

STA RETLEN

RTS

|  |
| --- |
| **AGET162 (sub)**  **Input**:  **WPAR1** = array address  (2b)  **WPAR2** = first dimension  index (2b)  **WPAR3** = second dimension  index (2b)  **Output**:  **.A** = element length  **RETURN** = element data  **RETLEN** = element length  **Destroys**: AXYNVZCM  **Cycles**: 410+  **Size**: 277 bytes |

**SUB.AGET162 >> AGET162**

The **AGET162** retrieves the data held in an element of a 16-bit, two-dimensional array and stores it in **RETURN**, with the element length held in **RETVAL**. This will work correctly only with arrays initialized with **ADIM162**.

\*

\*``````````````````````````````\*

\* AGET162 (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = ARRAY ADDRESS \*

\* WPAR2 = 1ST DIM INDEX \*

\* WPAR3 = 2ND DIM INDEX \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = ELEMENT LENGTH \*

\* RETURN: ELEMENT DATA \*

\* RETLEN: ELEMENT LENGTH \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 410+ \*

\* SIZE: 277 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]ADDR EQU WPAR1

]XIDX EQU WPAR2

]YIDX EQU WPAR3

\*

]ESIZE EQU VARTAB ; ELEMENT LENGTH

]MCAND EQU VARTAB+2 ; MULTIPLICAND

]MLIER EQU VARTAB+4 ; MULTIPLIER

]PROD EQU VARTAB+6 ; PRODUCT

]PBAK EQU VARTAB+10 ; ^BACKUP

]XLEN EQU VARTAB+12 ; X-DIM LENGTH

]YLEN EQU VARTAB+14 ; Y-DIM LENGTH

\*

AGET162

LDY #4 ; READ BYTE 4 FROM HEADER

LDA (]ADDR),Y ; TO GET ELEMENT SIZE

STA ]ESIZE

LDY #0 ; READ BYTE 0 FROM HEADER

LDA (]ADDR),Y ; TO GET X-DIM LENGTH LOBYTE

STA ]XLEN

LDY #1 ; READ BYTE 1 FROM HEADER

LDA (]ADDR),Y ; TO GET X-DIM LENGTH HIBYTE

STA ]XLEN+1

LDY #2 ; READ BYTE 2 FROM HEADER

LDA (]ADDR),Y ; TO GET Y-DIM LENGTH LOBYTE

STA ]YLEN

LDY #3 ; READ BYTE 3 OF HEADER

LDA (]ADDR),Y ; TO GET Y-DIM LENGTH HIBYTE

STA ]YLEN+1

LDY #0 ; RESET BYTE INDEX

\*

\*\* MULTIPLY Y-INDEX BY Y-LENGTH

\*

LDA ]YIDX ; PUT Y-INDEX INTO

STA ]MLIER ; MULTIPLIER

LDA ]YIDX+1 ; ALSO HIBYTE

STA ]MLIER+1

LDA ]YLEN ; PUT Y-DIM LENGTH LOBYTE

STA ]MCAND ; INTO MULTIPLICAND

LDA ]YLEN+1 ; ALSO HIBYTE

STA ]MCAND+1

LDA #00 ; RESET

STA ]PROD ; PRODUCT BYTES

STA ]PROD+1

STA ]PROD+2

STA ]PROD+3

LDX #$10 ; LOAD #16 INTO X REGISTER

:SHIFT\_R

LSR ]MLIER+1 ; DIVIDE MULTIPLIER BY 2

ROR ]MLIER ; ADJUST HIBYTE

BCC :ROT\_R ; IF 0 PUT INTO CARRY, ROTATE MORE

LDA ]PROD+2 ; LOAD PRODUCT 3RD BYTE

CLC ; CLEAR CARRY

ADC ]MCAND ; ADD MULTIPLICAND

STA ]PROD+2 ; STORE IN PRODUCT 3RD

LDA ]PROD+3 ; LOAD PRODUCT HIBYTE

ADC ]MCAND+1 ; ADD MULTIPLICAN HIBYTE

:ROT\_R

ROR ; ROTATE .A RIGHT

STA ]PROD+3 ; STORE IN PRODUCT HIBYTE

ROR ]PROD+2 ; ROTATE 3RD BYTE

ROR ]PROD+1 ; ROTATE 2ND BYTE

ROR ]PROD ; ROTATE LOBYTE

DEX ; DECREASE X COUNTER

BNE :SHIFT\_R ; IF NOT ZERO, SHIFT AGAIN

\*

\*\* NOW MULTIPLY XIDX BY ELEMENT SIZE

\*

LDA ]PROD ; BACKUP PREVIOUS PRODUCT

STA ]PBAK ; 1ST AND 2ND BYTES; THE

LDA ]PROD+1 ; 3RD AND 4TH ARE NOT USED

STA ]PBAK+1

LDA ]XIDX ; LOAD X-INDEX LOBYTE

STA ]MLIER ; AND STORE IN MULTIPLIER

LDA ]XIDX+1 ; LOAD HIBYTE AND STORE

STA ]MLIER+1

LDA ]ESIZE ; LOAD ELEMENT SIZE AND

STA ]MCAND ; STORE LOBYTE IN MULTIPLICAND

LDA #0 ; CLEAR MULTIPLICAND HIBYTE

STA ]MCAND+1

\*

STA ]PROD ; CLEAR ALL PRODUCT BYTES

STA ]PROD+1

STA ]PROD+2

STA ]PROD+3

LDX #$10 ; LOAD #16 IN COUNTER

:SHIFTR LSR ]MLIER+1 ; DIVIDE MULTIPLIER HIBYTE BY 2

ROR ]MLIER ; ADJUST LOBYTE

BCC :ROTR ; IF 0 PUT IN CARRY, ROTATE

LDA ]PROD+2 ; LOAD PRODUCT 3RD BYTE

CLC ; CLEAR CARRY

ADC ]MCAND ; ADD MULTIPLICAND LOBYTE

STA ]PROD+2 ; STORE PRODUCT 3RD BYTE

LDA ]PROD+3 ; LOAD PRODUCT HIBYTE

ADC ]MCAND+1 ; ADD MULTIPLICAND HIBYTE

:ROTR

ROR ; ROTATE .A RIGHT

STA ]PROD+3 ; STORE IN PRODUCT HIBYTE

ROR ]PROD+2 ; ROTATE PRODUCT 3RD BYTE

ROR ]PROD+1 ; ROTATE 2ND BYTE

ROR ]PROD ; ROTATE LOBYTE

DEX ; DECREMENT X COUNTER

BNE :SHIFTR ; IF != 0, SHIFT AGAIN

\*

\*\* NOW ADD X \* ESIZE TO RUNNING PRODUCT

\*

CLC ; CLEAR CARRY

LDA ]PROD ; ADD PREVIOUS PRODUCT

ADC ]PBAK ; LOBYTE TO CURRENT

STA ]PROD ; AND STORE IN PRODUCT

LDA ]PROD+1 ; DO THE SAME WITH HIBYTES

ADC ]PBAK+1

STA ]PROD+1

CLC ; CLEAR CARRY

LDA ]PROD ; ADD 5 BYTES TO PRODUCT

ADC #5 ; TO ACCOUNT FOR ARRAY HEADER

STA ]PROD

LDA ]PROD+1

ADC #0 ; ADJUST HIBYTE

STA ]PROD+1

\*

\*\* NOW ADD BASE ADDRESS OF ARRAY TO GET

\*\* THE ADDRESS OF THE INDEX VALUE

\*

CLC ; CLEAR CARRY

LDA ]PROD ; ADD PRODUCT TO ARRAY

ADC ]ADDR ; ADDRESS, LOBYTES

STA ADDR2 ; STORE IN ZERO PAGE

LDA ]PROD+1 ; DO THE SAME WITH HIBYTES

ADC ]ADDR+1

STA ADDR2+1

LDY #0 ; RESET BYTE INDEX

\*

\*\* COPY FROM SRC ADDR TO DEST ADDR

\*

:CLP

LDA (ADDR2),Y ; LOAD BYTE FROM ELEMENT

STA RETURN,Y ; AND STORE IN RETURN

INY ; INCREMENT BYTE COUNTER

CPY ]ESIZE ; IF != ELEMENT LENGTH,

BNE :CLP ; CONTINUE LOOPING

LDA ]ESIZE ; .A = ELEMENT SIZE

STA RETLEN ; ALSO IN RETLEN

LDY ADDR2+1 ; .Y = ELEMENT ADDRESS HIBYTE

LDX ADDR2 ; .X = ELEMENT ADDRESS LOBYTE

RTS

|  |
| --- |
| **APUT162 (sub)**  **Input**:  **WPAR1** = source address  (2b)  **WPAR2** = array address  (2b)  **WPAR3** = first dimension  index (2b)  **ADDR1** = second dimension  index (2b)  **Output**:  **.A** = element length  **.X** = element address  low byte  **.Y** = element address  high byte  **Destroys**: AXYNVZCM  **Cycles**: 404+  **Size**: 273 bytes |

**SUB.APUT162 >> APUT162**

The **APUT162** subroutine sets the data in a 16-bit, two-dimensional array’s element at the given 2D index. The length of the data to be copied to the element is determined by the length byte of the array.

\*

\*``````````````````````````````\*

\* APUT162 (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = SOURCE ADDRESS \*

\* WPAR2 = ARRAY ADDRESS \*

\* WPAR3 = 1ST DIM INDEX \*

\* ADDR1 = 2ND DIM INDEX \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = ELEMENT LENGTH \*

\* .X = ELEMENT ADDR LOBYTE \*

\* .Y = ELEMENT ADDR HIBYTE \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 404+ \*

\* SIZE: 273 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]ADDRS EQU WPAR1

]ADDRD EQU WPAR2

]XIDX EQU WPAR3

]YIDX EQU ADDR1

\*

]ESIZE EQU VARTAB ; ELEMENT LENGTH

]MCAND EQU VARTAB+6 ; MULTIPLICAND

]MLIER EQU VARTAB+8 ; MULTIPLIER

]PBAK EQU VARTAB+10 ; PRODUCT BACKUP

]XLEN EQU VARTAB+12 ; X-DIMENSION LENGTH

]YLEN EQU VARTAB+14 ; Y-DIMENSION LENGTH

]PROD EQU VARTAB+16 ; PRODUCT OF MULTIPLICATION

\*

APUT162

LDY #4 ; LOAD BYTE 4 OF ARRAY

LDA (]ADDRD),Y ; HEADER TO GET ELEMENT LENGTH

STA ]ESIZE

LDY #0 ; LOAD BYTE 0 TO GET

LDA (]ADDRD),Y ; X-DIMENSION LENGTH LOBYTE

STA ]XLEN

LDY #1 ; LOAD BYTE 1 TO GET

LDA (]ADDRD),Y ; X-DIMENSION LENGTH HIBYTE

STA ]XLEN+1

LDY #2 ; LOAD BYTE 2 TO GET THE

LDA (]ADDRD),Y ; Y-DIMENSION LENGTH LOBYTE

STA ]YLEN

LDY #3 ; LOAD BYTE 3 TO GET THE

LDA (]ADDRD),Y ; Y-DIMENSION LENGTH HIBYTE

STA ]YLEN+1

LDY #0 ; RESET BYTE INDEX

\*

\*\* MULTIPLY Y-INDEX BY Y-LENGTH

\*

LDA ]YIDX ; LOAD Y-INDEX LOBYTE

STA ]MLIER ; PUT IN MULTIPLIER LOBYTE

LDA ]YIDX+1 ; DO SAME FOR HIBYTES

STA ]MLIER+1

LDA ]YLEN ; PUT Y-DIM LENGTH LOBYTE

STA ]MCAND ; INTO MULTIPLICAND

LDA ]YLEN+1 ; DO SAME FOR HIBYTE

STA ]MCAND+1

LDA #00 ; CLEAR PRODUCT BYTES

STA ]PROD

STA ]PROD+1

STA ]PROD+2

STA ]PROD+3

LDX #$10 ; INIT COUNTER TO #16

:SHIFT\_R

LSR ]MLIER+1 ; DIVIDE MULTIPLIER HIBYTE BY 2

ROR ]MLIER ; ADJUST LOBYTE

BCC :ROT\_R ; IF 0 PUT IN CARRY, ROTATE PRODUCT

LDA ]PROD+2 ; LOAD PRODUCT 3RD BYTE

CLC ; CLEAR CARRY

ADC ]MCAND ; ADD MULTIPLICAND

STA ]PROD+2 ; STORE 3RD BYTE

LDA ]PROD+3 ; LOAD PRODUCT HIBYTE

ADC ]MCAND+1 ; ADD MULTIPLICAND HIBYTE

:ROT\_R

ROR ; ROTATE .A RIGHT

STA ]PROD+3 ; STORE IN PRODUCT HIBYTE

ROR ]PROD+2 ; ROTATE 3RD BYTE

ROR ]PROD+1 ; ROTATE 2ND

ROR ]PROD ; ROTATE LOBYTE

DEX ; DECREASE X COUNTER

BNE :SHIFT\_R ; IF NOT ZERO, LOOP AGAIN

\*

\*\* NOW MULTIPLY XIDX BY ELEMENT SIZE

\*

LDA ]PROD ; BACKUP PREVIOUS

STA ]PBAK ; PRODUCT FOR USE LATER

LDA ]PROD+1 ; DO SAME FOR HIBYTE

STA ]PBAK+1

LDA ]XIDX ; PUT X-INDEX LOBYTE

STA ]MLIER ; INTO MULTIPLIER

LDA ]XIDX+1 ; DO SAME FOR HIBYTE

STA ]MLIER+1

LDA ]ESIZE ; PUT ELEMENT SIZE

STA ]MCAND ; INTO MULTIPLICAND

LDA #0 ; CLEAR MULTIPLICAND HIBYTE

STA ]MCAND+1

\*

STA ]PROD ; CLEAR PRODUCT

STA ]PROD+1

STA ]PROD+2

STA ]PROD+3

LDX #$10 ; INIT X COUNTER TO #16

:SHIFTR LSR ]MLIER+1 ; DIVIDE MULTIPLIER BY 2

ROR ]MLIER ; ADJUST LOBYTE

BCC :ROTR ; IF 0 PUT INTO CARRY, ROTATE PROD

LDA ]PROD+2 ; LOAD PRODUCT 3RD BYTE

CLC ; CLEAR CARRY

ADC ]MCAND ; ADD MULTIPLICAND LOBYTE

STA ]PROD+2

LDA ]PROD+3 ; LOAD PRODUCT HIBYTE

ADC ]MCAND+1 ; HAD MULTIPLICAND HIBYTE

:ROTR

ROR ; ROTATE .A RIGHT

STA ]PROD+3 ; STORE PRODUCT HIBYTE

ROR ]PROD+2 ; ROTATE 3RD BYTE

ROR ]PROD+1 ; ROTATE 2ND BYTE

ROR ]PROD ; ROTATE LOBYTE

DEX ; DECREASE X COUNTER

BNE :SHIFTR ; IF NOT 0, KEEP LOOPING

\*

\*\* NOW ADD X \* ESIZE TO RUNNING PRODUCT

\*

CLC ; CLEAR CARRY

LDA ]PROD ; ADD CURRENT PRODUCT

ADC ]PBAK ; TO PREVIOUS PRODUCT

STA ]PROD ; AND STORE BACK IN PRODUCT

LDA ]PROD+1

ADC ]PBAK+1

STA ]PROD+1

CLC ; CLEAR CARRY

LDA ]PROD ; INCREASE LOBYTE BY 5

ADC #5 ; TO ACCOUNT FOR ARRAY

STA ]PROD ; HEADER

LDA ]PROD+1

ADC #0 ; ADJUST HIBYTE

STA ]PROD+1

\*

\*\* ADD ARRAY ADDRESS TO GET INDEX

\*

CLC ; CLEAR CARRY

LDA ]PROD ; ADD ARRAY ADDRESS

ADC ]ADDRD ; TO PRODUCT TO GET

STA ADDR2 ; ELEMENT ADDRESS; STORE

LDA ]PROD+1 ; ADDRESS ON ZERO PAGE

ADC ]ADDRD+1

STA ADDR2+1

LDY #0 ; RESET BYTE INDEX

:CLP

LDA (]ADDRS),Y ; LOAD BYTE FROM SOURCE

STA (ADDR2),Y ; STORE AT ELEMENT ADDRESS

INY ; INCREASE BYTE INDEX

CPY ]ESIZE ; IF != ELEMENT LENGTH, LOOP

BNE :CLP

LDY ADDR2+1 ; .Y = ELEMENT ADDRESS HIBYTE

LDX ADDR2 ; .X = ELEMENT ADDRESS LOBYTE

LDA ]ESIZE ; .A = ELEMENT LENGTH

RTS

**DEMO.ARRAYS**

DEMO.ARRAYS can be assembled into a program that illustrates how each macro works. This is not, however, an exhaustive test; for more complicated usage, see the integrated demos.

\*

\*``````````````````````````````\*

\* DEMO.ARRAYS \*

\* \*

\* A DECIDEDLY NON-EXHAUSTIVE \*

\* DEMO OF ARRAY FUNCTIONALITY \*

\* IN THE APPLEIIASM LIBRARY. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 14-JUL-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\*\* ASSEMBLER DIRECTIVES

\*

CYC AVE

EXP OFF

TR ON

DSK DEMO.ARRAYS

OBJ $BFE0

ORG $6000

\*

\*``````````````````````````````\*

\* TOP INCLUDES (HOOKS,MACROS) \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PUT MIN.HEAD.REQUIRED

USE MIN.MAC.REQUIRED

USE MIN.MAC.ARRAYS

PUT MIN.HOOKS.ARRAYS

\*

\*``````````````````````````````\*

\* PROGRAM MAIN BODY \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]VAR1 EQU $300

]COUNT1 EQU $320

]ARRAY1 EQU $4000

]ARRAY2 EQU $5000

]HOME EQU $FC58

\*

JSR ]HOME

\_PRN "1D AND 2D 8BIT/16BIT ARRAYS",8D

\_PRN "===========================",8D8D

\_PRN "THIS MACRO LIBRARY AND VARIOUS",8D

\_PRN "SUBROUTINES ARE USED FOR THE CREATION,",8D

\_PRN "ACCESS AND MANAGEMENT OF ARRAYS THAT",8D

\_PRN "CAN BE EITHER ONE OR TWO DIMENSIONS",8D

\_PRN "AND CAN HAVE EITHER 255 ELEMENTS PER",8D

\_PRN "DIMENSION IN THE CASE OF 8BIT ARRAYS OR",8D

\_PRN "UP TO 65,530 ELEMENTS IN THE CASE OF",8D

\_PRN "16BIT ARRAYS--AT LEAST, THEORETICALLY.",8D

\_PRN "SINCE THAT WOULD TAKE UP THE ENTIRETY",8D

\_PRN "OF RAM ON MOST APPLE ][ COMPUTERS,",8D

\_PRN "HAVING THAT MANY ELEMENTS IS NOT LIKELY.",8D8D

\_WAIT

JSR ]HOME

\_PRN "AT LEAST IN THIS LIBRARY, ARRAYS",8D

\_PRN "ARE FAIRLY SIMPLE DATA STRUCTURES.",8D

\_PRN "EVERY ARRAY HAS A HEADER THAT SPECIFIES",8D

\_PRN "THE NUMBER OF ELEMENTS PER DIMENSION",8D

\_PRN "AS WELL AS THE LENGTH OF EACH ELEMENT.",8D

\_PRN "THESE ARE SET WITH THE DIM MACROS AND",8D

\_PRN "SUBROUTINES:",8D8D

\_PRN "DIM81: INIT 1-DIMENSIONAL 8BIT ARRAY",8D

\_PRN "DIM82: INIT 2-DIMENSIONAL 8BIT ARRAY",8D

\_PRN "DIM161: INIT 1-DIMENSIONAL 16BIT ARRAY",8D

\_PRN "DIM162: INIT 2-DIMENSIONAL 16BIT ARRAY",8D8D

\_WAIT

\_PRN "IF YOU NEED FEWER THAN 255 ELEMENTS",8D

\_PRN "IN A DIMENSION, I HIGHLY SUGGEST",8D

\_PRN "USING THE 8BIT ARRAY MACROS AND,",8D

\_PRN "SUBROUTINES, AS THERE IS A SIGNIFICANT",8D

\_PRN "SAVING OF BYTES AND CPU CYCLES.",8D

\_WAIT

JSR ]HOME

\_PRN "LIKE THE DIM MACROS, EACH ARRAY",8D

\_PRN "TYPE ALSO HAS A GET AND PUT MACRO AND",8D

\_PRN "SET OF SUBROUTINES DEDICATED TO IT:",8D8D

\_WAIT

\_PRN "GET81: RETRIEVE THE DATA IN A GIVEN",8D

\_PRN " ELEMENT AND PUT IN RETURN.",8D

\_PRN "GET82: RETRIEVE DATA FROM ELEMENT AT",8D

\_PRN " X,Y AND PUT IN RETURN.",8D

\_PRN "GET161: GET DATA FROM 16-BIT ELEMENT",8D

\_PRN " AND PUT IN RETURN.",8D

\_PRN "GET162: GET DATA FROM ELEMENT AT 16BIT",8D

\_PRN " X,Y LOCATION AND PUT IN RETURN.",8D8D

\_WAIT

\_PRN "PUT81: PUT DATA FROM SOURCE LOCATION IN",8D

\_PRN " AN ARRAY'S ELEMENT.",8D

\_PRN "PUT82: PUT DATA FROM SOURCE ADDRESS IN",8D

\_PRN " ARRAY ELEMENT AT X,Y.",8D

\_PRN "PUT161: PUT DATA FROM SOURCE ADDRESS IN",8D

\_PRN " 16-BIT ARRAY ELEMENT.",8D

\_PRN "PUT162: PUT DATA FROM SOURCE INTO 16BIT",8D

\_PRN " ARRAY ELEMENT AT X,Y.",8D8D

\_WAIT

\*

JSR ]HOME

\_PRN "ONE-DIMENSIONAL, 8-BIT ARRAYS",8D

\_PRN "=============================",8D8D

\_PRN "DIM81, GET81, AND PU81 ARE USED FOR",8D

\_PRN "1D ARRAYS THAT DON'T NEED MORE THAN",8D

\_PRN "A SINGLE DIMENSION OF LESS THAN 255",8D

\_PRN "ELEMENTS. FOR MANY USES, THIS SUFFICES;",8D

\_PRN "THE FACT THAT THE APPLE ][ IS AN 8-BIT",8D

\_PRN "COMPUTER ATTESTS TO THIS FACT.",8D8D

\_WAIT

\_PRN "HOWEVER, THERE ARE A NUMBER OF CASES ",8D

\_PRN "IN WHICH 8-BIT INDEXING ISN'T ENOUGH.",8D

\_PRN "AGAIN, MAKE THE CHOICE BASED ON NEED,",8D

\_PRN "NOT CONVENIENCE. IF 255 ELEMENTS IS",8D

\_PRN "ENOUGH TO ACCOMPLISH THE TASK, USE ",8D

\_PRN "THESE MACROS AND SUBROUTINES.",8D8D

\_WAIT

JSR ]HOME

\_PRN "EIGHT BITS AND ONE DIMENSION: DIM",8D

\_PRN "=================================",8D8D

\_PRN "THE DIM81 MACRO CREATES A THREE",8D

\_PRN "BYTE HEADER THAT HOLDS, IN ORDER:",8D8D

\_PRN "BYTE 0: NUMBER OF ELEMENTS",8D

\_PRN "BYTE 1: ELEMENT SIZE",8D8D

\_PRN "THE GET81 AND PUT81 ROUTINES USE ",8D

\_PRN "THIS HEADER TO KNOW HOW MUCH DATA",8D

\_PRN "TO READ AND WRITE FROM AN ELEMENT.",8D

\_PRN "BASIC CHECKS AGAINST THE INTENDED",8D

\_PRN "NUMBER OF ELEMENTS CAN ALSO BE DONE",8D

\_PRN "USING THIS HEADER.",8D8D

\_WAIT

\_PRN " DIM81 #ARRAY1;#10;#2;#$FF",8D8D

\_PRN "CREATES AN 8BIT, 1D ARRAY AT THE",8D

\_PRN "ADDRESS OF #ARRAY1 WITH TEN ELEMENTS",8D

\_PRN "OF 2 BYTES EACH. ALL ELEMENTS ARE",8D

\_PRN "FILLED WITH THE LAS PARAMETER, $FF."

\_WAIT

JSR ]HOME

\_PRN "WE CAN DUMP #ARRAY1 BEFORE AND",8D

\_PRN "AFTER USING DIM81 TO SHOW THE",8D

\_PRN "DIFFERENCE:",8D8D

\_PRN "BEFORE:",8D8D

DUMP #]ARRAY1;#2

DUMP #]ARRAY1+2;#10

DUMP #]ARRAY1+12;#10

\_PRN " ",8D8D

\_WAIT

DIM81 #]ARRAY1;#10;#2;#$FF

\_PRN "AFTER:",8D8D

DUMP #]ARRAY1;#2

DUMP #]ARRAY1+2;#10

DUMP #]ARRAY1+12;#10

\_WAIT

JSR ]HOME

\_PRN "8 BITS AND ONE DIMENSION: PUT",8D

\_PRN "=============================",8D8D

\_PRN "THE PUT81 MACRO PUTS THE DATA FROM",8D

\_PRN "A SOURCE ADDRESS INTO AN 8BIT, 1D",8D

\_PRN "ARRAY ELEMENT. THE SOURCE ADDRESS,",8D

\_PRN "ARRAY ADDRESS AND THE ELEMENT NUMBER",8D

\_PRN "ARE SPECIFIED AS PARAMETERS, IN",8D

\_PRN "THAT ORDER. NOTE THAT THE NUMBER OF",8D

\_PRN "BYTES TO COPY INTO THE ELEMENT IS",8D

\_PRN "PREDETERMINED BY THE ELEMENT SIZE",8D

\_PRN "SET BY DIM81 IN THE HEADER.",8D8D

\_PRN "THUS:",8D8D

\_WAIT

\_PRN " LDA #0",8D

\_PRN " STA ]VAR1",8D

\_PRN " STA ]VAR1+1",8D

\_PRN " PUT81 #]VAR1;#ARRAY1;#3",8D8D

\_PRN "WILL PUT $0000 IN ARRAY1'S ",8D

\_PRN "ELEMENT 3, WHICH IS TECHNICALLY THE",8D

\_PRN "FOURTH ELEMENT DUE TO ZERO INDEXING."

LDA #0

STA ]VAR1

STA ]VAR1+1

PUT81 #]VAR1;#]ARRAY1;#3

\_WAIT

JSR ]HOME

\_PRN "IF WE DUMP THE ARRAY AGAIN, WE ",8D

\_PRN "CAN READILY SEE THE CHANGE:",8D8D

\_WAIT

DUMP #]ARRAY1;#2

DUMP #]ARRAY1+2;#10

DUMP #]ARRAY1+12;#10

\_WAIT

\_PRN " ",8D8D

\_PRN "OF COURSE, THIS IS OF LIMITED",8D

\_PRN "USE WITHOUT A FUNCTION TO EXTRACT",8D

\_PRN "THE ELEMENT INA USEFUL FASHION--",8D

\_PRN "RELYING ON THE DUMP MACRO ONLY GOES",8D

\_PRN "SO FAR. THAT'S WHERE OUR THIRD MACRO",8D

\_PRN "AND SUBROUTINE COMES IN..."

\_WAIT

JSR ]HOME

\_PRN "8-BIT, 1-DIMENSION ARRAYS: GET",8D

\_PRN "==============================",8D8D

\_PRN "THE GET81 MACRO GETS THE DATA",8D

\_PRN "STORED IN AN ELEMENT AND COPIES IT",8D

\_PRN "TO RETURN, STORING THE ELEMENT",8D

\_PRN "LENGTH IN RETLEN. THIS ALLOWS YOU",8D

\_PRN "TO USE THE ARRAY..WELL, LIKE AN",8D

\_PRN "ARRAY. SO:",8D8D

\_WAIT

\_PRN " GET81 #ARRAY1;#3",8D8D

\_PRN "RETRIEVES ELEMENT 3 OF ARRAY1 AND",8D

\_PRN "STORES IT IN RETURN FOR USE BY YOUR",8D

\_PRN "PROGRAM. WE CAN DUMP RETURN BEFORE",8D

\_PRN "AND AFTER USING GET81 TO SHOW",8D

\_PRN "THE DIFFERENCE:",8D8D

\_WAIT

\_PRN "BEFORE:",8D

DUMP #RETURN;RETLEN

\_WAIT

\_PRN " ",8D8D

\_PRN "AFTER: ",8D

GET81 #]ARRAY1;#3

DUMP #RETURN;RETLEN

\_WAIT

JSR ]HOME

\_PRN "16-BITS AND ONE DIMENSION: DIM161",8D

\_PRN "=================================",8D8D

\_PRN "DIM161 WORKS IN FORM AND FUNCTION JUST",8D

\_PRN "AS DIM81 DOES, EXCEPT IT ACCEPTS",8D

\_PRN "A TWO-BYTE VALUE FOR THE NUMBER",8D

\_PRN "OF ELEMENTS. BECAUSE OF THIS, THE ARRAY",8D

\_PRN "HEADER CREATED IS THREE BYTES INSTEAD",8D

\_PRN "OF THE TWO IN 8-BIT ARRAYS. SO:",8D8D

\_WAIT

\_PRN " DIM161 #ARRAY1;#300;#2;#$00",8D8D

\_PRN "WILL INITIALIZE AN ARRAY WITH 0..300",8D

\_PRN "ELEMENTS, ONE DIMENSION. AGAIN, THIS",8D

\_PRN "CAN TECHNICALLY USE A BIT MORE THAN",8D

\_PRN "65,000 ELEMENTS, BUT THIS IS BEYOND",8D

\_PRN "IMPRACTICAL FOR THE PURPOSES OF THIS",8D

\_PRN "LIBRARY, AS A CONSECUTIVE 64K OF BYTES",8D

\_PRN "IS UNLIKELY IN MOST APPLE II SYSTEMS.",8D8D

\_WAIT

DIM161 #]ARRAY1;#300;#2;#$00

JSR ]HOME

\_PRN "16-BITS AND ONE DIMENSION: PUT",8D

\_PRN "==============================",8D8D

\_PRN "NOW THAT WE HAVE CREATED OUR ARRAY,",8D

\_PRN "WE CAN USE PUT161 TO CHANGE THE DATA",8D

\_PRN "IN EACH ELEMENT. AGAIN, THIS WORKS",8D

\_PRN "EXACTLY LIKE PUT81, BUT WITH SOME",8D

\_PRN "EXTRA BYTES HERE AND THERE TO ACCOUNT",8D

\_PRN "FOR THE EXTRA BREADTH. LET'S FILL",8D

\_PRN "EACH ELEMENT 0..300 WITH ITS OWN VALUE--",8D

\_PRN "THAT IS, 0 WILL HOLD 0, 1 WILL HOLD 1,",8D

\_PRN "299 WILL HOLD 2999 AND 300 WILL HOLD",8D

\_PRN "300:",8D8D

\_WAIT

\_PRN " LDA #0",8D

\_PRN " STA ]COUNT",8D

\_PRN " STA ]COUNT+1",8D

\_PRN " TAX",8D

\_PRN " TAY",8D

\_PRN "LP ",8D

\_PRN " PUT161 #]COUNT'#]ARRAY1;]COUNT",8D

\_PRN " LDA ]COUNT",8D

\_PRN " CLC",8D

\_PRN " ADC #1",8D

\_PRN " STA ]COUNT",8D

\_PRN " LDA ]COUNT+1",8D

\_PRN " ADC #0",8D

\_PRN " STA ]COUNT+1",8D

\_PRN " CMP #$01",8D

\_PRN " BNE LP",8D

\_PRN " LDA ]COUNT",8D

\_PRN " CMP #$2C",8D

\_PRN " BNE LP"

\_WAIT

\*

LDA #0

STA ]COUNT1

STA ]COUNT1+1

TAX

TAY

LP161

PUT161 #]COUNT1;#]ARRAY1;]COUNT1

LDA ]COUNT1

DUMP #]COUNT1;#2

LDA ]COUNT1

CLC

ADC #1

STA ]COUNT1

LDA ]COUNT1+1

ADC #0

STA ]COUNT1+1

CMP #$01

BNE LP161

LDA ]COUNT1

CMP #$2D

BNE LP161

\_WAIT

\*

JSR ]HOME

\_PRN "WE CAN NOW DUMP THE ENTIRE ARRAY",8D

\_PRN "TO SEE HOW EACH ELEMENT IS STORED,"

\_PRN "ALONG WITH THE THREE BYTE HEADER:",8D8D

\_WAIT

DUMP #]ARRAY1;#3

\_WAIT

DUMP #]ARRAY1+3;#60

\_WAIT

DUMP #]ARRAY1+63;#60

\_WAIT

DUMP #]ARRAY1+123;#60

\_WAIT

DUMP #]ARRAY1+183;#60

\_WAIT

DUMP #]ARRAY1+243;#60

\_WAIT

DUMP #]ARRAY1+303;#60

\_WAIT

DUMP #]ARRAY1+363;#60

\_WAIT

DUMP #]ARRAY1+423;#60

\_WAIT

DUMP #]ARRAY1+483;#60

\_WAIT

DUMP #]ARRAY1+543;#64

\_PRN " ",8D8D

\_PRN "WELL THAT CERTAINLY WAS A DUMP...",8D8D

\_WAIT

JSR ]HOME

\_PRN "16-BITS IN ONE DIMENSION: GET",8D

\_PRN "=============================",8D8D

\_PRN "AND OF COURSE, WE HAVE THE SAME GET",8D

\_PRN "MACRO FOR 16-BIT, 1D ARRAYS, GET162. THIS",8D

\_PRN "AGAIN FUNCTIONS THE SAME AS ITS 8-BIT",8D

\_PRN "COUNTERPART, EXCEPT THE INDEX IS TWO ",8D

\_PRN "BYTES RATHER THAN ONE.",8D8D

\_PRN " ",8D8D

\_PRN "THUS:",8D8D

\_WAIT

\_PRN " GET161 #]ARRAY1;#270",8D8D

\_PRN "RETURNS: "

GET161 #]ARRAY1;#270

DUMP #RETURN;RETLEN

\_WAIT

JSR ]HOME

\_PRN "8-BIT, 2D ARRAYS: FML ANOTHER DIM",8D

\_PRN "=================================",8D8D

\_PRN "AT THIS POINT, YOU SHOULD HAVE A",8D

\_PRN "GOOD GRASP AS TO HOW ARRAYS WORK",8D

\_PRN "IN THIS LIBRARY. TWO-DIMENSIONAL",8D

\_PRN "ARRAYS DO NOT SIGNIFICANTLY DIFFER",8D

\_PRN "FROM ONE-DIMENSIONAL ARRAYS; IT JUST",8D

\_PRN "MEANS THAT AN EXTRA ELEMENT INDEX IS",8D

\_PRN "NEEDED AS A PARAMETER. AS SUCH, WE CAN",8D

\_PRN "MOSTLY BREEZE THROUGH THE REST OF THESE.",8D8D

\_WAIT

\_PRN "TO INITIALIZE A 2D, 8BIT ARRAY:",8D8D

\_PRN " DIM82 #ARRAY1;#10;#10;#1;#00",8D8D

\_PRN "THIS CREATES AN ARRAY OF TEN BY TEN",8D

\_PRN "ELEMENTS (TOTAL OF 100 ELEMENTS) WITH ",8D

\_PRN "A LENGTH OF ONE BYTE. EACH ELEMENT",8D

\_PRN "IS INITIALIZED TO A VALUE OF 0."

\_WAIT

DIM82 #]ARRAY1;#10;#10;#1;#0

JSR ]HOME

\_PRN "NOTE THAT WE HAVE A LONGER HEADER",8D

\_PRN "THANKS TO THE EXTRA ELEMENT INDEX. THE",8D

\_PRN "HEADER CONTAINS THE X-DIMENSION AS ",8D

\_PRN "BYTE ZERO, Y-DIMENSION AS BYTE ONE,",8D

\_PRN "AND ELEMENT LENGTH AS BYTE TWO, AS SUCH:",8D8D

DUMP #]ARRAY1;#3

\_WAIT

\_PRN " ",8D8D

\_PRN "AND THE REST OF THE ARRAY:",8D8D

DUMP #]ARRAY1+3;#10

DUMP #]ARRAY1+13;#10

DUMP #]ARRAY1+23;#10

DUMP #]ARRAY1+33;#10

DUMP #]ARRAY1+43;#10

DUMP #]ARRAY1+53;#10

DUMP #]ARRAY1+63;#10

DUMP #]ARRAY1+73;#10

DUMP #]ARRAY1+83;#10

DUMP #]ARRAY1+93;#10

\_WAIT

JSR ]HOME

\_PRN "8-BIT, 2-DIMENSIONAL ARRAYS: PUT, GET",8D

\_PRN "=====================================",8D8D

\_PRN "AND OF COURSE, JUST AS WITH 1D ARRAYS",8D

\_PRN "WE CAN USE PUT82 AND GET82 TO WRITE",8D

\_PRN "TO AND READ FROM THE ARRAY:",8D8D

\_WAIT

\_PRN " LDA #$FF",8D

\_PRN " STA ]VAR1",8D

\_PRN " PUT82 #]VAR1;#]ARRAY1;#4;#5",8D

\_PRN " GET82 #]ARRAY1;#4;#5",8D

\_PRN " DUMP #RETURN;RETLEN",8D8D

\_PRN "PRODUCES:",8D8D

\_WAIT

LDA #$FF

STA ]VAR1

PUT82 #]VAR1;#]ARRAY1;#4;#5

GET82 #]ARRAY1;#4;#5

DUMP #RETURN;RETLEN

\_WAIT

JSR ]HOME

\_PRN "16-BIT 2D ARRAYS: DIM, GET, PUT",8D

\_PRN "===============================",8D8D

\_PRN "AND LASTLY, WE CAN USE 16-BIT, TWO-",8D

\_PRN "DIMENSIONAL ARRAYS VIA THE DIM162,",8D

\_PRN "PUT162, AND GET162 MACROS:",8D8D

\_PRN " DIM162 #]ARRAY1;#300;#300;#1;#$00",8D

\_PRN " PUT162 #]VAR1;#]ARRAY1;#280;#280",8D

\_PRN " GET162 #]ARRAY1;#280;#280",8D

\_PRN " DUMP #RETURN;RETLEN",8D8D

\_PRN "PRODUCES:",8D8D

\_WAIT

DIM162 #]ARRAY1;#300;#2;#1;#$00

PUT162 #]VAR1;#]ARRAY1;#280;#1

GET162 #]ARRAY1;#280;#1

DUMP #RETURN;RETLEN

\_WAIT

JSR ]HOME

\_PRN " ",8D8D

\_PRN "FIN.",8D8D8D

\*

JMP REENTRY

\*

\*``````````````````````````````\*

\* BOTTOM INCLUDES (ROUTINES) \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PUT MIN.LIB.REQUIRED

\*

\*\* INDIVIDUAL SUBROUTINE INCLUDES

\*

\*\* 8-BIT 1-DIMENSIONAL ARRAY SUBROUTINES

\*

PUT MIN.SUB.ADIM81

PUT MIN.SUB.AGET81

PUT MIN.SUB.APUT81

\*

\*\* 8-BIT 2-DIMENSIONAL ARRAY SUBROUTINES

\*

PUT MIN.SUB.ADIM82

PUT MIN.SUB.AGET82

PUT MIN.SUB.APUT82

\*

\*\* 16-BIT 1-DIMENSIONAL ARRAYS

\*

PUT MIN.SUB.ADIM161

PUT MIN.SUB.APUT161

PUT MIN.SUB.AGET161

\*

\*\* 16-BIT 2-DIMENSIONAL ARRAYS

\*

PUT MIN.SUB.ADIM162

PUT MIN.SUB.APUT162

PUT MIN.SUB.AGET162

**Disk 4: MATH**

The fourth disk in the AppleIIAsm library contains macros and subroutines dedicated to 8-bit and 16-bit integer math. Additionally, hooks are provided to the various floating-point routine addresses built into Applesoft—but this should only be used when absolutely necessary, as these are substantially slower. It should also be noted that these routines are currently written to handle unsigned values, though in some cases signed values will work as well.

In the future, fixed-point mathematics routines will also be included here.

The disk contains the following:

* HOOKS.MATH
* MAC.MATH
* DEMO.MATH
* SUB.ADDIT16
* SUB.COMP16
* SUB.DIVD16
* SUB.DIVD8
* SUB.MULT16
* SUB.MULT8
* SUB.RAND16
* SUB.RAND8
* SUB.RANDB
* SUB.SUBT16

**HOOKS.MATH**

The HOOKS.MATH file contains various hooks useful to mathematical functions. Most of these are related to floating-point operations, which are built into Applesoft.

\*

\*``````````````````````````````\*

\* HOOKS.MATH \*

\* \*

\* THIS HOOKS FILE CONTAINS \*

\* HOOKS TO VARIOUS ROUTINES \*

\* RELATED TO MATHEMATICS. IN \*

\* PARTICULAR, WOZNIAK'S \*

\* FLOATING-POINT ALGORITHMS \*

\* ARE POINTED TO HERE, IF \*

\* INTEGER MATH IS NOT ENOUGH \*

\* FOR THE TASK AT HAND. \*

\* \*

\* NOTE THAT UNLESS ABSOLUTELY \*

\* NECESSARY, YOU SHOULD USE \*

\* THE INTEGER MATH ROUTINES, \*

\* AS THEY ARE MUCH FASTER. IN \*

\* THE FUTURE, FIXED-POINT MATH \*

\* MAY BE ADDED TO THE LIBRARY \*

\* AS WELL. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 15-JUL-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

GETNUM EQU $FFA7 ; ASCII TO HEX IN 3E & 3F

RNDL EQU $4E ; RANDOM NUMBER LOW

RNDH EQU $4F ; RANDOM NUMBER HIGH

\*

FAC EQU $9D ; FLOATING POINT ACCUM

FSUB EQU $E7A7 ; FLOATING POINT SUBTRACT

FADD EQU $E7BE

FMULT EQU $E97F ; FP MULTIPLY

FDIV EQU $EA66 ; FP DIVIDE

FMULTT EQU $E982

FDIVT EQU $EA69

FADDT EQU $E7C1

FSUBT EQU $E7AA

\*

MOVFM EQU $EAF9 ; MOVE FAC > MEM

MOVMF EQU $EB2B ; MOVE MEM > FAC

NORM EQU $E82E

CONUPK EQU $E9E3

\*

FLOG EQU $E941 ; LOGARITHM

FSQR EQU $EE8D ; SQUARE ROOT

FCOS EQU $EFEA ; FP COSINE

FSIN EQU $EFF1 ; SINE

FTAN EQU $F03A ; TANGENT

FATN EQU $F09E ; ATANGENT

\*

**MAC.MATH**

MAC.MATH contains all of the macros related to 8-bit and 16-bit integer math, as well as macros related to pseudo-random number generation. It contains the following macros:

* ADD8
* SUB8
* ADD16
* SUB16
* MUL16
* DIV16
* RAND
* CMP16
* MUL8
* DIV8
* RND16
* RND8

\*

\*``````````````````````````````\*

\* MAC.MATH \*

\* \*

\* THIS FILE CONTAINS ALL OF \*

\* THE INTEGER MATH MACROS. \*

\* GIVEN THAT THERE HAVE BEEN \*

\* 50 YEARS OF OPTIMIZATIONS \*

\* FOR 6502 MATH SUBROUTINES, \*

\* I WON'T BE REINVENTING THE \*

\* WHEEL TOO MUCH HERE. CREDIT \*

\* FOR INSPIRATION (OR JUST \*

\* PLAIN COPYING) IS GIVEN IN \*

\* THE SUBROUTINE FILES. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 15-JUL-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\* \*

\* SUBROUTINE FILES USED \*

\* \*

\* SUB.ADDIT16 \*

\* SUB.COMP16 \*

\* SUB.DIVD16 \*

\* SUB.DIVD8 \*

\* SUB.MULT16 \*

\* SUB.MULT8 \*

\* SUB.RAND16 \*

\* SUB.RAND8 \*

\* SUB.RANDB \*

\* SUB.SUBT16 \*

\* \*

\* LIST OF MACROS \*

\* \*

\* ADD8 : ADD 8BIT NUMBERS \*

\* SUB8 : SUBTRACT 8BIT NUMS \*

\* ADD16 : ADD 16BIT NUMBERS \*

\* SUB16 : SUBTRACT 16BIT NUMS \*

\* MUL16 : MULTIPLY 16BIT NUMS \*

\* DIV16 : DIVIDE 16BIT NUMS \*

\* RNDB : GET RANDOM BETWEEN \*

\* CMP16 : COMPARE 16BIT NUMS \*

\* MUL8 : MULTIPLY 8BIT NUMS \*

\* DIV8 : DIVIDE 8BIT NUMS \*

\* RND16 : RANDOM WORD \*

\* RND8 : RANDOM BYTE \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

**MAC.MATH >> ADD8**

|  |
| --- |
| **ADD8 (macro)**  **Input**:  ]1 = 1st addend (1b)  ]2 = 2nd addend (1b)  **Output**:  **.A** = sum  **RETURN** = sum  **RETLEN** = 1  **Destroys**: AXYNVZCM  **Cycles**: 22+  **Size**: 16 bytes |

The **ADD8** macro adds two 8-bit addends and returns a sum in **.A** as well as in **RETURN**, with **RETLEN** holding the byte-length of 1.

\*

\*``````````````````````````````\*

\* ADD8 (NATHAN RIGGS) \*

\* \*

\* DIRTY MACRO TO ADD TWO BYTES \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = ADDEND 1 \*

\* ]2 = ADDEND 2 \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* ADD8 #3;#4 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

ADD8 MAC

LDA #1

STA RETLEN

LDA ]1

CLC

ADC ]2

STA RETURN

<<<

**MAC.MATH >> SUB8**

|  |
| --- |
| **SUB8 (macro)**  **Input**:  ]1 = minuend (1b)  ]2 = subtrahend (1b)  **Output**:  **.A** = result  **RETURN** = result  **RETLEN** = 1  **Destroys**: AXYNVZCM  **Cycles**: 18+  **Size**: 16 bytes |

The **SUB8** macro subtracts a subtrahend from a minuend and stores the result in **.A** and **RETURN** with the byte-length of 1 in **RETLEN**.

\*

\*``````````````````````````````\*

\* SUB8 (NATHAN RIGGS) \*

\* \*

\* MACRO TO SUBTRACT TWO BYTES \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = MINUEND \*

\* ]2 = SUBTRAHEND \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* SUB8 #4;#3 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

SUB8 MAC

LDA #1

STA RETLEN

LDA ]1

SEC

SBC ]2

STA RETURN

<<<

**MAC.MATH >> ADD16**

|  |
| --- |
| **ADD16 (macro)**  **Input**:  ]1 = 1st addend (2b)  ]2 = 2nd addend (2b)  **Output**:  **.A** = sum low byte  **.X** = sum high byte  **RETURN** = sum  **RETLEN** = 2  **Destroys**: AXYNVZCM  **Cycles**: 83+  **Size**: 72 bytes |

The **ADD16** macro adds two 16-bit values and returns a 16-bit sum in **.A** (low byte) and **.X** (high byte). This is additionally stored in **RETURN**, with a **RETLEN** of 2. Note that if the sum is greater than a 16-bit value, only the lowest two bytes are returned.

\*

\*``````````````````````````````\*

\* ADD16 (NATHAN RIGGS) \*

\* \*

\* ADD TWO 16BIT VALUES, STORE \*

\* RESULT IN A, X (LOW, HIGH) \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = ADDEND 1 \*

\* ]2 = ADDEND 2 \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* ADD16 #3000;#4000 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

ADD16 MAC

\_MLIT ]1;WPAR1

\_MLIT ]2;WPAR2

JSR ADDIT16

<<<

**MAC.MATH >> SUB16**

|  |
| --- |
| **SUB16 (macro)**  **Input**:  ]1 = minuend (2b)  ]2 = subtrahend (2b)  **Output**:  **.A** = result low byte  **.X** = result high byte  **RETURN** = result  **RETLEN** = 2  **Destroys**: AXYNVZCM  **Cycles**: 69+  **Size**: 61 bytes |

The **SUB16** macro subtracts a 16-bit value subtrahend from a 16-bit value minuend, returning the result in **.A** (low byte) and **.X** (high byte). This result is also stored in **RETURN**, with a **RETLEN** of 2.

\*

\*``````````````````````````````\*

\* SUB16 (NATHAN RIGGS) \*

\* \*

\* SUBTRACTS ONE 16BIT INTEGER \*

\* FROM ANOTHER, STORING THE \*

\* RESULT IN A,X (LOW, HIGH) \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = MINUEND \*

\* ]2 = SUBTRAHEND \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* SUB16 #2000;#1500 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

SUB16 MAC

\_MLIT ]1;WPAR1

\_MLIT ]2;WPAR2

JSR SUBT16

<<<

**MAC.MATH >> MUL16**

|  |
| --- |
| **MUL16 (macro)**  **Input**:  ]1 = multiplicand (2b)  ]2 = multiplier (2b)  **Output**:  **.A** = product low byte  **.X** = product high byte  **RETURN** = product (4b)  **RETLEN** = 4  **Destroys**: AXYNVZCM  **Cycles**: 141+  **Size**: 109 bytes |

The **MUL16** macro multiplies two 16-bit values and returns the 16-bit product in **.A** (low byte) and **.X** (high byte). Additionally, a 32-bit product is stored in **RETURN** if the larger value is required. Note that this 32-bit value is only correct, however, when the values being multiplied are unsigned.

\*

\*``````````````````````````````\*

\* MUL16 (NATHAN RIGGS) \*

\* \*

\* MULTIPLIES TWO 16BIT NUMBERS \*

\* AND RETURNS THE PRODUCT IN \*

\* A,X (LOW, HIGH). \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = MULTIPLICAND \*

\* ]2 = MULTIPLIER \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* MUL16 #400;#500 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

MUL16 MAC

\_MLIT ]1;WPAR1

\_MLIT ]2;WPAR2

JSR MULT16

<<<

**MAC.MATH >> DIV16**

|  |
| --- |
| **DIV16 (macro)**  **Input**:  ]1 = dividend (2b)  ]2 = divisor (2b)  **Output**:  **.A** = result low byte  **.X** = result high byte  **RETURN** = result (2b)  **RETLEN** = 2  **Destroys**: AXYNVZCM  **Cycles**: 132+  **Size**: 101 bytes |

The **DIV16** macro divides a 16-bit dividend by a 16-bit divisor, returning the result in **.A** (low byte) and **.X** (high byte). The result is also stored in **RETURN** with a 2 byte length.

\*

\*``````````````````````````````\*

\* DIV16 (NATHAN RIGGS) \*

\* \*

\* DIVIDES ONE 16BIT NUMBER BY \*

\* ANOTHER AND RETURNS THE \*

\* RESULT IN A,X (LOW,HIGH). \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = DIVIDEND \*

\* ]2 = DIVISOR \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* DIV16 #3000;#300 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

DIV16 MAC

\_MLIT ]1;WPAR1

\_MLIT ]2;WPAR2

JSR DIVD16 ; UNSIGNED

FIN

<<<

**MAC.MATH >> RAND**

|  |
| --- |
| **RAND (macro)**  **Input**:  ]1 = low boundary (1b)  ]2 = high boundary (1b)  **Output**:  **.A** = pseudorandom value  **RETURN** = pseudorandom  value  **RETLEN** = 1  **Destroys**: AXYNVZCM  **Cycles**: 256+  **Size**: 482 bytes |

The **RAND** macro returns an 8-bit pseudorandom number in **.A** between the given low value and high value. This is also stored in **RETURN**.

\*

\*``````````````````````````````\*

\* RAND (NATHAN RIGGS) \*

\* \*

\* RETURNS A RANDOM NUMBER IN \*

\* REGISTER A THAT IS BETWEEN \*

\* THE LOW AND HIGH BOUNDARIES \*

\* PASSED IN THE PARAMETERS. \*

\* \*

\* NOTE THAT THIS RETURNS A \*

\* BYTE, AND THUS ONLY DEALS \*

\* WITH VALUES BETWEEN 0..255. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = LOW BOUNDARY \*

\* ]2 = HIGH BOUNDARY \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* RNDB #50;#100 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

RAND MAC

LDA ]1 ; LOW

LDX ]2 ; HIGH

JSR RANDB

<<<

**MAC.MATH >> CMP16**

|  |
| --- |
| **CMP16 (macro)**  **Input**:  ]1 = 1st word to compare  ]2 = 2nd word to compare  **Output**:  See description  **Destroys**: AXYNVZCM  **Cycles**: 91+  **Size**: 75 bytes |

The **CMP16** macro compares two 16-bit values and alters the status flags depending on the result of the comparison and whether values are signed or unsigned.

For **unsigned** values, the following flags are set under the given conditions:

* The **Z** flag is set to 1 if both values are equal.
* The **C** flag is set to 0 if the first parameter is greater than the second parameter.
* The **C** flag is set to 1 if the first parameter is less than or equal to the second parameter.

For **signed** values, the following flags are set under the given conditions:

* The **Z** flag is set to 1 if the both values are equal.
* The **N** flag is set to 1 if the first parameter is greater than the second parameter.
* The **N** flag is set to 0 if the first parameter is less than or equal to the second parameter.

\*

\*``````````````````````````````\*

\* CMP16 (NATHAN RIGGS) \*

\* \*

\* COMPARES TWO 16BIT VALUES \*

\* AND ALTERS THE P-REGISTER \*

\* ACCORDINGLY (FLAGS). \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = WORD 1 TO COMPARE \*

\* ]2 = WORD 2 TO COMPARE \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* CMP16 #1023;#3021 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

CMP16 MAC

\_MLIT ]1;WPAR1

\_MLIT ]2;WPAR2

JSR COMP16

<<<

**MAC.MATH >> MUL8**

|  |
| --- |
| **MUL8 (macro)**  **Input**:  ]1 = multiplicand (1b)  ]2 = multiplier (1b)  **Output**:  **.A** = product low byte  **.X** = product high byte  **RETURN** = product  **RETLEN** = 2  **Destroys**: AXYNVZCM  **Cycles**: 89+  **Size**: 53 bytes |

The **MUL8** macro multiplies two 8-bit values and returns a 16-bit product in **.A** (low byte) and **.X** (high byte). The product is also stored in **RETURN**.

\*

\*``````````````````````````````\*

\* MUL8 (NATHAN RIGGS) \*

\* \*

\* MULTIPLIES TWO 8BIT VALUES \*

\* AND RETURNS A 16BIT RESULT \*

\* IN A,X (LOW, HIGH). \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = MULTIPLICAND \*

\* ]2 = MULTIPLIER \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* MUL8 #10;#20 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

MUL8 MAC

LDA ]1

LDX ]2

JSR MULT8

<<<

**MAC.MATH >> DIV8**

|  |
| --- |
| **DIV8 (macro)**  **Input**:  ]1 = dividend (1b)  ]2 = divisor (1b)  **Output**:  **.A** = quotient  **.X** = remainder  **Destroys**: AXYNVZCM  **Cycles**: 66+  **Size**: 40 bytes |

The **DIV8** macro divides a first parameter by the second parameter and returns the quotient in **.A** with the remainder returned in **.X**. The quotient is also stored in **RETURN**.

\*

\*``````````````````````````````\*

\* DIV8 (NATHAN RIGGS) \*

\* \*

\* DIVIDES ONE 8BIT NUMBER BY \*

\* ANOTHER AND STORES THE \*

\* QUOTIENT IN A WITH THE \*

\* REMAINDER IN X. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = DIVIDEND \*

\* ]2 = DIVISOR \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* DIV8 #100;#10 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

DIV8 MAC

LDA ]1

LDX ]2

JSR DIVD8

<<<

**MAC.MATH >> RND16**

|  |
| --- |
| **RND16 (macro)**  **Input**:  none  **Output**:  **.A** = value low byte  **.X** = value high byte  **RETURN** = value (2b)  **RETLEN** = 2  **Destroys**: AXYNVZCM  **Cycles**: 96+  **Size**: 64 bytes |

The **RND16** macro returns a 16-bit pseudorandom number (1..65536) in **.A** (low byte) and **.X** (high byte).

\*

\*``````````````````````````````\*

\* RND16 (NATHAN RIGGS) \*

\* \*

\* RETURN A 16-BIT PSEUDORANDOM \*

\* NUMBER. \*

\* \*

\* PARAMETERS \*

\* \*

\* NONE \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* RND16 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

RND16 MAC

JSR RAND16

<<<

**MAC.MATH >> RND8**

|  |
| --- |
| **RND8 (macro)**  **Input**:  none  **Output**:  **.A** = value  **RETURN** = value  **RETLEN** = 1  **Destroys**: AXYNVZCM  **Cycles**: 50+  **Size**: 30 bytes |

The **RND8** macro generates an 8-bit pseudorandom value (1..255) and returns it in **.A**. This value is also held in **RETURN**.

\*

\*``````````````````````````````\*

\* RND8 (NATHAN RIGGS) \*

\* \*

\* RETURN AN 8-BIT PSEUDORANDOM \*

\* NUMBER. \*

\* \*

\* PARAMETERS \*

\* \*

\* NONE \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* RND8 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

RND8 MAC

JSR RAND8

<<<

\*

**SUB.ADDIT16 >> ADDIT16**

|  |
| --- |
| **ADDIT16 (sub)**  **Input**:  **WPAR1** = augend (2 bytes)  **WPAR2** = addend (2 bytes)  **Output**:  **.A** = summand low byte  **.X** = summand high byte  **RETLEN** = byte length (2)  **RETURN** = summand  **Destroys**: AXYNVZCM  **Cycles**: 43+  **Size**: 24 bytes |

The ADDIT16 subroutine adds the two 16-bit numbers held in **WPAR1** and **WPAR2** and stores the result (summand) in **RETURN**. The summand is also held in **.A** (low) and **.X** (high).

\*``````````````````````````````\*

\* ADDIT16 (NATHAN RIGGS) \*

\* \*

\* ADD TWO 16-BIT VALUES. \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = AUGEND \*

\* WPAR2 = ADDEND \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = SUMMAND LOW BYTE \*

\* .X = SUMMAND HIGH BYTE \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^ ^^^ \*

\* \*

\* CYCLES: 43+ \*

\* SIZE: 24 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]ADD1 EQU WPAR1

]ADD2 EQU WPAR2

\*

ADDIT16

LDA #2

STA RETLEN

LDA ]ADD1 ; ADD LOBYTES

CLC ; CLEAR CARRY

ADC ]ADD2

TAY ; TEMPORARY STORE IN .Y

LDA ]ADD1+1 ; ADD HIBYTES

ADC ]ADD2+1

TAX ; STORE IN .X

TYA ; XFER LOBYTE TO .A

STA RETURN

STX RETURN+1

RTS

**SUB.COMP16 >> COMP16**

|  |
| --- |
| **COMP16 (sub)**  **Input**:  **WPAR1** = 1st comparison  **WPAR2** = 2nd comparison  **Output**:  See description  **Destroys**: AXYNVZCM  **Cycles**: 51+  **Size**: 27 bytes |

The COMP16 subroutine provides the functionality of a CMP instruction for 16-bit values. The status flags are set under the following conditions:

* If first operand is equal to the second, then the **zero flag** is set to 1.
* If first unsigned operand is greater than the second unsigned operand, then the **carry flag** is set to zero.
* If the first unsigned operand is less than or equal to the second unsigned operand, then the **carry flag** is set to 1.
* If the first signed operand is greater than the second signed operand, then the **negative flag** is set to 1.
* If the first signed operand is less than or equal to the second signed operand, then the **negative flag** is set to 0.

\*

\*``````````````````````````````\*

\* COMP16 (NATHAN RIGGS) \*

\* \*

\* 16-BIT COMPARISON DIRECTIVE \*

\* \*

\* BASED ON LEVENTHAL AND \*

\* SAVILLE'S /6502 ASSEMBLY \*

\* LANGUAGE ROUTINES/ LISTING \*

\* \*

\* INPUT: \*

\* \*

\* ]WPAR1 = 16-BIT CMP VALUE \*

\* ]WPAR2 = 16-BIT CMP VALUE \*

\* \*

\* OUTPUT: \*

\* \*

\* Z FLAG = 1 IF VALUES EQUAL \*

\* C FLAG = 0 IF CMP1 > CMP2, \*

\* 1 IF CMP1 <= CMP2 \*

\* N FLAG = 1 IF SIGNED CMP1 > \*

\* SIGNED CMP2, 0 IF \*

\* SIGNED CMP1 <= \*

\* SIGNED CMP2 \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^ ^^^^^^^^ \*

\* \*

\* CYCLES: 51+ \*

\* SIZE: 27 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]CMP1 EQU WPAR1 ; COMPARISON VAR 1

]CMP2 EQU WPAR2 ; COMPARISON VAR 2

\*

COMP16

LDA ]CMP1 ; FIRST, COMPARE LOW BYTES

CMP ]CMP2

BEQ :EQUAL ; BRANCH IF EQUAL

LDA ]CMP1+1 ; COMPARE HIGH BYTES

SBC ]CMP2+1 ; SET ZERO FLAG TO 0,

ORA #1 ; SINCE LOW BYTES NOT EQUAL

BVS :OVFLOW ; HANDLE V FLAG FOR SIGNED

RTS

:EQUAL

LDA ]CMP1+1 ; COMPARE HIGH BYTES

SBC ]CMP2+1

BVS :OVFLOW ; HANDLE OVERFLOW FOR SIGNED

RTS

:OVFLOW

EOR #$80 ; COMPLEMENT NEGATIVE FLAG

ORA #1 ; IF OVERFLOW, Z = 0

RTS

**SUB.DIVD16 >> DIVD16**

|  |
| --- |
| **DIVD16 (sub)**  **Input**:  **WPAR1** = dividend (2)  **WPAR2** = divisor (2)  **Output**:  **.A** = result low byte  **.X** = result high byte  **RETURN** = result (2)  **RETLEN** = 2  **Destroys**: AXYNVZCM  **Cycles**: 92+  **Size**: 53 bytes |

The DIVD16 subroutine divides the first 16-bit operand (the dividend) by the second 16-bit operand (the divisor). A 16-bit result is then return in **.A** (low byte) and **.X** (high byte), as well as in the **RETURN** memory location.

\*

\*``````````````````````````````\*

\* DIVD16 (NATHAN RIGGS) \*

\* \*

\* DIVIDE WITH 16-BIT VALUES. \*

\* \*

\* ADAPTED FROM LISTINGS IN THE \*

\* C=64 MAGAZINES. \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = DIVIDEND \*

\* WPAR2 = DIVISOR \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = LOBYTE OF RESULT \*

\* .X = HIBYTE OF RESULT \*

\* RETURN = RESULT (2 BYTES) \*

\* RETLEN = RESULT BYTE LENGTH \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^ ^^^ \*

\* \*

\* CYCLES: 92+ \*

\* SIZE: 53 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]DVEND EQU WPAR1

]DVSOR EQU WPAR2

]REM EQU WPAR3

]RESULT EQU WPAR1

\*

DIVD16

LDA #0 ; RESET REMAINDER

STA ]REM

STA ]REM+1

LDX #16 ; NUMBER OF BITS

:DVLP

ASL ]DVEND ; LOBYTE \* 2

ROL ]DVEND+1 ; HIBYTE \* 2

ROL ]REM ; LOBYTE \* 2

ROL ]REM+1 ; HIBYTE \* 2

LDA ]REM

SEC ; SET CARRY

SBC ]DVSOR ; SUBTRACT DIVISOR

TAY ; TO SEE IF IT FITS IN DVEND,

LDA ]REM+1 ; HOLD LOBYTE IN .Y

SBC ]DVSOR+1 ; AND DO SAME WITH HIBYTES

BCC :SKIP ; IF C=0, DVSOR DOESN'T FIT

\*

STA ]REM+1 ; ELSE SAVE RESULT AS REM

STY ]REM

INC ]RESULT ; AND INC RES

:SKIP

DEX ; DECREASE BIT COUNTER

BNE :DVLP ; RELOOP IF > 0

LDA #2 ; LENGTH OF RESULT IN BYTES

STA RETLEN ; STORED IN RETLEN

LDA ]RESULT ; STORE RESULT LOBYTE

STA RETURN ; IN .A AND RETURN

LDX ]RESULT+1 ; STORE HIBYTE IN .X

STX RETURN+1 ; AND IN RETURN+1

RTS

**SUB.DIVD8 >> DIVD8**

|  |
| --- |
| **DIVD8 (sub)**  **Input**:  **WPAR1** = dividend  **WPAR2** = divisor  **Output**:  **.A** = result  **.X** = remainder  **RETURN** = result  **RETLEN** = 1  **Destroys**: AXYNVZCM  **Cycles**: 58+  **Size**: 34 bytes |

The DIVD8 subroutine divides one 8-bit number by another, returning the result in **.A** with the remainder in **.X**. The result is also stored in **RETURN** as a single byte.

\*

\*``````````````````````````````\*

\* DIVD8 (NATHAN RIGGS) \*

\* \*

\* DIVIDE WITH TWO 8-BIT VALUES \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = DIVIDEND \*

\* WPAR2 = DIVISOR \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = RESULT \*

\* .X = REMAINDER \*

\* RETURN = RESULT \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^ ^ ^^^ \*

\* \*

\* CYCLES: 58+ \*

\* SIZE: 34 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]DVEND EQU WPAR1 ; DIVIDEND

]DVSOR EQU WPAR2 ; DIVISOR

\*

DIVD8

STX ]DVEND ; .X HOLDS DIVIDEND

STA ]DVSOR ; .A HOLDS DIVISOR

LDA #$00 ; CLEAR ACCUMULATOR

LDX #8 ; COUNTER

ASL ]DVSOR ; SHIFT LEFT DIVISOR

:L1 ROL ; ROTATE LEFT .A

CMP ]DVEND ; COMPARE TO DIVIDEND

BCC :L2 ; IF NEXT BIT = 0, BRANCH :L2

SBC ]DVEND ; OTHERWISE, SUBTRACT DIVIDEND

:L2 ROL ]DVSOR ; ROTATE LEFT DIVISOR

DEX ; DECREMENT COUNTER

BNE :L1 ; IF > 0, LOOP

TAX ; REMAINDER IN .X

LDA #1

STA RETLEN

LDA ]DVSOR ; RESULT IN .A

STA RETURN

RTS

**SUB.MULT16 >> MULT16**

|  |
| --- |
| **MULT16 (sub)**  **Input**:  **WPAR1** = multiplier (2b)  **WPAR2** = multiplicand (2b)  **Output**:  **.A** = lowest product byte  **.X** = 2nd lowest prod byte  **RETURN** = 32-bit product  **RETLEN** = 4 (byte length)  **Destroys**: AXYNVZCM  **Cycles**: 101+  **Size**: 61 bytes |

The MULT16 subroutine multiplies two given 16-bit numbers passed via **WPAR1** and **WPAR2** and stores the 16-bit result in **.A** (low byte) and **.X** (high byte). If the multiplier and multiplicand are unsigned, a 32-bit product can be read from **RETURN** (4 bytes). If the values are signed, however, only the two lowest bits are reliable.

\*

\*``````````````````````````````\*

\* MULT16 (NATHAN RIGGS) \*

\* \*

\* MULTIPLY TWO 16-BIT VALUES. \*

\* NOTE THAT THIS ONLY WORKS \*

\* CORRECTLY WITH UNSIGNED \*

\* VALUES. \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = MULTIPLICAND \*

\* WPAR2 = MULTIPLIER \*

\* \*

\* OUTPUT: \*

\* \*

\* RETURN = 32-BIT PRODUCT \*

\* RETLEN = 4 (BYTE LENGTH) \*

\* .A = LOWEST PRODUCT BYTE \*

\* .X = 2ND LOWEST BYTE (COPY) \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^ ^ ^^^ \*

\* \*

\* CYCLES: 101+ \*

\* SIZE: 61 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]MCAND EQU WPAR1 ; MULTIPLICAND

]MLIER EQU WPAR2 ; MULTIPLIER

]HPROD EQU WPAR3 ; HIGH BYTES OF PRODUCT

\*

MULT16

LDA #0 ; ZERO OUT TOP TWO

STA ]HPROD ; HIGH BYTES OF 32-BIT

STA ]HPROD+1 ; RESULT

LDX #17 ; # OF BITS IN MLIER PLUS 1

; FOR LAST CARRY INTO PRODUCT

CLC ; CLEAR CARRY FOR 1ST TIME

; THROUGH LOOP.

:MLP

\*

\*\* IF NEXT BIT = 1, HPROD += 1

\*

ROR ]HPROD+1 ; SHIFT HIGHEST BYTE

ROR ]HPROD ; SHIFT 2ND-HIGHEST

ROR ]MLIER+1 ; SHIFT 3RD-HIGHEST

ROR ]MLIER ; SHIFT LOW BYTE

BCC :DX ; BRANCH IF NEXT BIT = 0

CLC ; OTHERWISE NEXT BIT =1,

LDA ]MCAND ; SO ADD MCAND TO PRODUCT

ADC ]HPROD

STA ]HPROD ; STORE NEW LOBYTE

LDA ]MCAND+1

ADC ]HPROD+1

STA ]HPROD+1 ; STORE NEW HIBYTE

:DX

DEX ; DECREASE COUNTER

BNE :MLP ; DO MUL LOOP UNTIL .X = 0

\*

\*\* NOW STORE IN RETURN, WITH LOWEST TWO

\*\* BYTES ALSO LEFT IN .A (LO) AND .X (HI)

\*

LDA #4 ; LENGTH OF PRODUCT

STA RETLEN ; STORED IN RETLEN

LDA ]HPROD+1

STA RETURN+3

LDA ]HPROD

STA RETURN+2

LDX ]MLIER+1

STX RETURN+1

LDA ]MLIER

STA RETURN

RTS

**SUB.MULT8 >> MULT8**

|  |
| --- |
| **MULT8 (sub)**  **Input**:  **WPAR1** = multiplier (1b)  **WPAR2** = multiplicand (1b)  **Output**:  **.A** = product low byte  **.X** = product high byte  **RETURN** = product (2b)  **RETLEN** = 2  **Destroys**: AXYNVZCM  **Cycles**: 81+  **Size**: 47 bytes |

The MULT8 subroutine accepts an 8-bit multiplier and an 8-bit multiplicand from **WPAR1** and **WPAR2**, respectively, and returns the 16-bit product in **.A** (low byte) and **.X** (high byte). This product is also placed in **RETURN** for retrieval.

\*

\*``````````````````````````````\*

\* MULT8 (NATHAN RIGGS) \*

\* \*

\* MULTIPLY TWO 8-BIT NUMBERS. \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = MULTIPLIER \*

\* WPAR2 = MULTIPLICAND \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = PRODUCT LOW BYTE \*

\* .X = PRODUCT HIGH BYTE \*

\* RETURN = PRODUCT (2 BYTES) \*

\* RETLEN = 2 \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^ ^^^ \*

\* \*

\* CYCLES: 81+ \*

\* SIZE: 47 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]MUL1 EQU WPAR1

]MUL2 EQU WPAR2

\*

MULT8

STA ]MUL1

STX ]MUL2

LDA #0 ; CLEAR REGISTERS

TAY

TAX

STA ]MUL1+1 ; CLEAR HIBYTE

BEQ :GOLOOP

:DOADD

CLC ; CLEAR CARRY

ADC ]MUL1 ; ADD MULTIPLIER

TAX ; HOLD IN .Y

TYA ; XFER .X TO .A

ADC ]MUL1+1 ; ADD MULTIPLIER HIBYTE

TAY ; HOLD BACK IN .X

TXA ; MOVE LOBYTE INTO .A

:LP

ASL ]MUL1 ; SHIFT LEFT

ROL ]MUL1+1 ; ROLL HIBYTE

:GOLOOP

LSR ]MUL2 ; SHIFT MULTIPLIER

BCS :DOADD ; IF 1 SHIFTED INTO CARRY, ADD AGAIN

BNE :LP ; OTHERWISE, LOP

LDA #2 ; 16-BIT LENGTH, 2 BYTES

STA RETLEN ; FOR RETURN LENGTH

STX RETURN ; STORE LOBYTE

STY RETURN+1 ; STORE HIBYTE

TXA ; LOBYTE TO .A

LDX RETURN+1 ; HIBYTE TO .X

RTS

**SUB.RAND16 >> RAND16**

|  |
| --- |
| **RAND16 (sub)**  **Input**:  none  **Output**:  **.A** = random value low  byte  **.X** = random value high  byte  **RETURN** = random value  **RETLEN** = 2 (byte length)  **Destroys**: AXYNVZCM  **Cycles**: 90+  **Size**: 60 bytes |

The RAND16 subroutine returns a 16-bit pseudo-random number with the low byte held in **.A** and the high byte stored in **.X**. This two-byte value is also stored in **RETURN**, with a **RETLEN** of 2.

\*``````````````````````````````\*

\* RAND16 : 16BIT RANDOM NUMBER \*

\*- -\*

\* GENERATE A 16BIT PSEUDO- \*

\* RANDOM NUMBER AND RETURN IT \*

\* IN Y,X (LOW, HIGH). \*

\* \*

\* ORIGINAL AUTHOR IS WHITE \*

\* FLAME, AS SHARED ON \*

\* CODEBASE64. \*

\* \*

\* NOTE: THERE ARE 2048 MAGIC \*

\* NUMBERS THAT COULD BE EOR'D \*

\* TO GENERATE A PSEUDO-RANDOM \*

\* PATTERN THAT DOESN'T REPEAT \*

\* UNTIL 65535 ITERATIONS. TOO \*

\* MANY TO LIST HERE, BUT SOME \*

\* ARE: $002D, $1979, $1B47, \*

\* $41BB, $3D91, $B5E9, $FFEB \*

\* \*

\* INPUT: \*

\* \*

\* NONE \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = RND VAL LOW BYTE \*

\* .X = RND VAL HIGH BYTE \*

\* RETURN = RND VALUE (2B) \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^ ^^^ \*

\* \*

\* CYCLES: 90+ \*

\* SIZE: 60 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]SEED EQU WPAR1

\*

RAND16

LDA RNDL ; GET SEED LOBYTE

STA ]SEED

LDA RNDH ; GET SEED HIBYTE

STA ]SEED+1

\*

LDA ]SEED ; CHECK IF $0 OR $8000

BEQ :LOW0

\*

\*\* DO A NORMAL SHIFT

\*

ASL ]SEED ; MUTATE

LDA ]SEED+1

ROL

BCC :NOEOR ; IF CARRY CLEAR, EXIT

:DOEOR ; HIGH BYTE IN A

EOR #>$0369 ; EXCLUSIVE OR WITH MAGIC NUMBER

STA ]SEED+1 ; STORE BACK INTO HIBYTE

LDA ]SEED

EOR #<$0369 ; DO THE SAME WITH LOW BYTE

STA ]SEED

JMP :EXIT

:LOW0

LDA ]SEED+1

BEQ :DOEOR ; IF HIBYTE IS ALSO 0, APPLY EOR

ASL

BEQ :NOEOR ; IF 00, THEN IT WAS $80

BCS :DOEOR ; ELSE DO EOR

:NOEOR

STA ]SEED+1

:EXIT

LDX ]SEED+1 ; VAL HIBYTE IN .X

LDY ]SEED ; LOBYTE TEMP IN .Y

STY RETURN ; TRANSFER TO RETURN AREA

STX RETURN+1

LDA #2 ; LENGTH OF RETURN IN BYTES

STA RETLEN

TYA ; TRANSFER LOBYTE TO .A

RTS

**SUB.RAND8 >> RAND8**

|  |
| --- |
| **RAND8 (sub)**  **Input**:  none  **Output**:  **.A** = random byte value  **RETURN** = random byte val  **RETLEN** = 1  **Destroys**: AXYNVZCM  **Cycles**: 44+  **Size**: 27 bytes |

The RAND8 subroutine returns a single-byte pseudo-random number in the **.A** register as well as in **RETURN**.

\*

\*``````````````````````````````\*

\* RAND8 (NATHAN RIGGS) \*

\* \*

\* GENERATE PSEUDO-RANDOM BYTE \*

\* \*

\* INPUT: \*

\* \*

\* NONE \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = RANDOM BYTE \*

\* RETURN = RANDOM BYTE \*

\* RETLEN = #1 \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^ ^^^ \*

\* \*

\* CYCLES: 44+ \*

\* SIZE: 27 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

RAND8

LDX #8 ; NUMBER OF BITS

LDA RNDL+0 ; GET SEED

:A

ASL ;SHIFT THE REG

ROL RNDL+1 ; ROTATE HIGH BYTE

BCC :B ; IF 1 BIT SHIFTED OUT,

EOR #$2D ; APPLY XOR FEEDBACK

:B

DEX ; DECREASE BIT COUNTER

BNE :A ; IF NOT ZERO, RELOOP

STA RNDL+0 ; STORE NEW SEED

STA RETURN ; STORE IN RETURN

LDY #1 ; RETURN BYTE LENGTH

STY RETLEN ; IN RETLEN

CMP #0 ; RELOAD FLAGS

RTS

\*

**SUB.RANDB >> RANDB**

|  |
| --- |
| **RANDB (sub)**  **Input**:  **BPAR1** = minimum boundary  **BPAR2** = maximum boundary  **Output**:  **.A** = random number  **RETURN** = random number  **RETLEN** = 1  **Destroys**: AXYNVZCM  **Cycles**: 248+  **Size**: 476 bytes |

The RANDB subroutine returns a single byte pseudo-random number between a low value of **BPAR1** and a high value of **BPAR2**. This number is returned in **.A** as well as in **RETURN**.

Note that this subroutine uses many more cycles than RAND8. Therefore, when the actual number matters less than the probability of its value being returned, it is best to use the RAND8 subroutine.

\*

\*``````````````````````````````\*

\* RANDB (NATHAN RIGGS) \*

\* \*

\* GET A RANDOM VALUE BETWEEN \*

\* A MIN AND MAX BOUNDARY. \*

\* \*

\* INPUT: \*

\* \*

\* BPAR1 = MINIMUM VALUE \*

\* BPAR2 = MAXIMUM VALUE \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = NEW VALUE \*

\* RETURN = NEW VALUE \*

\* RETLEN = 1 (BYTE COUNT) \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 248+ \*

\* SIZE: 476 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]NEWMIN EQU BPAR1 ; MINIMUM PARAMETER

]NEWMAX EQU BPAR2 ; MAXIMUM PARAMETER

]OLDMIN EQU WPAR1 ; OLD MINIMUM (1)

]OLDMAX EQU WPAR1+1 ; OLD MAXIMUM (255)

]OLDRNG EQU VARTAB ; OLD RANGE

]NEWRNG EQU VARTAB+2 ; NEW RANGE

]MULRNG EQU VARTAB+4 ; MULTIPLIED RANGE

]DIVRNG EQU VARTAB+6 ; DIVIDED RANGE

]VALRNG EQU VARTAB+8 ; VALUE RANGE

]OLDVAL EQU VARTAB+10 ; OLD VALUE

]NEWVAL EQU VARTAB+12 ; NEW VALUE

]NUM1HI EQU VARTAB+14 ; MULTIPLICATION HI BYTE

]REMAIN EQU VARTAB+16 ; REMAINDER

\*

RANDB

STX ]NEWMAX ; NEW HIGH VALUE

STA ]NEWMIN ; NEW LOW VALUE OF RANGE

\*

\*\* GET OLDMIN,OLDMAX,OLDVAL

\*

LDA #1 ; OLD LOW IS ALWAYS 1

STA ]OLDMIN

LDA #255 ; OLD HIGH IS ALWAYS 255

STA ]OLDMAX

\*

LDX #8 ; NUMBER OF BITS IN #

LDA RNDL+0 ; LOAD SEED VALUE

:AA

ASL ; SHIFT ACCUMULATOR

ROL RNDL+1

BCC :BB ; IF NEXT BIT IS 0, BRANCH

EOR #$2D ; ELSE, APPLY XOR FEEDBACK

:BB

DEX ; DECREASE .X COUNTER

BNE :AA ; IF > 0, KEEP LOOPING

STA RNDL+0 ; OVERWRITE SEED VALUE

CMP #0 ; RESET FLAGS

STA ]OLDVAL ; STORE RANDOM NUMBER

\*

\*\* NEWVALUE = (((OLDVAL-NEWMIN) \* (NEWMAX-NEWMIN) /

\*\* (OLDMAX-OLDMIN)) + NEWMIN

\*

\*\* OLDRANGE = (OLDMAX-OLDMIN)

\*\* NEWRANGE = (NEWMAX - NEWMIN)

\*\* NEWVAL = (((OLDVAL-OLDMIN) \* NEWRANGE) / OLDRANGE) + NEWMIN

\*

LDA ]OLDMAX ; SUBTRACT OLDMIN

SEC ; FROM OLDMAX, STORE

SBC ]OLDMIN ; IN OLDRANGE

STA ]OLDRNG

\*

LDA ]NEWMAX ; SUBTRACT NEWMIN

SEC ; FROM NEWMAX, THEN

SBC ]NEWMIN ; STORE IN NEWRANGE

STA ]NEWRNG

\*

LDA ]OLDVAL ; SUBTRACT OLDMIN

SEC ; FROM OLDVAL AND

SBC ]OLDMIN ; STORE IN VALRANGE

STA ]VALRNG

\*

\*\* GET MULRANGE: VALRANGE \* NEWRANGE

\*

LDA #00 ; CLEAR ACCUMULATOR,

TAY ; .Y AND THE HIGH BYTE

STY ]NUM1HI

BEQ :ENTLP ; IF ZERO, BRANCH

:DOADD

CLC ; CLEAR CARRY

ADC ]VALRNG ; ADD VALUE RANGE TO .A

TAX ; HOLD IN .X

TYA ; .Y BACK TO .A

ADC ]NUM1HI ; ADD HIBYTE

TAY ; MOVE BACK TO .Y

TXA ; .X BACK TO .A

:MLP

ASL ]VALRNG ; SHIFT VALUE RANGE

ROL ]NUM1HI ; ADJUST HIGH BYTE

:ENTLP

LSR ]NEWRNG ; SHIFT NEW RANGE

BCS :DOADD ; IF LAST BIT WAS 1, LOOP ADD

BNE :MLP ; IF ZERO FLAG CLEAR, LOOP SHIFT

STA ]MULRNG ; STORE RESULT LOW BYTE

STY ]MULRNG+1 ; STORE HIGH BYTE

\*

\*\* NOW GET DIVRANGE: MULRANGE / OLDRANGE

\*

:DIVIDE

LDA #0 ; CLEAR ACCUMULATOR

STA ]REMAIN ; AND THE REMAINDER LOBYTE

STA ]REMAIN+1 ; AND REMAINDER HIBYTE

LDX #16 ; NUMBER OF BYTES

\*

:DIVLP

ASL ]MULRNG ; LOW BYTE \* 2

ROL ]MULRNG+1 ; HIGH BYTE \* 2

ROL ]REMAIN ; REMAINDER LOW BYTE \* 2

ROL ]REMAIN+1 ; HIGH BYTE \* 2

LDA ]REMAIN ; SUBTRACT OLDRANGE

SEC ; FROM REMAINDER

SBC ]OLDRNG

TAY ; HOLD IN .Y

LDA ]REMAIN+1 ; SUBTRACT HIGH BYTES

SBC ]OLDRNG+1

BCC :SKIP ; IF NO CARRY, THEN NOT DONE

\*

STA ]REMAIN+1 ; SAVE SBC AS NEW REMAINDER

STY ]REMAIN

INC ]DIVRNG ; INCREMENT THE RESULT

\*

:SKIP DEX ; DECREMENT COUNTER

BNE :DIVLP ; IF ZERNO, RELOOP

\*

\*\* NOW ADD NEWMIN TO DIVRANGE

\*

LDA ]DIVRNG ; USE LOW BYTE ONLY

CLC ; AND ADD TO ]NEWMIN

ADC ]NEWMIN ; TO GET THE NEW VALUE

STA ]NEWVAL

STA RETURN ; COPY TO RETURN

LDX #1 ; RETURN LENGTH

STX RETLEN

RTS

**SUB.SUBT16 >> SUBT16**

|  |
| --- |
| **SUBT16 (sub)**  **Input**:  **WPAR1** = minuend (2b)  **WPAR2** = subtrahend (2b)  **Output**:  **.A** = difference low byte  **.X** = difference high byte  **RETURN** = difference  **RETLEN** = 2 (byte length)  **Destroys**: AXYNVZCM  **Cycles**: 43+  **Size**: 24 bytes |

The SUBT16 subroutine subtracts a 16-bit subtrahend stored in **WPAR2** from the 16-bit minuend in **WPAR1**. The difference is stored in **.A** (low byte) and **.X** (high byte), as well as in **RETURN**. **RETLEN** contains the byte-length of **RETURN**, which is always 2.

This subroutine is likely to be supplemented with a macro that achieves the same result, allowing the programmer to decide between speed of execution versus the length of the program in bytes.

\*

\*``````````````````````````````\*

\* SUBT16 (NATHAN RIGGS) \*

\* \*

\* SUBTRACT A 16-BIT SUBTRAHEND \*

\* FROM A MINUEND. \*

\* \*

\* INPUT \*

\* \*

\* WPAR1 = MINUEND \*

\* WPAR2 = SUBTRAHEND \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = DIFFERENCE LOW BYTE \*

\* .X = DIFFERENCE HIGH BYTE \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^ ^^^ \*

\* \*

\* CYCLES: 43+ \*

\* SIZE: 24 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]MINU EQU WPAR1 ; MINUEND

]SUBT EQU WPAR2 ; SUBTRAHEND

\*

SUBT16

LDA #2

STA RETLEN

LDA ]MINU ; SUBTRACT SUBTRAHEND

SEC ; LOBYTE FROM MINUEND

SBC ]SUBT ; LOBYTE

TAY ; HOLD LOBYTE IN .Y

LDA ]MINU+1 ; SUBTRACT SUBTRAHEND

SBC ]SUBT+1 ; HIBYTE FROM MINUEND

TAX ; HIGH BYTE, PASS IN .X

TYA ; LOBYTE BACK IN .A

STA RETURN

STX RETURN+1

RTS

**DEMO.MATH**

The DEMO.MATH program showcases the functionality of the SUB.MATH subroutines and macros. These are not exhaustive, and are intended to simply illustrate how the library works rather than test the limits of each subroutine.

\*

\*``````````````````````````````\*

\* DEMO.MATH \*

\* \*

\* A DEMO OF THE INTEGER MATH \*

\* MACROS INCLUDED AS PART OF \*

\* THE APPLEIIASM LIBRARY. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 16-JUL-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\*\* ASSEMBLER DIRECTIVES

\*

CYC AVE

EXP OFF

TR ON

DSK DEMO.MATH

OBJ $BFE0

ORG $6000

\*

\*``````````````````````````````\*

\* TOP INCLUDES (HOOKS,MACROS) \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PUT MIN.HEAD.REQUIRED

USE MIN.MAC.REQUIRED

PUT MIN.HOOKS.MATH

USE MIN.MAC.MATH

]HOME EQU $FC58

\*

\*``````````````````````````````\*

\* PROGRAM MAIN BODY \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

JSR ]HOME

\_PRN "INTEGER MATH DEMO",8D

\_PRN "=================",8D8D

\_PRN "THIS DISK CONTAINS MACROS AND",8D

\_PRN "SUBROUTINES RELATED TO INTEGER",8D

\_PRN "MATH (UNSIGNED ONLY, SO FAR), AS",8D

\_PRN "WELL AS HOOKS TO USE THE STANDARD",8D

\_PRN "APPLESOFT FLOATING-POINT ",8D

\_PRN "SUBROUTINES.",8D8D

\_PRN "THE FLOATING-POINT ROUTINES",8D

\_PRN "ARE NOT COVERED HERE.",8D8D

\_WAIT

JSR ]HOME

\_PRN "16-BIT INTEGER MATH",8D

\_PRN "===================",8D8D

\_PRN "ADD16, SUB16, MUL16, DIV16",8D8D

\_PRN "THE 16-BIT INTEGER MATH MACROS",8D

\_PRN "ARE USED TO CALCULATE UNSIGNED VALUES",8D

\_PRN "BETWEEN 0 AND 65,025. THESE ARE TWO-",8D

\_PRN "BYTE VALUES.",8D8D

\_PRN "NOTE THAT BECAUSE OF INCREASED BYTE",8D

\_PRN "AND CPU CYCLE EXPENSES, THESE SHOULD",8D

\_PRN "ONLY BE USED IF 8-BIT CALCULATION ISN'T",8D

\_PRN "ADEQUATE.",8D

\_WAIT

JSR ]HOME

\_PRN "LET'S START WITH ADDING TWO 16-BIT",8D

\_PRN "NUMBERS. THE ADD16 MACRO ACCEPTS TWO",8D

\_PRN "16-BIT PARAMETERS, ADDS THEM TOGETHER,",8D

\_PRN "AND THEN HOLDS THE VALUE IN RETURN,",8D

\_PRN "WITH THE BYTE-LENGTH STORED IN RETLEN.",8D8D

\_PRN "NOTE THAT THE SUM RETURNED IS ALSO A",8D

\_PRN "16-BIT VALUE; THUS, A TOTAL SUM CAN BE",8D

\_PRN "NO HIGHER THAN 65,025. THE SUM IS",8D

\_PRN "ALSO RETURNED IN .A (LOW BYTE) AND",8D

\_PRN ".X (HIGH BYTE) FOR FASTER REFERENCE.",8D8D

\_WAIT

\_PRN "THUS, THE FOLLOWING CODE:",8D8D

\_PRN " ADD16 #10000;#20000",8D8D

\_PRN "WILL RESULT IN:",8D8D

\_WAIT

ADD16 #10000;#20000

DUMP #RETURN;RETLEN

\_WAIT

JSR ]HOME

\_PRN "16-BIT SUBTRACTION WORKS MUCH THE",8D

\_PRN "SAME. THE DIFFERENCE IS STORED IN",8D

\_PRN "RETURN AS WELL AS IN .A (LOW) AND",8D

\_PRN ".X (HIGH), AND RETLEN CONTAINS",8D

\_PRN "THE BYTE-LENGTH OF THE DIFFERENCE.",8D8D

\_PRN "THUS, THE FOLLOWING CODE:",8D8D

\_PRN " SUB16 #20000;#10000",8D8D

\_PRN "PRODUCES:",8D8D

\_WAIT

SUB16 #20000;#10000

DUMP #RETURN;RETLEN

\_WAIT

JSR ]HOME

\_PRN "16-BIT MULTIPLICATION AGAIN WORKS",8D

\_PRN "MUCH LIKE ADDITION AND SUBTRACTION,",8D

\_PRN "EXCEPT THE ORDER OF THE PARAMETERS DOES",8D

\_PRN "NOT MATTER.",8D8D

\_WAIT

\_PRN "UNLIKE 16-BIT ADDITION AND 16-BIT",8D

\_PRN "SUBTRACTION, THE MUL16 MACRO ",8D

\_PRN "RETURNS A 32-BYTE VALUE (4 BYTES). NOTE",8D

\_PRN "THAT IF EITHER OF THE PARAMETERS ARE",8D

\_PRN "SIGNED, THE TWO HIGHEST BYTES WILL BE",8D

\_PRN "WRONG.",8D8D

\_WAIT

\_PRN "THUS, MULTIPLYING TWO NUMBERS IS AS",8D

\_PRN "EASY TO ACCOMPLISH AS:",8D8D

\_PRN " MUL16 #300;#1000",8D8D

\_PRN "WHICH OUTPUTS THE PRODUCT TO RETURN:",8D8D

\_WAIT

MUL16 #300;#1000

DUMP #RETURN;RETLEN

\_WAIT

JSR ]HOME

\_PRN "FINALLY, THE DIV16 MACRO HANDLES ",8D

\_PRN "16-BIT DIVISION, STORING THE RESULT",8D

\_PRN "IN RETURN. THIS IS ALSO STORED IN",8D

\_PRN ".A (LOW BYTE) AND .X (HIGH BYTE). THE ",8D

\_PRN "REMAINDER OF THE OPERATION IS STORED",8D

\_PRN "IN .Y.",8D8D

\_WAIT

\_PRN "THUS:",8D8D

\_PRN " DIV16 #10000;#1000",8D8D

\_PRN "WILL RETURN:",8D8D

\_WAIT

DIV16 #10000;#1000

DUMP #RETURN;RETLEN

\_WAIT

JSR ]HOME

\_PRN "8-BIT INTEGER MATHEMATICS",8D

\_PRN "=========================",8D8D

\_PRN "8-BIT MATH MOSTLY WORKS THE SAME",8D

\_PRN "AS 16-BIT MATH MACROS, BUT SINCE",8D

\_PRN "8-BIT ADDITION AND SUBTRACTION ARE",8D

\_PRN "MUCH SIMPLER IN 6502, THEY ARE ONLY",8D

\_PRN "MACROS WITHOUT SUBROUTINES, AND ",8D

\_PRN "STRICTLY USE THE REGISTERS FOR PASSING",8D

\_PRN "DATA.",8D8D

\_PRN "SINCE THEY ARE SO SIMILAR IN FORM",8D

\_PRN "AND FUNCTION, WE WILL COVER THOSE",8D

\_PRN "TOGETHER.",8D8D

\_WAIT

JSR ]HOME

\_PRN "THE ADD8 AND SUB8 MACROS ADD AND",8D

\_PRN "SUBTRACT 8-BIT VALUES, RESPECTIVELY.",8D

\_PRN "THE RESULT OF BOTH OPERATIONS IS",8D

\_PRN "STORED IN THE ACCUMULATOR. AS SUCH:",8D8D

\_WAIT

\_PRN " ADD8 #10;#20",8D8D

\_PRN "WILL RETURN:",8D8D

ADD8 #10;#20

DUMP #RETURN;RETLEN

\_PRN "AND:",8D8D

\_WAIT

\_PRN " SUB8 #20;#10",8D8D

\_PRN "WILL RETURN:",8D8D

SUB8 #20;#10

DUMP #RETURN;RETLEN

\_WAIT

JSR ]HOME

\_PRN "THE DIV8 AND MUL8 MACROS WORK AS",8D

\_PRN "EXPECTED: LIKE DIV16 AND MUL16, BUT",8D

\_PRN "WORK ONLY WITH 8-BIT VALUES INSTEAD.",8D8D

\_PRN "THUS:",8D8D

\_PRN " MUL8 #10;#10",8D8D

\_PRN "RETURNS:",8D8D

\_WAIT

MUL8 #10;#10

DUMP #RETURN;RETLEN

\_WAIT

\_PRN "AND:",8D8D

\_PRN " DIV8 #100;#10",8D8D

\_PRN "RETURNS:",8D8D

\_WAIT

DIV8 #100;#10

DUMP #RETURN;RETLEN

\_WAIT

JSR ]HOME

\_PRN "PSEUDO-RANDOM NUMBERS",8D

\_PRN "=====================",8D8D

\_PRN "THERE ARE THREE MACROS DEDICATED TO",8D

\_PRN "PSEUDO-RANDOM NUMBER GENERATION:",8D

\_PRN "RND8, RND16, AND RAND. ",8D8D

\_WAIT

\_PRN "RND8 RETURNS A PSEUDO-RANDOM BYTE IN",8D

\_PRN ".A AND IN RETURN (0..255), WHEREAS",8D

\_PRN "RND16 RETURNS A 16-BIT VALUE (2 BYTES)",8D

\_PRN "IN RETURN AND IN .A (LOW BYTE) AND .X",8D

\_PRN "(HIGH BYTE). LASTLY, THE RAND MACRO",8D

\_PRN "RETURNS A BYTE VALUE BETWEEN A GIVEN ",8D

\_PRN "LOW VALUE AND HIGH VALUE.",8D8D

\_WAIT

\_PRN "RND8 AND RND16 DO NOT ACCEPT ANY",8D

\_PRN "PARAMETERS; ONLY RAND ACCEPTS ANY INPUT",8D

\_PRN "WHATSOEVER. THUS:",8D8D

\_WAIT

\_PRN " RAND #10;#20",8D8D

\_PRN "RETURNS A NUMBER BETWEEN 10 AND 20:",8D8D

RAND #10;#20

DUMP #RETURN;RETLEN

\_WAIT

JSR ]HOME

\_PRN "16-BIT COMPARISON",8D

\_PRN "=================",8D8D

\_PRN "LASTLY, THE ODD MACRO OUT IN THIS",8D

\_PRN "MACRO COLLECTION IS CMP16, WHICH",8D

\_PRN "PERFORMS THE EQUIVALENT OF THE 6502",8D

\_PRN "ASSEMBLY CMP COMMAND (COMPARE) BUT ON A",8D

\_PRN "16-BIT VALUE. THIS IS ACHIEVED BY",8D

\_PRN "SETTING FLAG BITS IN THE .P REGISTER",8D

\_PRN "BASED ON WHETHER THE TWO VALUES ARE",8D

\_PRN "EQUAL, OR ONE IS LESS THAN OR GREATER",8D

\_PRN "THAN THE OTHER. ",8D8D

\_WAIT

\_PRN "THE FOLLOWING FLAGS ARE SET BASED",8D

\_PRN "ON THE RELATIONSHIP OF THE PARAMETERS:",8D8D

\_PRN "UNSIGNED PARAMETERS:",8D8D

\_PRN " Z = 1 IF PARAMETERS ARE EQUAL",8D

\_PRN " C = 0 IF FIRST PARAMETER > SECOND",8D

\_PRN " 1 IF FIRST PARAMETER <= SECOND",8D8D

\_WAIT

\_PRN "SIGNED PARAMETERS:",8D8D

\_PRN " N = 1 IF FIRST PARAMETER > SECOND",8D

\_PRN " 0 IF FIRST PARAMETER <= SECOND",8D

\*

\_WAIT

JSR ]HOME

\_PRN "WE ARE DONE HERE.",8D8D8D

JMP REENTRY

\*

\*``````````````````````````````\*

\* BOTTOM INCLUDES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\*\* BOTTOM INCLUDES

\*

PUT MIN.LIB.REQUIRED

\*

\*\* INDIVIDUAL SUBROUTINE INCLUDES

\*

\*\* 8-BIT MATH SUBROUTINES

\*

PUT MIN.SUB.MULT8

PUT MIN.SUB.DIVD8

PUT MIN.SUB.RAND8

PUT MIN.SUB.RANDB

\*

\*\* 16-BIT MATH SUBROUTINES

\*

PUT MIN.SUB.ADDIT16

PUT MIN.SUB.SUBT16

PUT MIN.SUB.COMP16

PUT MIN.SUB.MULT16

PUT MIN.SUB.DIVD16

PUT MIN.SUB.RAND16

\*

**STRINGS LIBRARY**

The strings library holds macros and subroutines dedicated to string manipulation. Currently, this only covers 8-bit strings: strings with a single preceding byte that defines the length, followed by the characters in the string (not to exceed 255). Null-terminated strings are handled mostly in the STDIO library, but 16-bit or larger strings may be handled here in the future.

* HOOKS.STRINGS
* MAC.STRINGS
* DEMO.STRINGS
* SUB.PRNSTR
* SUB.STRCAT
* SUB.STRCOMP
* SUB.SUBCOPY
* SUB.SUBDEL
* SUB.SUBINS
* SUB.SUBPOS

**HOOKS.STRINGS** includes hooks related to string manipulation. Currently, there aren’t too many of these.

**MAC.STRINGS** contains all of the macros related to string manipulation.

**DEMO.STRINGS** is a demo of all of the string manipulation macros.

**SUB.PRNSTR** holds the subroutine for printing a string with a preceding length byte. This is pretty much identical to the PRNSTR routine in the STDIO library; one or the other may be deleted in future iterations.

**SUB.STRCAT** contains the subroutine dedicated to string concatenation.

**SUB.STRCOMP** includes the subroutine used for string comparison.

**SUB.SUBCOPY** contains the subroutine dedicated to copying a substring from a source string.

**SUB.SUBINS** holds the SUBINS subroutine, which inserts a substring into another string at the given position.

**SUB.SUBPOS** includes the subroutine that finds the position of a substring in a given source string.

**HOOKS.STRINGS**

This file contains hooks related to string manipulation. Currently, this is very limited. Future revisions will include some hooks to basic Applesoft routines.

\*

\*``````````````````````````````\*

\* HOOKS.STRINGS \*

\* \*

\* THIS FILE CONTAINS ALL OF \*

\* THE HOOKS REQUIRED BY THE \*

\* STRING LIBRARY. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 19-SEP-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

SCOUT1 EQU $FDF0

\*

**MAC.STRINGS**

MAC.STRINGS contains all of the macros related to 8-bit string manipulation. 16-bit and 32-bit routines may be included in the future, as well as macros and subroutines dedicated to parsing strings for tasks like command line interaction, breaking down mathematical expressions stored as strings, and so on.

\*``````````````````````````````\*

\* MAC.STRINGS \*

\* \*

\* THIS FILE CONTAINS ALL OF \*

\* THE MACROS RELATED TO STRING \*

\* MANIPULATION. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 17-SEP-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\* \*

\* SUBROUTINE FILES USED \*

\* \*

\* SUB.PRNSTR \*

\* SUB.STRCAT \*

\* SUB.STRCOMP \*

\* SUB.SUBCOPY \*

\* SUB.SUBDEL \*

\* SUB.SUBINS \*

\* SUB.SUBPOS \*

\* \*

\* LIST OF MACROS \*

\* \*

\* SCMP : STRING COMPARE \*

\* SCAT : STRING CONCATENATE \*

\* SPRN : PRINT STRING \*

\* SPOS : FIND SUBSTRING POS \*

\* SCOP : SUBSTRING COPY \*

\* SDEL : SUBSTRING DELETE \*

\* SINS : SUNBSTRING INSERT \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

**MAC.STRINGS >> SCMP**

|  |
| --- |
| **SCMP (macro)**  **Input**:  **]1** = 1st string  **]2** = 2nd string  **Output**:  See description  **Destroys**: AXYNVZCM  **Cycles**: 113+  **Size**: 88 bytes |

The **SCMP** macro compares one string to another and changes the status register in response. First, the strings are tested to be equal or not. If so, the **ZERO** flag is set to **1**; if not, the **ZERO** flag is set to **0**.

If the strings do not match, further testing is done on the lengths of the strings, with the results affecting the carry flag. If the first string has fewer characters than the second string, the **CARRY** flag is set to **0**; otherwise, it is set to **1**.

\*``````````````````````````````\*

\* SCMP (NATHAN RIGGS) \*

\* \*

\* COMPARES TWO STRINGS AND \*

\* CHANGES THE ZERO FLAG TO 1 \*

\* IF THE STRINGS ARE EQUAL. IF \*

\* UNEQUAL, THE MACRO THEN \*

\* COMPARES THE LENGTHS; IF THE \*

\* FIRST IS LESS THAN SECOND, \*

\* THE CARRY FLAG IS SET TO 0. \*

\* OTHERWISE, IT IS SET TO 1. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = 1ST STRING TO COMPARE \*

\* ]2 = 2ND STRING TO COMPARE \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* SCMP "TEST";"TEST" \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

SCMP MAC

STY SCRATCH

\_MSTR ]1;WPAR1

\_MSTR ]2;WPAR2

JSR STRCMP

LDY SCRATCH

<<<

**MAC.STRINGS >> SCAT**

|  |
| --- |
| **SCAT (macro)**  **Input**:  **]1** = 1st string  **]2** = 2nd string  **Output**:  **.A** = length byte  **RETURN** = new string chars  **RETLEN** = length byte  **Destroys**: AXYNVZCM  **Cycles**: 167+  **Size**: 130 bytes |

The **SCAT** macro takes two strings and concatenates the second string onto the first. This new string is then stored in **RETLEN**/**RETURN**, with the length byte also being passed back via **.A**.

\*

\*``````````````````````````````\*

\* SCAT (NATHAN RIGGS) \*

\* \*

\* CONCATENATE TWO STRINGS \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = FIRST STRING \*

\* ]2 = SECOND STRING \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* SCAT "I AM";" TIRED" \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

SCAT MAC

STY SCRATCH

\_MSTR ]1;WPAR1

\_MSTR ]2;WPAR2

JSR STRCAT

LDY SCRATCH

<<<

**MAC.STRINGS >> SPRN**

|  |
| --- |
| **SPRN (macro)**  **Input**:  **]1** = string to print  **Output**:  **.A** = string length  **Destroys**: AXYNVZCM  **Cycles**: 64+  **Size**: 37 bytes |

The **SPRN** macro simply prints an 8-bit string with a preceding length byte held at a certain address to the screen, via the **COUT1** hook.

\*

\*``````````````````````````````\*

\* SPRN : PRINT STRING \*

\* \*

\* PRINT A STRING TO THE SCREEN \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = STRING TO PRINT \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* SPRN "TESTING" \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

SPRN MAC

STY SCRATCH

\_AXLIT ]1

JSR PRNSTR

LDY SCRATCH

<<<

**MAC.STRINGS >> SPOS**

|  |
| --- |
| **SPOS (macro)**  **Input**:  **]1** = source string  **]2** = substring  **Output**:  **.A** = substring index  **RETURN** = substring index  **RETLEN** = 1  **Destroys**: AXYNVZCM  **Cycles**: 150+  **Size**: 103 bytes |

The **SPOS** macro finds the position of a substring within a larger string and returns that index via **.A** and **RETURN**.

\*

\*``````````````````````````````\*

\* SPOS (NATHAN RIGGS) \*

\* \*

\* FIND THE POSITION OF A SUB- \*

\* STRING IN A GIVEN STRING. \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = SOURCE STRING \*

\* ]2 = SUBSTRING \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* SPOS "A TEST";"TEST" \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

SPOS MAC

STY SCRATCH

\_MSTR ]1;WPAR2

\_MSTR ]2;WPAR1

JSR SUBPOS

LDY SCRATCH

<<<

**MAC.STRINGS >> SCPY**

|  |
| --- |
| **SCPY (macro)**  **Input**:  **]1** = source string  **]2** = substring index  **]3** = substring length  **Output**:  **.A** = new string length  **RETURN** = new string chars  **RETLEN** = length byte  **Destroys**: AXYNVZCM  **Cycles**: 160+  **Size**: 72 bytes |

The **SCPY** macro copies a substring from a source string and stores it in **RETLEN**/**RETURN** as a new string. The length byte is also passed back via **.A**.

\*

\*``````````````````````````````\*

\* SCPY (NATHAN RIGGS) \*

\* \*

\* COPY SUBSTRING FROM STRING \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = SOURCE STRING \*

\* ]2 = SUBSTRING INDEX \*

\* ]3 = SUBSTRING LENGTH \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* SCPY "HELLO WORLD";#7;#5 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

SCPY MAC

STY SCRATCH

\_MSTR ]1;WPAR1

LDA ]2

STA BPAR2

LDA ]3

STA BPAR1

JSR SUBCOPY

LDY SCRATCH

<<<

**MAC.STRINGS >> SDEL**

|  |
| --- |
| **SDEL (macro)**  **Input**:  **]1** = source string  **]2** = substring index  **]3** = substring length  **Output**:  **.A** = new string length  **Destroys**: AXYNVZCM  **Cycles**: 133+  **Size**: 90 bytes |

The **SDEL** macro deletes a substring starting at a given index in a source string for a given length of bytes and then stores the resulting string in **RETLEN**/**RETURN**. The length byte is additionally set back via **.A**.

\*

\*``````````````````````````````\*

\* SDEL (NATHAN RIGGS) \*

\* \*

\* DELETE SUBSTRING FROM STRING \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = SOURCE STRING \*

\* ]2 = SUBSTRING INDEX \*

\* ]3 = SUBSTRING LENGTH \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* SUBDEL "12345";#2;#2 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

SDEL MAC

STY SCRATCH

\_MSTR ]1;WPAR1

LDA ]2

STA BPAR2

LDA ]3

STA BPAR1

JSR SUBDEL

LDY SCRATCH

<<<

**MAC.STRINGS >> SINS**

|  |
| --- |
| **SINS (macro)**  **Input**:  **]1** = string address  **]2** = substring address  **]3** substring index  **Output**:  **.A** = new string length  **RETURN** = new string chars  **RETLEN** = length byte  **Destroys**: AXYNVZCM  **Cycles**: 161+  **Size**: 128 bytes |

The **SINS** macro inserts a substring into another string and holds the result in **RETLEN**/**RETURN**, while also holding the new length in **.A**.

\*

\*``````````````````````````````\*

\* SINS (NATHAN RIGGS) \*

\* \*

\* INSERT SUBSTRING INTO STRING \*

\* \*

\* PARAMETERS \*

\* \*

\* ]1 = STRING ADDRESS \*

\* ]2 = SUBSTRING ADDRESS \*

\* ]3 = SUBSTRING INDEX \*

\* \*

\* SAMPLE USAGE \*

\* \*

\* SINS "1245";"3";#3 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

SINS MAC

STY SCRATCH

\_MSTR ]1;WPAR2

\_MSTR ]2;WPAR1

LDA ]3

STA BPAR1

JSR SUBINS

LDY SCRATCH

<<<

**SUB.PRNSTR >> PRNSTR**

|  |
| --- |
| **PRNSTR (sub)**  **Input**:  **.A** = address low byte  **.X** = address high byte  **Output**:  **.A** = string length  **Destroys**: AXYNVZCM  **Cycles**: 46+  **Size**: 26 bytes |

The **PRNSTR** subroutine prints an 8-bit string with a preceding length byte from the specified address to the screen via **COUT1**, at the current cursor position. The length of the printed string is returned in **.A**.

Note that this is used for strings with a preceding byte length only. Zero-terminated strings, in their limited use, are covered by the **STDIO** library.

\*``````````````````````````````\*

\* PRNSTR (NATHAN RIGGS) \*

\* \*

\* PRINTS STRING TO SCREEN. \*

\* \*

\* INPUT: \*

\* \*

\* .A = ADDRESS LOBYTE \*

\* .X = ADDRESS HIBYTE \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = STRING LENGTH \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 46+ \*

\* SIZE: 26 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]LEN EQU VARTAB ; STRING LENGTH

]STR EQU ADDR1 ; ZERO-PAGE ADDRESS POINTER

\*

PRNSTR

\*

STA ]STR ; STORE LOW BYTE OF STRING ADDR

STX ]STR+1 ; STORE HIGH BYTE OF ADDR

LDY #0 ; RESET .Y COUNTER

LDA (]STR),Y ; GET STRING LENGTH

STA ]LEN ; STORE LENGTH

:LP

INY ; INCREASE COUNTER

LDA (]STR),Y ; GET CHARACTER FROM STRING

JSR SCOUT1 ; PRINT CHARACTER TO SCREEN

CPY ]LEN ; IF Y < LENGTH

BNE :LP ; THEN LOOP

LDA ]LEN ; RETURN LENGTH IN .A

RTS

**SUB.STRCAT >> STRCAT**

|  |
| --- |
| **STRCAT (sub)**  **Input**:  **WPAR1** = 1st string addr  **WPAR2** = 2nd string addr  **Output**:  **.A** = new string length  **RETURN** = new string  **RETLEN** = length byte  **Destroys**: AXYNVZCM  **Cycles**: 115+  **Size**: 75 bytes |

The **STRCAT** subroutine concatenates two strings and stores the new string in **RETURN**, holding the length byte in **RETLEN** as well as in **.A**.

Note that when printing or copying the new string, you should reference it at **RETLEN** in order to include the length byte as part of the string. As such:

SPRN #RETURN

Will cause an error, whereas the proper way to print the returned string is:

SPRN #RETLEN

\*``````````````````````````````\*

\* STRCAT (NATHAN RIGGS) \*

\* \*

\* CONCATENATE TWO STRINGS AND \*

\* STORE THE NEW STRING IN \*

\* RETURN, WITH THE LENGTH BYTE \*

\* AT RETLEN. \*

\* \*

\* NOTE THAT THE WHOLE STRING \*

\* IS ACTUALLY PLACED IN RETLEN \*

\* TO ACCOUNT FOR THE LENGTH \*

\* BYTE THAT PRECEDES IT. \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = 1ST STRING \*

\* WPAR2 = 2ND STRING ADDRESS \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = NEW STRING LENGTH \*

\* RETURN = NEW STRING ADDRESS \*

\* RETLEN = NEW STRING LENGTH \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 115+ \*

\* SIZE: 75 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]S1LEN EQU VARTAB+1 ; FIRST STRING LENGTH

]S2LEN EQU VARTAB+3 ; SECOND STRING LENGTH

]INDEX EQU WPAR3 ; ADDRESS TO PLACE 2ND STRING

]STR2 EQU WPAR2 ; POINTER TO 2ND STRING

]STR1 EQU WPAR1 ; POINTER TO 1ST STRING

\*

STRCAT

\*

LDY #0 ; CLEAR INDEX POINTER

LDA (]STR1),Y ; GET LENGTH OF 1ST STRING

STA ]S1LEN ; STORE IN 1ST STRING LENGTH

LDA (]STR2),Y ; GET LENGTH OF 2ND STRING

STA ]S2LEN ; STORE 2ND STRING LENGTH

\*

\*\* DETERMINE NUMBER OF CHAR

\*

LDA ]S2LEN ; GET 2ND STRING LENGTH

CLC ; CLEAR CARRY

ADC ]S1LEN ; ADD TO LENGTH OF 1ST STRING

STA RETLEN ; SAVE SUM OF TWO LENGTHS

BCC :DOCAT ; NO OVERFLOW, JUST CONCATENATE

LDA #255 ; OTHERWISE, 255 IS MAX

STA RETLEN

\*

:DOCAT

\*

LDY #0 ; OFFSET INDEX BY 1

:CAT1

INY

LDA (]STR1),Y ; LOAD 1ST STRING INDEXED CHAR

STA RETLEN,Y ; STORE IN RETURN AT SAME INDEX

CPY ]S1LEN ; IF .Y < 1ST STRING LENGTH

BNE :CAT1 ; THEN LOOP UNTIL FALSE

\*

TYA ; TRANSFER COUNTER TO .A

CLC ; CLEAR CARRY

ADC #<RETLEN ; ADD LOW BYTE OF RETLEN ADDRESS

STA ]INDEX ; STORE AS NEW ADDRESS LOW BYTE

LDA #0 ; NOW ADJUST HIGH BYTE

ADC #>RETLEN+1 ; OF NEW ADDRESS

STA ]INDEX+1 ; AND STORE HIGH BYTE

CLC ; RESET CARRY

LDY #0

\*

:CAT2

INY

LDA (]STR2),Y ; LOAD 2ND STRING INDEXED CHAR

STA (]INDEX),Y ; STORE AT NEW ADDRESS

CPY RETLEN ; IF .Y < 2ND STRING LENGTH

BEQ :EXIT

BNE :CAT2 ; LOOP UNTIL FALSE

:EXIT

LDA RETLEN ; RETURN NEW LENGTH IN .A

RTS

**SUB.STRCOMP >> STRCMP**

|  |
| --- |
| **STRCMP (sub)**  **Input**:  **WPAR1** = 1st string  **WPAR2** = 2nd string  **Output**:  See description  **Destroys**: AXYNVZCM  **Cycles**: 61+  **Size**: 32 bytes |

The **STRCMP** subroutine takes two strings and compares them, setting the status flags accordingly. First, the strings are tested for being a perfect match. If so, then the **Z** flag is set to **1**; otherwise, it is set to **0**.

Further, if the strings do not match, then the strings are tested regarding length. If the first string has a length smaller than the 2nd, then the **carry** flag is set to **0**; otherwise, it is set to **1**.

\*``````````````````````````````\*

\* STRCMP (NATHAN RIGGS) \*

\* \*

\* COMPARES A STRING TO ANOTHER \*

\* STRING AND SETS THE FLAGS \*

\* ACCORDINGLY: \*

\* \*

\* Z = 1 IF STRINGS MATCH \*

\* Z = 0 IF STRINGS DON'T MATCH \*

\* \*

\* IF THE STRINGS MATCH UP TO \*

\* THE LENGTH OF THE SHORTEST \*

\* STRING, THE STRING LENGTHS \*

\* ARE THEN COMPARED AND THE \*

\* CARRY FLAG IS SET AS SUCH: \*

\* \*

\* C = 0 IF 1ST STRING < 2ND \*

\* C = 1 IF 1ST STRING >= 2ND \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = 1ST STRING ADDRESS \*

\* WPAR2 = 2ND STRING ADDRESS \*

\* \*

\* OUTPUT: \*

\* \*

\* SEE DESCRIPTION \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 61+ \*

\* SIZE: 32 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]STR1 EQU WPAR1 ; ZP POINTER TO 1ST STRING

]STR2 EQU WPAR2 ; ZP POINTER TO 2ND STRING

\*

STRCMP

\*

LDY #0 ; RESET .Y COUNTER

LDA (]STR1),Y ; GET LENGTH OF 1ST STRING

CMP (]STR2),Y ; IF STR1 LENGTH < STR2 LENGTH

BCC :BEGCMP ; THEN BEGIN COMPARISON; ELSE

LDA (]STR2),Y ; USE STR2 LENGTH INSTEAD

:BEGCMP

TAX ; X IS LENGTH OF SHORTER STRING

BEQ :TSTLEN ; IF LENGTH IS 0, TEST LENGTH

LDY #1 ; ELSE SET .Y TO FIRST CHAR OF STRINGS

:CMPLP

LDA (]STR1),Y ; GET INDEXED CHAR OF 1ST STRING

CMP (]STR2),Y ; COMPARE TO INDEXED CHAR OF 2ND

BNE :EXIT ; EXIT IF THE CHARS ARE NOT EQUAL

; Z,C WILL BE PROPERLY SET

INY ; INCREASE CHARACTER INDEX

DEX ; DECREMENT COUNTER

BNE :CMPLP ; CONTINUE UNTIL ALL CHARS CHECKED

:TSTLEN

LDY #0 ; NOW COMPARE LENGTHS

LDA (]STR1),Y ; GET LENGTH OF 1ST STRING

CMP (]STR2),Y ; SET OR CLEAR THE FLAGS

:EXIT

RTS

**SUB.SUBCOPY >> SUBCOPY**

|  |
| --- |
| **SUBCOPY (sub)**  **Input**:  **BPAR1** = substring length  **BPAR2** = substring index  **WPAR1** = source address  **Output**:  **.A** = substring length  **RETURN** = substring chars  **RETLEN** = substring length  **Destroys**: AXYNVZCM  **Cycles**: 46+  **Size**: 27 bytes |

The **SUBCOPY** subroutine copies a substring from a source string and stores the new string into **RETLEN**/**RETURN**. The substring length is additionally returned in **.A**.

\*``````````````````````````````\*

\* SUBCOPY (NATHAN RIGGS) \*

\* \*

\* COPY A SUBSTRING FROM A \*

\* STRING AND STORE IN RETURN. \*

\* \*

\* INPUT: \*

\* \*

\* BPAR1 = SUBSTRING LENGTH \*

\* BPAR2 = SUBSTRING INDEX \*

\* WPAR1 = SOURCE STRING ADDR \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = SUBSTRING LENGTH \*

\* RETURN = SUBSTRING \*

\* RETLEN = SUBSTRING LENGTH \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 46+ \*

\* SIZE: 27 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]SUBLEN EQU BPAR1 ; SUBSTRING LENGTH

]SUBIND EQU BPAR2 ; SUBSTRING INDEX

]STR EQU WPAR1 ; SOURCE STRING

\*

SUBCOPY

\*

LDY ]SUBIND ; STARTING COPY INDEX

LDA ]SUBLEN ; SUBSTRING LENGTH

STA RETLEN ; STORE SUBSTRING LENGTH IN RETLEN

LDX #0

:COPY

LDA (]STR),Y ; GET SUBSTRING CHARACTER

STA RETURN,X ; STORE CHAR IN RETURN

CPX ]SUBLEN ; IF .X COUNTER = SUBSTRING LENGTH

BEQ :EXIT ; THEN FINISHED WITH LOOP

INY ; OTHERWISE, INCREMENT .Y

INX ; AND INCREMENT .X

CLC ; CLEAR CARRY FOR FORCED BRANCH

BCC :COPY ; LOOP

:EXIT

LDA ]SUBLEN ; RETURN SUBSTRING LENGTH IN .A

RTS

**SUB.SUBDEL >> SUBDEL**

|  |
| --- |
| **SUBDEL (sub)**  **Input**:  **BPAR1** = substring length  **BPAR2** = substring index  **WPAR1** = source address  **Output**:  **.A** = string length  **RETURN** = new string chars  **RETLEN** = length byte  **Destroys**: AXYNVZCM  **Cycles**: 79+  **Size**: 47 bytes |

The **SUBDEL** subroutine deletes a substring at a given index and length from a source string, placing the resulting new string in **RETLEN**/**RETURN**.

\*``````````````````````````````\*

\* SUBDEL (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* .A = ADDRESS LOBYTE \*

\* .X = ADDRESS HIBYTE \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = STRING LENGTH \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 79+ \*

\* SIZE: 47 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]SUBLEN EQU BPAR1

]SUBIND EQU BPAR2

]STR EQU WPAR1

\*

SUBDEL

\*

DEC ]SUBIND

INC ]SUBLEN

LDY #0 ; RESET .Y INDEX

LDA (]STR),Y ; GET STRING LENGTH

SEC ; SET CARRY

SBC ]SUBLEN ; SUBTRACT SUBSTRING LENGTH

STA RETLEN ; STORE NEW LENGTH IN RETLEN

INC RETLEN

:LP1

INY ; INCREASE .Y INDEX

LDA (]STR),Y ; LOAD CHARACTER FROM STRING

STA RETLEN,Y ; STORE IN RETURN

CPY ]SUBIND ; IF .Y != SUBSTRING INDEX

BNE :LP1 ; THEN CONTINUE LOOPING

\*

LDX ]SUBIND ; OTHERWISE, .X INDEX = SUBSTRING INDEX

TYA ; TRANSFER .Y INDEX TO .A

CLC ; CLEAR CARRY

ADC ]SUBLEN ; ADD .Y INDEX TO SUBSTRING LENGTH

TAY ; FOR NEW POSITION, THEN BACK TO .Y

DEX

DEY

:LP2

INY ; INCREMENT .Y INDEX

INX ; INCREMEMNT .X INDEX

LDA (]STR),Y ; GET CHAR AT STARTING AFTER SUBSTRING

STA RETURN,X ; STORE IN RETURN AT SEPARATE INDEX

CPX RETLEN ; IF .X != NEW STRING LENGTH,

BNE :LP2 ; CONTINUE LOOPING

:EXIT

LDA RETLEN ; LOAD NEW STRING LENGTH IN .A

RTS

CPY #255 ; IF AT LENGTH MAX

BEQ :EXIT ; THEN QUIT COPYING

**SUB.SUBINS >> SUBINS**

|  |
| --- |
| **SUBINS (sub)**  **Input**:  **WPAR1** = substring addr  **WPAR2** = string address  **BPAR1** = insertion index  **Output**:  **.A** = new string length  **RETURN** = new string chars  **RETLEN** = length byte  **Destroys**: AXYNVZCM  **Cycles**: 106+  **Size**: 67 bytes |

The **SUBINS** subroutine inserts a substring into a destination string at a given index. The new string is stored in **RETLEN**/**RETURN**, with the string length additionally held in **.A**.

\*``````````````````````````````\*

\* SUBINS (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = SUBSTRING ADDRESS \*

\* WPAR2 = STRING ADDRESS \*

\* BPAR1 = INSERTION INDEX \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = NEW STRING LENGTH \*

\* RETURN = NEW STRING CHARS \*

\* RETLEN = NEW STRING LENGTH \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 106+ \*

\* SIZE: 67 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]SUB EQU WPAR1

]STR EQU WPAR2

]INDEX EQU BPAR1

]OLDIND EQU VARTAB

]TMP EQU VARTAB+2

]SUBLEN EQU VARTAB+4

\*

SUBINS

\*

DEC ]INDEX

LDY #0 ; SET .Y INDEX TO 0

LDA (]STR),Y ; GET STRING LENGTH

STA ]TMP ; TEMPORARILY STORE

LDA (]SUB),Y ; GET SUBSTRING LENGTH

STA ]SUBLEN

CLC ; CLEAR CARRY

ADC ]TMP ; ADD SOURCE STRING LENGTH

STA RETLEN ; STORE NEW STRING LENGTH

BCC :CONT ; IF NO OVERFLOW, CONTINUE

LDA #255 ; ELSE, NEW STRING LENGTH IS 255

STA RETLEN ; STORE IN RETLEN

:CONT

\*

LDA ]INDEX ; IF INDEX IS 0, GO STRAIGHT

BEQ :SUBCOPY ; TO COPYING SUBSTRING FIRST

:LP1

INY ; INCREASE INDEX

LDA (]STR),Y ; GET SOURCE STRING CHARACTER

STA RETLEN,Y ; STORE IN RETURN

CPY ]INDEX ; IF WE DON'T HIT SUBSTRING INDEX

BNE :LP1 ; KEEP ON COPYING

:SUBCOPY

STY ]OLDIND ; STORE CURRENT STRING INDEX

TYA ; TRANSFER .Y COUNTER TO

TAX ; .X COUNTER TEMPORARILY

LDY #0 ; RESET .Y COUNTER

:SUBLP

INY ; INCREASE .Y SUBSTRING INDEX

INX ; CONTINUE INCREASING .X INDEX

LDA (]SUB),Y ; LOAD INDEXED CHAR FROM SUBSTRING

STA RETLEN,X ; STORE INTO RETURN AT CONTINUING INDEX

CPY ]SUBLEN ; IF .Y != SUBSTRING LENGTH

BNE :SUBLP ; THEN CONTINUE COPYING

\*

LDY ]OLDIND ; RESTORE OLD INDEX

:FINLP

INY ; INCREASE ORIGINAL INDEX

INX ; INCREASE NEW INDEX

LDA (]STR),Y ; LOAD NEXT CHAR FROM STRING

STA RETLEN,X ; AND STORE AFTER SUBSTRING

CPY ]STR ; IF ORIGINAL STRING LENGTH

BNE :FINLP ; IS NOT YET HIT, KEEP LOOPING

:EXIT

LDA RETLEN ; RETURN NEW LENGTH IN .A

RTS

**DEMO.STRINGS**

The DEMO.STRINGS listing illustrates the usage of each macro in the strings library. It should be remembered that this demo does not exhaustively test the macros and routines in question, nor does it illustrate multiple ways to pass parameters (literal, address, pointer, etc.).

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* \*

\* -< STRINGS DEMO >- \*

\* \*

\* VERSION 00.03.00 \*

\* \*

\* 20-JAN-2019 \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* \*

\* NATHAN D. RIGGS \*

\* NATHAN.RIGGS@OUTLOOK.COM \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\*\* ASSEMBLER DIRECTIVES

\*

CYC AVE

EXP OFF

TR ON

DSK DEMO.STRINGS

OBJ $BFE0

ORG $6000

\*

\*``````````````````````````````\*

\* TOP INCLUDES (PUTS, MACROS) \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PUT MIN.HEAD.REQUIRED

USE MIN.MAC.REQUIRED

USE MIN.MAC.STRINGS

PUT MIN.HOOKS.STRINGS

]HOME EQU $FC58

\*

\*``````````````````````````````\*

\* PROGRAM MAIN BODY \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

JSR ]HOME

\_PRN "STRING MACROS AND SUBROUTINES",8D

\_PRN "=============================",8D8D

\_PRN "THIS DEMO ILLUSTRATES THE USAGE",8D

\_PRN "OF MACROS RELATED TO STRING",8D

\_PRN "MANIPULATION. CURRENTLY, THIS IS ",8D

\_PRN "LIMITED TO 8-BIT STRINGS WITH",8D

\_PRN "A PRECEDING LENGTH BYTE, BUT MAY",8D

\_PRN "ENCOMPASS OTHER TYPES IN THE FUTURE.",8D8D

\_PRN "THE FOLLOWING MACROS WILL BE COVERED:",8D8D

\_PRN " - SPRN",8D

\_PRN " - SCAT",8D

\_PRN " - SCPY",8D

\_PRN " - SDEL",8D

\_PRN " - SINS",8D

\_PRN " - SPOS",8D

\_PRN " - SCMP",8D8D

\_WAIT

JSR ]HOME

\_PRN "THE FIRST AND EASIEST MACRO TO",8D

\_PRN "USE AND EXPLAIN IS SPRN, WHICH ",8D

\_PRN "STANDS FOR STRING PRINT. AS THE",8D

\_PRN "NAME IMPLIES, THIS MACRO PRINTS",8D

\_PRN "THE STRING AT A GIVEN ADDRESS USING",8D

\_PRN "COUT. THUS:",8D8D

\_PRN " SPRN #STR1",8D8D

\_PRN "WILL RETURN:",8D8D

\_WAIT

SPRN #STR1

\_WAIT

JSR ]HOME

\_PRN "THE NEXT MACRO, SCAT, IS USED",8D

\_PRN "TO CONCATENATE ONE STRING TO",8D

\_PRN "ANOTHER, STORING THE NEW STRING",8D

\_PRN "IN RETURN. EITHER A LITERAL",8D

\_PRN "STRING OR AN ADDRESS CAN BE USED",8D

\_PRN "IN EACH PARAMETER. THUS:",8D8D

\_PRN " SCAT 'HELLO,';' WORLD!'",8D

\_PRN " SPRN #RETLEN",8D8D

\_PRN "WILL RETURN:",8D8D

\_WAIT

SCAT "HELLO,";" WORLD!"

SPRN #RETLEN

\_WAIT

JSR ]HOME

\_PRN "THE NEXT MACRO IS SCPY, WHICH",8D

\_PRN "STANDS FOR SUBSTRING COPY. THIS",8D

\_PRN "MACRO COPIES A SUBSTRING FROM A",8D

\_PRN "GIVEN STRING (LITERAL OR ADDRESS)",8D

\_PRN "AT THE GIVEN INDEX AND LENGTH,",8D

\_PRN "STORING IT IN RETURN. THUS:",8D8D

\_PRN " SCPY 'KILL ALL HUMANS';#1;#8",8D

\_PRN " SPRN #RETLEN",8D8D

\_PRN "RETURNS:",8D8D

\_WAIT

SCPY "KILL ALL HUMANS";#1;#8

SPRN #RETLEN

\_WAIT

JSR ]HOME

\_PRN "THE NEXT MACRO, SDEL, DELETES",8D

\_PRN "A SUBSTRING FROM A GIVEN STRING",8D

\_PRN "AND RETURNS THE NEW STRING IN",8D

\_PRN "RETURN. THUS:",8D8D

\_PRN " SDEL 'HELLO, WORLD!';#6;#8",8D

\_PRN " SPRN #RETLEN",8D8D

\_PRN "RETURNS:",8D8D

\_WAIT

SDEL "HELLO, WORLD!";#6;#8

SPRN #RETLEN

\_WAIT

JSR ]HOME

\_PRN "THE SPOS MACRO LOOKS FOR A",8D

\_PRN "GIVEN SUBSTRING WITHIN A GIVEN",8D

\_PRN "STRING, RETURNING 0 IF NO MATCH ",8D

\_PRN "IS FOUND OR RETURNING THE INDEX AT",8D

\_PRN "WHICH THE SUBSTRING IS FOUND. THUS:",8D8D

\_PRN " SPOS 'I HATE CAPITALISM';'CAPITALISM'",8D

\_PRN " ",8D

\_PRN "WILL RETURN:",8D8D

\_WAIT

SPOS "I HATE CAPITALISM";"CAPITALISM"

DUMP #RETURN;#1

\_WAIT

JSR ]HOME

\_PRN "NEXT WE HAVE THE SINS MACRO, WHICH",8D

\_PRN "STANDS FOR 'SUBSTRING INSERT.' THIS",8D

\_PRN "MACRO INSERTS A SUBSTRING INTO A ",8D

\_PRN "SOURCE STRING AT A GIVEN POSITION AND",8D

\_PRN "PUTS THE NEW STRING IN RETURN. THUS:",8D8D

\_PRN " SINS 'I LOVE BABIES';' TO HATE';#7",8D8D

\_PRN "WILL RETURN:",8D8D

\_WAIT

SINS "I LOVE BABIES";" TO HATE";#7

SPRN #RETLEN

\_WAIT

JSR ]HOME

\_PRN "LASTLY WE HAVE THE SCMP MACRO, WHICH",8D

\_PRN "STANDS FOR 'STRING COMPARE.' THIS MACRO",8D

\_PRN "COMPARES TWO STRINGS AND SETS STATUS",8D

\_PRN "FLAGS ACCORDINGLY, MAINLY THE ZERO",8D

\_PRN "FLAG AND THE CARRY FLAG.",8D8D

\_WAIT

\_PRN "THE ZERO FLAG IS SET TO 0 IF THE",8D

\_PRN "STRINGS ARE AN EXACT MATCH; OTHERWISE",8D

\_PRN "THE ZERO FLAG IS SET TO 1. IF THE",8D

\_PRN "STRINGS DON'T MATCH, THEY ARE TESTED",8D

\_PRN "TO SEE IF THEY ARE THE SAME LENGTH.",8D

\_PRN "IF THE FIRST STRING IS SMALLER, THEN",8D

\_PRN "THE CARRY IS SET TO 0; IF IT IS ",8D

\_PRN "EQUAL TO OR LARGER THAN THE 2ND, THEN",8D

\_PRN "THE CARRY IS SET TO 1.",8D8D

\_WAIT

\_PRN "THESE CAN BE TESTED BY USING",8D

\_PRN "BRANCH INSTRUCTIONS LIKE BEQ FOR THE ",8D

\_PRN "ZERO FLAG OR BCC FOR THE CARRY. THUS:",8D8D

\_WAIT

\_PRN " SCMP 'TEST';'TEST'",8D

\_PRN " BEQ :NOMATCH",8D

\_PRN " \_PRN 'THE STRINGS MATCH!'",8D

\_PRN " JMP :EXIT",8D

\_PRN " :NOMATCH",8D

\_PRN " \_PRN 'STRINGS DO NOT MATCH!'",8D

\_PRN " :EXIT",8D8D

\_PRN "WILL RETURN:",8D8D

\_WAIT

SCMP "TEST";"TEST"

BEQ NOMATCH

\_PRN "THE STRINGS MATCH!",8D8D

JMP EXIT1

NOMATCH

\_PRN "THE STRINGS DO NOT MATCH!",8D8D

EXIT1

\_WAIT

JSR ]HOME

\_PRN "FIN.",8D8D

\*

JMP $3D0

\*

\*``````````````````````````````\*

\* BOTTOM INCLUDES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\*\* BOTTOM INCLUDES

\*

PUT MIN.LIB.REQUIRED

\*

\*\* INDIVIDUAL SUBROUTINE INCLUDES

\*

\*\* STRING SUBROUTINES

\*

PUT MIN.SUB.PRNSTR

PUT MIN.SUB.STRCAT

PUT MIN.SUB.STRCOMP

\*

\*\* SUBSTRING SUBROUTINES

\*

PUT MIN.SUB.SUBCOPY

PUT MIN.SUB.SUBDEL

PUT MIN.SUB.SUBINS

PUT MIN.SUB.SUBPOS

\*

STR1 STR "TEST STRING 1"

STR2 STR "TEST STRING 2"

SUB1 STR "-SUBTEST1-"

STR3 STR "TEST STRING 2"

SUB2 STR "STRING"

**DISK 6: FILEIO**

The FILEIO library contains macros and subroutines dedicated to file input and output. For the most part, these use the standard DOS 3.3 and Applesoft commands in order to keep compatibility with most systems. These will not work without DOS.

It should be noted that any executables that use this library should be BLOADED into memory and then run through the monitor, rather than using BRUN. Alternately, the MAKEEXEC utility included on the disk can be used to create an EXEC file that automatically does this upon execution.

The FILEIO disk includes the following files:

* DEMO.FILEIO
* HOOKS.FILEIO
* MAC.FILEIO
* SUB.BINLOAD
* SUB.BINSAVE
* SUB.DISKRW
* SUB.FINPUT
* SUB.FPRINT
* SUB.FPSTR

**HOOKS.FILEIO**

The HOOKS.FILEIO file contains hooks related to reading and writing to the disk. Many of these are unused by the library, but are included for use by the programmer.

\*

\*``````````````````````````````\*

\* HOOKS.FILEIO \*

\* \*

\* THIS FILE CONTAINS MANY OF \*

\* THE HOOKS RELATED TO FILE \*

\* INPUT AND OUTPUT. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 21-SEP-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

STEP00 EQU $C080 ; DISK STEPPER PHASE 0 OFF

STEP01 EQU $C081 ; DISK STEPPER PHASE 0 ON

STEP10 EQU $C082 ; DISK STEPPER PHASE 1 OFF

STEP11 EQU $C083 ; DISK STEPPER PHASE 1 ON

STEP20 EQU $C084 ; DISK STEPPER PHASE 2 OFF

STEP21 EQU $C085 ; DISK STEPPER PHASE 2 ON

STEP30 EQU $C086 ; DISK STEPPER HAPSE 3 OFF

STEP31 EQU $C087 ; DISK STEPPER PHASE 3 ON

MOTON EQU $C088 ; DISK MAIN MOTOR OFF

MOTOFF EQU $C089 ; DISK MAIN MOTOR ON

DRV0EN EQU $C08A ; DISK ENABLE DRIVE 1

DRV1EN EQU $C08B ; DISK ENABLE DRIVE 2

Q6CLR EQU $C08C ; DISK Q6 CLEAR

Q6SET EQU $C08D ; DISK Q6 SET

Q7CLR EQU $C08E ; DISK Q7 CLEAR

Q7SET EQU $C08F ; DISK Q7 SET

CWRITE EQU $FECD ; WRITE TO CASSETTE TAPE

CREAD EQU $FEFD ; READ FROM CASSETTE TAPE

IOB EQU $B7E8 ; INPUT/OUTPUT AND CONTROL

; BLOCK TABLE

IOB\_SLOT EQU $B7E9 ; SLOT NUMBER

IOB\_DRIV EQU $B7EA ; DRIVE NUMBER

IOB\_EVOL EQU $B7EB ; EXPECTED VOLUME NUMBER

IOB\_TRAK EQU $B7EC ; DISK TRACK

IOB\_SECT EQU $B7ED ; DISK SECTOR

IOB\_DCTL EQU $B7EE ; LOW ORDER BYTE OF THE

; DEVICE CARACTERISTIC TBL

IOB\_DCTH EQU $B7EF ; HIGH ORDER OF DCT

IOB\_BUFL EQU $B7F0 ; LOW ORDER OF BUFFER

IOB\_BUFH EQU $B7F1 ; HIGH

IOB\_COMM EQU $B7F4 ; COMMAND CODE; READ/WRITE

IOB\_ERR EQU $B7F5 ; ERROR CODE

IOB\_AVOL EQU $B7F6 ; ACTUAL VOL NUMBER

IOB\_PRES EQU $B7F7 ; PREVIOUS SLOT ACCESSED

IOB\_PRED EQU $B7F8 ; PREVIOUS DRIVE ACCESSED

RWTS EQU $3D9 ; DOS RWTS ROUTINE

FCOUT EQU $FDED ; COUT SUBROUTINE

LANG EQU $AAB6 ; DOS LANGUAGE INDICATOR

CURLIN EQU $75

PROMPT EQU $33

FGET EQU $FD0C ; MONITOR GETKEY ROUTINE

FGETLN EQU $FD6F ; MON GETLN ROUTINE

DOSERR EQU $DE ; DOS ERROR LOC

**MAC**.**FILEIO**

The MAC.FILEIO library holds all of the macros related to disk input and output. This currently includes:

* BSAVE
* BLOAD
* AMODE
* CMD
* FPRN
* FINP
* SLOT
* DRIVE
* TRACK
* SECT
* DSKR
* DSKW
* DBUFF
* DRWTS

\*``````````````````````````````\*

\* FILEIO.MAC \*

\* \*

\* THIS IS A MACRO LIBRARY FOR \*

\* FILE INPUT AND OUTPUT, AS \*

\* WELL AS DISK OPERATIONS. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 21-SEP-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\* \*

\* SUBROUTINE FILES USED \*

\* \*

\* SUB.BINLOAD \*

\* SUB.BINSAVE \*

\* SUB.DISKRW \*

\* SUB.DOSCMD \*

\* SUB.FINPUT \*

\* SUB.FPRINT \*

\* SUB.FPSTR \*

\* \*

\* LIST OF MACROS \*

\* \*

\* BSAVE : BINARY SAVE \*

\* BLOAD : BINARY LOAD \*

\* AMODE : TURN ON APPLESOFT \*

\* CMD : EXECUTE DOS COMMAND \*

\* FPRN : PRINT TO FILE \*

\* FINP : INPUT LINE FROM FILE \*

\* SLOT : SET RWTS SLOT \*

\* DRIVE : SET RWTS DRIVE \*

\* TRACK : SET RWTS TRACK \*

\* SECT : SET RWTS SECTOR \*

\* DSKR : SET RWTS READ \*

\* DSKW : SET RWTS WRITE \*

\* DBUFF : SET BUFFER ADDRESS \*

\* DRWTS : CALL THE RWTS ROUTE \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

**MAC.FILEIO >> BLOAD**

|  |
| --- |
| **BLOAD (mac)**  **Input**:  **]1** = string pointer  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 158+  **Size**: 110 bytes |

The **BLOAD** macro works in the same way as the **BLOAD** command in **DOS**: it simply loads data from a binary file into its appropriate location in memory.

\*

\*``````````````````````````````\*

\* BLOAD (NATHAN RIGGS) \*

\* \*

\* LOAD INTO THE GIVEN ADDRESS \*

\* THE SPECIFIED BINARY FILE. \*

\* \*

\* PARAMETERS: \*

\* \*

\* ]1 = COMMAND STRING OR PTR \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* BLOAD "TEST,A$300" \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

BLOAD MAC

STY SCRATCH

\_MSTR ]1;WPAR1

JSR BINLOAD

LDY SCRATCH

<<<

**MAC.FILEIO >> BSAVE**

|  |
| --- |
| **BSAVE (mac)**  **Input**:  **]1** = string pointer  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 124+  **Size**: 82 bytes |

The **BSAVE** macro saves a given range of memory at a given address. This works the same as the **DOS** **BSAVE** command. The address and length are sent as part of the string, as such:

BSAVE “file,A$6000,L256”

\*

\*``````````````````````````````\*

\* BSAVE (NATHAN RIGGS) \*

\* \*

\* SAVE THE GIVEN ADDRESS RANGE \*

\* TO THE SPECIFIED FILE NAME. \*

\* \*

\* PARAMETERS: \*

\* \*

\* ]1 = ADDRESS OF CDM STR \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* BSAVE "TEST,A$300,L$100" \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

BSAVE MAC

STY SCRATCH

\_MSTR ]1;WPAR1

JSR BINSAVE

LDY SCRATCH

<<<

**MAC.FILEIO >> AMODE**

|  |
| --- |
| **AMODE (mac)**  **Input**:  none  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 8+  **Size**: 9 bytes |

The **AMODE** macro “tricks” **DOS** into thinking it is in Applesoft mode. This is primarily used with **FILEIO** operations because they require **DOS** to run in non-immediate mode.

\*

\*``````````````````````````````\*

\* AMODE (NATHAN RIGGS) \*

\* \*

\* FOOLS DOS INTO THINKING THAT \*

\* WE ARE IN INDIRECT MODE TO \*

\* ALLOW FOR TEXT FILE READ AND \*

\* WRITE OPERATIONS. \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* AMODE \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

AMODE MAC

LDA #1

STA $AAB6 ; DOS LANG FLAG

STA $75+1 ; NOT IN DIRECT MODE

STA $33 ; NOT IN DIRECT MODE

<<<

**MAC.FILEIO >> CMD**

|  |
| --- |
| **CMD (mac)**  **Input**:  **]1** = string pointer  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 76+  **Size**: 52 bytes |

The **CMD** macro executes a **DOS** command that is passed via string.

\*

\*``````````````````````````````\*

\* CMD (NATHAN RIGGS) \*

\* \*

\* SIMPLY EXECUTES THE DOS CMD \*

\* AS IT IS PROVIDED IN THE \*

\* STRING PASSED AS PARAMETER 1 \*

\* \*

\* PARAMETERS: \*

\* \*

\* ]1 = COMMAND STRING \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* CMD "CATALOG" \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

CMD MAC

STY SCRATCH

\_MSTR ]1;WPAR1

JSR DOSCMD

LDY SCRATCH

<<<

**MAC.FILEIO >> FPRN**

|  |
| --- |
| **FPRN (mac)**  **Input**:  **]1** = string  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 75+  **Size**: 69 bytes |

The **FPRN** macro outputs a null-terminated string to the open file.

\*

\*``````````````````````````````\*

\* FPRN (NATHAN RIGGS) \*

\* \*

\* PRINTS THE GIVEN STRING TO \*

\* THE FILE THAT IS OPEN FOR \*

\* WRITING. IF MEMORY ADDRESS \*

\* IS PASSED, THEN PRINT THE \*

\* STRING THAT IS AT THAT \*

\* LOCATION. \*

\* \*

\* PARAMETERS: \*

\* \*

\* ]1 = EITHER A STRING OR \*

\* MEMLOC OF STRING \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* FPRN "TESTING" \*

\* FPRN $300 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

FPRN MAC

STY SCRATCH

IF ",]1

JSR FPRINT

ASC ]1

HEX 8D00

ELSE ; IF PARAM IS ADDR

\_ISLIT ]1

JSR FPSTR ; PRINT STRING

FIN

LDY SCRATCH

<<<

**MAC.FILEIO >> FSPRN**

|  |
| --- |
| **FSPRN (mac)**  **Input**:  **]1** = string or address  **Output**:  **.A** = string length  **Destroys**: AXYNVZCM  **Cycles**: 70+  **Size**: 25 bytes |

The **FSPRN** macro outputs the contents of a string with a preceding length byte to an open file. Only the characters are written to the file; the length byte is not.

\*

\*``````````````````````````````\*

\* FSPRN (NATHAN RIGGS) \*

\* \*

\* PRINTS A STRING WITH A \*

\* PRECEDING LENGTH BYTE TO A \*

\* FILE. \*

\* \*

\* PARAMETERS: \*

\* \*

\* ]1 = EITHER A STRING OR \*

\* MEMLOC OF STRING \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* FPRN "TESTING" \*

\* FPRN $300 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

FSPRN MAC

STY SCRATCH

\_MLIT ]1;WPAR1

JSR FPSTR

LDY SCRATCH

<<<

**MAC.FILEIO >> FINP**

|  |
| --- |
| **FINP (mac)**  **Input**:  none  **Output**:  **.A** = string length  **RETURN** = string chars  **RETLEN** = length byte  **Destroys**: AXYNVZCM  **Cycles**: 64+  **Size**: 49 bytes |

The **FINP** macro reads a line of input from a text file (ended with a carriage return), and transfers it to **RETURN**. The length byte is stored in **RETLEN** and in **.A**.

\*

\*``````````````````````````````\*

\* FINP (NATHAN RIGGS) \*

\* \*

\* GETS A LINE OF TEXT FROM THE \*

\* FILE OPEN FOR READING AND \*

\* STORES IT AD THE ADDRRESS \*

\* SPECIFIED IN THE PARAMETERS. \*

\* \*

\* PARAMETERS: \*

\* \*

\* NONE, SAVE FOR OPEN FILE \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* FINP $300 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

FINP MAC

STY SCRATCH

JSR FINPUT

LDY SCRATCH

<<<

**MAC.FILEIO >> SLOT**

|  |
| --- |
| **SLOT (mac)**  **Input**:  **]1** = slot number  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 14+  **Size**: 14 bytes |

Change the slot for **RWTS** routines. In terms of this library, that refers primarily to **DSKRW**.

\*

\*``````````````````````````````\*

\* SLOT (NATHAN RIGGS) \*

\* \*

\* CHANGES THE SLOT VALUE IN \*

\* THE IOB TABLE FOR THE RWTS \*

\* ROUTINE. JUST USES DOS IOB. \*

\* \*

\* PARAMETERS: \*

\* \*

\* ]1 = SLOT NUMBER \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* SLOT #6 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

SLOT MAC

\*

LDA ]1

STA SCRATCH

ASL SCRATCH

ASL SCRATCH

ASL SCRATCH

ASL SCRATCH ; MUL BY 16

LDA SCRATCH

STA IOB\_SLOT

<<<

**MAC.FILEIO >> DRIVE**

|  |
| --- |
| **DRIVE (mac)**  **Input**:  **]1** = drive number  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 6+  **Size**: 5 bytes |

Change the drive for **RWTS** routines. In terms of this library, that refers primarily to **DSKRW**.

\*

\*``````````````````````````````\*

\* DRIVE (NATHAN RIGGS) \*

\* \*

\* CHANGES THE DRIVE VALUE IN \*

\* THE IOB TABLE FOR THE RWTS \*

\* ROUTINE. JUST USES DOS IOB. \*

\* \*

\* PARAMETERS: \*

\* \*

\* ]1 = DRIVE NUMBER \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* DRIVE #1 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

DRIVE MAC

\*

LDA ]1

STA IOB\_DRIV

<<<

**MAC.FILEIO >> TRACK**

|  |
| --- |
| **TRACK (mac)**  **Input**:  **]1** = track number  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 4+  **Size**: 4 bytes |

Change the track for **RWTS** routines. In terms of this library, that refers primarily to **DSKRW**.

\*

\*``````````````````````````````\*

\* TRACK (NATHAN RIGGS) \*

\* \*

\* CHANGES THE TRACK VALUE IN \*

\* THE IOB TABLE FOR THE RWTS \*

\* ROUTINE. JUST USES DOS IOB. \*

\* \*

\* PARAMETERS: \*

\* \*

\* ]1 = TRACK NUMBER \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* TRACK #5 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

TRACK MAC

\*

LDA ]1

STA IOB\_TRAK

<<<

**MAC.FILEIO >> SECT**

|  |
| --- |
| **SECT (mac)**  **Input**:  **]1** = sector number  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 4+  **Size**: 4 bytes |

Change the sector for **RWTS** routines. In terms of this library, that refers primarily to **DSKRW**.

\*

\*``````````````````````````````\*

\* SECT (NATHAN RIGGS) \*

\* \*

\* CHANGES THE SECTOR VALUE IN \*

\* THE IOB TABLE FOR THE RWTS \*

\* ROUTINE. JUST USES DOS IOB. \*

\* \*

\* PARAMETERS: \*

\* \*

\* ]1 = SECTOR NUMBER \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* SECT #3 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

SECT MAC

\*

LDA ]1

STA IOB\_SECT

<<<

**MAC.FILEIO >> DSKR**

|  |
| --- |
| **DSKR (mac)**  **Input**:  none  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 5+  **Size**: 5 bytes |

Sets the **DRTWS** subroutine to read mode.

\*

\*``````````````````````````````\*

\* DSKR (NATHAN RIGGS) \*

\* \*

\* CHANGES THE RWTS COMMAND TO \*

\* READ ($01). \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* SETDR \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

DSKR MAC

\*

LDA $01

STA IOB\_COMM

<<<

**MAC.FILEIO >> DSKW**

|  |
| --- |
| **DSKW (mac)**  **Input**:  none  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 4+  **Size**: 5 bytes |

Sets the **DRWTS** subroutine to write mode.

\*

\*``````````````````````````````\*

\* DSKW (NATHAN RIGGS) \*

\* \*

\* CHANGES THE RWTS COMMAND TO \*

\* WRITE ($02). \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* SETDW \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

DSKW MAC

\*

LDA $02

STA IOB\_COMM

<<<

**MAC.FILEIO >> DBUFF**

|  |
| --- |
| **DBUFF (mac)**  **Input**:  **]1** = address  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 13+  **Size**: 10 bytes |

Set the disk buffer address.

\*

\*``````````````````````````````\*

\* DBUFF (NATHAN RIGGS) \*

\* \*

\* CHANGES THE BUFFER ADDRESS \*

\* FOR THE RWTS SUBROUTINE \*

\* \*

\* PARAMETERS: \*

\* \*

\* ]1 = BUFFER ADDRESS \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* DBUFF $300 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

DBUFF MAC

\*

LDA #<]1

STA IOB\_BUFL

LDA #>]1

STA IOB\_BUFH

<<<

**MAC.FILEIO >> DRWTS**

|  |
| --- |
| **DRWTS (mac)**  **Input**:  none  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 45+  **Size**: 38 bytes |

The **DRWTS** macro either reads or writes to the disk at the sector, track, volume, slot and drive that is set by the preceding macros. If **DSKR** is invoked, then **DRWTS** is set to read mode; if **DSKW** is invoked, then the macro writes to the disk.

\*

\*``````````````````````````````\*

\* DRWTS (NATHAN RIGGS) \*

\* \*

\* RUNS THE RWTS ROUTINE AFTER \*

\* THE APPROPRIATE VARIABLES IN \*

\* THE IOB TABLE ARE SET. \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* DRWTS \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

DRWTS MAC

\*

STY SCRATCH

JSR DISKRW

LDY SCRATCH

<<<

**SUB.BINLOAD >> BINLOAD**

|  |
| --- |
| **BINLOAD (sub)**  **Input**:  **WPAR1** = string address  pointer  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 124+  **Size**: 82 bytes |

The **BINLOAD** subroutine loads a binary file into memory. The string passed as a parameter should follow the exact same conventions as is used in **DOS**.

\*``````````````````````````````\*

\* BINLOAD (NATHAN RIGGS) \*

\* \*

\* SIMPLY BLOADS FILE IN MEMORY \*

\* AS SPECIFIED BY THE STRING \*

\* PASSED AS A PARAMETER. \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = STRING ADDRESS PTR \*

\* \*

\* OUTPUT: \*

\* \*

\* NONE \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 124+ \*

\* SIZE: 82 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]SLEN EQU VARTAB

]ADDR EQU WPAR1

\*

BINLOAD

\*

LDA #1 ; TELL DOS TO ENTER APPLESOFT

STA $AAB6 ; MODE; SWITCH DOS LANG FLAG

STA $75+1 ; NOT IN DIRECT MODE

STA $33 ; NOT IN DIRECT MODE

LDA #$8D ; CARRIAGE RETURN

JSR FCOUT ; SEND TO COUT

LDA #$84 ; CTRL-D FOR DOS COMMAND

JSR FCOUT ; SEND TO COUT

LDA #$C2 ; B

JSR FCOUT ; SEND TO COUT

LDA #$CC ; L

JSR FCOUT ; SEND TO COUT

LDA #$CF ; O

JSR FCOUT ; SEND TO COUT

LDA #$C1 ; A

JSR FCOUT ; SEND TO COUT

LDA #$C4 ; D

JSR FCOUT ; SEND TO COUT

LDA #$A0 ; [SPACE]

JSR FCOUT ; SEND TO COUT

LDY #0 ; RESET .Y INDEX

LDA (]ADDR),Y ; GET STRING LENGTH

STA ]SLEN ; STORE IN ]SLEN

LDY #1 ; SET INDEX TO FIRST CHAR

:LP

LDA (]ADDR),Y ; GET CHAR

JSR FCOUT ; SEND TO COUT

INY ; INCREASE INDEX

CPY ]SLEN ; IF .Y < STRING LENGTH,

BCC :LP ; CONTINUE LOOPING

BEQ :LP ; IF =, LOOP

LDA #$8D ; CARRIAGE RETURN

JSR FCOUT ; SEND TO COUT

RTS

**SUB.BINSAVE >> BINSAVE**

|  |
| --- |
| **BINSAVE (sub)**  **Input**:  **WPAR1** = string address  pointer  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 124+  **Size**: 82 bytes |

The **BINSAVE** subroutine retrieves the data at a given memory location and stores it on the disk under the given filename. The string passed should follow the same format as **BSAVE** on the command line, with the address and length specified as **DOS** parameters as so:

“file,A$6000,L256”

\*``````````````````````````````\*

\* BINSAVE (NATHAN RIGGS) \*

\* \*

\* SIMPLY DOES A BINARY SAVE \*

\* WITH THE COMMAND LINE PARAMS \*

\* SPECIFIED IN THE STRING AT \*

\* THE GIVEN ADDRESS. \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = STRING ADDRESS PTR \*

\* \*

\* OUTPUT: \*

\* \*

\* NONE \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 124+ \*

\* SIZE: 82 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]SLEN EQU VARTAB

]ADDR EQU WPAR1

\*

BINSAVE

\*

LDA #1 ; SET APPLESOFT MODE

STA $AAB6 ; 1ST, SET DOS LANG FLAG

STA $75+1 ; NOT IN DIRECT MODE

STA $33 ; NOT IN DIRECT MODE

LDA #$8D ; CARRIAGE RETURN

JSR FCOUT ; SEND TO COUT

LDA #$84 ; CTRL-D FOR DOS COMMAND

JSR FCOUT ; SEND TO COUT

LDA #$C2 ; B

JSR FCOUT ; SEND TO COUT

LDA #$D3 ; S

JSR FCOUT ; SEND TO COUT

LDA #$C1 ; A

JSR FCOUT ; SEND TO COUT

LDA #$D6 ; V

JSR FCOUT ; SEND TO COUT

LDA #$C5 ; E

JSR FCOUT ; SEND TO COUT

LDA #$A0 ; [SPACE]

JSR FCOUT ; SEND TO COUT

LDY #0 ; RESET INDEX TO 0

LDA (]ADDR),Y ; GET STRING LENGTH

STA ]SLEN ; STORE IN SLEN

LDY #1 ; SET INDEX TO 1ST CHAR

:LP

LDA (]ADDR),Y ; LOAD CHAR

JSR FCOUT ; SEND TO COUT

INY ; INCREASE INDEX

CPY ]SLEN ; IF .Y <= STRING LENGTH,

BCC :LP ; THEN CONTINUE LOOPING

BEQ :LP

LDA #$8D ; ELSE LOAD CARRIAGE RETURN

JSR FCOUT ; SEND TO COUT

RTS

**SUB.DISKRW >> DISKRW**

|  |
| --- |
| **DISKRW (sub)**  **Input**:  See description  **Output**:  **.A** = error code  **RETURN** = byte  read/written  **RETLEN** = 1  **Destroys**: AXYNVZCM  **Cycles**: 41+  **Size**: 34 bytes |

The **DISKRW** subroutine initiates either a read or a write to the disk, depending on whether the programmer has used the **DSKR** macro to set read mode or **DSKW** to set write mode. The slot, drive, volume and sector to be written to or read from are also set by the appropriate macros.

If read mode is set by **DSKR**, then **DISKRW** passes the byte read via **RETURN**. If write mode is set by **DSKW**, however, then the byte to write to the disk is first put into **RETURN**.

\*``````````````````````````````\*

\* DISKRW (NATHAN RIGGS) \*

\* \*

\* GENERAL PURPOSE ROUTINE FOR \*

\* READING AND WRITING TO A \*

\* \*

\* INPUT: \*

\* \*

\* SLOT, DRIVE, VOLUME AND \*

\* SECTOR, AS WELL AS READ OR \*

\* WRITE FLAG, SHOULD BE SET \*

\* BEFORE CALLING SUBROUTINE \*

\* \*

\* RETURN = BYTE TO WRITE, IF \*

\* IN WRITE MODE \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = ERROR CODE, IF ANY \*

\* RETURN = BYTE RETURNED, IF \*

\* IN READ MODE \*

\* RETLEN = 1 \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 41+ \*

\* SIZE: 34 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

DISKRW

\*

:CLEAR

LDA #00 ; CLEAR EXPECTED

STA IOB\_EVOL ; VOLUME BYTE

LDA #1 ; BUFFER IS ALWAYS

STA RETLEN ; A SINGLE BYTE

LDA #>RETURN ; PASS BUFFER TO RWTS, WHICH

LDY #<RETURN ; IS THE MOMLOC WHERE THE READ

JSR RWTS ; OR WRITE DATA IS PASSED; CALL RWTS

LDA #0 ; CLEAR .A TO INDICATE NO ERRORS

BCC :EXIT ; IF CARRY IS CLEAR, NO ERRORS

:ERR LDA IOB\_ERR ; .A HOLDS ERROR CODE

:EXIT

LDX #00 ; CLEAR THE SCRATCH LOCATION

STX $48 ; USED BY RWTS

RTS

**SUB.DOSCMD >> DOSCMD**

|  |
| --- |
| **DOSCMD (sub)**  **Input**:  **WPAR1** = string address  pointer  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 76+  **Size**: 52 bytes |

The **DOSCMD** subroutine simply executes the **DOS** command specified in the string passed.

\*``````````````````````````````\*

\* DOSCMD (NATHAN RIGGS) \*

\* \*

\* EXECUTES A DOS COMMAND THAT \*

\* IS PASSED VIA A STRING ADDR \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = STRING ADDRESS PTR \*

\* \*

\* OUTPUT: \*

\* \*

\* NONE \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 76+ \*

\* SIZE: 52 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]SLEN EQU VARTAB

]ADDR EQU WPAR1

\*

DOSCMD

\*

LDA #1 ; SET DOS TO APPLESOFT MODE

STA $AAB6 ; BY SWITCHING DOS LANG FLAG

STA $75+1 ; AND SETTING INDIRECT MODE

STA $33 ; NOT DIRECT MODE

LDA #$8D ; CARRIAGE RETURN

JSR FCOUT ; SEND TO COUT

LDA #$84 ; CTRL-D FOR DOS COMMAND

JSR FCOUT ; SEND TO COUT

LDY #0 ; RESET INDEX

LDA (]ADDR),Y ; GET STRING LENGTH

STA ]SLEN ; HOLD IN ]SLEN

LDY #$01 ; SET INDEX TO FIRST CHARACTER

:LP

LDA (]ADDR),Y ; LOAD CHARACTER

JSR FCOUT ; SEND TOU COUT

INY ; INCREASE INDEX

CPY ]SLEN ; IF .Y <= STRING LENGTH

BCC :LP ; THEN KEEP LOOPING

BEQ :LP

LDA #$8D ; OTHERWISE, LOAD CARRIAGE RETURN

JSR FCOUT ; AND SEND TO COUT

RTS

**SUB.FINPUT >> FINPUT**

|  |
| --- |
| **FINPUT (sub)**  **Input**:  none  **Output**:  **.A** = string length  **RETURN** = string read  **RETLEN** = length byte  **Destroys**: AXYNVZCM  **Cycles**: 54+  **Size**: 41 bytes |

The **FINPUT** subroutine reads a string from an opened text file and stores it in **RETLEN**/**RETURN**.

\*``````````````````````````````\*

\* FINPUT (NATHAN RIGGS) \*

\* \*

\* INPUTS A LINE FROM A TEXT \*

\* FILE AND STORES IT AS A \*

\* STRING IN RETLEN/RETURN. \*

\* \*

\* INPUT: \*

\* \*

\* OPEN FILE TO BE READ \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = STRING LENGTH \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 54+ \*

\* SIZE: 41 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]SLEN EQU VARTAB ; STRING LENGTH

\*

FINPUT

\*

LDX #0 ; INIT LENGTH

JSR FGETLN ; GET A LINE OF INPUT, ENDED BY $8D

STX ]SLEN ; STORE LENGTH IN ]SLEN

CPX #0 ; IF X = 0, NO STRING TO READ

BEQ :EXIT ; THEREFORE, EXIT

:INP\_CLR

LDY #0 ; CLEAR OUTPUT INDEX

LDA ]SLEN ; STORE LENGTH BYTE

STA RETLEN,Y ; PUT LENGTH AT START

:LP

LDA $0200,Y ; READ KEYBOARD BUFFER

INY ; INCREASE OUTPUT INDEX

STA RETLEN,Y ; STORE CHARACTER IN RETURN

CPY ]SLEN ; IF .Y != STRING LENGTH

BNE :LP ; KEEP LOOPING

:EXIT

LDA ]SLEN ; RETURN LENGTH IN .A

RTS

**SUB.FPRINT >> FPRINT**

|  |
| --- |
| **FPRINT (sub)**  **Input**:  See description  **Output**:  none  **Destroys**: AXYNVZCM  **Cycles**: 63+  **Size**: 37 bytes |

The **FPRINT** subroutine outputs to the open file a null-terminated **ASC** that follows the call to the subroutine, as so:

JSR FPRINT

ASC “testing”,8D00

For outputting strings with preceding length bytes, use the **FPSTR** subroutine.

\*``````````````````````````````\*

\* FPRINT (NATHAN RIGGS) \*

\* \*

\* PRINTS A NULL-TERMINATED \*

\* STRING TO A TEXT FILE. THIS \*

\* STRING SHOULD BE AN ASC THAT \*

\* FOLLOWS THE JSR TO THIS \*

\* SUBROUTINE. \*

\* \*

\* INPUT: \*

\* \*

\* AN ASC FOLLOWS THE CALL \*

\* TO THIS, FOLLOWED BY 00 \*

\* \*

\* OUTPUT: \*

\* \*

\* NONE \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^^ \*

\* \*

\* CYCLES: 63+ \*

\* SIZE: 37 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

FPRINT

\*

PLA ; GET RETURN ADDRESS LOW BYTE

STA RETADR ; STORE IN RETURN ADDRESS

PLA ; GET RETURN ADDRESS HIGH BYTE

STA RETADR+1 ; STORE HIGH BYTE

LDY #$01 ; POINT TO INSTRUCTION AFTER RETURN ADDR

:LP

LDA (RETADR),Y ; GET CHARACTER FROM STRING

BEQ :DONE ; IF CHAR IS 00, EXIT LOOP

JSR FCOUT ; SEND CHARACTER TO COUT

INY ; INCREASE STRING INDEX

BNE :LP ; LOOP IF INDEX != 0

:DONE

CLC ; NOW RESTORE INSTRUCTION POINTER

TYA ; MOVE INDEX TO .A FOR ADDING

ADC RETADR ; ADD INDEX TO OLD ADDRESS

STA RETADR ; STORE AS NEW ADDRESS

LDA RETADR+1 ; DO THE SAME FOR THE HIGH BYTE

ADC #$00 ; THEN PUSH HIGH BYTE

PHA ; TO THE STACK

LDA RETADR ; PUSH RETURN ADDRESS LOW BYTE

PHA ; TO THE STACK

RTS

**SUB.FPSTR >> FPSTR**

|  |
| --- |
| **FPSTR (sub)**  **Input**:  **WPAR1** = string address  pointer  **Output**:  **.A** = string length  **Destroys**: AXYNVZCM  **Cycles**: 38+  **Size**: 25 bytes |

The **FPSTR** subroutine writes a string with a preceding byte length to a file. The byte length itself is not written.

\*``````````````````````````````\*

\* FPSTR (NATHAN RIGGS) \*

\* \*

\* PRINTS THE SPECIFIED STRING \*

\* AT GIVEN LOCATION TO THE \*

\* FILE OPEN AND SET TO BE \*

\* WRITTEN. \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = STRING ADDRESS PTR \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = STRING LENGTH \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 38+ \*

\* SIZE: 25 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]SLEN EQU VARTAB ; STRING LENGTH

]ADDR EQU WPAR1 ; STRING ADDRESS POINTER

\*

FPSTR

\*

LDY #0 ; RESET INDEX

LDA (]ADDR),Y ; GET STRING LENGTH

STA ]SLEN ; STORE IN ]SLEN

:LP

INY ; INCREASE INDEX

LDA (]ADDR),Y ; GET CHARACTER

JSR FCOUT ; STORE IN FILE

CPY ]SLEN ; IF .Y != STRING LENGTH

BNE :LP ; THEN KEEP LOOPING

:EXIT

TYA ; STRING LENGTH TO .A

RTS

**DEMO.FILEIO**

This demo contains illustrations of how to use the macros in the **FILEIO** library. These are not meant to be exhaustive demonstrations.

\*

\*``````````````````````````````\*

\* DEMO.FILEIO \*

\* \*

\* A DEMO OF THE FILE INPUT AND \*

\* OUTPUT MACROS. RWTS ROUTINES \*

\* ARE NOT DEMONSTRATED. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 21-SEP-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\*\* ASSEMBLER DIRECTIVES

\*

CYC AVE

EXP OFF

TR ON

DSK DEMO.FILEIO

OBJ $BFE0

ORG $6000

\*

\*``````````````````````````````\*

\* TOP INCLUDES (HOOKS,MACROS) \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PUT MIN.HEAD.REQUIRED

USE MIN.MAC.REQUIRED

USE MIN.MAC.FILEIO

PUT MIN.HOOKS.FILEIO

\*

\*``````````````````````````````\*

\* PROGRAM MAIN BODY \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* \*

\* NOTE: FOR THIS TO WORK \*

\* PROPERLY, THE DEMO HAS TO BE \*

\* BLOADED, THEN EXECUTED VIA \*

\* THE MONITOR (6000G). IF THIS \*

\* IS NOT DONE, YOU WILL GET A \*

\* "FILE NOT FOUND" ERROR WHEN \*

\* DOING FILE OPERATIONS. \*

\* \*

\* FOR YOUR OWN PROJECTS, A WAY \*

\* TO WORK AROUND THIS IS TO \*

\* USE AN EXEC FILE TO BLOAD \*

\* AND EXECUTE THE CODE. \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\_PRN " ",8D8D8D8D8D

\_PRN "FILE INPUT/OUTPUT MACROS",8D

\_PRN "------------------------",8D8D

\_PRN "THE BSAVE MACRO SAVES THE GIVEN",8D

\_PRN "ADDRESS RANGE UNDER THE SPECIFIED",8D

\_PRN "BINARY FILE. THE ARGUMENT IS SIMPLY",8D

\_PRN "A STRING THAT WOULD MATCH THE ARGUMENTS",8D

\_PRN "OF A TYPICAL BSAVE STATEMENT IN DOS.",8D8D

\_PRN "BSAVE 'TEST,A$800,L$100' SAVES THE",8D

\_PRN "$100 BYTES LOCATED AT $800 IN THE FILE",8D

\_PRN "TEST.",8D8D

\_PRN "LET'S PUT SOMETHING INTO $300 TO",8D

\_PRN "TEST IT OUT.",8D8D

LDY #0

LP

TYA

STA $800,Y

INY

CPY #$100

BNE LP

\_WAIT

DUMP #$800;#$100

\_WAIT

\_PRN " ",8D8D

\_PRN " BSAVE 'TEST,A$800,L$100'...."

BSAVE "TEST,A$800,L$100"

\_PRN "DONE!",8D8D

\_PRN "NOW LET'S CLEAR $100 BYTES AT",8D

\_PRN "$800 BEFORE WE RELOAD IT WITH BLOAD.",8D8D

LDY #0

LP2

LDA #0

STA $800,Y

INY

CPY #$100

BNE LP2

DUMP #$800;#$100

\*

\_PRN " ",8D8D

\_PRN "NOW WE CAN BLOAD TEST TO GET $800",8D

\_PRN "BACK INTO THE STATE WE PUT IT.",8D8D

\_PRN "BLOAD 'TEST'...",8D

\_WAIT

BLOAD "TEST"

\_PRN " ",8D8D

\_PRN "DONE!",8D8D

DUMP #$0800;#$100

\_PRN " ",8D8D

\_WAIT

\*

\_PRN "THE CMD MACRO SIMPLY EXECUTES A",8D

\_PRN "DOS COMMAND, ALONG WITH ANY ARGUMENTS",8D

\_PRN "PASSED TO IT. CMD 'CATALOG', FOR INSTANCE,",8D

\_PRN "RETURNS:",8D8D

\_WAIT

CMD "CATALOG"

\_WAIT

\*

\*\* IF WE ARE TO READ OR WRITE FILES, WE HAVE TO FOOL

\*\* THE COMPUTER TO THINK IT'S IN APPLESOFT MODE. THIS

\*\* IS ACCOMPLISHED WITH THE AMODE MACRO. WITH BINSAVE

\*\* AND BINLOAD, THIS IS ALREADY DONE, SO TECHNICALLY

\*\* WE DON'T HAVE TO DO IT HERE. HOWEVER, THE CMD

\*\* ROUTINE DOESN'T SET IT UP AUTOMATICALLY, SO BE SURE

\*\* TO INCLUDE THIS BEFORE OPENING TEXT FILES.

\*

AMODE

\*

\_PRN " ",8D8D8D

\_PRN "TYPICALLY, THE CMD MACRO IS ALSO",8D

\_PRN "USED FOR PREPARING TO READ OR WRITE",8D

\_PRN "TEXT FILES. HOWEVER, BEFORE THIS CAN",8D

\_PRN "BE ACCOMPLISHED, THE TMODE MACRO",8D

\_PRN "MUST BE RUN TO TRICK APPLESOFT INTO",8D

\_PRN "BELIEVING IT ISN'T IN IMMEDIATE MODE.",8D8D

\_PRN "TMODE HAS NO ARGUMENTS. THUS, THE",8D

\_PRN "FOLLOWING PREPARES US TO OPEN A TEXT",8D

\_PRN "FILE TO BE WRITTEN TO:",8D8D

\_PRN "AMODE",8D

\_PRN "CMD 'OPEN T.TEXTFILE'",8D

\_PRN "CMD 'WRITE T.TEXTFILE'",8D8D

\_WAIT

\*

\_PRN "WE CAN NOW PRINT TO THIS FILE WITH",8D

\_PRN "THE FPRN MACRO. THIS MACRO EITHER",8D

\_PRN "PRINTS A GIVEN LINE OF TEXT TO THE FILE,",8D

\_PRN "FOLLOWED BY A RETURN ($8D), OR PRINTS",8D

\_PRN "THE CHARACTERS IN A STRING AT A GIVEN",8D

\_PRN "ADDRESS. IN THE LATTER CASE, THE LENGTH",8D

\_PRN "OF THE STRING IS NOT PRESERVED; ONLY",8D

\_PRN "THE ASCII IS.",8D8D

\_PRN "FPRN 'ALL IS WELL THAT ENDS WELL.'",8D

\_PRN "FPRN RETORT",8D8D

CMD "OPEN T.TEXTFILE"

CMD "WRITE T.TEXTFILE"

FPRN "ALL IS WELL THAT ENDS WELL."

FPRN #RETORT

CMD "CLOSE T.TEXTFILE"

\_PRN " ",8D8D8D

\_PRN "PUTS THE LITERAL PHRASE AND A PHRASE",8D

\_PRN "STORED IN THE RETORT ADDRESS INTO",8D

\_PRN "THE FILE.",8D

\_WAIT

\_PRN " ",8D8D8D

\_PRN "THEN, LIKE ALWAYS, WE MUST CLOSE",8D

\_PRN "THE FILE VIA CMD:",8D8D

\_PRN "CMD 'CLOSE T.TEXTFILE'",8D8D8D

\_WAIT

\_PRN "FINALLY, TO READ THIS TEXT FILE",8D

\_PRN "WE SIMPLY NEED TO OPEN THE",8D

\_PRN "FILE FOR READING VIA THE CMD MACRO,",8D

\_PRN "THEN USE THE FINP MACRO TO READ A ",8D

\_PRN "LINE OF TEXT AND STORE IT IN",8D

\_PRN "MEMORY:",8D8D

\_PRN "CMD 'OPEN T.TEXTFILE'",8D

\_PRN "CMD 'READ T.TEXTFILE'",8D

\_PRN "FINP",8D

\_PRN "CMD 'CLOSE T.TEXTFILE'",8D8D

CMD "OPEN T.TEXTFILE"

CMD "READ T.TEXTFILE"

FINP

CMD "CLOSE T.TEXTFILE"

\_WAIT

DUMP #RETURN;RETLEN

\_WAIT

\*

\_PRN " ",8D8D

\_PRN "THE STRING IS NOW STORED IN",8D

\_PRN "[RETURN], WITH A PRECEDING LENGTH BYTE.",8D

\_PRN "THESE CAN BE PRINTED WITH THE SPRN MACRO",8D

\_PRN "FOUND IN THE STRINGS LIBRARY.",8D8D8D

\_WAIT

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* \*

\* W A R N I N G \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\_PRN "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*",8D

\_PRN "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*",8D8D

\_PRN " WARNING!!!",8D8D

\_PRN "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*",8D

\_PRN "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*",8D8D

\_PRN "AT THIS POINT, YOU WANT TO EJECT",8D

\_PRN "THE CURRENT DISK, AND PUT IN",8D

\_PRN "A DISK THAT YOU DON'T MIND ",8D

\_PRN "HAVING TO REFORMAT. ",8D8D

\_PRN "THE REST OF THE ROUTINES ARE",8D

\_PRN "LOW LEVEL DISK ACCESS PROCEDURES,",8D

\_PRN "AND CAN SERIOUSLY DAMAGE A DISK!",8D8D

\_PRN "<<< PRESS A KEY ONCE YOU'RE READY >>>",8D8D

\_WAIT

\*

\_PRN "LOW-LEVEL DISK ACCESS IS DONE VIA",8D

\_PRN "THE STANDARD RWTS ROUTINE, WITH A",8D

\_PRN "FEW MACROS THROWN IN TO MAKE IT \*FEEL\*",8D

\_PRN "MORE SERIALIZED. THE FOLLOWING MACROS",8D

\_PRN "ALTER THE RWTS ROUTINE'S BEAHVIOR:",8D8D

\_PRN "SLOT : SETS THE RWTS SLOT",8D

\_PRN "DRIVE: SETS THE RWTS DRIVE",8D

\_PRN "TRACK: SETS THE TRACK TO BE WRITTEN/READ",8D

\_PRN "SECT : SETS THE SECTOR TO BE READ/WRITTEN",8D

\_PRN "SETDR: SET RWTS TO READ MODE",8D

\_PRN "SETDW: SET RWTS TO WRITE MODE",8D

\_PRN "DBUFF: SET THE READ/WRITE BUFFER ADDRESS",8D8D

\_WAIT

\_PRN "EACH OF THESE SETTINGS ARE INHERITED",8D

\_PRN "FROM THE PREVIOUS STATE; IF YOU ARE",8D

\_PRN "ALREADY USING SECTOR 6, DRIVE 1, FOR",8D

\_PRN "EXAMPLE, THEN YOU DON'T HAVE TO SET IT AGAIN",8D

\_PRN "UNLESS YOU WANT THOSE SETTINGS CHANGED.",8D

\_PRN "THIS LIBRARY ALSO USES THE SAME IOB",8D

\_PRN "TABLE AS THE OPERATING SYSTEM (DOS OR",8D

\_PRN "PRODOS) TO CARRY OVER ANY PREVIOUS SETTINGS.",8D8D

\_WAIT

\*

\_PRN "ONCE THE SETTINGS ARE AS DESIRED,",8D

\_PRN "YOU USE THE DRWTS MACRO TO CALL",8D

\_PRN "THE RWTS ROUTINE TO MAKE THE ",8D

\_PRN "APPROPRIATE READ OR WRITE CHANGE TO",8D

\_PRN "THE DISK.",8D8D

\_PRN "FOR THE SAKE OF PLAYING IT SAFE,",8D

\_PRN "WE WON'T BE DOING THAT HERE--YOU CAN",8D

\_PRN "EXPERIMENT ON YOUR OWN WITH THESE CALLS;",8D

\_PRN "THAT WAY IF SOMETHING BAD HAPPENS,",8D

\_PRN "IT'S ON YOU--NOT ME! :)",8D8D8D

\_WAIT

\*

JMP REENTRY

\*

RETORT STR "IF YOU ARE RICH, ANYHOW..."

\*

\*``````````````````````````````\*

\* BOTTOM INCLUDES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PUT MIN.LIB.REQUIRED

\*

\*\* INDIVIDUAL SUBROUTINES

\*

\*\* FILEIO SUBROUTINES

\*

PUT MIN.SUB.BINLOAD

PUT MIN.SUB.BINSAVE

PUT MIN.SUB.DISKRW

PUT MIN.SUB.DOSCMD

PUT MIN.SUB.FINPUT

PUT MIN.SUB.FPRINT

PUT MIN.SUB.FPSTR

**DISK 7: CONVERSION UTILITIES**

This disk contains macros and subroutines dedicated to converting strings with numerals into their actual numeric values and converting numeric values into their string equivalents. This comes in three flavors: integer, hexadecimal, or binary.

This disk contains the following files:

* HOOKS.CONVERT
* MAC.CONVERT
* DEMO.CONVERT
* SUB.BINASC2HEX
* SUB.HEX2BINASC
* SUB.HEX2HEXASC
* SUB.HEX2INTASC
* SUB.HEXASX2HEX
* SUB.INTASC2HEX

**HOOKS.CONVERT**

The HOOKS.CONVERT file holds hooks related to string and numeral conversion. So far, there are no hooks, but the file is still included to keep consistent with the rest of the library.

Note that the NOP instruction is included because Merlin 8 Pro will crash if a file is included without any instructions.

\*``````````````````````````````\*

\* HOOKS.CONVERT \*

\* \*

\* HOOKS TO AID IN CONVERTING \*

\* STRINGS TO NUMBERS AND VICE \*

\* VERSA, AND ALSO IN BETWEEN. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 25-SEP-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* LICENSE: APACHE 2.0 \*

\* OS: DOS 3.3 \*

\* \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

NOP ; OTHERWISE, MERLIN WILL CRASH

; DUE TO EMPTY FILE

**MAC.CONVERT**

This file contains all of the macros pertaining to string and numeric conversion. They are the following:

* I2STR
* STR2I
* H2STR
* STR2H
* B2STR
* STR2B

\*``````````````````````````````\*

\* MAC.CONVERT \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 25-SEP-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\* \*

\* SUBROUTINE FILES NEEDED \*

\* \*

\* SUB.BINASC2HEX \*

\* SUB.HEX2BINASC \*

\* SUB.HEX2HEXASC \*

\* SUB.HEX2INTASC \*

\* SUB.HEXASC2HEX \*

\* SUB.INTASC2HEX \*

\* \*

\* LIST OF MACROS \*

\* \*

\* I2STR: INTEGER TO STRING \*

\* STR2I: STRING TO INTEGER \*

\* H2STR: HEXADECIMAL TO STRING \*

\* STR2H: STRING TO HEXADECIMAL \*

\* B2STR: BINARY TO STRING \*

\* STR2B: STRING TO BINARY \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

**MAC.CONVERT >> I2STR**

|  |
| --- |
| **I2STR (mac)**  **Input**:  **]1** = value to convert  **Output**:  **.A** = string length  **RETURN** = string chars  **RETLEN** = length byte  **Destroys**: AXYNVZCM  **Cycles**: 258+  **Size**: 383 bytes |

The **I2STR** macro converts a numeric value into a string holding its integer representation. This value can be 8-bit or 16-bit, and the sign of the value is preserved.

\*``````````````````````````````\*

\* I2STR \*

\* \*

\* CONVERTS A 16BIT INTEGER TO \*

\* ITS STRING EQUIVALENT. \*

\* \*

\* PARAMETERS: \*

\* \*

\* ]1 = VALUE TO CONVERT \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* I2STR #11111 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

I2STR MAC

STY SCRATCH

\_MLIT ]1;WPAR1

JSR HEX2INTASC

LDY SCRATCH

<<<

**MAC.CONVERT >> STR2I**

|  |
| --- |
| **STR2I (mac)**  **Input**:  **]1** = string or address  **Output**:  **.A** = value low byte  **.X** = value high byte  **RETURN** = value  **RETLEN** = value length  **Destroys**: AXYNVZCM  **Cycles**: 298+  **Size**: 227 bytes |

The **STR2I** macro converts a string with an integer representation of a value into its actual value. The string may contain a representation of an 8-bit or 16-bit signed integer, and the real value is passed back via **.A** (low byte) and **.X** (high byte). The value is additionally held in **RETURN**.

\*

\*``````````````````````````````\*

\* STR2I \*

\* \*

\* CONVERTS A STRING TO A 16BIT \*

\* NUMBER EQUIVALENT. \*

\* \*

\* PARAMETERS: \*

\* \*

\* ]1 = STRING OR ITS ADDRESS \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* STR2I "1024" \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

STR2I MAC

STY SCRATCH

\_MSTR ]1;WPAR1

JSR INTASC2HEX

LDY SCRATCH

<<<

**MAC.CONVERT >> H2STR**

|  |
| --- |
| **H2STR (mac)**  **Input**:  **]1** = hex value or address  **Output**:  **RETURN** = string  **RETLEN** = 2  **Destroys**: AXYNVZCM  **Cycles**: 98+  **Size**: 87 bytes |

The **H2STR** macro converts a numeric value into a string containing its hexadecimal representation, passing back the string vial **RETLEN**/**RETURN**. This macro only handles 8-bit values, meaning that the string length byte will always be 2.

\*

\*``````````````````````````````\*

\* H2STR \*

\* \*

\* CONVERTS A HEX BYTE INTO AN \*

\* EQUIVALENT STRING IN HEX. \*

\* \*

\* PARAMETERS: \*

\* \*

\* ]1 = HEX VALUE TO CONVERT \*

\* OR THE ADDRESS \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* H2STR #FF \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

H2STR MAC

STY SCRATCH

LDA ]1

JSR HEX2HEXASC

LDY SCRATCH

<<<

**MAC.CONVERT >> STR2H**

|  |
| --- |
| **STR2H (mac)**  **Input**:  **]1** = string or address  **Output**:  **.A** = value returned  **RETURN** = value returned  **RETLEN** = 1  **Destroys**: AXYNVZCM  **Cycles**: 114+  **Size**: 92 bytes |

The **STR2H** macro converts a string holding a hexadecimal representation of an 8-bit numeric value into its actual value. This value is passed back via **.A** and **RETURN**.

\*

\*``````````````````````````````\*

\* STR2H \*

\* \*

\* CONVERTS A HEX STRING TO ITS \*

\* EQUIVALENT HEX BYTE. \*

\* \*

\* PARAMETERS: \*

\* \*

\* ]1 = STRING OR ITS ADDRESS \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* STR2H "FE" \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

STR2H MAC

STY SCRATCH

\_MSTR ]1;WPAR1

JSR HEXASC2HEX

LDY SCRATCH

<<<

**MAC.CONVERT >> B2STR**

|  |
| --- |
| **B2STR (mac)**  **Input**:  **]1** = hex value to convert  **Output**:  **RETURN** = string chars  **RETLEN** = length byte  **Destroys**: AXYNVZCM  **Cycles**: 152+  **Size**: 171 bytes |

The **B2STR** macro converts an 8-bit numeric value into a string holding its binary representation. The string is returned via **RETLEN**/**RETURN**.

\*

\*``````````````````````````````\*

\* B2STR \*

\* \*

\* CONVERTS A HEX VALUE TO ITS \*

\* EQUIVALENT BINARY STRING. \*

\* \*

\* PARAMETERS: \*

\* \*

\* ]1 = HEX VALUE OR ADDRESS \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* B2STR #$FE \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

B2STR MAC

STY SCRATCH

LDA ]1

STA BPAR1

JSR HEX2BINASC

LDY SCRATCH

<<<

**MAC.CONVERT >> STR2B**

|  |
| --- |
| **STR2B (mac)**  **Input**:  **]1** = string or address  **Output**:  **.A** = converted value  **RETURN** = converted value  **RETLEN** = 1  **Destroys**: AXYNVZCM  **Cycles**: 432+  **Size**: 351 bytes |

The **STR2B** macro converts a string holding a binary representation of an 8-bit value into its corresponding numeric value. This value is then passed back via **.A** as well as in **RETURN**.

\*

\*``````````````````````````````\*

\* STR2B \*

\* \*

\* CONVERTS A BINARY STRING TO \*

\* EQUIVALENT HEX VALUE. \*

\* \*

\* PARAMETERS: \*

\* \*

\* ]1 = STRING OR ITS ADDRESS \*

\* \*

\* SAMPLE USAGE: \*

\* \*

\* STR2B "00110101" \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

STR2B MAC

STY SCRATCH

\_MSTR ]1;WPAR1

JSR BINASC2HEX

LDY SCRATCH

<<<

**SUB.BINASC2HEX >> BINASC2HEX**

|  |
| --- |
| **BINASC2HEX (sub)**  **Input**:  **WPAR1** = string address  **Output**:  **.A** = hexadecimal value  **RETURN** = hex value  **RETLEN** = 1  **Destroys**: AXYNVZCM  **Cycles**: 400+  **Size**: 320 bytes |

The **BINASC2HEX** subroutine translates a string containing a representation of eight bits into its actual numerical byte value. The value is passed back via **RETURN** and **.A** as well.

\*``````````````````````````````\*

\* BINASC2HEX (NATHAN RIGGS) \*

\* \*

\* CONVERTS A STRING HOLDING \*

\* 8 CHARACTERS OF 0S AND 1S \*

\* THAT SIGNIFY A BYTE INTO THE \*

\* APPROPRIATE HEX VALUE. \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = STRING ADDRESS PTR \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = HEXADECIMAL VALUE \*

\* RETURN = HEX VALUE \*

\* RETLEN = 1 (BYTE LENGTH) \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 400+ \*

\* SIZE: 320 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]HIGH EQU VARTAB

]LOW EQU VARTAB+2

]NIB EQU VARTAB+4

]STR EQU WPAR1

\*

BINASC2HEX

\*

JSR :TESTNIB ; FIRST CHECK HIGH NIBBLE

LDA ]NIB ; (1ST 4 'BITS' IN THE STRING)

STA ]HIGH ; AND STORE HEX IN ]HIGH

LDA ]STR ; ADD 4 TO THE STRING ADDRESS

CLC ; TO GET THE LOW NIBBLE

ADC #4 ; STRING ADDRESS

STA ]STR

LDA ]STR+1 ; MAKE SURE TO ADJUST

ADC #0 ; THE HIGH BYTE

STA ]STR+1

JSR :TESTNIB ; TEST THE LOW NIBBLE OF THE STRING

LDA ]NIB

STA ]LOW ; AND STORE THE LOW NIBBLE HEX

\*

LDA #1 ; STORE BYTE LENGTH

STA RETLEN ; IN RETLEN

LDA ]HIGH ; LOAD HIGH NIBBLE AND

ORA ]LOW ; EXCLUSIVE-OR IT WITH LOW NIBBLE

STA RETURN ; TO GET COMPLETE BYTE

JMP :EXIT

\*

\*\* THE :TESTNIB SUBROUTINE TRANSLATES

\*\* A BINARY NIBBLE STRING REPRESENTATION INTO

\*\* ITS EQUIVALENT HEXADECIMAL CODE

\*

:TESTNIB

LDY #0 ; START AT FIRST BINARY DIGIT

LDA (]STR),Y ; GET EITHER A 0 OR A 1 CHARACTER

CMP #'0' ; IF = 0

BEQ :\_07 ; THEN THE NIBBLE IS BETWEEN 0 AND 7

JMP :\_8F ; ELSE IT IS BETWEEN 8 AND F

:\_07

LDY #1 ; CHECK SECOND STRING DIGIT

LDA (]STR),Y ; AGAIN, GET 0 OR 1

CMP #'0' ; IF = 0

BEQ :\_03 ; THEN NIBBLE BETWEEN 0 AND 3

JMP :\_47 ; ELSE IT IS BETWEEN 4 AND 7

:\_03

LDY #2 ; THIRD DIGIT OF NIBBLE

LDA (]STR),Y ; GET 0 OR 1 FROM STRING

CMP #'0' ; IF = 0,

BEQ :\_01 ; NIBBLE IS EITHER 0 OR 1

JMP :\_23 ; ELSE EITHER 2 OR 3

:\_01

LDY #3 ; LAST BIT OF NIBBLE STRING

LDA (]STR),Y ; GET EITHER 0 OR 1

CMP #'0' ; IF IT IS 0,

BEQ :\_00 ; FIRST NIBBLE IS 0

LDA #1 ; ELSE IT IS 1

STA ]NIB ; STORE NIBBLE

JMP :EXIT

:\_00 LDA #0 ; NIBBLE IS 0000

STA ]NIB

JMP :EXIT

\*

:\_23 LDY #3 ; READ 4TH BIT IN NIBBLE

LDA (]STR),Y

CMP #'0' ; IF = "0",

BEQ :\_02 ; THEN THE FIRST NIBBLE IS 2

LDA #3 ; ELSE IT IS 3

STA ]NIB

JMP :EXIT

:\_02 LDA #$2 ; NIBBLE IS 2

STA ]NIB

JMP :EXIT

:\_47

LDY #2 ; READ 3RD BIT FROM STRING

LDA (]STR),Y

CMP #'0' ; IF = "0",

BEQ :\_45 ; THEN THE 1ST NIBBLE IS 4 OR 5

JMP :\_67 ; ELSE IT IS 6 OR 7

:\_45

LDY #3 ; CHECK 4TH BIT OF BINARY STRING

LDA (]STR),Y

CMP #'0' ; IF = "0",

BEQ :\_4 ; THEN FIRST NIB IS 4

LDA #$5 ; ELSE IT IS 5

STA ]NIB

JMP :EXIT

:\_4 LDA #$4 ; NIBBLE = 4

STA ]NIB

JMP :EXIT

:\_67

LDY #3 ; CHECK 4TH BIT IN STRING

LDA (]STR),Y

CMP #'0' ; IF = "0"

BEQ :\_6 ; THEN THE FIRST NIB IS 6

LDA #$7 ; ELSE IT IS 7

STA ]NIB

JMP :EXIT

:\_6 LDA #$6 ; NIBBLE = 6

STA ]NIB

JMP :EXIT

\*

:\_8F ; CHECK VALUE BETWEEN 8 AND F

LDY #1 ; CHECK SECOND BIT

LDA (]STR),Y

CMP #'0' ; IF = "0",

BEQ :\_8B ; THEN NIBBLE IS BETWEEN 8 AND B

JMP :\_CF ; OTHERWISE BETWEEN C AND F

:\_8B ; CHECK VALUES 8-B

LDY #2 ; CHECK 3RD BIT

LDA (]STR),Y

CMP #'0' ; IF = "0",

BEQ :\_89 ; NIBBLE IS EITHER 8 OR 9

JMP :\_AB ; ELSE IT IS BETWEEN A AND B

:\_89 ; TEST WHETHER 8 OR 9

LDY #3 ; CHECK 4TH BIT

LDA (]STR),Y

CMP #'0' IF = "0",

BEQ :\_8 THEN NIBBLE IS 8

LDA #9 ; ELSE, IS 9

STA ]NIB

JMP :EXIT

:\_8 LDA #$8 ; NIBBLE = 8

STA ]NIB

JMP :EXIT

:\_AB ; NIBBLE IS EITHER A OR B

LDY #3 ; CHECK 4TH BIT

LDA (]STR),Y

CMP #'0' ; IF = "0"

BEQ :\_A ; THEN NIBBLE IS A

LDA #$B ; OTHERWISE, IT'S B

STA ]NIB

JMP :EXIT

:\_A LDA #$A ; NIBBLE IS A

STA ]NIB

JMP :EXIT

:\_CF ; NIBBLE IS BETWEEN C AND F

LDY #2 ; CHECK 3RD BIT

LDA (]STR),Y

CMP #'0' ; IF = "0",

BEQ :\_CD ; THEN IT IS EITHER C AND D

JMP :\_EF ; OTHERWISE, BETWEEN E AND F

:\_CD ; NIBBLE IS EITHER C OR D

LDY #3 ; CHECK 4TH BIT

LDA (]STR),Y

CMP #'0' ; IF IT IS "0",

BEQ :\_C ; THEN NIBBLE IS C

LDA #$D ; OTHERWISE, IT'S D

STA ]NIB

JMP :EXIT

:\_C LDA #$C ; NIBBLE IS C

STA ]NIB

JMP :EXIT

:\_EF ; NIBBLE IS EITHER E OR F

LDY #3 ; CHECK 4TH BIT

LDA (]STR),Y

CMP #'0' ; IF IT IS "0",

BEQ :\_E ; THEN NIBBLE IS E

LDA #$F ; OTHERWISE, F

STA ]NIB

JMP :EXIT

:\_E LDA #$E ; SET TO E

STA ]NIB

:EXIT

RTS

**SUB.HEX2BINASC >> HEX2BINASC**

|  |
| --- |
| **HEX2BINASC (sub)**  **Input**:  **BPAR1** = hexadecimal byte  **Output**:  **RETURN** = hex string  **RETLEN** = 8  **Destroys**: AXYNVZCM  **Cycles**: 134+  **Size**: 159 bytes |

The **HEX2BINASC** subroutine converts a single byte numeric value into a string carrying the value’s binary representation.

\*``````````````````````````````\*

\* HEX2BINASC (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* BPAR1 = HEX BYTE TO CONVERT \*

\* \*

\* OUTPUT: \*

\* \*

\* NONE \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 134+ \*

\* SIZE: 159 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]BINTAB ASC "0000" ; 0

ASC "0001" ; 1

ASC "0010" ; 2

ASC "0011" ; 3

ASC "0100" ; 4

ASC "0101" ; 5

ASC "0110" ; 6

ASC "0111" ; 7

ASC "1000" ; 8

ASC "1001" ; 9

ASC "1010" ; A

ASC "1011" ; B

ASC "1100" ; C

ASC "1101" ; D

ASC "1110" ; E

ASC "1111" ; F

\*

]LEFT EQU VARTAB ; LEFT NIBBLE

]RIGHT EQU VARTAB+2 ; RIGHT NIBBLE

]HBYTE EQU BPAR1 ; HEX BYTE

\*

HEX2BINASC

\*

LDA ]HBYTE

AND #$F0 ; FIRST, MASK THE RIGHT NIBBLE

LSR ; SHIFT RIGHT

LSR ; SHIFT RIGHT

LSR ; SHIFT RIGHT

LSR ; SHIFT RIGHT

STA ]LEFT ; STORE AS LEFT NIBBLE

LDA ]HBYTE

AND #$0F ; NOW MASK LEFT NIBBLE

STA ]RIGHT ; STORE AS RIGHT NIBBLE

\*

\*\* GET LEFT FROM LOOKUP TABLE

\*

ASL ]LEFT ; MULTIPLY ]LEFT NIBBLE

ASL ]LEFT ; BY FOUR

LDX ]LEFT ; TO GET LOOKUP TABLE OFFSET

LDA ]BINTAB,X ; TRANSFER APPROPRIATE

STA RETURN ; PART OF THE TABLE TO RETURN

LDA ]BINTAB,X+1

STA RETURN+1

LDA ]BINTAB,X+2

STA RETURN+2

LDA ]BINTAB,X+3

STA RETURN+3

\*

\*\* NOW GET RIGHT

\*

ASL ]RIGHT ; MULTIPLY ]RIGHT BY 4

ASL ]RIGHT ; TO GET LOOKUP TABLE OFFSET

LDX ]RIGHT

LDA ]BINTAB,X ; AND TRANSFER APPROPRIATE

STA RETURN+4 ; STRING TO RETURN AFTER

LDA ]BINTAB,X+1 ; THE PREVIOUS NIBBLE

STA RETURN+5

LDA ]BINTAB,X+2

STA RETURN+6

LDA ]BINTAB,X+3

STA RETURN+7

\*

LDA #8 ; LENGTH IN .A AND RETLEN

STA RETLEN

RTS

**SUB.HEX2HEXASC >> HEX2HEXASC**

|  |
| --- |
| **HEX2HEXASC (sub)**  **Input**:  **.A** = hexadecimal value  **Output**:  **RETURN** = hex string  **RETLEN** = 2  **Destroys**: AXYNVZCM  **Cycles**: 80+  **Size**: 77 bytes |

The **HEX2HEXASC** subroutine converts a single byte numeric value into its string equivalent in hexadecimal representation.

\*``````````````````````````````\*

\* HEX2HEXASC (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* .A = HEX TO CONVERT \*

\* \*

\* OUTPUT: \*

\* \*

\* RETURN = HEX STRING \*

\* RETLEN = 2 \*

\* \*

\* DESTROY: AXYNVBDIZCMS \*

\* ^^^^^ ^^^ \*

\* \*

\* CYCLES: 80+ \*

\* SIZE: 77 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]LEFT EQU VARTAB ; LEFT NIBBLE

]RIGHT EQU VARTAB+2 ; RIGHT NIBBLE

]HBYTE EQU VARTAB+4 ; HEX BYTE TO CONVERT

]HEXTAB ASC "0123456789ABCDEF" ; HEX LOOKUP TABLE

\*

HEX2HEXASC

\*

STA ]HBYTE ; STORE HEX PASSED VIA .A

AND #$F0 ; MASK RIGHT

LSR

LSR

LSR

LSR

STA ]LEFT ; STORE LEFT NIBBLE

LDA ]HBYTE

AND #$0F ; MASK LEFT

STA ]RIGHT ; STORE RIGHT NIBBLE

LDX ]LEFT ; GET THE LEFT CHARACTER

LDA ]HEXTAB,X ; FROM LOOKUP TABLE

STA ]LEFT

LDX ]RIGHT ; GET THE RIGHT CHARACTER

LDA ]HEXTAB,X ; FROM LOOKUP TABLE

STA ]RIGHT

LDA ]LEFT ; STORE LEFT IN RETURN

STA RETURN

LDA ]RIGHT ; STORE RIGHT IN NEXT BYTE

STA RETURN+1

LDA #2 ; LENGTH IN RETLEN AND .A

STA RETLEN

RTS

**SUB.HEX2INTASC >> HEX2INTASC**

|  |
| --- |
| **HEX2INTASC (sub)**  **Input**:  **WPAR1** = 16-bit value  **Output**:  **.A** = string length  **RETURN** = integer chars  **RETURN** = string length  **Destroys**: AXYNVZCM  **Cycles**: 226+  **Size**: 352 bytes |

The **HEX2INTASC** subroutine converts an 8-bit or 16-bit value into its string equivalent, using decimal notation. Note that if the value is negative, the string will be prepended with a “-“ character.

\*``````````````````````````````\*

\* HEX2INTASC (NATHAN RIGGS) \*

\* \*

\* CONVERT A SIGNED HEXADECIMAL \*

\* VALUE TO AN INTEGER STRING. \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = HEX TO CONVERT \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = STRING LENGTH \*

\* RETURN = INTEGER CHARACTERS \*

\* RETLEN = LENGTH BYTE \*

\* \*

\* DESTROYS: AXYNVBDIZCMS \*

\* ^^^^ ^^^ \*

\* \*

\* CYCLES: 226+ \*

\* SIZE: 352 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]NGFLAG EQU VARTAB ; NEGATIVE FLAG

]VALSTR EQU WPAR1 ; HEXADECIMAL TO CONVERT

]MOD10 EQU VARTAB+2 ; VALUE MODULUS 10

\*

HEX2INTASC

\*

LDA ]VALSTR+1 ; STORE VALUE HIGH BYTE

STA ]NGFLAG ; IN THE NEGATIVE FLAG

BPL :GETBP ; IF VALUE IS POSITIVE, BRANCH

LDA #0 ; ELSE SUBTRACT LOW BYTE

SEC

SBC ]VALSTR

STA ]VALSTR ; STORE AS NEW LOW BYTE

LDA #0 ; ADJUST HIGH BYTE

SBC ]VALSTR+1

STA ]VALSTR+1

:GETBP

LDA #0 ; SET BUFFER TO EMPTY

LDY #0

STA RETLEN,Y ; BUFFER(0) = 0

\*

:CNVERT ; CONVERT VALUE TO STRING

LDA #0 ; RESET MODULUS

STA ]MOD10

STA ]MOD10+1

LDX #16

CLC ; CLEAR CARRY

:DVLOOP

ROL ]VALSTR ; SHIFT CARRY INTO DIVBIT 0

ROL ]VALSTR+1 ; WHICH WILL BE THE QUOTIENT

ROL ]MOD10 ; + SHIFT DIV AT SAME TIME

ROL ]MOD10+1

SEC ; SET CARRY

LDA ]MOD10 ; SUBTRACT #10 (DECIMAL) FROM

SBC #10 ; MODULUS 10

TAY ; SAVE LOW BYTE IN .Y

LDA ]MOD10+1 ; ADJUST HIGHBYTE

SBC #0 ; SUBTRACT CARRY

BCC :DECCNT ; IF DIVIDEND < DIVISOR, DECREASE COUNTER

STY ]MOD10 ; ELSE STORE RESULT IN MODULUS

STA ]MOD10+1 ; NEXT BIT OF QUOTIENT IS A 1,

; DIVIDEND = DIVIDEND - DIVISOR

:DECCNT

DEX ; DECREASE .X COUNTER

BNE :DVLOOP ; IF NOT 0, CONTINUE DIVIDING

ROL ]VALSTR ; ELSE, SHIFT IN LAST CARRY FOR QUOTIENT

ROL ]VALSTR+1

:CONCH

LDA ]MOD10

CLC ; CLEAR CARRY

ADC #$B0 ; ADD '0' CHARACTER TO VALUE

; TO GET ACTUAL ASCII CHARACTER

JSR :CONCAT ; CONCATENATE TO STRING

\*

\*\* IF VALUE <> 0 THEN CONTINUE

\*

LDA ]VALSTR ; IF VALUE STILL NOT 0,

ORA ]VALSTR+1 ; OR HIGH BIT, THEN KEEP DIVIDING

BNE :CNVERT ;

\*

:EXIT

LDA ]NGFLAG ; IF NEGATIVE FLAG IS SET

BPL :POS ; TO ZERO, THEN NO SIGN NEEDED

LDA #173 ; ELSE PREPEND THE STRING

JSR :CONCAT ; WITH A MINUS SIGN

\*

:POS ; VALUE IS POSITIVE

RTS ; RETLEN

\*

:CONCAT ; STRING CONCATENATION SUBROUTINE

PHA ; SAVE CHAR ON STACK

\*

\*\* MOVE BUFFER RIGHT ONE CHAR

\*

LDY #0 ; RESET INDEX

LDA RETLEN,Y ; GET CURRENT STRING LENGTH

TAY ; CURRENT LENGTH IS NOW THE INDEX

BEQ :EXITMR ; IF LENGTH = 0, EXIT CONCATENATION

\*

:MVELP

LDA RETLEN,Y ; GET NEXT CHARACTER

INY ; INCREASE INDEX

STA RETLEN,Y ; STORE IT

DEY ; DECREASE INDEX BY 2

DEY

BNE :MVELP ; LOOP UNTIL INDEX IS 0

:EXITMR

PLA ; GET CHAR BACK FROM STACK

LDY #1

STA RETLEN,Y ; STORE THE CHAR AS FIRST CHARACTER

LDY #0 ; RESET INDEX

LDA RETLEN,Y ; GET LENGTH BYTE

CLC ; CLEAR CARRY

ADC #1 ; INC LENGTH BY ONE

STA RETLEN,Y ; UPDATE LENGTH

\*

LDA RETLEN

RTS

**SUB.HEXASC2HEX >> HEXASC2HEX**

|  |
| --- |
| **HEXASC2HEX (sub)**  **Input**:  **WPAR1** = string address  **Output**:  **.A** = hex value  **RETURN** = hex value  **RETLEN** = 1  **Destroys**: AXYNVZCM  **Cycles**: 82+  **Size**: 61 bytes |

The **HEX2HEXASC** subroutine converts a 2-byte string of a number in hexadecimal format to its numeric equivalent. This value is passed back via **.A** and **RETURN**.

\*``````````````````````````````\*

\* HEXASC2HEX \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = HEX STRING ADDRESS \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = HEX BYTE VALUE \*

\* RETURN = HEX BYTE VALUE \*

\* RETLEN = 1 \*

\* \*

\* DESTROYS: AXYNVBDIZCMS \*

\* ^^^^ ^^^ \*

\* \*

\* CYCLES: 82+ \*

\* SIZE: 61 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]HI EQU VARTAB ; HIGH BYTE

]LO EQU VARTAB+2 ; LOW BYTE

]STR EQU WPAR1 ; ADDR OF STRING TO CONVERT

\*

HEXASC2HEX

LDY #1 ; GET FIRST HEX CHARACTER

LDA (]STR),Y

STA ]HI ; STORE IN HIBYTE

INY ; INCREASE INDEX

LDA (]STR),Y ; TO GET SECOND HEX CHARACTER

STA ]LO ; AND STORE THAT IN LOW BYTE

\*

SEC ; SET CARRY

SBC #'0' ; SUBTRACT '0' CHAR FROM ]LO CHAR

CMP #10 ; ASCII NUMERALS OFFSET

BCC :CONT ; IF NUMERAL, CONTINUE

SBC #7 ; OTHERWISE SUBTRACT LETTER OFFSET

:CONT

STA ]LO ; STORE VALUE INTO LOW BYTE

LDA ]HI ; NO WORK ON HIGH BYTE

SEC ; SET CARRY

SBC #'0' ; SUBTRACT '0' ASCII

CMP #10 ; IS NUMBER?

BCC :C2 ; THEN DONE

SBC #7 ; OTHERWISE LETTER OFFSET

:C2

STA ]HI ; STORE HIGH BYTE VALUE

ASL ; CLEAR LOW BYTE OF ]HI

ASL

ASL

ASL

ORA ]LO ; OR OPERATION TO INSERT

; LOW BYTE INTO RESULT

LDY #1 ; SET LENGTH OF RETURN

STY RETLEN

STA RETURN ; PASS BACK VIA RETURN AND .A

RTS

**SUB.INTASC2HEX >> INTASC2HEX**

|  |
| --- |
| **INTASC2HEX (sub)**  **Input**:  **WPAR1** = string address  **Output**:  **.A** = hex value low byte  **.X** = hex val high byte  **RETURN** = hex value  **RETLEN** = 2  **Destroys**: AXYNVZCM  **Cycles**: 266+  **Size**: 196 bytes |

The **INTASC2HEX** subroutine converts a string of numbers representing an integer value into its equivalent value, which is returned in **.A** (low byte) and **.X** (high byte) as well as in **RETURN**. The string must be no larger than a 16-bit integer, and the sign is preserved.

\*``````````````````````````````\*

\* INTASC2HEX (NATHAN RIGGS) \*

\* \*

\* INPUT: \*

\* \*

\* WPAR1 = STRING ADDRESS \*

\* \*

\* OUTPUT: \*

\* \*

\* .A = HEX VALUE LOW BYTE \*

\* .X = HEX VALUE HIGH BYTE \*

\* RETURN = HEX VALUE \*

\* RETLEN = 2 \*

\* \*

\* DESTROYS: AXYNVBDIZCMS \*

\* ^^^^ ^^^ \*

\* \*

\* CYCLES: 266+ \*

\* SIZE: 196 BYTES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]NACCUM EQU VARTAB

]SIGN EQU VARTAB+4

]NINDEX EQU VARTAB+6

]STR EQU WPAR1

\*

INTASC2HEX

\*

LDY #0 ; INIT INDEX

LDA (]STR),Y ; GET STRING LENGTH

TAX ; TRANSFER TO .X

LDA #1 ; SET NINDEX TO 1

STA ]NINDEX ;

LDA #0 ; INIT ]NACCUM LOW, HIGH

STA ]NACCUM ; ACCUM = 0

STA ]NACCUM+1

STA ]SIGN ; INIT SIGN TO 0 (POSITIVE)

TXA ; TRANSFER .X BACK TO .A

BNE :INIT1 ; IF .A != 0, CONTINUE INIT

JMP :EREXIT ; ELSE, EXIT WITH ERROR--NO STRING

:INIT1

LDY ]NINDEX ; INITIALLY, SET TO 1

LDA (]STR),Y ; LOAD FIRST CHARACTER

CMP #173 ; IF .A != "-"

BNE :PLUS ; THEN NUMBER IS POSITIVE

LDA #$0FF ; ELSE SET FLAG TO NEGATIVE

STA ]SIGN

INC ]NINDEX ; INCREASE INDEX

DEX ; DECREMENT LENGTH COUNT

BEQ :EREXIT ; EXIT WITH ERROR IF .X = 0

JMP :CNVERT

:PLUS

CMP #'+'

BNE :CHKDIG ; START CONVERSION IF 1ST

; CHARACTER IS NOT A +

INC ]NINDEX ; INCREASE NEW INDEX

DEX ; DEC COUNT; IGNORE + SIGN

BEQ :EREXIT ; ERROR EXIT IF ONLY

; + IN THE BUFFER

:CNVERT

LDY ]NINDEX ; GET NEW INDEX

LDA (]STR),Y ; GET NEXT CHARACTER

:CHKDIG ; CHECK DIGIT

CMP #$B0 ; "0"

BMI :EREXIT ; ERROR IF NOT A NUMERAL

CMP #$BA ; '9'+1; TECHNICALLY :

BPL :EREXIT ; ERR IF > 9 (NOT NUMERAL)

PHA ; PUSH DIGIT TO STACK

\*

\*\* VALID DECIMAL DIGIT SO

\*\* ACCUM = ACCUM \* 10

\*\* = ACCUM \* (8+2)

\*\* = (ACCUM \* 8) + (ACCUM \* 2)

\*

ASL ]NACCUM

ROL ]NACCUM+1 ; TIMES 2

LDA ]NACCUM

LDY ]NACCUM+1 ; SAVE ACCUM \* 2

ASL ]NACCUM

ROL ]NACCUM+1

ASL ]NACCUM

ROL ]NACCUM+1 ; TIMES 8

CLC

ADC ]NACCUM ; SUM WITH \* 2

STA ]NACCUM

TYA

ADC ]NACCUM+1

STA ]NACCUM+1 ; ACCUM=ACCUM \* 10

\*

PLA ; GET THE DIGIT FROM STACK

SEC ; SET CARRY

SBC #$B0 ; SUBTRACT ASCII '0'

CLC ; CLEAR CARRY

ADC ]NACCUM ; ADD TO ACCUMULATION

STA ]NACCUM ; STORE IN ACCUMULATION

LDA #0 ; NOW ADJUST HIGH BYTE

ADC ]NACCUM+1

STA ]NACCUM+1

INC ]NINDEX ;INC TO NEXT CHARACTER

DEX ; DECREMENT .X COUNTER

BNE :CNVERT ; IF .X != 0, CONTINUE CONVERSION

LDA ]SIGN ; ELSE LOAD SIGN FLAG

BPL :OKEXIT ; IF POSITIVE, EXIT WITHOUT ERROR

LDA #0 ; ELSE SET THE VALUE TO NEGATIVE

SEC ; SET CARRY

SBC ]NACCUM ; 0 - ]NACCUM

STA ]NACCUM ; STORE AS ]NACCUM

LDA #0 ; ADJUST HIGHBYTE

SBC ]NACCUM+1

STA ]NACCUM+1

\*

:OKEXIT

CLC ; CLEAR CARRY TO SIGNIFY NO ERRORS

BCC :EXIT

:EREXIT

SEC ; SET CARRY TO INIDICATE ERROR

:EXIT

LDA #2 ; BYTE LENGTH IS 2

STA RETLEN

LDX ]NACCUM+1 ; LOAD HIGH BYTE INTO .X

LDA ]NACCUM ; AND LOW BYTE INTO .A

STA RETURN ; ALSO STORE RESULT IN RETURN

STX RETURN+1

RTS

**DEMO.CONVERT**

This demo shows how to use the conversion macros. Note that this is by no means exhaustive; it is meant to quickly illustrate how to you the macros only.

\*

\*``````````````````````````````\*

\* DEMO.CONVERT \*

\* \*

\* A DEMO OF THE CONVERSION \*

\* MACROS. \*

\* \*

\* AUTHOR: NATHAN RIGGS \*

\* CONTACT: NATHAN.RIGGS@ \*

\* OUTLOOK.COM \*

\* \*

\* DATE: 25-SEP-2019 \*

\* ASSEMBLER: MERLIN 8 PRO \*

\* OS: DOS 3.3 \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\*\* ASSEMBLER DIRECTIVES

\*

CYC AVE

EXP OFF

TR ON

DSK DEMO.CONVERT

OBJ $BFE0

ORG $6000

\*

\*``````````````````````````````\*

\* TOP INCLUDES (PUTS, MACROS) \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

PUT MIN.HEAD.REQUIRED

USE MIN.MAC.REQUIRED

USE MIN.MAC.CONVERT

PUT MIN.HOOKS.CONVERT

\*

\*``````````````````````````````\*

\* PROGRAM MAIN BODY \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

]HOME EQU $FC58

]XCOUT EQU $FDF0

\*

JSR ]HOME

\_PRN "CONVERSION LIBRARY",8D

\_PRN "==================",8D8D

\_PRN "THIS DEMO SHOWCASES HOW TO USE",8D

\_PRN "THE MACROS IN THE CONVERSION LIBRARY.",8D8D

\_PRN "THESE MACROS ARE USED FOR CONVERTING",8D

\_PRN "NUMBERS INTO STRINGS AND VICE VERSA",8D

\_PRN "IN THREE NUMBERING SYSTEMS: ",8D

\_PRN "DECIMAL, HEXADECIMAL, AND BINARY.",8D8D

\_WAIT

\*

JSR ]HOME

\_PRN "INTEGERS AND STRINGS",8D

\_PRN "====================",8D8D

\_PRN "TO CONVERT BETWEEN NUMERALS",8D

\_PRN "AND THEIR INTEGER-BASED EQUIVALENTS.",8D

\_PRN "TO CONVERT FROM A NUMBER TO AN INTEGER",8D

\_PRN "STRING, YOU WOULD USE THE I2STR MACRO,",8D

\_PRN "WHICH STANDS FOR 'INTEGER TO STRING.'",8D

\_PRN "TO CONVERT AN INTEGER STRING TO ITS",8D

\_PRN "NUMERICAL 16-BIT EQUIVALENT, YOU WOULD",8D

\_PRN "USE THE STR2I MACRO--WHICH OF COURSE",8D

\_PRN "STANDS FOR 'STRING TO INTEGER.",8D8D

\_PRN "LET'S TEST THESE TO SEE HOW THEY WORK.",8D

\_WAIT

JSR ]HOME

\_PRN "IN CONVERTING AN INTEGER TO A STRING,",8D

\_PRN "YOU WOULD USE THE I2STR MACRO AS SUCH:",8D8D

\_PRN " I2STR #5309",8D8D

\_PRN "WHICH WILL PRODUCE THE FOLLOWING STRING:",8D8D

\_WAIT

I2STR #5309

LDA RETURN

JSR ]XCOUT

LDA RETURN+1

JSR ]XCOUT

LDA RETURN+2

JSR ]XCOUT

LDA RETURN+3

JSR ]XCOUT

\_WAIT

\*

JSR ]HOME

\_PRN "THE STR2I MACRO DOES THE OPPOSITE:",8D

\_PRN "IT TAKES AN INTEGER STRING AND",8D

\_PRN "CONVERTS IT TO A NUMERIC VALUE. THUS:",8D8D

\_PRN " STR2I '255'",8D

\_PRN " DUMP #RETURN;#2",8D8D

\_PRN "WILL RETURN:",8D8D

STR2I "255"

\_WAIT

DUMP #RETURN;#2

\_WAIT

JSR ]HOME

\_PRN "HEXADECIMAL TO STRING",8D

\_PRN "=====================",8D8D

\_PRN "TO CONVERT A HEX VALUE TO A",8D

\_PRN "HEX STRING AND VICE VERSA, YOU",8D

\_PRN "WOULD USE THE H2STR AND STR2H MACROS.",8D8D

\_PRN "THE H2STR MACRO CONVERTS A HEX BYTE",8D

\_PRN "TO ITS STRING EQUIVALENT, AS SUCH:",8D8D

\_PRN " H2STR #$FF",8D

\_PRN " LDA RETURN",8D

\_PRN " JSR ]XCOUT",8D8D

\_PRN "RETURNS:",8D8D

\_WAIT

H2STR #$FF

LDA RETURN

JSR ]XCOUT

LDA RETURN+1

JSR ]XCOUT

\_WAIT

\_PRN " ",8D8D

\_PRN "TO TURN A HEX STRING BACK",8D

\_PRN "INTO ITS NUMERIC VALUE, YOU WOULD",8D

\_PRN "THE STR2H MACRO AS SUCH:",8D8D

\_PRN " STR2H 'FF'",8D

\_PRN " DUMP #RETURN;#1",8D8D

\_PRN "WHICH RETURNS:",8D8D

\_WAIT

STR2H "FF"

DUMP #RETURN;#1

\_WAIT

\*

JSR ]HOME

\_PRN "BINARY STRING CONVERSION",8D

\_PRN "========================",8D8D

\_PRN "LASTLY, WE HAVE MACROS FOR THE",8D

\_PRN "CONVERSION OF BINARY STRINGS TO THEIR",8D

\_PRN "NUMERIC EQUIVELENT AND VICE VERSA:",8D

\_PRN "STR2B AND B2STR.",8D8D

\_WAIT

\_PRN "STR2B TAKES A STRING OF ZEROS AND",8D

\_PRN "ONES AND CONVERTS THAT INTO ITS",8D

\_PRN "NUMERIC VALUE, AS SUCH:",8D8D

\_PRN " STR2B '00110011'",8D

\_PRN " DUMP #RETURN;#1",8D8D

\_PRN "WHICH RETURNS:",8D8D

\_WAIT

STR2B "00110011"

DUMP #RETURN;#1

\_WAIT

\_PRN "TO CONVERT A NUMERIC VALUE TO",8D

\_PRN "A BINARY STRING, USE THE B2STR",8D

\_PRN "MACRO AS SUCH:",8D8D

\_PRN " B2STR #$FF",8D8D

\_PRN "WHICH RETURNS THE STRING:",8D8D

\_WAIT

B2STR #$FF

LDA RETURN

JSR ]XCOUT

LDA RETURN+1

JSR ]XCOUT

LDA RETURN+2

JSR ]XCOUT

LDA RETURN+3

JSR ]XCOUT

LDA RETURN+4

JSR ]XCOUT

LDA RETURN+5

JSR ]XCOUT

LDA RETURN+6

JSR ]XCOUT

LDA RETURN+7

JSR ]XCOUT

\_WAIT

JSR ]HOME

\_PRN "FIN.",8D8D8D

\*

JMP REENTRY

\*

\*``````````````````````````````\*

\* BOTTOM INCLUDES \*

\*,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\*

\*

\*\* BOTTOM INCLUDES

\*

PUT MIN.LIB.REQUIRED

\*

\*\* INDIVIDUAL SUBROUTINE INCLUDES

\*

\*\* STRING SUBROUTINES

\*

PUT MIN.SUB.HEX2INTASC

PUT MIN.SUB.INTASC2HEX

PUT MIN.SUB.HEX2BINASC

PUT MIN.SUB.BINASC2HEX

PUT MIN.SUB.HEX2HEXASC

PUT MIN.SUB.HEXASC2HEX

\*