"SUPERFORTH 64"™

for Total Control of the Commodore 64*.

Disk Enclosed

Published by
PARSEC RESEARCH
Drawer 1766
Fremont, CA 94538
(415) 651-3160

Available at the Vintage Volts website

VINTAGE

VOLTS

http://www.vintagevolts.com

http://www.vintagevolts.com

Copyright Notice

Errata

1.3 Getting Started (p 3.)

Note: The Master disk has two sides to it... a system side and a source code side. The System side is to be Loaded First.

Whenever the instructions in the manual requests that you list a source screen # or Load a source screen # (ie. 10 LIST, 10 LOAD or 10 20 THRU) then you insert the Source Code Side of the master disk into your drive and type the appropriate command.

To make a complete backup of the master disk you will need 2 blank disks, one for the system side and one for the source code side. Follow the instructions on page 3 to copy the system, and page 72, 73 to make a backup copy of the source code.

Your working copy should consist of <u>two additional</u> disks, one system disk and one blank formatted disk. Your program listings are stored on the blank formatted disk and your compiled program becomes part of the system disk.

To make a backup copy of the AI and Math Modules follow the instructions on pages 72 & 73.

TABLE OF CONTENTS

APPENDICES

1. Introduction To the System

SUPER FORTH 64 is a superset of the Version 1.xx.03 MVP-FORTH system as defined in the annotated glossary, ALL ABOUT FORTH by Glen B. Hayden. Questions concerning definitions and implementations within the MVP FORTH Kernel can be resolved by referring to ALL ABOUT FORTH. ALL ABOUT FORTH also describes the implementation of various FIG-FORTH words which are not used or have been renamed.

SUPER FORTH 64 is a complete implementation of the FORTH-79 Required and Extension Word Sets. It contains extensions from FIG FORTH, STARTING FORTH by Leo Brodie, and special extensions to take advantage of the particular hardware in the Commodore 64 computer.

This manual contains definitions for each ALL ABOUT FORTH word used in the SUPER FORTH 64 system. Therefore, ALL ABOUT FORTH is NOT a required guide to definitions in this system, but will be useful to the advanced FORTH programmer for the detailed information it contains about each MVP FORTH word.

1.1 System Support

We have attempted to bring you a correctly working system and a manual which is error-free. However, questions on system usage will arise from time to time. We encourage you to contact Parsec Research (Drawer 1766, Fremont, CA 94538: 415-651-3160) to report any anomolies in the system or particular problems which you may be encountering with it. We will attempt to answer all questions as quickly as possible. We are also interested in user feedback as to potential changes or enhancements which would facilitate use of the system.

It is IMPORTANT to send in the registration card which is enclosed with the system. This card will enable us to send users periodic updates concerning the SUPER FORTH 64 system. These updates will contain useful information, such as bug fixes, enhancements to the system and answers to commonly asked questions.

1.2 About This Manual

This is a reference manual for the SUPER FORTH implementation of the FORTH language on the Commodore 64 computer. It is NOT meant to be used as a tutorial for learning how to program in FORTH. It is assumed that the user either has some familiarity with the FORTH language, or will use STARTING FORTH or a similar tutorial (see appendix for a list of recommended books) to learn enough to use the system. It is also assumed that the user has a basic operating knowledge of the Commodore 64. This can be acquired through use of the Commodore 64 User's Guide which comes with the machine.

Introduction To the System

To fully understand the Graphics and Sound capabilities of the Commodore 64, and thus to utilize the special extensions for these features, it is recommended that the user read one of the books which deals with C64 sound and/or graphics (see appendix) and use it in conjunction with the sections in this manual on Sound and Graphics Extension words (Sections 5.7 and 5.8).

Since the hexadecimal number system (base 16) is often used when dealing with computer hardware (addresses which often seem strange in decimal are usually even multiples of 16, 16**2 or 16**3) I have included the hex conversion for addresses given in parenthesis after the decimal address. Hex numbers are numbers which are prefixed with a "$" (such as $12AB). In FORTH the base can easily be switched by using the FORTH words HEX and DECIMAL.

1.3 Getting Started

1.3.1 Disk Based Systems

To get started working with SUPER FORTH perform the following steps :

1. Turn on the computer and disk drive, and insert the SUPER FORTH 64 system diskette into the drive (use device 8 in a multi-drive system).

2. To load the demo system, skip to section 1.4, "Demo Program"; to load a normal SUPER FORTH system perform the following commands :

```
LOAD "SUPER FORTH 64",8 {return}
RUN {return}
```

The SUPER FORTH system should now be running.

It is recommended that you make a working copy of your master SUPER FORTH 64 system immediately and keep the master in a safe place. To make the working copy you must first format a blank diskette (Commodore refers to this as "NEWing" a diskette). Place a blank diskette in the drive and type the following to format a "working copy" diskette:

```
" N0:diskname,id" DOS
```

where "diskname" and "id" are as defined in the VIC-1541 User's Manual. The operation will take about two minutes.

When formatting is finished type the following:

```
SAVE-FORTH
```

Your system will be saved to disk as a file named "SUPER FORTH 64". This diskette should now be used as your working copy of SUPER FORTH.

You will need a separate formatted diskette in order to save source code for the FORTH programs you will write. Place a blank diskette in the drive and type the following to format a "source screens" diskette:

```
" N0:diskname,id" DOS
```

This diskette may be used as your working "source screens" diskette.

1.3.2 Cassette Based Systems

To get started working with SUPER FORTH 64 perform the following steps :

1. Turn on the computer and insert the SUPER FORTH 64 system cassette into the cassette drive.

2. Perform the following commands :

```
LOAD "SUPER FORTH 64" {return}
```

Follow procedure for loading a program from cassette. When the program is done loading type:

```
RUN {return}
```

The FORTH system should now be running.

It is recommended that you make a backup copy of your master SUPER FORTH 64 system immediately and work from the backup copy. Place a new cassette in the drive and key in the following:

```
SAVE-FORTH
```

A copy will be made to the new cassette. This should be used as your working copy of the system.

1.3.3 Once the System Is Running

It is worthwhile (and probably a lot of fun) to go through the program examples in the manual. Especially useful for the beginner are the Graphics and Music Editor examples in sections 5.7 and 5.9. If you want to start saving programs quickly go through the editing tutorial in section 5.1, Editor Words. Section 4 provides a tutorial on the FORTH Assembler. It is intended for programmers who have a general assembler experience, but not using a FORTH assembler. By following the examples on your C64 and seeing what they do, you will quickly develop a "feel" for FORTH without really knowing how to program in it.

If you are a beginning FORTH programmer, use the instructional guide STARTING FORTH (see book list in Appendix V) by Leo Brodie, Prentice Hall, 1981, or THE COMPLETE FORTH by Alan Winfield, Wiley Press. STARTING FORTH is a more comprehensive guide to the FORTH language and internals, while THE COMPLETE FORTH is aimed towards users of BASIC who wish to program in FORTH.

Note: This version of FORTH is intended to be compatible with the vocabulary used in STARTING FORTH. However, there are a few instances where the STARTING FORTH word is in conflict with the FORTH 79 Standard. In those cases, the 79 Standard is adhered to.

The most notable difference is in the word, ' (tick). In this system, ' returns the PFA of a word, in STARTING FORTH it returns the CFA. Therefore, to follow examples in the book, follow use of ' with CFA.

Other words which differ are S0 and ?STACK.

The FORTH source screen editor differs from that in the book. See section 5.1 on The Editor before attempting to edit source screens.

The really adventurous FORTH programmer may want to read ALL ABOUT FORTH by Dr. Glen Hayden. ALL ABOUT FORTH is a complete annotated description of this particular implementation of the FORTH Kernel. It contains the definition of each Kernel word and a description of its use.

If you are an experienced FORTH programmer you will probably be interested in the various extension words for writing applications programs using SUPER FORTH. This manual describes all extension words particular to the SUPER FORTH system.

1.4 Demo Program

Included in SUPER FORTH 64 is a demonstration program called DEMO The source code screens are provided as examples of how the various extensions, and FORTH itself, can be used to program the C64 (see DEMO screens in Appendix II).

DEMO includes use of high-res graphics, sprite graphics and the SID sound chip. It also demonstrates advanced programming techniques, such as recursion and co-routines.

The complete SUPER FORTH system, including the demos, is provided in the program file "DEMO-SYSTEM". Enter the following to execute the demo:

```
        LOAD "DEMO",8  {return}
        RUN
        DEMO
```

After the demo completes the system is left waiting for FORTH commands.

1.5 Warm Starting

The RUN-STOP/RESTORE key sequence will cause a warm start to FORTH. After warm starting the system will be left in the following state: Newly entered definitions will remain available. Vectors which have been changed will remain intact. The I/O re-direction variables (OUTLFN and INPLFN) will be intialized to their default values. The parameter and return stacks will be emptied, and execution proceeds from the interpreter.

A system warm start may be effected from the BASIC interpreter by entering the BASIC command SYS 2067.

1.6 Exiting and Cold Starting

To exit FORTH and restore the system to its initial state use the word BYE. This performs a system cold start. FORTH can be re-entered only if no numbered BASIC lines have been entered in the default BASIC program area, since that is where the FORTH dictionary resides.

The default BASIC area starts at 2049 ($801). The FORTH dictionary area is defined to start at 2064 ($0810), leaving room for the BASIC instruction 10 SYS(2064) to be saved and loaded with the FORTH system file, enabling the user to start up the FORTH system by typing RUN. Placing FORTH at 2064 also insures that a cold start will not destroy the beginning of FORTH.

After cold start if a BASIC program is to be entered or a directory listing is to be performed and re-entry to FORTH will be desired, the BASIC program space must be moved so that it won't overwrite the beginning of the FORTH dictionary. Moving BASIC to 32768 ($8000) is probably safe and can be accomplished by the following BASIC commands:

```
        POKE 44,128: POKE 32768,0: NEW
```

After performing the BASIC functions, reset the BASIC area by entering the following:

```
        POKE 44,8: NEW
```

To re-enter FORTH type the following:

```
        10 SYS(2064)
        RUN
```

1.7 Resetting the System

The 6510 microprocessor (the heart of this system) has a very nice feature built in whereby you can reset the system without losing your program. Unfortunately, Commodore did not see fit to provide any way for the end user (you) to utilize this feature. However, there are expansion cards available which have a "reset" button built on which when pressed will activate the reset feature.

It is recommended, especially for the beginner, that you purchase one of these boards, since beginners often tend to crash the system while learning how to program in FORTH. This way, your system would be recovered at the click of a switch instead of having to re-boot the system from disk.

1.8 FORTH and the Commodore Screen Editor

Input in this FORTH system makes use of the Commodore screen editor. This is the same editor used by Commodore Basic. It allows a user to enter data anywhere on the screen by using the Commodore Screen Editing keys to move the cursor and edit the line on which the cursor was moved to.

When the RETURN key is depressed the editor copies the line where the cursor was residing into an input buffer, from which FORTH get its input a character at a time. Only the line in which the cursor was placed (it doesn't matter where on the line the cursor resides) is moved into the input buffer. Thus, the screen editing functions can be used during immediate FORTH input to correct and/or re-enter FORTH lines.

1.9 FORTH and Machine Language

FORTH supports both externally and internally coded machine language routines. Internally coded routines are created using the FORTH 6510 Assembler. Externally coded routines are loaded and linked to using various SUPER FORTH words described below.

1.9.1 FORTH 6510 Assembler

A 6510 machine language assembler for use within the FORTH environment is included in this implementation. It is essentially the one written by William Ragsdale and published in FORTH Dimensions Vol. III, No. 5. This assembler also includes extensions for the FORTH constructs BEGIN...AGAIN, BEGIN...WHILE...REPEAT and BEGIN...UNTIL and a conditional specifier for the overflow status bit, VS.

A feature of this system is that the assembler may be made to be "remote", that is, it is not compiled into the main dictionary space. Thus, applications may use the assembler to assemble machine code, but need not have it in memory during execution. See the article in the appendix for detailed operating instructions and the section on Assembler Usage (Section 4.) for a tutorial on usage.

1.9.2 External Machine Language Routines

External routines can be loaded and used by the SUPER FORTH system.

Before entering external routines into the system it must be verified that they will not be loaded over an area which is used by the system. Many external machine language routines are loaded into the area at $C000, since in the default machine configuration this is an unused area above BASIC. SUPER FORTH, however, allows use of all memory below $D000. Therefore, if routines reside in the $C000 area, the top of SUPER FORTH must be moved below it (see CHANGE).

If the routines are in the form of program files on a diskette, LOADRAM can be used to load them into memory. Otherwise, PATCH may be used to "hand enter" machine code into memory.

Once the routines are in memory, they may be linked to using one of various words- SYS and SYSCALL preserve the 6510's registers and allow data to be placed in them before calling an external routine. GO calls a routine without preserving registers. A routine must end with an RTS instruction in order to return to SUPER FORTH.

WARNING! AN EXTERNAL ROUTINE MAY BE USING ZERO-PAGE AREAS WHICH SUPER FORTH EXPECTS TO BE LEFT UNTOUCHED. THIS WOULD RESULT IN A CRASH OF THE SYSTEM AND SHOULD BE CHECKED BEFORE USING THE EXTERNAL ROUTINE.

1.10 FORTH Editor

An Editor is included for entering SUPER FORTH source code; the edited code can be saved to disk or cassette. The Editor is a screen editor which works functionally the way the Commodore Basic Editor works. It includes words for copying screens, copying sections of screens and moving sections of screens. See the section titled "Editor Words" for more detailed information.

1.11 FORTH Code Storage

FORTH code exists in the system in two forms: source code and compiled code. Source code format is used to store user readable code in the form of source screens. FORTH source words can be individually entered or modified by using the SUPER FORTH Editor (Note: The Editor is NOT the STARTING FORTH editor- see "Editor Words" for details).

The source code is stored using one of two modes of storage, Standard Mode (as in "standard" FORTH I/O systems) or File Mode. Disk users should use Standard Mode. Cassette users MUST use File Mode. A by-product of the implementation, however, is that File Mode MAY be used by disk users, but is not recommended since Standard Mode generally provides greater flexibility. SUPER FORTH programs which are stored in File Mode (using F-SAVE) may, however, be mixed with BASIC files on the same diskette.

Compiled code is generated by the FORTH system either by entering FORTH word definitions interactively or by loading source code screens using LOAD or F-LOAD. A source word must be compiled into the system before it can be executed. Once compiled, it can be invoked (called upon for execution) interactively or can be used as part of the definition of a new word. When a source screen is compiled it in effect becomes part of a new FORTH system which contains all previous definitions AND the newly compiled definitions.

VLIST can be used to display the words which have been compiled. Compiled words may be "decompiled" by using the word DECOMPILE. In this way the user can examine the definitions of words which have been entered interactively along with compiled source screens.

SAVE-FORTH and APPLICATION are words which can be used to save the complete compiled FORTH system as a Commodore program file. All user defined words which have been compiled will be saved along with the SUPER FORTH 64 original system. In this way, user extensions can be added, compiled and saved without having to re-compile them each time the system needs to be loaded in.

Note: Disk users must use a FORMATTED Commodore diskette for EITHER mode of operation.

1.11.1 Standard Mode (Disk Only)

Standard Mode provides compatibility with other MVP-FORTH systems by utilizing the standard FORTH definition, BLOCK, to provide the typical FORTH virtual disk system. Using Standard Mode, all FORTH blocks available (the total in all drives) may be looked upon as a single fast random access file. If an application requires a data base which spans multiple disk drives, Standard Mode may be the way to go. Standard Mode is the mode of operation described in STARTING FORTH in both the Editor and the I/O chapters.

Users whose applications require a random access database will find that a Standard Mode system will run faster than a Commodore relative file system because there is no sector lookup to determine where a particular sector lies. Relative files may take up to three disk accesses to read or write a single sector. Standard Mode will make only one disk access per sector.

Standard Mode allows greater area usage of the diskette for FORTH source code, since 680 of the 683 sectors are available for use as source screens. It also allows minimal disk buffer allocation for loading of a very large source program since a program is loaded by specifying actual screen numbers. However, because these screens are written and read using direct sector I/O routines, source screens created on a diskette using this mode cannot be mixed with standard Commodore files. The Commodore directory may, in fact, have been used to store FORTH source information (see section titled "Standard Mode")!

Standard Mode utilizes a virtual buffer system- the number of buffers defined in the system simply determines how many screens may reside in memory before the system must flush (write out to disk) a screen in order to make room for a new one. This process is carried out automatically by the system, so the user need not concern himself with buffer management. Buffer management is reduced to a matter of system efficiency- if there are more buffers available the system will have to perform less I/O while the user is editing screens. Initially 8 buffers are allocated to the system.

Cassette users cannot use Standard Mode since a random access device must be available.

1.11.2 File Mode (Cassette or Disk)

File Mode enables the user to create and edit a group of FORTH source screens as a named file. File Mode provides the ability to use the Commodore file system to keep track of groups of screens as files, but has the limitation that all screens for a particular file name must fit in memory (just as a complete BASIC program must fit in available memory - see section titled "File Mode").

Using File Mode, the user loses the use of FORTH's virtual buffer facility. The user must insure that there are enough buffers available to completely contain a file before the editing begins or the file is loaded in. When all buffers are full new screens can no longer be added and the file must be written out. As in Standard Mode, 8 buffers are allocated for File Mode initially.

1.11.3 Initial Mode Settings

Cassette users MUST use File Mode since Standard Mode assumes availability of a disk unit. Cassette based systems will come up in File Mode configured for cassette. If a cassette user wants to change over to a disk based system, he should type 8#SYSDEV#! (this changes the system device to disk 8) and type SAVE-FORTH to save the system to disk. The saved system will come up disk based.

Disk based systems initially come up in Standard Mode. If you prefer File Mode, type F-NEW to enter File Mode and save the FORTH system by typing SAVE-FORTH. The saved system will come up in File Mode. If a disk based user wants to change over to a cassette based system (there's always one in every crowd), he should type 1 SYSDEV ! (this changes the system device to 1- cassette) and type SAVE-FORTH to save the system to cassette. The saved system will come up cassette based.

1.12 Special Features

The following are special utilities and features implemented in SUPER FORTH 64.

1.12.1 Saving Applications

After the FORTH program designer develops an application using SUPER FORTH 64, the word APPLICATION can be used to save the system as a special "application" program file which can be loaded by end users. This file, when loaded and run, will immediately execute the application word. The FORTH system is not visible to a user running the application. In this way the applications designer may market the application program without infringing on the copyright of the FORTH system (see section titled "APPLICATION" for more detailed information).

NOTE: The word APPLICATION must be entered in order to release you from infringing on the Author's copyrights.

1.12.2 Debugging Features

Several utilities are provided to ease debugging of programs written using SUPER FORTH 64:

A decompiler is provided which enables the user to decompile any high-level word in the system (including FORTH Kernel words). This Decompiler was written by Robert Dudley Ackerman and published in FORTH Dimensions Vol. IV, No. 2. See the article in Appendix VII for a complete description of its use.

A trace facility is provided to aid developers in their debugging of newly defined FORTH words. This trace was adapted from the one written by Paul van der Eijk and published in FORTH Dimensions Vol. III, No. 2. See the article in Appendix VII and section 5.6.24 for operating instructions.

A non-destructive stack dump, .S is provided to enable users to see what is on the parameter stack without losing any data values. The stack may be displayed from latest to oldest value, or oldest to latest See the descriptions in the Kernel Glossary (.S, .SR and .SL)

A memory dump facility, DUMP is provided to enable users to display both the hex and ASCII values of sections of memory. See the Kernel Glossary for a description of DUMP.

Example:

```
11730 32 DUMP
```

Dumps 32 locations starting at 11730.

1.12.3 Math Routines

A simple floating point package, fixed point sin/cos routines (resolution down to a degree) and a fixed point square root routine have been provided to aid in developing graphics and mathematical applications. See the appropriate articles in Appendix VII and the section on Math Extension Words for usage.

1.12.4 High RAM Access

Special extension words are provided to enable easy access to the 12K of RAM which resides underneath the Kernel ROM and I/O Memory Map areas. This is greater than 1/5 the total memory of the machine which is normally inaccessable (see section titled "C64 High RAM Access Words")!

1.12.5 Graphics and Sound Words

Extensions are provided to take much of the difficulty out of programming graphics and sound on the Commodore 64.

The low level extensions are intended to take the user a level away from the hardware so that he does not have to deal with the details of chip addresses and alignment of data within hardware registers (if you've tried programming the graphics or sound chips using BASIC you know what I mean).

Higher level graphics words included in the system allow the user to plot hi-res lines, arc, ellipses, circles and mirrored images. A Turtle Graphics package is included as part of the high level word support. There is a simple sprite editor included (S-EDITOR) which will enable you to enter sprites and incorporate them into the system directly.

Included in the set of sound words is a music editor which can be used to compose and play back three part music. The music editor is implemented as an extension vocabulary to the SUPER FORTH 64 system, therefore, music can actually by programmed using SUPER FORTH structures!

1.12.6 I/O Redirection

The SUPER FORTH system has the ability to redirect standard I/O to devices other than keyboard and screen. Thus, for example, output can be directed to a printer or both input and ouput could be directed to an RS-232 device (see sec 4.4).

1.12.7 Diskette Backup Utilities

A utility word, BACKUP, is provided with this FORTH system to allow users to make backup copies of diskettes. Words are provided to perform either partial or complete backups, using either a single disk drive or two drives.

The copy is accomplished by using the allocated buffer space to read a section of the "from" disk and then write that section to the "to" disk. This process is repeated as many times as necessary to complete the copy. In a basic FORTH system this can be accomplished in five section read/writes. This copy performs direct sector I/O and thus cannot be used to copy particular files, but can copy a complete Commodore system disk in about twenty minutes (see section titled "Backup Utilities").

1.13 Vectored FORTH System Routines

Vectoring is a method which enables FORTH programmers to customize a FORTH System Kernel without having to re-assemble the system. Certain FORTH words are referenced by vector addresses, so by changing the address of the vector, the system will reference a new system routine. The following table lists the vectored words, their vector address words, and the offset of the vector within the User Area (see User Area description).

WORD	VECTOR ADDRESS	USER AREA OFFSET
-FIND	'-FIND	$16
?TERMINAL	'?TERMINAL	$18
ABORT	'ABORT	$1A
BLOCK	'BLOCK	$1C
CR	'CR	$1E
EMIT	'EMIT	$20
EXPECT	'EXPECT	$22
INTERPRET	'INTERPRET	$24
KEY	'KEY	$26
LOAD	'LOAD	$28
NUMBER	'NUMBER	$2A
PAGE	'PAGE	$2C
R/W	'R/W	$2E
T&SCALC	'T&SCALC	$30
VOCABULARY	'VOCABULARY	$32
WORD	'WORD	$34

As an example, let's say we want to change the system definition of PAGE to emit a form feed (ASCII code 12) instead of a clear screen character. This way, if output is to a printer, PAGE would cause the printer to go to a new page. We could enter the following:

```
: NEW-PAGE  12 EMIT ; ( New definition of PAGE )
' NEW-PAGE CFA        ( Puts code field adr of NEW-PAGE on stack)
'PAGE !               ( and stores its addr in vector for PAGE)
```

Using PAGE would now vector to the definition of NEW-PAGE. Cold starting would reset PAGE to its original definition.

Other uses of re-vectored words may be to add a disk handler for a non-1541 type disk drive. T&SCALC and/or R/W may be rewritten to use definitions which would handle these non-standard drives.

1.14 Vocabularies

Both the FORTH-79 and the FIG types of vocabularies are supported by the MVP-FORTH Kernel. The system comes up assuming FORTH-79 vocabulary structure. If the FIG vocabulary structure is preferred, VOCABULARY can be re-vectored to point to the FIG definition instead. The following will accomplish this:

```
' <VOCABULARYFIG> CFA 'VOCABULARY !
```

To get back to the FORTH-79 vocabulary structure enter the following:

```
' <VOCABULARY79> CFA 'VOCABULARY !
```

2. System Memory Usage

Part of the FORTH start-up procedure is to swap out the system's C64 BASIC ROM, making available the 8K of RAM located at the ROM's address space. Thus the user memory available for FORTH extensions is a contiguous area of memory starting from the present top of FORTH to 53120 ($CF80 - where the FORTH USER variables area is located). In the initial SUPER FORTH 64 system about 30K of RAM is available for use in creating new word definitions and/or for disk buffer space. A "stripped" system (see section 2.4, FORTH Dictionary Area) uses just under 10K of RAM. In such a system, about 41K of RAM is initially available new word definitions and/or disk buffer space.

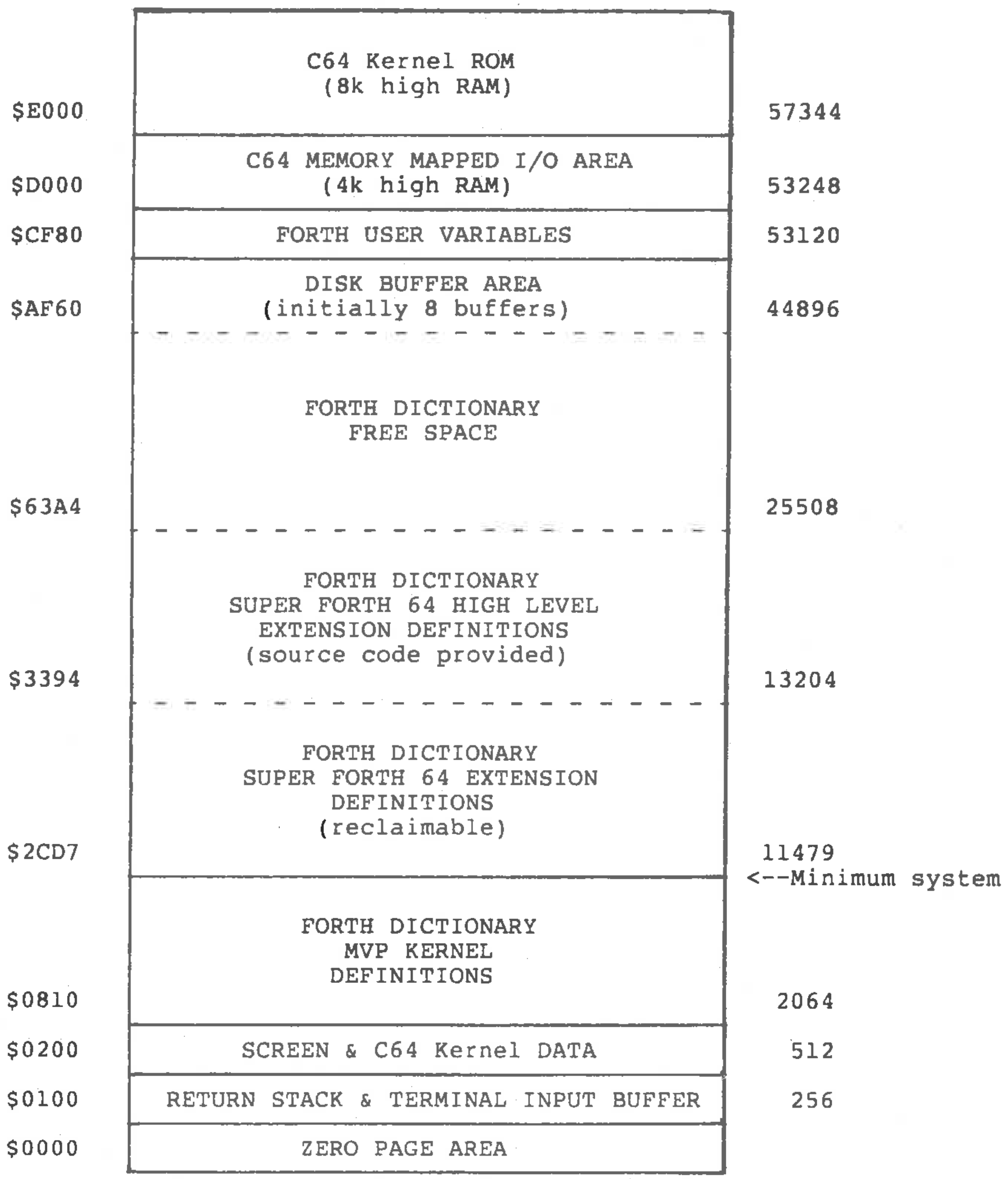
2.1 System Memory Organization

Memory is partitioned into various areas both for the FORTH system and for the Commodore Kernel. The memory map on the following page describes the initial FORTH system.

The 8k of RAM underneath the C64 Kernel and the 4k of RAM under the I/O Memory Map area are available to the user through use of the High RAM word extensions.

The SUPER FORTH dictionary can be stripped partially or completely down to the MVP Kernel area. A selection of the source screens provided can then be re-compiled into the system and the new system saved using SAVE-FORTH. In this way the user may customize his initial SUPER FORTH system (see section 2.4).

System Memory Usage



Dashed lines are used to represent movable memory boundaries.

2.2 Zero Page Usage

This section describes zero page usage of the SUPER FORTH system. The information will generally be of use to programmers who wish to write assembly code routines utilizing zero page memory.

The FORTH stack uses the zero page area from $78 down to $07. Since the stack grows down from $78 the user can generally use lower zero page locations from $07 on up. How much can be used depends on the FORTH stack usage. If recursive words are not used then the stack will probably not go TOO far down, but the user will have to experiment (recursive definitions can quickly eat up both parameter and return stack space).

The following table describes zero page usage by FORTH:

ADDRESS	DESCRIPTION
$0002-$0003	IP : Interpretive Pointer
$0004-$0005	UAP : User Area Pointer
$0007	Bottom of parameter stack
$0078	Top of parameter stack
$007A-$007B	Address of system interrupt routine
$007C-$007D	Address of user interrupt routine
$007E	Address of interrupt parameter stack
$007F	Interrupt return flag
$0080	W-1 : Indirect jump instruction
$0081-$0082	W : CFA address of word being interpreted
$0083	XSAVE : 6502 .X register save area
$0084	Raster split screen line #
$0086	N-1
$0087-$008F	N to N+8 : System zero page work area

2.3 Return Stack and Input Buffer

The system hardware stack is used as the FORTH return stack. As per normal usage it uses page one of memory ($0100-$01FF), and grows down from the top of page one. The FORTH Terminal Input Buffer is also located in page one. It works its way up from the bottom of page one at 256 ($0100).

2.4 FORTH Dictionary Area

The FORTH dictionary resides in the locations from 2064 ($0810) through the beginning of the disk buffer area. It is partitioned into two areas: the FORTH system, including any user extensions, is located at the lower end of the dictionary area. The dictionary free space area (that which is available for defining new FORTH words) is located at the upper end of the dictionary area. As the FORTH system grows, the free space area shrinks.

System Memory Usage

Initially, the top of the dictionary free space area is location 44896 ($AF60), however this may be changed by the user (see Disk Buffer Area). The bottom of the free space area (which is also the top of the FORTH system area) is at location 25508 ($63A4) in the initial system. Parts of the system space may be reclaimed by the user, as described below.

Included within the system definitions on disk are the MVP FORTH Kernel words, C64 extension words (graphics, sound, etc.), the FORTH Editor, a FORTH format Assembler, and the utilities and supplemental screens defined in ALL ABOUT FORTH. Initially, the system occupies about 23.5K of RAM. If not required for an application, parts or everything except for the Kernel words may be "forgotten" by the system, thus reclaiming the memory which had been used for those words. The order of these "forgettable" definitions is as follows:

```
     ( Source code is not supplied for the following [1725 bytes] )
      CATNIB SPLIT SAVE SYS SAVENAME SAVE-FORTH
      APPLICATION MASK SBIT CBIT FBIT SWAPOUT SWAPIN
      'BANK 'SCREEN 'BITMAP 'CHARBASE BANK SCREEN
      BITMAP CHARBASE B-X B-Y B-PEN B-ERASE B-DRAW
      B-PLOT SPRITE S-POSITION SID VOICE SID! V!
      F-FREQ PADDLE@ OSC3@ ENV3@
      D-SPLIT D-POSITION D-READ D-CLEAR
      I-INIT I-SYSTEM I-USER I-SET I-CLEAR

     ( Source code is supplied for the following words )
      THRU
      C64 Utility words
      C64 Kernel Interface words
      String Extensions
      File Mode Extensions
      Low Level Graphics Extensions
      Low Level Sound Extensions
      MVP Utility words
        .S .SL .SR .SS .INDEX <ROT MAX-BUFFS BMOVE COPY SCOPY
        DSWAP D- D0= D= D> D@ DCONSTANT
        DMAX DMIN DOVER DU< DVARIABLE
        PAUSE INDEX ?LOADING --> DUMP 'TITLE TITLE TRIAD
        <EMIT7> ID. VTAB VLEN VLIST
      MVP Supplemental word set
        'S 2! 2@ 2CONSTANT 2DROP 2DUP 2OVER 2SWAP 2VARIABLE
        >BINARY EMPTY ERASE FLUSH H U.R [']
      Local ASSEMBLER
      SUPER FORTH Screen Editor
      DECOMPILE Utility
      Math Routines- Trig, Square Root and Floating Point
      C64 Data & Constants
      Data Structure Examples
      Hi Level Graphics, I/O & Utilities
      Music Editor
      Sprite Editor
      Turtle Graphics Extensions
```

2.5 Reducing SUPER FORTH 64 To Minimum Size

Typing FORGET CATNIB would reduce the system to its minimum of about 9.5k bytes. Prior to invoking FORGET, VLIST may be used to determine the actual order of FORTH words in the dictionary.

Note that typing FORGET CATNIB removes not only words for which the source is supplied, but also those for which source is NOT supplied (from CATNIB thru I-CLEAR). It is probably more useful to type FORGET THRU which would leave a system which could be modified and rebuilt by loading in the appropriate source screens.

If File Mode is being used, it may be appropriate to save an editing system which consists of only the Kernel words, extensions thru File Mode words, and the editor in order to make available the greatest amount of buffers for editing (see section titled " Source Screen File Mode Words"). This should not be a concern unless the source file is quite sizable.

2.6 Disk Buffer Area

A FORTH disk buffer (including its 4 byte header) is 1028 ($0404) bytes long. The buffer area ends at 53120 ($CF80) and begins at 53120 minus the number of disk buffers * 1028. The FORTH disk system requires a minimum of two disk buffers but is initialized with eight buffers for ease of editing. Thus, in the initial configuration the start of the disk buffers area is 53120 - 8*1028 = 44896 ($AF60).

#BUFF is a constant which returns the number of buffers the system is set up to use. If it is determined that a different number is required, the following FORTH line may be used to change the system to use a different number of buffers :

```
        new-number  '  #BUFF  !  CHANGE
```

Example:

```
        16  ' #BUFF !  CHANGE
```

will change the number of disk buffers to 16.

This may be done to increase the number of buffers (for editing a large number of screens for instance) or decrease the number of buffers to gain full dictionary usage of memory. A word, MAX-BUFFS is provided which automatically calculates the maximum amount of buffer space available and re-configures the system for it

2.7 User Area

The FORTH User Area variables and vectors are located starting at 53120 ($CF80). These are typically not used directly by the FORTH user, but through invocation of a user variable name. Unused User Area positions ($52 - $7E) may be assigned through use of the USER defining word (HEX 52 USER name - 7E USER name).

The User Area (and the start of the disk buffers area) may be moved by changing the value of the constant LIMIT and invoking the word CHANGE. The system will cold start (losing unflushed buffers) and use the new User Area for its system variables. This procedure is useful for freeing up the 4K block of $C000-$CFFF which may be used by independent machine language routines or I/O peripherals.

Example: We want to move the User Area to 48896 ($BF00), freeing up the area from 49152 ($C000) to 53247 ($CFFF). Perform the following:

```
48896 ' LIMIT !      ( SETS NEW USER AREA ADDR. )
CHANGE               ( RE-CONFIGURES SUPER FORTH )
```

Entering the following will switch back to the default area:

```
53120 ' LIMIT !  CHANGE
```

The position and order of the User Area data is as follows:

HEX Offset	Description
$00	- Last name addr
$02	- Delete character (not used)
$04	UAP - Pointer to start of user area ($CF80)
$06	SP0 - Pointer to top of stack
$08	R0 - Pointer to top of return stack
$0A	TIB - Pointer to terminal input buffer
$0C	WIDTH - Name field max. width (31)
$0E	WARNING - Warning flag (not FIG usage)
$10	FENCE - Fence against forgetting Kernel
$12	DP - Dictionary Pointer
$14	VOC-LINK - Vocabulary link
$16 - $35 :	Vector addresses (see vector table)
$36	>IN - Input stream character offset
$38	BASE - Current number base
$3A	BLK - Current block being LOADed
$3C	CONTEXT - Current CONTEXT vocabulary link
$3E	CSP - Check stack position variable
$40	CURRENT - Current DEFINITIONS vocabulary link
$42	DPL - Decimal Point Location (digits to right)
$44	FLD - Field length for numeric output conversion
$46	HLD - Holds latest char during numeric conversion

$48	OFFSET - Block offset to determine actual block #
$4A	OUT - Value incremented by EMIT
$4C	R# - Number of current editing line
$4E	SCR - Number of current editing screen
$50	STATE - Non-zero = compiling
$52 - $7E :	Available to user

2.8 High RAM

The area from 53248 ($D000) through the end of memory is generally occupied by the I/O Memory Map area and Commodore Kernel ROM. However, "hidden" in these areas is RAM which the user can access using special extension words in this FORTH system. The special extensions are listed in the section titled "C64 High RAM Access Words". The extension words take care of inhibiting interrupts, swapping the RAM in performing the operation, and returning the system to its usable state.

2.9 Running Out of Memory

The following lists some steps which may be taken when the system seems to be running low on memory.

1. Reduce the number of disk buffers in use, if possible. Each buffer uses over 1K of RAM. Initially, there are 8 buffers allocated. The system may be reduced to having only two buffers (see CHANGE).

2. Make use of the 12K of RAM available by placing large arrays and data structures in that space. Use the High RAM Access words to maintain the data.

3. Strip unused words from the system and re-compile only those necessary for a particular application (see next section).

2.10 Stripping System For An Application

To strip a system perform the following steps:

1. Determine which words are necessary to compile an application.

2. Create a "Load Script Screen" which contains the commands required to build the system.

3. LOAD the Load Script. If there are load errors do the following: load the EDITOR screens, correct the errors, and re-do this step.

An example Load Script is provided on screen #122. This Script was used to create the "DEMO4TH" program file on your master disk. Note the initial "FORGET D-READ". Since the Interrupt words and two Display words are not required, we may FORGET some words preceding THRU. Note also that IMMEDIATE words (such as RECURSE) may be compiled into the Transient Assembler area, since their definitions are only needed during compilation.

The following was entered in order to use the Load Script and save the stripped system as a program file:

```
( PLACE MASTER SCREENS DISKETTE IN DRIVE )
122 LOAD        ( CREATE STRIPPED SYSTEM)

( REPLACE SCREENS DISKETTE WITH FORMATTED PROGRAM DISKETTE )
APPLICATION "DEMO4TH"
```

3. FORTH Source Disk

FORTH differs from BASIC in numerous ways. One of these is the way that code is stored and executed. While using BASIC, the code is stored in what is essentially a source code format. The BASIC interpreter reads the program source code and "interprets" it into an executable machine form. Whenever it is time for a particular BASIC statement to be executed it must be re-interpreted. This is probably the major reason why BASIC runs so slowly (blank spaces and remarks also must be interpreted).

FORTH systems, however, pre-compile their source code into a more readily executable form. In interactive mode, for example, definitions are compiled upon execution of ";". This is why a decompiler (see DECOMPILE) must be used to re-construct the FORTH definition if the source code is not available. A result of this compilation is that the source code need not reside in memory after it has been compiled.

Source code must be read in and compiled before execution. This is generally accomplished using the FORTH word LOAD. As mentioned in the introduction on disks, this system provides two distinct modes for storing and retrieving source code. Cassette users MUST use File Mode, since Standard Mode assumes availability of a random access device (such as a disk).

3.1 SUPER FORTH 64 System Diskette

Along with the FORTH system itself, the system diskette comes with various FORTH source screens. These screens have already been compiled into the SUPER FORTH system. They are provided to allow flexibility for the advanced FORTH user who may wish to customize his system, and as an educational tool for the beginner who desires examples of FORTH programs and routines (see Appendix II).

Specifically, the Graphics/Sound Demo program is an example of how high resolution graphics, sprites, and the SID sound chip may be controlled by the FORTH programmer. The C64 Constants screens define many useful constants relating to the hardware registers on the C64. The Data Structures screens present examples of how to define data structures using FORTH. Single and two dimensional array structures, a string constant structure and a structure for initializing sprite areas are defined.

See the source screens index in the appendix for a complete list of what source screens are provided.

After starting up SUPER FORTH 64, insert the master diskette into the drive (if it is not there already). Perform the following commands:

FORTH Source Disk

```
        1 LIST
        2 LIST
```

will list screens 1 and 2 on the display. These screens contains a directory of the source screens on the master diskette.

You may have noticed that the SUPER FORTH 64 system diskette contains both Standard Mode screens AND a Commodore program file (namely SUPER FORTH 64)! I highly recommend NOT trying to mix these formats unless you really have a good idea about where things will end up.

3.2 Screens and Blocks

FORTH systems historically use their own type of disk format. A standard FORTH disk is divided up into 1024 byte blocks (when listing or editing "blocks" are also referred to as "screens") and each block is referenced by its block number.

3.3 File Mode

A FORTH source file consists of a group of FORTH screens which can be saved and loaded as a named file. For editing purposes, the screens are numbered sequentially starting at 1. Screen numbers are used only as reference to allow editing various parts of the file.

An extension set of FORTH words is used to operate in this mode. See File Mode extension words, section 5.2, for a complete description of File Mode word usage.

3.4 Standard Mode

To enter Standard Mode from File Mode use F-EXIT. In Standard Mode FORTH system knows nothing about the Commodore file system, therefore FORTH style blocks should not be mixed with Commodore style files when using this mode. In particular, the FORTH system itself is saved as a Commodore program file, therefore, when saving a new version of FORTH make sure that a Commodore diskette and NOT a FORTH screens diskette is in the drive!

The advantage, for the FORTH programmer, is that the former disk directory track is now available for use, allowing full use of virtually all sectors on a diskette with the following exceptions:

There are 683 sectors on a standard Commodore disk, however sector 0 of track 18 must not be written over by the programmer (specifically, there is a byte which specifies that the diskette has been formatted on a 1541 disk drive. If that byte is changed the diskette is still readable, but it can no longer be updated on a 1541 drive. I know of no method of changing this byte back short of re-formatting the diskette [which will wipe it clean] therefore I strongly recommend not changing that byte). Since there are 4 sectors per FORTH screen, the diskette is divided up into 170 FORTH screens, numbered 0 to 169.

The astute user will have noticed that 680 sectors are allocated as FORTH screens, leaving three unused by the FORTH disk system. The FORTH word T&SCALC performs the calculations to skip those sectors, so the user need not worry about them. The three sectors are track 17 sector 20, track 18 sector 0, and track 18 sector 1. The primitive disk word RWTS (normally not called directly by the user) allows access to all sectors on the disk, so the brave (and foolish) can still manage to destroy their disk despite system protection. This direct access by the system is necessary, however to perform a complete diskette to diskette BACKUP.

3.5 Multiple Disk Systems

FORTH can accommodate up to five disk drives, Commodore device numbers 8 through 12. Initially, the system is set up to use device 8 as the system drive (for file saves). The system device number can be changed by storing a new device number in the FORTH variable SYSDEV (see section 5.4.25). The FORTH system uses the names DR0, DR1, DR2, DR3 and DR4 to refer to the drives.

Initially, the system comes up assuming a single drive. If more than one drive is in use, CONFIGURE (see Glossary, Section 6.) must be entered to set the system to the correct number (if desired, the system may then be saved with the new configuration parameters set up, eliminating the need for reconfiguring each time the system is brought up). On Warm/Cold Starts or after using CONFIGURE, DR0 is selected.

FORTH screens are specified as a relative offset to block 0 of the currently selected drive. Thus, if DR0 is selected, the following table could be used to reference screens on any drive relative to drive 0:

		Relative to DR0		Relative to Self	
FORTH #	Device #	First	Last	First	Last
DR0	8	0	169	0	169
DR1	9	170	339	0	169
DR2	10	340	509	0	169
DR3	11	510	679	0	169
DR4	12	680	849	0	169

The "Relative to Self" column is used if the block to be accessed is on the currently selected drive.

Example: Two ways of listing the same screen are as follows:

```
DR0 200 LIST              ( LIST BLOCK 30 ON DR1 )
DR1 30 LIST               ( LIST BLOCK 30 ON DR1 )
```

3.6 4040 Drives

If you own a 4040 drive, configure your system to use two drives (see "CONFIGURE") and type ON DUAL. Your dual drive will now work with SUPER FORTH.

3.7 Disk Error Recovery

If a disk operation should result in an error during "Standard Mode" operation, the SUPER FORTH disk system will automatically display to disk error channel so the user can immediately ascertain what the problem is. On errors the system re-initializes the disk (DOS command "I") for the user.

If an error has occurred in reading the disk, there may be erroneous information in the disk buffers. EMPTY-BUFFERS should be used to clear out the buffers.

If repeated "NO CHANNEL" error messages appear, try entering the following:

```
9 CLOSE             ( CLOSES THE DISK CHANNEL )
CLRCHN              ( CLEARS I/O CHANNELS )
" I" DOS            ( INITIALIZES THE DISK )
```

Note that turning a drive on after the system has come up results in an error message and the drive should be initialized interactively.

4. FORTH Assembler Usage

This system includes the Bill Ragsdale 6510 (ne 6502) FORTH Assembler. The original article describing the Assembler is included in the Appendix. The article contains all information necessary for an experienced FORTH programmer to use the Assembler, however, programmers who are experienced in "typical" assemblers and NOT FORTH may look at the article and say "This is crazy!". This section is intended to ease the user into the world of the FORTH Assembler. The article should be used as reference while going through this introduction. After finishing the introduction the article should be re-read and the examples within it tried out on the machine.

The FORTH Assembler is part of an integrated applications development system (SUPER FORTH 64). It enables users to develop programs completely in high level FORTH and then, if desired, speed up the program by using the FORTH assembler to re-write only the words which are necessary to have running in machine language.

Another feature of the SUPER FORTH 64 system is that unlike other FORTH systems which provide assemblers which must reside in the dictionary in order to use them, the SUPER FORTH 64 Assembler may be removed from the dictionary after the application program is loaded, before using APPLICATION to save it to disk. This reduces the size of the program which must be loaded in by the end-user, or the amount of ROM memory to be used if the application is to be ROM'd. See the word A-REMOVE for instructions on removing the assembler.

It is highly recommended that beginning machine language programmers obtain one of the many 6502 machine language introductions which can be found in bookstores which carry computer books. It is also recommended that beginning programmers do not attempt to understand the Assembler before understanding how to use FORTH in general.

4.1 How the Assembler Works

First, some background on assembling in FORTH. In a typical assembler environment you enter code using an editor, assemble the module producing an object module, load the object module in and call it. If you wish to link the machine code with a high level language, there is usually a somewhat clumsy interface (if at all) with the machine language routine.

In the FORTH environment you enter code either interactively or editing to a screen. Just as in entering regular FORTH definitions, each method works the same way, but in one case you enter code directly (interactive) and in the other you edit a screen which contains the code to be loaded into the system later.

As assembly code is entered it is compiled into the dictionary. The same mechanism is used as when compiling definitions. The difference is that when compiling definitions, addresses of other FORTH words are compiled into the dictionary. When using the Assembler, actual machine language is compiled into the dictionary. CODE and END-CODE replace : and ; for delimiting a FORTH module.

4.2 Entering Assembly Code

The first thing you need to know in order to use the Assembler is how to enter code. Two words are used to control the interaction of assembly code with high level FORTH code: CODE and END-CODE. As described in the preceding section, CODE and END-CODE delimit an assembly module. The length of the module may be arbitrary, but it must be delimited by CODE and END-CODE.

CODE performs various tasks for the assembly coder. First, it calls in the ASSEMBLER vocabulary. Thus, the user need not concern himself about "entering" the Assembler- he simply uses CODE in the beginning of the routine. CODE expects a routine name following it, such as:

```
        CODE FOOBAR
```

CODE uses CREATE to create a header for a definition which will be called FOOBAR (in this example). It performs virtually the same functions as :, but whereas : leaves the run-time routine DOCOLON as its code field address, CODE leaves the start of the assembler module as its code field. You really needn't concern yourself with the technical aspects at the moment, just understand that an assembler module must start with CODE.

END-CODE, as you might guess, is used to end an assembly code definition. It restores whatever vocabulary was in effect when CODE was called (usually FORTH) and performs some verification that things are in order. If things are not in order a message will be displayed.

Example:

```
        CODE FOOBAR       ( CREATE HEADER )
        NEXT JMP,         ( LINK TO FORTH INTERPRETER )
        END-CODE          ( END OF CODE )
```

The example defines an assembly routine named FOOBAR with a minimum amount of code defined within it.

The word, FOOBAR, could be called from the keyboard or from within another FORTH routine. The second line NEXT JMP, is required to link to the system and will be explained in a bit. Try executing FOOBAR. It may not do much, but it's your first assembly code linked with the FORTH system.

4.3 Opcodes, Operands and Addressing Modes

One of the immediately recognizable confusions arises from the FORTH Assemblers use of Reverse Notation in specifying the operands and opcodes. The confusion gets worse in the more complicated addressing modes. The typical assembler has the following fields which may be entered (ignoring the comments field):

```
LABEL   OPCODE  OPERAND
```

where OPERAND itself is usually of various forms depending on the addressing mode used. The FORTH Assembler expects code in the following form:

```
OPERAND  OPCODE,
```

where again the OPERAND is broken down into various addressing modes to be discussed later in this section. There is no LABEL field. This is because labels are not required in this assembler, since code labels are used as places to branch to and branches will be generated by special control structures.

The OPCODE...OPERAND fields are reversed, and opcodes are always ended with a comma. Thus, as in the previous example, where a typical assembler would say:

```
JMP     NEXT
```

to specify an unconditional jump to the routine NEXT, our FORTH Assembler requires

```
NEXT    JMP,
```

here are some examples of typical vs. FORTH assembler for one mode and simple address mode opcodes:

```
Typical             FORTH
 BRK                        BRK,
 PHA                        PHA,
 LDA ADDR            ADDR   LDA,     ( ABSOLUTE ADDRESSING )
 INC A               .A     INC,     ( ACCUMULATOR ADDRESSING )
```

The article lists three reasons for having a comma following the opcode, none of which I particularly subscribe to. I heartily encourage experimentation with the Assembler source code (source screens have been provided- If an easier to use implementation is come up with I'm sure the 6502 based FORTH world would be very grateful. Please let us know the results if any).

Simple addressing modes, such as an absolute addresses, pose little problem (as seen in the above example). Complex operands, however, can start to look very unfamiliar when coded in the FORTH assembler.

The immediate mode address mode reverses the value and the #. Also, a space must be left between them. It is worth knowing that the value is left on the stack for processing. Therefore, FORTH words may be used to compute an immediate value. This is equivalent to calculating an immediate expression with a typical assembler, but since ALL FORTH WORDS are available to perform calculations with, the FORTH Assembler probably is much more flexible:

```
   TYPICAL                                   FORTH
OPCODE  #VALUE                          VALUE # OPCODE,
LDA     #3                                  3 # LDA,
LDX     #>DTA.ADDR          DTA.ADDR 255 AND # LDX,
LDY     #<DTA.ADDR         DTA.ADDR 8 RSHIFT # LDY,
LDA     #ROUT.NFA          ' ROUT NFA        # LDA,
ADC     #8660              60 SIN            # ADC,
```

The first example is the more typical use of immediate mode, that is specifying a simple value. Examples two and three describe ways of specifying the low and high bytes of a data address (DTA.ADDR leaves its PFA on the stack when executed). Example 4 describes a way of loading the name field address of a FORTH routine: ' ROUT leaves the PFA of ROUT on the stack and NFA converts it to the name field address. Example 5 is included just to give you an idea of some of the interesting (strange?) things that can be done by using other FORTH words. Here, we want the integer sine of 60 degrees added to the accumulator.

Okay, at this point we know enough to construct simple programs. The next example will increment location 32768 by 5 each time it is called. Following the code definition is a high level definition which tests the code out.

```
CODE INCR32K
        CLC,      ( CLEAR CARRY FOR ADD )
 32768  LDA,      ( GET DATA FROM 32768 )
 5 #    ADC,      ( INCREMENT IT BY 5 )
 32768  STA,      ( PUT IT BACK )
 NEXT   JMP,      ( GO BACK TO FORTH )
END-CODE

0 32768 !         ( CLEAR 32768 INITIALLY )
: TEST1           ( TEST THE CODE )
   20 0 DO
    INCR32K 32768 ?
   LOOP ;
TEST1
```

Note that we can call our machine language routine from within a FORTH definition, just as if it was another FORTH word. INCR32K can be executed interactively from the keyboard also. Notice too that by looking at TEST1 we would have NO IDEA that INCR32K is a machine language routine! This is one of the niceties of FORTH- programs can be completely written and debugged in high level and then optimized by re-writing the time critical words in machine code WITHOUT AFFECTING THE STRUCTURE OF THE PROGRAM. Try that in another language!

By this point you should be starting to get the idea of how addressing modes work. If you don't understand the previous part, go over it again before going on.

Indexed X and indexed Y are similar. The operands are actually very close to "typical":

```
     TYPICAL                          FORTH
   LDA     ARRAY,X           ARRAY ,X       LDA,
   STA     ARRAY,Y           ARRAY ,Y       STA,
```

Other than the OPERAND OPCODE reversal the only difference is really a space between the operand and the index.

The final three modes are just slightly stranger than immediate addressing:

```
      TYPICAL                      FORTH
   JMP    (VECTOR)           VECTOR )   JMP,
   EOR    (6,X)              6 ,X)      EOR,
   CMP    (DATA),Y           DATA ),Y   CMP,
```

As can be seen, the left open parenthesis is gone, the address is separated from the address modifier by a space, and of course the operand appears before the opcode and the modifier.

In general, the jump indirect instruction will rarely be used (and in fact it is a good idea to avoid it on any 6502 based chips- there is a hardware bug [excuse me, feature] involving indirect jumps whose addresses span page boundaries- JMP ($xxFF) will produce effectively indeterminate results. It does not matter what page xx is!).

At this point we can write programs using any instruction other than branches. This code initializes variable D1 with 6325, the value of constant C1.

```
6325 CONSTANT C1
VARIABLE D1

CODE MOVEDATA
 C1 255 AND #   LDA,      ( LO BYTE OF C1 )
         D1     STA,      ( LO BYTE OF D1 )
 C1 8 RSHIFT #  LDA,      ( HI BYTE OF C1 )
         D1 1+  STA,      ( HI BYTE OF D1 )
         NEXT   JMP,
END-CODE
MOVEDATA D1 ?
```

C1 255 AND isolates the low byte of C1, by ANDing its value with $FF (255). C1 8 RSHIFT isolates the high byte of C1 by performing an 8 bit right shift on its value. D1 leaves the address of the low byte of its data area on the stack. D1 1+ leaves the address of the high byte of the D1 data area on the stack.

4.4 Interfacing With FORTH

On entry to the machine language routine the 6510 registers are set up as follows: The .A register is undefined and may be freely used. The .X register points to the bottom of the parameter stack. On return it must point to a proper parameter stack value. The .Y register is zero. It may be freely used. The stack pointer points to one byte below the bottom return stack item.

Certain values are set up for use by the assembly language programmer: XSAVE may be used to save the value of the .X register. N defines a 9 byte area from N-1 through N+7 which can be used for temporary calculations. SETUPN can be used to move values from the parameter stack to the N area. The bottom stack value is moved to N, the second to N+2, etc.

Example:

```
    CODE OVERADD  ( N1 N2 N3 --- N1 N2 N3 N1+N4)
     XSAVE  STX,     ( SAVE PARAM STACK PTR. )
     3 #    LDA,     ( # OF VALUES TO BE MOVED TO N )
     SETUPN JSR,     ( MOVE THEM )
            CLC,     ( CLEAR CARRRY FOR ADD )
     N 2+   LDA,     ( GET VALUE OF N2 LO )
     N 4 +  ADC,     ( ADD N1 LO )
            TAY,     ( SAVE LO BYTE )
     N 3 +  LDA,     ( GET N2 HI )
     N 5 +  ADC,     ( ADD N1 HI )
     XSAVE  LDX,     ( RESTORE STACK POINTER )
            DEX,     ( MAKE ROOM FOR NEW STACK VALUE )
            DEX,
     0 ,X   STY,     ( PUT LO ON STACK )
     1 ,X   STA,     ( PUT HI ON STACK )
     NEXT   JMP,     ( PUSH N2 ONTO STACK )
    END-CODE

    400 500 600 OVERADD .S DDROP DDROP
```

This example is a somewhat oblique way of adding together the 2nd and 3rd numbers on the stack and leaving them on the bottom of the stack. Two words, BOT and SEC are provided to ease referencing the bottom and second to bottom values of the parameter stack. BOT is equivalent to 0,X. SEC is equivalent to 2,X. Let's re-write the OVERADD routine to use the values directly from the stack:

Example:

```
    CODE OVERADD2  ( N1 N2 N3 --- N1 N2 N3 N1+N2 )
              CLC,
     SEC      LDA,     ( GET VALUE OF N2 LO )
     4 ,X     ADC,     ( ADD N1 LO )
              TAY,     ( SAVE IT )
     SEC 1+   LDA,     ( GET N2 HI )
     5 ,X     ADC,     ( ADD N1 HI )
              DEX,     ( MAKE ROOM ON STACK )
              DEX,
     BOT      STY,     ( PUT LO ON STACK )
     BOT 1+   STA,     ( PUT HI ON STACK )
     NEXT     JMP,
    END-CODE

     600 700 800 OVERADD2 .S DDROP DDROP
```

Notice that when we decrement X twice (once for each byte) to point to a new bottom that using BOT reflects this change. Now that we can manipulate values on the stack, we can easily interface with either high level routines or other machine language routines which use the stack. All the outside world needs to know is what goes on the stack and what remains after the routine is run.

4.5 Returning to Interpreter

Several return points to the FORTH interpreter are provided. A routine which is to link back to FORTH must perform a JMP to one of the return points. The alternate return points provide easy ways of leaving the parameter stack in the proper form:

NEXT : Has no effect on the parameter stack.

POP : Discards the bottom parameter on the stack. POP is equivalent to INX, INX, NEXT JMP,.

POPTWO : Discards the bottom two parameters on the stack. POPTWO is equivalent to INX, INX, INX, INX, NEXT JMP,.

PUSH : Adds a value to the parameter stack. The low byte of the value must be on the return stack, the high byte is in the .A register. PUSH is equivalent to DEX, DEX, 1 ,X STA, PLA, 0 ,X STA,.

PUT : Copies a value over the bottom parameter on the stack. The low byte of the value must be on the return stack, the high byte is in the .A register. PUT is equivalent to 1 ,X STA, PLA, 0 ,X STA,.

Let's use an alternate return point to shorten our OVERADD example even more. Remember, the last thing we want to do is to push the new 2 byte value onto the parameter stack:

Example:

```
CODE OVERADD3   ( N1 N2 N3 --- N1 N2 N3 N1+N2 )
           CLC,
 SEC       LDA,    ( GET VALUE OF N2 LO )
 4 ,X      ADC,    ( ADD N1 LO )
           PHA,    ( SET UP FOR PUSH )
 SEC 1+    LDA,    ( GET N2 HI )
 5 ,X      ADC,    ( ADD N1 HI )
 PUSH      JMP,    ( PUSH VALUE AND RETURN )
END-CODE

 123 456 789 OVERADD3 .S DDROP DDROP
```

Well, that certainly simplifies it! How does this compare with high level FORTH code? I ran a test comparing OVERADD3 with the following high level routine:

```
 : HILVL  >R DDUP + R> SWAP ;
```

Over 10000 executions, OVERADD3 ran in 1.4167 seconds, while HILVL ran in 4.7333 seconds. So OVERADD3 ran about 3 1/3 times as fast (BASIC, by the way, takes about 30 seconds to add two numbers together).

4.6 Code Structures

Now that we can write straight line code, it would be nice if we could use branches. This assembler provides branches around code the same way the high level FORTH does: using program constructs. The following table lists the names of the high and low level constructs:

High Level	Low Level
scond IF...ELSE...THEN	mcond IF,...ELSE,...THEN,
BEGIN...scond UNTIL	BEGIN,...mcond UNTIL,
BEGIN...scond WHILE...REPEAT	BEGIN,...mcond WHILE,...REPEAT,
BEGIN...AGAIN	BEGIN,...AGAIN,

There are two differences to the user, the low level word all end with a comma, and the high level words check for a stack condition (scond) while the low level words check for a machine status condition (mcond).

The stack condition is specified by a value on the stack. If the value is 0 the condition is considered false. Anything other than 0 is true. The machine condition is tested by specifying a condition testing word. The following table lists these words:

Words	Condition Tested
CS	Carry status flag set?
0<	Negative flag set (<0)
0=	Zero flag set (=0)
CS NOT	Carry flag clear?
0< NOT	Negative flag clear (>=0)?
VS	Overflow flag set?
VS NOT	Overflow flag clear?

The following example implements a routine which compares two numbers for equality non-destructively and leaves the result on the stack.

```
CODE A=   ( N1 N2 --- N1 N2 FLAG )
 BOT     LDA,     ( GET LO BYTE N1 )
 SEC     CMP,     ( = LO N2? )
 0= IF,
  BOT 1+ LDA,     ( GET HI BYTE N1 )
  SEC 1+ CMP,     ( = HI BYTE N2? )
  0= IF,
   INY,           ( SET Y=1 )
  THEN,
 THEN,
 TYA, PHA,        ( SET UP FOR PUSH )
 0 # LDA,         ( ZERO HI BYTE )
 PUSH JMP,
END-CODE

1435 1235 A= .S DDROP DROP
1435 1435 A= .S DDROP DROP
```

The machine level routine will run quicker, but look at the simplicity of the high level routine:

```
: NON=  DDUP =  ;
```

4.7 Subroutines

Subroutines may be created and linked to by creating a header, entering the assembler, entering the code and ending the subroutine with an RTS:

```
ASSEMBLER     ( MUST INVOKE MANUALLY, SINCE NOT USING "CODE" )
CREATE ONE+   ( B --- B+1 )
 BOT    LDY,
        INY,
 BOT    STY,
        RTS,

CODE TWO+  ( B --- B+2 )
 ONE+   JSR,
 ONE+   JSR,
 NEXT   JMP,
END-CODE

5 TWO+ .
```

The example implements a subroutine, ONE+ which performs a byte increment by one, and an assembler routine, TWO+ which performst a byte incrment of two by calling ONE+ twice.

WARNING! DO NOT CALL SUBROUTINES FROM HIGH LEVEL- THE SYSTEM WILL CRASH!

4.8 Macros

A macro is a code definition which compiles code when it is called. Macros may be created by enclosing code within a regular colon definition. When the macro name is invoked, the code within the definition will be generated:

```
: LSHIFT16        ( --- ) during assembly time
                  ( N --- N+1 ) during execution
  ASSEMBLER       ( MUST BE INVOKED ) .
  BOT  ASL,       ( SHIFT LO BYTE )
  BOT 1+ ROL,     ( SHIFT CARRY INTO HI BYTE )
;

CODE 32MULT  ( N --- N*32)
 LSHIFT16
 LSHIFT16
 LSHIFT16
 LSHIFT16
 LSHIFT16
 NEXT   JMP,
END-CODE

10 32MULT .
```

This example defines a macro, LSHIFT16 which performs a 1 bit shift of a two byte value. Since a 1 bit left shift is equivalent to multiplying by 2, we can define a routine, 32MULT which multiplies a value by 32 by invoking LSHIFT16 5 times to generate code for 5 1 bit left shifts.

Within the macro definition, FORTH can be used to control what actually gets generated when the macro is invoked. For instance, an alternative version of the LSHIFT16 macro could accept a number on the stack during assembly time and generate that many left shifts:

```
: MLSHIFT16       ( M --- ) during assembly time
                  ( N --- N LSHIFT M ) during execution
  ASSEMBLER
  0 DO
   BOT ASL,
   BOT 1+ ROL,
  LOOP ;

CODE 32MULTB      ( N --- N*32 )
 5 MLSHIFT16      ( GENERATES 5 LEFT SHIFTS )
 NEXT JMP,
END-CODE

10 32MULTB .
```

Both 32MULT and 32MULTB have generated exactly the same code! This can be verified by using DUMP to examine them:

```
        ' 32MULT 32 DUMP
        ' 32MULTB 32 DUMP
```

Well, there you have it. The only thing left to do is start writing some machine language routines. Be warned, it is VERY EASY to crash the system when writing in machine language! Here are some things to be careful about:

1) Always be sure to end your code with a JMP to NEXT, PUT, PUSH, POP or POPTWO. If the jump is missing you will crash since the system will not know how to link back to FORTH.

2) Be sure you have preserved the .X register and restore it properly when you are ready to exit your routine.

3) A PHA, without a corresponding PLA, (or a call to PUT or PUSH) will crash the system since the return stack will have an improper value in it.

4) A PLA, without a PHA, will crash because you have probably "trashed" the systems return address.

There are probably thousands of imaginative ways to crash a system using machine language, but the above four seem to be the ones which turn up most often.

5. Implementation Specific Words

The Standard MVP FORTH Word Set is listed in Section 6. This section details the MVP FORTH words which are specific to this implementation of the MVP FORTH Kernel and the extension words which have been designed to handle specific Commodore 64 functions. The combination of these two sets of words comprise the SUPER FORTH 64 Word Set.

The implementation specific words and extensions fall into the following categories:

1. Editor Words
2. File Mode Words
3. C64 Primitive Words
4. C64 Specific I/O Words and Extensions
5. C64 Kernel Interface Words
6. C64 System Utility Words
7. Graphics Words
8. Turtle Graphics Words
9. Sound Words
10. Music Editor Words
11. String Words
12. Interrupt Words
13. Display Screen Words
14. High RAM Access Words
15. Data Structure Words
16. Math Words

The notation used to describe FORTH words is as follows:

1) The word name and a brief description of its use.

2) A stack description of the following format:

```
( input --- output)    text
```

where "input" is a list of the values which are expected on the stack upon entry to the word and "output" is a list of values which are left on the stack by the word at the end of its execution. "Text", if included, is entered after the word's name. Text in brackets, such as [filename] indicates that the text following the word is optional.

3) A description of the usage and operation of the word.

4) An example of usage, where appropriate.

5.1 Editor Words

New FORTH definitions may be saved by editing them onto a FORTH screen. Editing within a screen remains the same with either mode (File Mode or Standard Mode). The differences are in initial access to screens and loading. In Standard Mode screens are accessed as described in STARTING FORTH, that is, individually, by block number. In File Mode a file workspace is set up either by using F-NEW to create an area for editing a new file, or by using F-EDIT to read in a previously created file for editing.

The editor vocabulary is entered either by listing a screen, using LIST, or by typing the word EDITOR. A line is edited by LISTing out a block (the LISTed block becomes the current screen) and using the Commodore screen edit keys to enter or modify a particular line of a screen (that line becomes the current line). The FORTH words "1) 2)...15)" are used to insert the text following them on that numbered line. LIST can be stopped prematurely by hitting any key while the LIST is occuring. This may be useful for editing a screen which does not completely fit on the monitor

The FORTH editor extensions are provided to ease copying lines, moving lines around, etc. The current screen number is stored in SCR. The current line number is stored in R#.

Example: This example describes a typical Standard Mode editing session. Most of the common editing commands will be used, therefore it is recommended that the beginning user follow the example referring to the appropriate section to understand the commands which are being performed. Let's say we wish to perform the following edits for our session:

1) Enter new text on screens #100, 101, 102 and 103.
2) Re-edit 102.
3) Enter new text on 104.
4) Re-edit 100 and 101.
5) Load in screens 100 to 104 and test definitions.

Follow this example on a formatted (NEW'd) blank diskette:

```
100 LIST ( SETS THE EDITOR SCREEN POINTER )
W ( CLEAR SCREEN 100 FOR EDITING )

( POSITION CURSOR AND ENTER THE FOLLOWING )
0) ( EDITOR EXERCISE: SCREEN 100 )
1) : TEST1
2)    ." THIS IS SCREEN 100 "
3)    CR ;
D-CLEAR   ( THIS SHOULD CLEAR THE BOTTOM OF SCREEN )
L ( LIST OUT 100 TO VERIFY - IF NOTHING IS ON IT
  BE SURE YOU ENDED EDITING LINES WITH A CARRIAGE RETURN )
```

```
1 12 C      ( COPIES LINE 1 TO LINE 12 )
2 13 C      ( COPIES LINE 2 TO LINE 13)
3 14 C      ( COPIES LINE 3 TO LINE 14 )
L           ( LIST SCREEN TO VERIFY EDITS )

12 5 M      ( MOVES LINE 12 TO LINE 5 & EXTRACTS 12 )
12 6 M      ( MOVES LINE 12- WAS PREVIOUSLY LINE 13 )
12 7 M      ( MOVES LINE 12- WAS 14 )
L           ( LIST SCREEN TO VERIFY EDITS )

2 X L       ( EXTRACT LINE 2 AND LIST SCREEN )
1 K 2 K L   ( KILL LINES 1 AND 2 AND LIST SCREEN )
4 O L       ( OPEN LINE 4 AND LIST )

N W         ( THIS WRITES OUT 100, CLEARS & LISTS 101 )
100 7 SC    ( COPIES LINES 5-7 FROM PREVIOUS SCREEN INTO )
100 6 SC    ( LINES 4-6 )
100 5 SC    ( LINE 4 WAS PREVIOUSLY EDITED LINE )
L           ( LIST SCREEN )
F           ( WRITE OUT SCREEN 101 )

101 102 COPY 102 LIST     ( COPY SCREEN 101 TO 102 )
N W         ( WRITE OUT 102, CLEAR & LIST 103 )

( POSITION CURSOR TO AND ENTER THE FOLLOWING )
0) ( EDITOR EXERCISE: SCREEN 103 )
1)
2) : TEST3
3)    THIS LINE WILL BE REMOVED
4)    ." SCREEN #103 "
5)    ." ANOTHER LINE "   ;
D-CLEAR
5 O L       ( OPEN LINE 5 & LIST SCREEN )

( POSITION CURSOR TO LINE 5 )
5)    CR

( POSITION CURSOR TO BOTTOM OF SCREEN )
3 X L       ( REMOVE LINE 3 & LIST SCREEN )
P L         ( WRITE 103 & LIST 102 )

( POSITION CURSOR TO LINE 0 )
0) EDITOR EXERCISE: SCREEN 102 )

( POSITION CURSOR TO LINE 4 )
4) : TEST2
5)    ." THIS IS SCREEN 102 "
D-CLEAR   ( LEAVE A SPACE AFTER D-CLEAR )
104 L W   ( WRITE 102, LIST CLEAR & LIST 104 )

( POSITION CURSOR & ENTER TEXT )
0) ( EDITOR EXERCISE: SCREEN 104 )
```

```
1) : TEST4 ." SCREEN 104 " ;
D-CLEAR
100 L       ( WRITE OUT 104 AND GO BACK TO 100 )
2 X 2 X 2 X L  ( MOVE TEXT DOWN TO LINE 2 )

( POSITION CURSOR TO LINE 2 )
2) : TEST0
D-CLEAR
N L         ( WRITE OUT 100 & LIST 101 )

( POSITION CURSOR & ENTER TEXT )
0) ( EDITOR EXERCISE: SCREEN 101 )

( POSITION CURSOR & EDIT TEXT )
5)    ." THIS IS SCREEN 101 "
D-CLEAR
F           ( WRITE OUT 101 )

( IF THE PREVIOUS WAS CORRECTLY ENTERED NOTHING SHOULD )
(  BE WRITTEN WHEN THE FOLLOWING COMMAND IS EXECUTED )
SAVE-BUFFERS

( INDEX CAN BE USED TO CHECK LINE ZERO OF A SET OF SCREENS )
100 104 INDEX

100 104 THRU       ( LOADS DEFINITIONS INTO DICTIONARY )
( WE COULD HAVE LOADED SCREENS INDIVIDUALLY WITH      )
(   100 LOAD  101 LOAD  102 LOAD  103 LOAD  104 LOAD )
VLIST              ( VERIFIES DEFINITIONS HAVE BEEN LOADED )

( TEST OUT NEW DEFINTIONS )
TEST0
TEST1
TEST2
TEST3
TEST4

FORGET TEST0       ( FORGETS ALL TEST DEFINITIONS )
```

Notice that during normal editing, screens are automatically flushed (written out to the disk) by using the editing commands N, P, L or F. It is a good practice to use SAVE-BUFFERS (also called FLUSH in STARTING FORTH) at the end of an editing session to insure that all screens have been written out to disk.

5.1.1 Configuring the Editing Screen

SUPER FORTH 64 provides the user the flexibility of determining his optimal screen format and configuring the system for that format. A screen always occupies 1024 bytes in memory, but how it is displayed is determined by the word C/L, a word which returns the number of characters per listing line. The number of lines to be listed is determined by the number of characters in a screen buffer (1024) divided by C/L. The initial system is configured for 64 characters per line (typing C/L . should display 64). Therefore, the initial screen format is 16 lines by 64 characters (1024/64=16 lines).

A user may wish to change the default format for various reasons. If a user never overflows his lines on the screen, for instance, he may prefer 35 characters per line, not wasting the 29 characters per line (positions 36 thru 64) which are always blank filled by the system. A format of 35 characters by 29 lines may also be useful for defining sprites using S-DEF. If the number base is changed to binary, a pictoral representation of the sprite may be entered as data (see S-DEF).

To change the listing format, the number of characters per line must be placed in C/L. This can be accomplished as follows:

```
        chars  ' C/L !
```

The additional Editor words 16), 17) ... will have to be defined to enable the Editor to handle the extra lines on the screen. The editing screen which defines 0) ... 15) also contains auxillary definitions for lines 16) ... 24), but these are not compiled into the initial system.

For example, to change the listing format to 40 characters by 25 lines, the following must be done:

```
        ( DEFINE EDITOR COMMANDS FOR LINES 16 THRU 24 )
        : 16) 16 SE ;   : 17) 17 SE ;   : 18) 18 SE ;
        : 19) 19 SE ;   : 20) 20 SE ;   : 21) 21 SE ;
        : 22) 22 SE ;   : 23) 23 SE ;   : 24) 24 SE ;

        ( CHANGE CHARACTERS PER LINE TO 40 )
        40 ' C/L !
```

Once the change has been made, screens which have been entered using a different format will appear jumbled. However, once the line definitions have been entered, changing format back (to list screens on the master disk, for instance) is easy:

```
        64 ' C/L !        ( CHANGES BACK TO DEFAULT FORMAT )
        ( LIST OR EDIT 64 CHARACTER FORMAT SCREENS )
        40 ' C/L !        ( CHANGES TO 40 CHARACTER FORMAT )
```

I recommend that if you change to another format, that format be stuck to. This will avoid screen format confusion

5.1.2 C : Copy A Line On A Screen

(FROM# TO# ---)

On the current screen copy the line at FROM# over the line at TO#. The line at FROM# remains the same.

Example: 14 3 C will copy line 14 over line 3 on the current screen. Line 14 will be untouched.

5.1.3 COPY : Copy Screen

(FROM# TO# ---)

Copies screen FROM# to screen TO#.

5.1.4 EDITOR : Use Editor Vocabulary

(---)

This word may be used prior to editing to insure that the EDITOR vocabulary is invoked. The EDITOR is kept as a separate vocabulary so that no conflict will arise with other words having the same names as EDITOR words.

5.1.5 F : Flush (Save) The Current Screen

(---)

In Standard Mode, this is used to flush (save) the current screen to the disk if the screen has been updated since the last flush. Unlike using SAVE-BUFFERS (or FLUSH as STARTING FORTH recommends) the flushed screen will remain accessible without re-reading it. F performs no action if the screen has not been updated since the last flush.

F is automatically invoked by N and P. This insures that when moving to the next or previous screens, the information in the current screen will be saved. F only need be used if LIST is used to get the next screen to update.

5.1.6 K : Kill A Line

(LINE# ---)

Kill (replace with blanks) the line at LINE# on the current screen.

Example: 10 K replaces line 10 with blanks.

5.1.7 L : List A Screen

([N] ---)

LISTs the ASCII symbolic contents of a screen to the current output device. If there is no parameter on the stack then the current screen is listed to the current output device. If there is a parameter on the stack then the current screen is flushed if it has been updated and screen N is listed to the current output device. Edits may be made directly to the lines listed by using the CRSR keys to position the cursor and the INST/DEL key for inserting for deleting text from the line. Edits on a line must be ended with a carriage return (this form of editing is the same as in BASIC).

Example:

```
L        ( LISTS CURRENT SCREEN )
20 L     ( FLUSHES CURRENT SCREEN AND LISTS SCREEN 20)
```

5.1.8 LIST : List A Screen

(N ---)

Lists the ASCII symbolic contents of screen N on the current output device, setting N as the current screen. N is stored in the current screen variable, SCR. Also invokes the EDITOR vocabulary if it is not already invoked.

Example: 1 LIST lists screen 1.

5.1.9 M : Move A Line On A Screen

(FROM# TO# ---)

Copies line at FROM# to line at TO# and extracts line at FROM# from the current screen.

Example: 4 13 M will insert line 4 under line 13 and removes line 4.

5.1.10 N : Next Screen

(---)

Flushes the current screen if it has been updated (see F) and sets the current screen to be the next sequential screen number.

Example: If the current screen is 5, N L would flush screen 5, set the current screen to 6 and list it.

5.1.11 O : Open A Line For Input

(LINE# ---)

Opens up the line at LINE# by moving from LINE# to 14 down 1 line. Line 15 is lost and LINE# is set to blanks.

Example: 6 O will open up line 6 by moving lines 6-14 to lines 7-15 and blanking out line 6.

5.1.12 P : Previous Screen

(---)

Flushes the current screen if it has been updated (see F) and sets the current screen to be the previous screen number.

Example: If the current screen is 5, P L would flush the current screen, set the current screen to 4 and list it.

5.1.13 SC : Copy Line From Different Screen

(SCR# LINE# ---)

Opens the current line in the current screen and copies LINE# from SCR# screen into it. The current line and current screen remain the same.

Example: If the current screen is 5 and the current line is 10, 8 4 SC will move lines 10-14 to 11-15 on screen 5 and copy line 4 from screen 8 into line 10 of screen 5.

5.1.14 SCOPY : Copy A Group of Screens

(FR-START FR-END TO-START ---)

Copies the group of screens from FR-START thru FR-END to the area starting at TO-START. This is useful for re-arranging areas of a standard disk. The copy proceeds from low to high, so be careful of overlapping areas!

Example: 5 10 14 SCOPY copies screens 5-10 to screens 14-19.

5.1.15 SM : Move Line From Different Screen

(SCR# LINE# ---)

Opens the current line in the current screen, extracts LINE# from SCR#

and copies it into the current line in the current screen. The current line and current screen remain the same.

Example: If the current screen is 5 and the current line is 8, typing 10 2 SM will open line 8 (moving 8-14 down one line), copy line 2 of screen 10 into line 8 of screen 5, and remove line 2 of screen 10.

5.1.16 W : Wipe the Current Screen Clear

(---)

Sets the contents of the current screen to blanks and LISTs it. This should always be used prior to initially editing a screen in Standard Mode. In File Mode W is not necessary since the buffers are initialized to blanks.

Example: N W will flush the current screen, set the next to current, wipe it and list out the blank screen. This is useful when initially entering a set of sequential screens.

5.1.17 X : eXtract A Line

(LINE# ---)

Extracts (removes) the line at LINE# from the current screen. All lines from LINE# until the end of the block are moved down one line. Line 15 is blanked out.

Example: 3 X Extracts line 3 of the current screen, moving lines 4-15 to 3-14 and blanking out line 15.

5.2 Source Screen File Mode Words

The words in this section are used to control source screen editing when in File Mode. The parameter [filename] is an optional filename which may be entered in quotes after the word is entered. If [filename] is entered it becomes the default filename, that is, the filename which is used to name the source file in following commands where [filename] is not entered.

Since File Mode defaults to program files to store source code, using the system for either cassette or disk is simply a matter of specifying the system device number in SYSDEV where cassette is device 1 and disk is one of device numbers 8 through 12.

Example:

```
SYSDEV !
```

will set the system to use cassette for file mode.

A File Mode type file can easily be converted to Standard Mode on disk by using F-EDIT to read the file in, use F-NUMBER to renumber the screens to the new Standard Mode area, F-EXIT to exit File Mode, and SAVE-BUFFERS to save the screens out in Standard Mode. See F-NUMBER for an example of File Mode to Standard Mode conversion.

A brief description of various File Mode words follows: F-NEW is used to create a new FORTH source file. After editing screens F-SAVE is used to save the file. F-EDIT is used to read an existing source file into memory for editing. F-APPEND will append a source file to screens already in memory. F-LOAD loads the source file in and compiles it into the dictionary. F-NUMBER is useful for re-numbering screens for conversion to or from Standard Mode. Typing F-NEW, F-EDIT, F-SAVE and F-LOAD automatically enter File Mode from Standard Mode.

Because the complete file must reside in memory for editing, if File Mode is to be used extensively it is recommended that the user create a special editing system which would make full use of the buffer space available. The following code produces such a system and saves it as a program file.

```
FORGET S-MULTIR
( INSERT FORTH SOURCE SCREENS DISKETTE )
25 26 THRU        ( LOAD MAX-BUFFS, DSWAP)
42 45 THRU        ( LOADS EDITOR )
MAX-BUFFS         ( ALLOCATES THE MAX # OF BUFFERS )
F-NEW             ( INITIALIZES SYSTEM FOR FILE MODE )
( INSERT A BLANK, FORMATTED DISKETTE )
SAVE-FORTH "SUPER FORTH.EDIT"
```

Note: The new system is an "edit only" system and should not be used to attempt loading in the edit program since only a limited amount of dictionary space is available and many of the higher level extensions (which may be required by the application) have been removed.

5.2.1 F-APPEND : Append A File To Block Buffers

(---) [filename]

Assumes the block buffers have already been set up and contain information. Uses variable FLAST to determine which block buffer to start reading the file into and reads the file in, appending it to screens which are already in the block buffer area. FLAST is updated to point to the last screen number assigned for the file.

For example:

```
F-EDIT  "FILE1"       ( Reads first file into buffers )
F-APPEND "FILE2"      ( Appends another file to first )
```

```
    { perform editing of screens }
F-SAVE "NEWFILE"        ( Saves the concatenated file )
    or
" S0:FILE1" DOS         ( Scratches FILE1 )
F-SAVE "FILE1"   ( Replaces FILE1 with the new file )
```

5.2.2 F-EDIT : Set Up To Edit File

(---) [filename]

Sets File Mode, initializes the block buffers, sets the default filename (if given) and reads the default file into the block buffer area for editing. Variable FLAST is set to the number of the last screen read in. An error condition exists if a file of the given name does not exist. The screens numbers will go from 1 to the screen number stored in FLAST.

Note: There must be enough buffers available to read in the file. If not, the end of the file will not be read in and the file must be read in again after changing the number of buffers.

Example: F-EDIT "FILE" reads FILE into the block buffers for editing.

5.2.3 F-EXIT : Exits File Mode

(---)

Sets up system to use "standard" mode for editing source screens. Does not affect the block buffer contents.

5.2.4 F-LOAD : Load File Into System

(---) [filename]

Opens file of name "filename", reads the file in and performs a LOAD of each block as it is read in. Closes file when load is finished. F-LOAD requires a minimum of 2 buffers to be allocated for loading. Therefore, if a large program is to be loaded, set the number of system buffers to 2 to minimize buffer space.

F-LOAD must be used AFTER a file has be saved using F-SAVE, since it reads the file in as it loads it. LOAD or THRU may be used to load in screens directly from the block buffers while still in edit mode.

Example:

```
F-LOAD "filename"
```

5.2.5 F-NEW : File Mode Initialization

(---)

Initializes the system for File Mode. Calls FILE-MODE, empties and resets block buffer pointers, initializes FLAST to 0 and initializes SCR to 1. This word is called by F-EDIT and F-LOAD, so File Mode is automatically entered by invoking one of those words. Can be used to insure the block buffers are initialized correctly.

5.2.6 F-NUMBER : Renumber the Block Buffer Screens

(start ---)

F-NUMBER will renumber the used blocks of the block buffer area. Numbering proceeds sequentially from "start". The buffers are checked starting with the buffer at FIRST. If the buffer has been used, it is assigned the next sequential number. All used blocks will be marked at UPDATEd.

This word is useful for conversion between Standard Mode and File Mode. File Mode screens are always numbered starting from 1, therefore if more than one File Mode file is to be put onto a Standard Mode disk, the screens must be renumbered first. Since F-NUMBER also marks screens as UPDATEd, SAVE-BUFFERS can be used to save the newly re-numbered screens to a standard disk after using F-EXIT to leave File Mode.

Example: If a File Mode file TESTFILE is 8 screens long and we wish to convert the file to Standard Mode and place the screens at screen 40 through 47, the following sequence may be used:

```
F-EDIT "TESTFILE"        ( Reads TESTFILE into buffers )
40 F-NUMBER              ( Renumbers screens & sets update )
F-EXIT                   ( Leaves File Mode, enters standard )
SAVE-BUFFERS             ( Saves 8 screens at 40 through 47 )
```

5.2.7 F-SAVE : Save Source Screen File

(---) [filename]

Sets the default filename if given. Opens the default file and writes the screens in the block buffers to that file. Closes the file when done.

If the file already exists and a replacement is desired, first scratch the old file before saving the new one.
Examples:

```
        F-SAVE  "TESTFILE"
```

saves the buffers to a new file called TESTFILE.

```
        " S0:TESTFILE"  DOS    F-SAVE  "TESTFILE"
```

replaces the previous TESTFILE with the contents of the block buffers.

5.2.8 FILE-MODE : Invoke File Mode

(---) [filename]

This is the primitive used by other File Mode words to invoke File Mode and set up the default filename. It invokes DR0, sets the mode to 1 and replaces <R/W> (the standard disk read/write routine) with FR/W (a dummy read/write routine which performs no I/O and drops all arguments passed to it) assuring that standard block I/O cannot occur (this can still be over-ridden, however, by words such as INDEX and BACKUP which use RWTS directly, bypassing the standard system.

5.2.9 FLAST : Last Screen Variable

(--- ADDR)

Variable which contains the highest screen number accessed in File Mode. This variable can be examined after using F-EDIT to determine how many screens were read in.

5.2.10 FNAME : Default File Name

(--- ADDR)

String variable which is set up by FILE-MODE to contain the default file name.

5.2.11 F-OPEN : Open Default File

(FLAG ---)

Primitive invoked by various File Mode words to open logical file 9 as the file in FNAME on the current system device. If FLAG=0 the file is opened as a "read" file. If FLAG=1 the file is opened as a "write" file.

5.2.12 READB : Read Block Into Buffer

(ADDR ---)

Primitive invoked by F-APPEND and F-LOAD to read 1024 characters from the currently opened file into the buffer at ADDR.

5.2.13 WRITEB : Write Block Into File

(ADDR ---)

Primitive invoked by F-SAVE to write 1024 characters from the buffer at ADDR into the currently opened file.

5.3 C64 Bit/Byte Manipulation Words

Many of the words in this section were defined to implement particular functions related to the Commodore 64. They are available for use by the user.

5.3.1 CATNIB : Concatenate Two Nibbles

(NH NL --- BYTE)

Concatenate the two nibble (4-bit) values, NH and NL into the byte (8-bit) value BYTE.

Example: HEX 8 9 CATNIB leaves 89 on the stack.

5.3.2 CBIT : Clear Bits in Byte

(ADDR MASK ---)

Clears the bits in the byte at ADDR according to the bits set in MASK. If a bit is set to 1 in MASK the corresponding bit of the byte at ADDR will be cleared. The remaining bits will be unchanged. This word can be used to clear one or more bits of an I/O register.

Example: HEX D016 18 CBIT clears the multi-color mode and the 40 column select bits in the VIC Control Register (see memory map in C64 reference manual).

5.3.3 FBIT : On Flag CBIT/SBIT in Byte

(FLAG ADDR MASK ---)

If FLAG is true (1) calls SBIT to set the MASKed bits in the byte at ADDR. If FLAG is false (0) calls CBIT to clear the MASKed bits in the byte at ADDR.

Example: HEX 1 D016 18 FBIT sets the multi-color mode and the 40 column select bits in the VIC Control Register (see memory map in C64 reference manual).

5.3.4 LSHIFT : Perform a 16-bit Left Shift

(N #BITS --- N)

Shifts the 16-bit value N left the number of bits specified in #BITS. Zeroes are shifted in from the right. This can be used to effect a fast unsigned multiply by a power of 2. For example, to multiply N by 256 (2**8), N 8 LSHIFT can be used. This will execute many times faster than N 256 *.

5.3.5 MASK : Calculate 2**N

(N --- 2**N)

Given N leaves the value of 2**N (2 to the power of N) on the stack. This utility can be used to convert a bit number into a bit mask which can be used to set or clear a particular bit of a memory byte.

Example: HEX D016 4 MASK SBIT creates a MASK and sets the multi-color bit in the VIC Control Register.

5.3.6 RSHIFT : Perform a 16-bit right shift

(N #BITS --- N)

Shifts the 16-bit value N right the number of bits specified in #BITS. Zeroes are shifted in from the left. This can be used to effect a fast unsigned divide by a power of 2. For example, to divide N by 256 (2**8), N 8 RSHIFT can be used. This will execute many times faster than N 256 /.

5.3.7 SBIT : Set Bits in Byte

(ADDR MASK ---)

Sets the bits in the byte at ADDR according to the bits set in MASK. If a bit is set to 1 in MASK the corresponding bit of the byte at ADDR will be set. The remaining bits will be unchanged. This word can be used to set one or more bits of an I/O register.

Example: HEX D016 18 FBIT sets the multi-color mode and the 40 column select bits in the VIC Control Register (see memory map in C64 reference manual).

5.3.8 SPLIT : Split A Cell Into Two Bytes

(N --- BH BL)

Split the 16-bit value into its component byte values, leaving on the stack hi byte, low byte.

Example: HEX 89AB SPLIT leaves 89 AB on the stack.

5.4 C64 Specific I/O Words & Extensions

The SUPER FORTH 64 I/O system has been designed to provide flexibility in dealing with I/O devices on the C64. The system provides the capability of re-directing I/O to any opened device. The words EMIT, ." , PRINT# and PUT# (which provide the FORTH output) and ?TERMINAL, GET#, KEY, CHARIN, INPUT and INPUT# (which provide input) use the C64 Kernel to perform I/O to the device which has been specified.

CMDI or words which use CMDI, such as GET# and INPUT#, automatically set INPLFN. CMD or words which use CMD, such as PUT# and PRINT#, automatically set OUTLFN.

For example, to talk to an RS-232 device instead of the C64 keyboard and monitor, the user would open the RS-232 device and store the logical file number in the Standard I/O variables, INPLFN and OUTLFN:

```
10 " <ctl-f>" RS232    ( opens a 300 baud RS-232 logical file )
10 CMD                 ( directs output to logical file 10 )
10 CMDI                ( directs input from logical file 10 )
```

To set defaults either perform a warm start or set INPLFN and OUTLFN to 0 (0 CMDI and 0 CMD will perform the function).

To write data to a Commodore sequential file the user could open the file, direct output to that file, and restore defaults when done. To read from that file the user could direct input from the file, get the input and then re-direct input for the interpreter.

Files should not be opened with logical file number 9 since this is reserved for system disk/cassette operation.

A word, PRINTER, is provided to open a file and direct output to a standard Commodore 1525 printer.

5.4.1 ?TERMINAL : Query Current Input Device For Character

(--- C)

This word performs the BASIC GET function- it returns an ASCII value from the keyboard. If no key has been depressed a zero is returned.

Example:

```
: TEST
   10000 0 DO
    ?TERMINAL ?DUP
     IF . LEAVE THEN
   LOOP ;
```

When invoked, TEST loops, waiting for a key to be depressed. If a key is depressed its ASCII value is printed, otherwise eventually the loop ends and nothing is printed.

5.4.2 CMD : Set File Number As Current Output Device

(LFN ---)

This word performs a function similar to the BASIC CMD. The logical file number LFN is stored in OUTLFN, the current output device variable. Output will then be directed to the file referenced by LFN until the value of OUTLFN is changed or the file is closed.

Example:

```
10 4 0 "" OPEN            ( OPENS CHANNEL TO PRINTER)
10 CMD                    ( DIRECTS OUTPUT TO PRINTER)
```

Causes all output to be directed to the printer until the value of OUTLFN is changed.

```
0 CMD
```

re-directs output back to the screen.

5.4.3 CMDI : Set File Number As Current Input Device

(LFN ---)

This word performs the input version of CMD. The logical file number LFN is stored in INPLFN, the current input device variable. Input will then be directed from the file referenced by LFN until the value of INPLFN is changed or the file is closed.

Example:

```
10 " {ctrl-f}" RS232      ( OPENS 300 BAUD RS-232 CHANNEL)
10 CMDI                   ( DIRECTS OUTPUT TO RS-232)
```

Causes all input to be directed from the RS232 device until the value of INPLFN is changed.

5.4.4 EMIT : Output Character

(ASCII-VALUE ---)

Transmit an 8-bit character value to the current output device. The current output device is the logical file number stored in OUTLFN (see OUTLFN for setting alternate output devices).

Example:

```
65 EMIT
```

Sends an "A" to the current output device.

5.4.5 EMIT7 : Output 7-Bit Character

This word zeros the left most bit in the character and then sends it to EMIT. Its primary use is in printing out FORTH word name fields using the word ID. . Since the high order bit in the last character of the name field is used as a flag to signal the end of the field, that bit must be zeroed in order to print a correct representation of the name field.

5.4.6 EXPECT : Get Input Line

(ADDR N ---)

As defined in ALL ABOUT FORTH except it uses CHARIN for input, thus allowing screen editing functions to be performed before a line is sent to the system input buffer. Uses the logical file number stored in INPLFN as the device to take input from (see INPLFN for setting alternate input devices).

5.4.7 FRE : Display Amount of Free Space Available

(--- BYTES)

Displays to the current output device the amount of space available from the top of the dictionary thru the value of LIMIT.

5.4.8 GET# : Get A Character From File

(LFN --- N)

This word performs the BASIC GET# function. LFN is first stored into the variable INPLFN, making LFN the current input file. GET# then returns an ASCII value from the keyboard. If no key has been depressed a zero is returned.

Example:

```
: TEST
 10 " {ctrl-f}" RS232   ( OPENS A 300 BAUD RS-232 CHANNEL)
 BEGIN
  10 GET#               ( GET CHAR FROM RS-232 )
  ?DUP IF 0 PUT# THEN   ( ECHO IT TO SCREEN )
  0 GET#                ( GET KEYBOARD CHAR )
  ?DUP IF 10 PUT# THEN  ( SEND TO RS-232 )
 AGAIN ;
```

This example performs the function of a simple terminal program. After using a "run-stop/restore" sequence to exit the example, type "10 CLOSE" to close the RS-232 file.

5.4.9 INPLFN : Input Device Logical File Number

(--- ADDR)

Returns address of the system variable INPLFN. The logical file number (LFN) of the current input device is stored in INPLFN. System input words KEY and CHARIN call the Kernel routine CHKIN with the LFN stored in INPLFN before performing input, thus enabling system input from devices other than keyboard. To use other devices (such as RS-232), first use OPEN to set up the LFN and open the device, then store the LFN of the opened device in INPLFN. To reset to the system default either store a 0 in INPLFN, CLOSE the opened input channel, or depress RUN-STOP/RESTORE to perform a system warm start.

5.4.10 INPUT : Input A Number From Current Input Device

(--- N)

Upon execution of this word, the system pauses and waits for the user to input a number from the current input file (see INPLFN). The single precision number is left on the stack.

Example:

```
: TEST
   CR ." ENTER NUMBER: "  INPUT
   CR 0 DO I . LOOP ;
```

Invoking TEST will cause the system to prompt and wait for entry of a number. The second part of TEST uses the entered number as the final value of the DO loop (use small numbers if you want to try this).

5.4.11 INPUT# : Input A Number From File

(LFN --- N)

Upon execution of this word LFN is stored into INPLFN as the new current input file. The system then pauses and waits for the user to input a number from this input file (see INPLFN). The single precision number is left on the stack.

Example:

```
10 " {ctrl-f}" RS232
10 INPUT# (user enters number) .
0 CMDI            ( RESTORE INPUT FROM KEYBOARD)
```

The first line of the example opens an RS-232 device as a logical file. The next line sets up the RS-232 device as the current input file, waits for a number to come from it and prints the number. The "0 CMDI" resets the system to accept input from the keyboard.

5.4.12 JOY1 : Joystick Constant

(--- $DC01)

Returns the address of joystick 1. Fetching from this address will get the latest value of joystick 1.

Example: JOY1 C@ gets the value of the current position of joystick 1.

5.4.13 JOY2 : Joystick Constant

(--- $DC00)

Returns the address of joystick 2. Fetching from this address will get the latest value of joystick 2.

Example: JOY2 C@ gets the value of the current position of joystick 2.

5.4.14 KEY : Input Character

(--- CHAR)

Get a character from the current input devce. The contents of INPLFN are used to set up the input device.

5.4.15 MODE : Source File Mode Variable

(--- ADDR)

Variable which determines the systems block buffer mode of operation. If MODE=0, Standard Mode is used. If MODE=1, File Mode is used. FILE-MODE and F-EXIT automatically change MODE to enter and exit File Mode.

5.4.16 OUTLFN : Output Device Logical File Number

(--- ADDR)

Returns address of the system variable OUTLFN. The logical file number (LFN) of the current output device is stored in OUTLFN. System output word EMIT calls the Kernel routine CHKOUT with the LFN stored in OUTLFN before performing output, thus enabling system output from devices other than keyboard. To use other devices (such as RS-232), first use OPEN to set up the LFN and open the device, then store the LFN of the opened device in OUTLFN. To reset to the system default either store a 0 in OUTLFN, CLOSE the opened output channel, or depress RUN-STOP/RESTORE to perform a system warm start.

5.4.17 PADDLE@ : Fetch Paddle X,Y Values

(--- XVALUE YVALUE)

Gets the values of the X-paddle and Y-paddle A/D outputs. XVALUE and YVALUE range from 0 to 255.

Example: PADDLE@ . . will print the x and y values of the paddle registers.

5.4.18 PRINT# : Print A Number To File

(N LFN ---)

Upon execution of this word LFN is stored into OUTLFN as the new current output file. The value of N is then displayed on the current output device.

Example:

```
10 4 0 "" OPEN
1234 10 PRINT#
0 CMD
```

This example opens a printer as a logical file, re-directs output to the printer, prints the number "1234" and finally resets output to go to the display screen.

5.4.19 PRINTER : Open a Printer File and Re-direct Output

(FLAG ---)

This word is included to enable users to easily direct output to a 1525E printer. When invoked with FLAG=ON, PRINTER opens logical file number 127 as a printer output unit and re-directs output to the printer. All system output will go to the printer until one of the following occurs: 1) OFF PRINTER is invoked, 2) RUN-STOP/RESTORE sequence warm starting the system, 3) 127 CLOSE is entered, closing the printer file, 4) a command is entered which directs output to a non-printer file (such as 0 CMD).

When invoked with FLAG=OFF, PRINTER closes logical file 127 and directs output to the display screen (performs a 0 CMD).

Example:

```
ON PRINTER          ( OPENS PRINTER FILE )
." TO PRINTER "
0 CMD               ( DIRECTS OUTPUT TO SCREEN )
." TO SCREEN "
127 CMD             ( DIRECTS OUTPUT TO PRINTER )
12 EMIT             ( SEND PAGE EJECT TO PRINTER )
0 TRIAD             ( PRINT THREE SCREENS )
12 EMIT             ( SEND PAGE EJECT )
0 10 INDEX          ( PRINT INDEX OF SCREENS )
OFF PRINTER
```

This example demonstrates various things which can be done by re-directing output between the printer and the display screen.

5.4.20 PUT# : Set Output File and Send Character

(CHAR LFN ---)

LFN is first stored into the variable OUTLFN, making LFN the current output file. PUT# then sends CHAR to the output file.

Example: See GET# for an example of a simple terminal program which utilizes PUT#.

5.4.21 RS232 : Open An RS-232 Channel

(LFN ADDR ---)

This word is used to open an RS-232 channel for I/O operations to use. LFN specifies the logical file number of the channel. ADDR specifies the address of a string which contains command information for the channel. Use of the immediate string word, "", will leave a proper address on the stack. The channel is opened to device number 2 with a secondary address of 0.

Note : In opening RS-232 files characters with the numeric values 1-26 may be entered by depressing ctrl-a to ctrl-z.

Example:

```
10 " {ctrl-f}" RS232
10 CMDI  10 CMD
```

opens a 300 baud RS-232 channel and directs output to it and input from it.

Opening an RS-232 channel automatically causes the allocation of 512 bytes of memory for input and output buffers. Initially, this area would be 40448 ($9E00) to 40959 ($9FFF), the top of the default BASIC memory area. This area would not be interfered with in the initial system, since the disk buffers only extend down to 44896 ($AF60). If more buffers must be allocated, however, the RS-232 area may have to be moved. The safest way to do this would be to move the top of the FORTH user memory down 512 bytes and move the RS-232 buffers to the top of FORTH memory. This can be accomplished as follows:

```
LIMIT 512 - ' LIMIT ! CHANGE  ( MOVE FORTH DOWN 200 BYTES )
HEX D000 283 ! DECIMAL        ( SET NEW TOP OF MEMORY )
```

5.4.22 RWTS : Read/Write Track & Sector

(ADDR R/W DRIVE SECTOR TRACK #PAGES --- ERRCOUNT)

This is the disk primitive used to interface with the 1541 disk drives. It is used by <R/W> to implement FORTH's "virtual memory" system. Aside from user memory, RWTS may perform reads and writes to the High RAM areas, 53248 ($D000) - 65535 ($FFFF).

ADDR: Address operation is to be performed on

R/W: 0 = write operation, 1 = read operation.

DRIVE SECTOR TRACK: Disk address for begin operation.

PAGES: Number of disk pages (256 byte sectors) operation is to be performed on (1 FORTH screen = 4 disk pages [1024 bytes])

ERRCOUNT: Number of errors detected during operation (the error channel is automatically displayed on error and on the initial disk access after the disk is first powered on).

Note: RWTS is not normally used unless track/sector I/O is required. For block I/O see BLOCK and R/W.

Example:

```
PAD 1 0 0 18 1 RWTS .
PAD 256 DUMP
```

Reads track 18, sector 0 (the bitmap) into memory and lists it.

5.4.23 SAVE-FORTH : Save A Compiled System

(---) [filename]

This word FREEZEs the system, determines memory boundaries of the system and invokes SAVE to save the system to the last specified file (the initial default filename is "SUPER FORTH 64") or the new file specification, on the device specified by SYSDEV (see Kernel SAVE).

Examples:

SAVE-FORTH saves the system to the default filename.

SAVE-FORTH "NEW-FILENAME" saves the system to the filename "NEW-FILENAME" and sets "NEW-FILENAME" as new default.

5.4.24 SECTRKT : Sectors/Track Table

(--- ADDR)

Table used by T&SCALC to determine the track and sector number of a disk sector number relative to the beginning of the disk. Since the number of sectors per track varies on the 1541 disk, a table lookup (as opposed to a simple multiplication) must be used to determine the position of a sector on the disk.

5.4.25 SYSDEV : System Device Variable

(--- ADDR)

Variable which contains the number of the system device. Routines which perform file or direct sector I/O use the device number stored here to perform their operation to (such as saving the system, application, etc.) SYSDEV should be set to 1 for cassette, or 8-12 for disk I/O.

Example: 1 SYSDEV ! sets the system device to cassette.

5.4.26 UPORT : User Port Constant

(--- $DD01)

Returns address of the C64 User Port. This address can be used to fetch values from or write byte into the User Port.

5.5 C64 Kernel Interface Words

The following words are used to interface to the C64 Kernel ROM routines. Kernel routines which are not provided can usually be easily implemented by using SYS or SYSCALL.

Examples: The following will implement TALK and ACPTR

```
HEX
: TALK  ( DEVICE --- )
   0 0 FFB4 SYSCALL ;

: ACPTR ( --- BYTE )
   0 0 0 FFA5 SYS DDROP DROP ;
DECIMAL
```

5.5.1 CHARIN : Character Input

(--- CHAR)

Word used to call the system Kernel CHARIN routine- it sets the input device from the logical file number stored in INPLFN (set to keyboard on cold/warm starts) and performs character input utilizing the C64 screen editing functions if device is the keyboard.

5.5.2 CLALL : Close All Files

(---)

Calls Kernel CLALL routine to close all opened files.

5.5.3 CLOSE : Close A Logical File

(LFN ---)

Calls Kernel CLOSE routine to close a previously opened logical file.

5.5.4 CLRCHN : Close I/O Channels

(---)

Calls Kernel CLRCHN routine to close all open channels and restore the I/O channels to their default values. This routine is automatically called by CLALL.

5.5.5 LOADRAM : Load A Program File Into Memory

(LOADADDR FILEADDR ---)

The program file whose name is in the string at FILEADDR is loaded into the memory at LOADADDR. If LOADADDR is zero, the file is loaded into the address in the first two bytes of the program file. This word can be used to load machine language sub-routines which can then be called from the SUPER FORTH system.

Example:

```
HEX C000 " ML.PROGRAM" LOADRAM
    0 " ML.PROGRAM" LOADRAM DECIMAL
```

Both of the above statements would result in the program file "ML.PROGRAM" being loaded into the area at $C000 (assuming the file was originally saved from $C000). Note: Loading a file over the FORTH dictionary area will likely result in a crash of the system.

5.5.6 OPEN : Open A Logical File

(LFN DEVICE SECOND ADDR ---)

This word is used to OPEN a logical file which can then be used for input/ output operations by storing the logical file number in INPLFN or OUTLFN. Calls SETLFS with the logical file number (LFN), device address (DEVICE) and the secondary address (SECOND), calls SETNAM with the name of the file to be opened and finally calls the Kernel OPEN routine to open the file. The proper ADDR value will be left on the stack by using the immediate string word, "" to specify the file name (see Example).

Example:

```
10 4 0 "" OPEN
10 CMD
." HELLO!"  CR
0 CMD
10 CLOSE
```

could be used to open a channel and direct system output to a serial printer.

```
: RS-232  10 2 0 " {CTRL-F}"  OPEN  3 OUTLFN ! ;
```

could be used to define a word to open an RS-232 channel and direct output to it.

5.5.7 SAVE : Save Memory to Device

(START END ---)

This interface to the Kernel save routine is used to save a span of memory locations as a program file to a device. It is called by SAVE-FORTH and APPLICATION to save the system. The memory is saved to the file name stored in the string variable SAVENAME. The device to be saved to is taken from the system variable SYSDEV.

START: Starting address in memory to be saved.

END: Ending address in memory to be saved.

5.5.8 SAVENAME : Name of File To Be Saved

This string variable contains the file name which is used by the SAVE routine to save a span of memory to a program file. SAVENAME $. CR will print out the name currently stored there. SAVENAME has room to hold a name of up to 19 characters (replacement prefix-"@0:" plus 16 characters max for a filename).

The following can be used to store a new text string in SAVENAME:

```
SAVENAME $CLR SAVENAME " NEWFILE" $CONCAT
```

5.5.9 SETLFS : Set Logical, First, Second

(LFN DEVICE SECOND ---)

This word is used to set up a system logical file. The Kernel routine SETLFS is called, passing the logical file number (LFN), the device address (DEVICE) and the secondary address (SECOND).

5.5.10 SETNAM : Set Name of File

(ADDR COUNT ---)

This word calls the Kernel SETNAM routine. ADDR is the address of a text string to be passed to SETNAM. COUNT is the length of the string.

Example:

```
$INPUT COUNT SETNAM
```

would accept an input string from the keyboard, determine its address and length and pass the parameters to the Kernel SETNAM routine.

5.5.11 ST : Get Kernel I/O Status Byte

(--- STATUS)

Gets the C64 Kernel status byte. It is useful to examine the status byte after a Kernel routine call (see programmers reference guide).

5.6 C64 Utility Words

The various words in this section are "utility" words, useful definitions which were changed or not included in the original MVP-FORTH definition. It is recommended that even the beginning user look through this section since many of these words are useful as "stand-alone" words, that is, they are complete SUPER FORTH programs in themselves and probably will not be used within other word definitions. The following words fall into this category: A-REMOVE, APPLICATION, CHANGE, BACKUP, DECOMPILE, DIR, DOS, DOSERR, PATCH, Tracing, and VLIST.

The remainder of the words are intended to be used within other word definitions (they can, of course, be used interactively also). These are: <ROT, ?DEPTH, D2*, OFF, ON, RDTIM, RECURSE, SETTIM, SWAPIN, SWAPOUT, SYS, SYSCALL and WAIT.

5.6.1 <ROT : Reverse Rotate Stack

(N1 N2 N3 --- N3 N1 N2)

This word effects a relatively common stack operation, that of a "reverse rotate". It is equivalent to executing the sequence ROT ROT.

5.6.2 ?DEPTH : Check Stack Depth

(N ---)

If stack depth is less than N (before N was entered) ABORT" is called printing out a message specifying EMPTY STACK.

5.6.3 A-REMOVE : Remove Assembler From System

(---)

This word is used to remove a remote Assembler vocabulary from the system prior to saving the system as an application. This feature lets users mix machine language routines (written using the FORTH Assembler) and high level FORTH routines without keeping the overhead of an Assembler in the dictionary.

When the procedure outlined below is followed, the Assembler and any user defined macro are loaded into memory starting at location 36864 ($9000). Words compiled after the remote Assembler is loaded in are compiled into the dictionary in the original dictionary area.

When the application screens have been loaded, A-REMOVE is invoked prior to saving the application. The dictionary is re-linked without the Assembler being linked in.

In order to utilize this feature follow these steps:

1) Fully debug your application, including any assembler routines required.

2) Load the initial system WITHOUT your application screens.

3) Enter FORGET ASSEMBLER to remove the system's assembler.

4) Place the Master Source Screens diskette (or backup copy) into the drive and type 114 120 THRU to load the remote assembler.

5) Place a user Source Screens diskette (if any) into the drive and load any assembler macro screens you may have- these definitions will also be removed after assembly (if none then ignore this step).

6) Replace the Master Source Screens diskette and type 121 LOAD to load the word A-REMOVE.

7) Type 50 92 THRU and 94 97 THRU to compile a system with no assembler or editor.

8) Replace the user Source Screens diskette and load your application screens.

9) Type A-REMOVE to remove the Assembler and any macros and re-link the system.

10) Put a newly formatted diskette in the drive (or format one)

11) Type APPLICATION "program-name" save your application out onto disk.

12) Load the application program to verify that it runs properly. If not, go back to step one.

WARNING: DO NOT USE SAVE-FORTH ON A SYSTEM HAVING A REMOTE ASSEMBLER WHICH IS STILL LINKED- THE DICTIONARY LINKS WILL NOT EXIST WHEN THE SYSTEM IS LOADED BACK IN. THIS WILL PROBABLY RESULT IN AN UNRECOVERABLE CRASH!!!

5.6.4 APPLICATION : Save System As An Application

(---) [filename]

Save a program which when loaded and run will automatically start up the latest word defined in the system. The given filename is used and becomes the new default. If no filename is given, the last default filename will be used.

Note: A new, formatted diskette should be placed into the drive prior to using this word.

The FORTH system will be rendered "invisible" through use of this word. This is accomplished by the system blanking out the name fields of FORTH words in the system, and re-defining the interpreter so that when the application file is loaded and run the application word will be immediately executed.

Note: This is NOT a Meta-compilation- the application system will remain the same size as the complete FORTH system. Prior to loading, the application can be kept to a minimum size by FORGETting THRU and recompiling only those source screens from the master diskette which are necessary in order to compile the application.

This utility is given in order to allow the FORTH applications designer to write a FORTH application and save it away to be used without allowing access to the underlying FORTH system. In this way the application can be distributed without copyright infringement. (The word APPLICATION must be executed to prevent copyright infringment!)

After use of this word the system may appear to be "hung". Upon warm starting (see Section 1.4) the application will be left running. The FORTH system must be reloaded in order to continue FORTH program development.

Programs saved using APPLICATION may be ROM'd as long as they occupy less than 16384 ($4000) bytes in memory. See the section below on E-PROMing code. If more information is needed, contact Parsec Research.

Example:

```
( PUT THE MASTER DISKETTE IN THE DRIVE )
99 110 THRU      ( LOAD FRACTALS DEMO )

( PUT A BLANK DISKETTE IN DRIVE )
" N0:FORTH-DEMO,D1" DOS     ( FORMAT DISKETTE )
APPLICATION "DEMOFILE"      ( SAVE DEMO AS APPLICATION )

( TURN OFF THE MACHINE & TURN IT ON AGAIN )
LOAD "DEMOFILE",8         ( LOADS IN DEMO PROGRAM )
SYS 2064                  ( STARTS UP DEMO )
```

5.6.4.1 E-PROMing APPLICATION Programs

To transfer SUPER FORTH application programs to E-PROM, two additonal short routines are needed. The auto-start routine described in the PROM-QUEEN instruction booklet, and a routine to transfer your program from the I/O area to the normal memory location for SUPER FORTH programs. The program listing below contains both of these routines, and should be added to the front of your application program.

AUTO-START/TRANSFER ROUTINE.

This routine should be assembled using the PROM-QUEEN assembler starting at $8000 and then placed on a disk as instructed in the PROM-QUEEN manual.

```
;AUTO-START ROUTINE.
10    .BY 0A 80 9F FF C3 C2 CD 38 30 FF
```

```
 20     LDX #$00
 30     STX $D016
 40     JSR $FDA3
 50     JSR $FD50
 60     JSR $FD15
 70     JSR $E518
 80     CLI
 90     JSR $8022           ;($8022=ADDRESS OF TRANSFER ROUTINE)
100     JMP $A000           ;END OF AUTO-START ROUTINE.
110     LDY #$00            ;START OF TRANSFER ROUTINE ($8022).
120     LDA #$61            ;LO-BYTE OF START OF FORTH PROGRAM>
130     STA $59
140     LDA #$80            ;HI-BYTE OF START OF FORTH PROGRAM.
150     STA $5A
160     LDA #$01            ;LO-BYTE OF WHERE TO XFER FORTH PROGRAM
170     STA $5B
180     LDA #$08            ;HI-BYTE OF WHERE TO XFER FORTH PROGRAM.
190     STA $5C
200LOOP LDA ($59),Y
210     STA ($5B),Y
220     CLC
230     LDA $59
240     ADC #$01
250     STA $59
260     LDA $5A
270     ADC #$00
280     STA $5A
290     CLC
300     LDA $5B
310     ADC #$01
320     STA $5B
330     LDA $5C
340     ADC #$00
350     STA $5C
360     LDA $5B
370     CMP #L,ENDADDRS
380     BNE LOOP
390     LDA $5C
400     CMP #H,ENDADDRS
410     BNE LOOP            ;END OF TRANSFER ROUTINE.
420     JMP $0810 ;STARTING ADDRESS OF RELOCATED APPLICATION.
430     .EN
```

ENDADDRS=Address of last byte+1 of your application program after transfer.

After assembling the above code, and saving to disk, plug the PROM-QUEEN into the back of your Commodore-64. Insert the autohex ROM into the PROM-QUEEN, and turn on your computer. Using the D command, load the auto-start/transfer routing into location $2000. Next, load your application program starting at location $2061. the two programs are now linked and may be burned onto the E-PROM as described in the PROM-QUEEN manual.

5.6.5 CHANGE : Change SUPER FORTH Configuration

(---)

Modify the SUPER FORTH 64 memory configuration based on the values in LIMIT and #BUFF. LIMIT specifies the start of the USER area and the top of the buffers area. #BUFF is used to determine the value of FIRST, the start of the buffers area. At least 96 ($60) bytes should be left above LIMIT for the USER variables. CHANGE ends with a call to COLD to re-write vectors in the new user area.

Example:

```
20 ' #BUFF !          ( set system for 20 buffers )
48896 ' LIMIT !       ( move top of system down 4k )
CHANGE
```

This example will change buffer allocation to 20 buffers and place the USER area at 48896 ($BF00).

5.6.6 CASE Structure Extensions

A program control structure, CASE, has been added to SUPER FORTH 64 to facilitate choosing one of a set of operations based on a value. CASE is an extremely useful construct for the beginning as well as the advanced FORTH user. Once the standard FORTH structures are understood (IF...THEN, DO...LOOP, BEGIN...WHILE...REPEAT, BEGIN...UNTIL, BEGIN...AGAIN), the user should learn and start using the CASE construct. For implementation details see the article in Appendix VII.

Example:

```
: CASE-TEST  ( VALUE --- )
   CR
   CASE
    0 OF ." CASE ZERO " CR ;;
    2 OF ." CASE TWO " CR ;;
    4 OF ." CASE FOUR " CR ;;
     ." VALUE OUT OF RANGE "
   ENDCASE ;
2 CASE-TEST
3 CASE-TEST
```

This example sets up three cases, one of which will be chosen if upon entering CASE-TEST the value on the stack is one of 0, 2 or 4. If the value is anything else, the words between the last ;; and ENDCASE will be executed. Also see the example in the article in Appendix VII.

5.6.6.1 CASE : Begin Case Structure

The starting word in a CASE program control structure. Must be "paired" with the word ENDCASE at the end of the CASE structure.

5.6.6.2 OF : Test For Particular Case

OF tests the value which is on the stack (the "select" value) against the value associated with the OF. If the select value is equal to the OF value, then the FORTH words between OF and ;; are executed and no other case is tested.

5.6.6.3 ;; : Specify End of Particular Case

;; is used to specify the end of a set of words which follow a particular OF in a CASE structure. If this is the last ;; before the ENDCASE word, then the words which are entered between ;; and ENDCASE will be executed if no OF...;; has been selected for execution.

5.6.6.4 ENDCASE : Specify End of CASE Structure

ENDCASE is used at the end of the CASE structure to specify completion of the CASE. Any words compiled between the last ;; and ENDCASE are executed if the select value does not match the values of any OF's.

5.6.7 Backup Utilities

Several words have been included to enable users to make partial or complete disk backups, using either a single disk drive or two drives. The backup utilities must be loaded from the master diskette in order to be used.

> NOTE: Owners of 4040 or other dual drives should use the Duplicate command which is internal to the drive, since that command will perform a complete backup in two minutes!

In general, the following instructions apply to all backup words:
After forgetting ASSEMBLER and
Prior to performing a backup, type HERE . to determine where the top of the dictionary is currently. The dictionary must be below 17152 ($4300) in order for a complete diskette copy to perform in five parts. If less space is available the copy will be divided into six or more parts. The following example will set up the dictionary and buffers and load the backup utility from the master diskette:

Using the source code side of the disk type ;

```
        FORGET ASSEMBLER          ( MAKE ROOM FOR 5 PART BACKUP )
        112 LOAD                  ( LOAD IN SINGLE DRIVE BACKUP )
        MAX-BUFFS                 ( SET UP MAX # OF BUFFERS )
```

For users with two disk drives, perform 111 LOAD instead of 112 LOAD. If you intend to use this utility often, it will probably pay to save the system using SAVE-FORTH. The single drive backup will prompt the user when source and destination diskettes should be inserted in the drive. The backup procedure will take approximately 20 minutes.

You must use a formatted diskette to perform the backup to (see section titled "Getting Started"). Place your diskette to be copied in the drive and type:

```
        BACKUP
```

If you are using the single drive utility you will be prompted when it is time to place a different diskette in the drive.

Since FORTH uses sector reads and writes, this utility can be used to copy either a FORTH format disk or a standard C64 format disk.

5.6.7.1 BACKUP : Complete Diskette Backup Utility

(---)

Utility which can be used to perform a complete backup of one diskette to another diskette. The backup may be performed either using a single disk drive or two drives. Since a complete diskette contains 170k bytes and there are at most 41k bytes available for buffer space a full disk copy must be performed in at least five parts.

5.6.7.2 COPYBUF : Copy Up to #BUFF Screens

(FLAG END START ---)

This routine is used to copy screens from STARTing screen number to ENDing screen number + 1 to a second diskette.

If FLAG is 1, the three sectors which cannot be addressed by screen number are also copied. If FLAG is 0 they are not copied.

In general this routine will be called by PCOPY and need not be called directly by the user. There are two versions of COPYBUF. The single drive version is loaded from screen #112. The two drive version is loaded from screen #111.

5.6.7.3 PCOPY : Perform A Partial Disk Backup

(FLAG START END ---)

This routine is used to specify the starting and ending screens to be spanned for a partial disk backup. If FLAG is 1, the three sectors which cannot be addressed by screen number are also copied. If FLAG is 0 they are not copied. PCOPY is invoked by BACKUP and SOURCE-BACKUP to copy a complete diskette or just the source code area of the master diskette, respectively.

5.6.7.4 SOURCE-BACKUP : Perform A Backup Copy of Source

This word may be used to backup only the source code section of the master diskette.

5.6.8 D2* : Double Word Mult. By Two

(D --- D*2)

This word implements a fast double precision multiply by two by performing a left shift of the value D.

Example:

```
     1234. D2* D.
Results in the value 2468 being displayed.
```

5.6.9 DECOMPILE : Source Decompiler Utility

(---) [namefield]

This word is used to decompile a definition from the dictionary. In this way the user can determine the components of a definition without having the source code available. DECOMPILE is set up to enable decompiling of the called definition by using the SPACE bar to step through its components, or decompiling the components themselves by using RETURN to thread through the components.

Example : DECOMPILE SETLFS would allow the user to decompile the SUPER FORTH 64 word SETLFS to examine its components. Multiple depressions of the SPACE bar will perform the decompilation.

5.6.10 DIR : Display Disk Directory

(---)

Causes the Commodore DOS file directory to be displayed from the diskette in the drive whose device number is in SYSDEV. The directory itself is loaded into the PAD area.

This word incorporates the PAUSE feature: a single keystroke will pause the directory display- another single keystroke resumes it, or a double keystroke aborts it.

5.6.11 DOS : Send A Command To the Disk

(ADDR ---)

This word is used to send a DOS command to the disk. It opens the command channel, sends the command string located at ADDR to the disk at SYSDEV and closes the command channel.

Using the immediate string word, ", will leave the proper address. All of the commands listed on page 41 of the 1541 User's Manual may be sent using this word.

Example:

```
" N0:SUPER FORTH 64,64" DOS
```

This example will cause the DOS "NEW" command to be sent to the disk, causing the disk to be formatted.

5.6.12 DOSERR : Read and Print the Disk Error Channel

(---)

Displays the disk error channel. If an error has occured, performs a disk init.

5.6.13 DUAL : Dual Drive Specifier

(FLAG ---)

This word is used to specify to the system that a 4040 type dual drive is being used. Flag is ON for a dual drive and OFF for a non-dual drive. The system initially assumes a non-dual drive.

5.6.14 MAX-BUFFS : Re-configure System For Maximum Buffers

Invoking MAX-BUFFS will automatically allocate all available dictionary space for use as buffers. This may be useful before performing a BACKUP to to set up an editing system under File Mode.

5.6.15 OFF : Leave Constant Zero On Stack

(--- 0)

This word may be used to symbolically represent a 0 condition.

5.6.16 ON : Leave Constant One On Stack

(--- 1)

This word may be used to symbolically represent a 1 condition.

5.6.17 PATCH : Patch Memory

(SADDR --- EADDR+1)

This word may be used to enable easy entry of patches. The word is used as follows:

```
    <patch-addr>  PATCH  <cr>
    p1 p2 p3 p4 p5 ... <cr>
```

where <patch-addr> is the starting address to patch and p1, p2 etc. are byte values to be entered starting at <patch-addr>. Up to 80 characters of patches may be entered for a single PATCH command. PATCH leaves on the stack the address of the next location to be patched (EADDR+1), so a follow-up call to PATCH will start patching where the previous call left off. The final call should be followed by a DROP.

Example:

```
        HEX 8000 PATCH
        0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12 13 14 15 16 17
        PATCH
        18 19 1A 1B 1C 1D 1E 1F
        DROP DECIMAL
```

This example will set locations $8000-$801F to values 0-$1F. Note, if the patches fit on a single 80 character input line the second PATCH call would not be needed.

5.6.18 RDTIM : Read the 60 Cycle Clock

(--- D)

Returns the value of the system 60 cycle clock as a double word value D.

Example:

```
      : TEST 1000 0 DO LOOP ;
      RDTIM TEST RDTIM DSWAP D- D.
```

The second line is used to time the number of clock ticks (1/60's of a second) it requires to execute the TEST definition.

5.6.19 RECURSE : Call A Definition Recursively

(---)

Used within a definition to invoke a recursive call, that is, a call to the definition which is being defined. Care must be taken within the word to allow an end to the recursion, otherwise the program will end up in an "endless" loop which will probably result in the parameter stack being quickly used up! This word may appear in other systems as RECUR or MYSELF.

Example:

```
      : FACTOR ( N --- )
         ?DUP IF
           DUP 1- >R  1 M*/  R>  RECURSE
         ELSE  D.  THEN ;
      : FACTORIAL ( N --- ) 1. ROT FACTOR ;
      12 FACTORIAL
```

This example defines a simple recursive word which calculates and displays the factorial of an integer between 1 and 12.

5.6.20 SETTIM : Set the 60 Cycle Clock

(D ---)

Sets the system 60 cycle clock with the double precision value D.

Example:

```
      : TEST 1000 0 DO LOOP ;
      0. SETTIM TEST RDTIM D.
```

The second line is used to time the number of clock ticks (1/60's of a second) it requires to execute the TEST definition.

5.6.21 SWAPIN : Swap Kernel ROM & I/O Area In

(---)

Swaps the C64 Kernel ROM and I/O Memory Map Area back in and re-enables interrupts. Must be used after SWAPOUT is used prior to calling any C64 Kernel routines.

Example:

```
HEX SWAPOUT D000 1000 FILL SWAPIN DECIMAL
```

would zero out the 4K RAM area underneath the I/O Memory Map Area. Interrupts would not occur until the FILL was finished and SWAPIN was invoked.

5.6.22 SWAPOUT : Swap Kernel ROM & I/O Area Out

(---)

Disables all interrupts (including NMI) and swaps out the C64 Kernel ROM and I/O Memory Map Area, making available the RAM underneath for use by the FORTH system.

Note: SWAPIN must be used before attempting to use any words which call any C64 Kernel routines or use the I/O registers. If SWAPIN is not used the FORTH system is not likely to run correctly.

Example:

```
HEX SWAPOUT D000 1000 FILL SWAPIN DECIMAL
```

would zero out the 4K RAM area underneath the I/O Memory Map Area. Interrupts would not occur until the FILL was finished and SWAPIN was invoked.

5.6.23 SYS : Call Machine Language Routine

(.A .X .Y ADDR --- .A .X .Y STATUS)

Word used to interface with machine language sub-routines which are external to the FORTH system (such as C64 Kernel routines). .A .X .Y are values which will be loaded into those 6510 registers prior to performing a JSR to the routine located at ADDR. Upon return from the routine all registers and the processor status word are returned on the parameter stack.

The called routine must end with a RTS to return to the system and the hardware stack must be left in the same condition as when the routine was first called.

Example:

```
1 2 3 32768 SYS . . . .
```

sets hardware registers .A=1, .X=2 and .Y=3 before calling the machine language routine at address 32768 ($8000). Upon return from the called routine, the values of the processor status, .Y, .X and .A are sent to the current output device.

5.6.24 SYSCALL : Call Machine Language Routine

(.A .X .Y ADDR ---)

An alternate to SYS which does not return anything on the stack when the call is finished.

Example:

```
1 2 3 32768 SYSCALL
```

sets hardware registers .A=1, .X=2 and .Y=3 before calling the machine language routine at address 32768 ($8000).

5.6.25 TRACE

(FLAG ---)

Turns on or off the tracing variable, TFLAG based on FLAG. If FLAG is ON (1), then tracing is set on. If FLAG is OFF (0), then tracing is set off (see Tracing).

5.6.26 Tracing Forth Definitions

(---)

These words must be loaded in to be used. The ":" command is re-defined so that any words which are defined after loading the tracing routines will cause the name of the command to be printed along with the contents of the stack (see Appendix VII).

To control tracing, a word has been defined, TRACE, which turns on or off tracing mode. During compilation of definitions, if TFLAG is set ON (1), the word will be compiled with the trace option. If TFLAG is set OFF (0), the word will be compiled without the trace option. Thus, by invoking ON TRACE or OFF TRACE before loading a definition tracing may be selectively enabled.

During execution if TFLAG is set ON the words which were compiled with the trace option will be traced, that is, the following will occur:

1. The name of the traced word is displayed.
2. The contents of the stack is displayed (according to the setting of .SS [see also .S]).
3. Tracing pauses and waits for the user to depress any key to continue.

If TFLAG is set OFF during execution no tracing will occur.

> Note: When loading the screen with the trace routines, the message NOT UNIQUE will print out, reminding the user that a word (in this case, :) is being redefined. It is simply an informative message!

Example:

```
98 LOAD           ( LOADS TRACE WORDS IN )
OFF TRACE
: TEST1 ." --TEST1-- " CR ;
ON TRACE          ( TURNS TRACING ON )
: TEST2 TEST1 ." --TEST2-- " CR ;
: TEST3 TEST2 ." --TEST3-- " CR ;
OFF TRACE         ( TURNS TRACING OFF )
: TEST4 2 0 DO I TEST3 LOOP ;
TEST4 DDROP       ( TRACING IS OFF )
ON TRACE          ( TURN ON TRACE DURING EXECUTION )
TEST4 DDROP       (TRACING IS ON )
```

The above example will cause tracing of words TEST2 and TEST3, but not TEST1 or TEST4 while tracing is set to occur.

5.6.27 VLEN : VLIST Line Length Variable

(--- ADDR)

Variable which controls the length of the VLIST line. Initially set to 40 for printing to a 40 column screen.

5.6.28 VLIST : List Vocabulary Words

(---)

List the word names of the CONTEXT vocabulary starting with the most recent definition. Use VLEN and VTAB to determine line length and column formatting when outputting the vocabulary list.

This word incorporates a PAUSE feature, by which pressing any key will freeze the display. Once suspended, the VLIST my be resumed by a single keystroke, or aborted by striking any two keys in rapid succession.

5.6.29 VTAB : VLIST Tab Length Variable

(--- ADDR)

Variable which controls the length of tabs which denote columns during a VLIST printout. Initially set to 13 (three columns on a 40 column screen).

5.6.30 WAIT : Pause N Clock Ticks

(N ---)

Causes the system to pause until N clock ticks (1/60's of a second) have passed. Even multiples of 60 will wait an even number of seconds.

Example:

```
." TESTING... "  5 60 * WAIT  ." THE WAIT FUNCTION" CR
```

This example causes the system to pause five seconds in between printing out the messages.

5.7 Graphics Related Words

The following graphics primitives have been defined in order to give the user a basis with which to write graphics routines. They are designed to take some of the pain out of dealing with the VIC-II graphics chip at the lowest level.

Words which start with the prefix "B-" are bitmap (hi-resolution) graphics related words. These words should only be used in bitmap mode because they may produce undesirable results. They can't be used in the normal mode because the character screen is used to determine colors in bitmap mode. Also, the 8K bitmap screen area may fall in an undesirable place (such as in the middle of your FORTH program) if you simply start using the bitmap words without being sure of where the bitmap resides.

Therefore, I recommend that prior to experimenting with any of these words you first set up the bitmap area in a known place (this is easily accomplished using BITMAP) and set up a word to restore things to a known useable state. Warm starting should get the system back to a useable state.

Words which start with the prefix "S-" are sprite related words. These words use the contents of the variable SPRITE to determine which sprite they should act upon, therefore SPRITE must be set up with a sprite number (0-7) before using any of these words. The sprite referenced by the contents of SPRITE is referred to as "the active sprite". The words S1, S2, S3, S4, S5, S6, S7 and S8 are provided to automatically set one of the eight sprites as the active sprite (Refer to the Commodore 64 Programmers Reference Guide and Commodore 64 Graphics & Sound Programming listed in Appendix VIII).

Sprites may be entered into the system in one of two ways: The Sprite Editor word, S-EDITOR, may be used to design a sprite on the display screen. This word automatically moves the sprite data for the newly created sprite into the dictionary. Raw data (such as output from an external sprite editing program or from BASIC DATA statements) may be entered by using the sprite defining word, S-DEF to create the definition and load the sprite upon execution. (see S-DEF in the Data Structures section).

The cursor color may be changed from the keyboard by depressing the proper key sequence for chosing a cursor color (see Commodore User's Guide- "Printing Colors", p.56). The cursor color may be changed under program control by EMITting the character code of the color. This can be done directly (such as 28 EMIT to turn the cursor red) or within a ." string:

Example:

```
: BLUE-CHARS ." {ctrl-7}THIS IS A VERY BLUE STRING" CR ;
: DIFF-CHARS ." {ctrl-3}BUT THIS {ctrl-6}STRING CHANGES" CR ;
BLUE-CHARS      ( DISPLAY BLUE STRING )
DIFF-CHARS      ( DISPLAY RED & GREEN STRING )
{ckey-4}        ( RETURNS TO DEFAULT CURSOR )
```

The following is a graphics example intended to provide the user with a basic understanding of graphics command usage. Refer to the appropriate sections for details of particular words.

first "prepare" the system for hi-res usage. The following example provides a complete step-by-step description of a simple hi-res pattern routine. It may be entered as a defined word, or interactively so that you can see the effect of each word as it is entered.

First we will define some generally useful words for working in the hi-res (or bitmap) graphics mode. Comments (WORDS IN PARENTHESIS, LIKE THESE) need not be entered if you are working interactively at the display. If you set up source screens however, (see section on Editing to do this) I highly recommend including them.

```
( SET UP A WORD TO PUT US INTO BITMAP MODE )
: GRAPHICS  ( --- )
   7 BITMAP            ( SWITCH VIDEO TO BANK 2 UPPER [$E000])
   22 D-SPLIT          ( SET UP SPLIT GRAPHICS SCREEN )
   PAGE                ( CLEAR TEXT SCREEN )
   0 22 D-POSITION ;   ( POSITION CURSOR IN TEXT AREA )

( SET UP A WORD TO GET US BACK TO NORMAL )
: NORMAL  ( --- )
   MED.GRAY BORDER     ( MAKE BORDER GRAY )
   OFF D-SPLIT ;       ( SWITCH US BACK TO NORMAL SCREEN )
```

GRAPHICS and NORMAL allow us to switch back and forth easily between the bitmap area which we want to use and our normal viewing area. Using a "split-screen" method, when we invoke GRAPHICS the screen will be partitioned into two areas: The hi-res graphics area will lie between lines 0 and 21. The text area will lie between lines 22 and 24. Using this method, we can enter commands interactively in the text area and watch them execute in the graphics area.

Okay, we're ready to start, but first type GRAPHICS from the keyboard. On the screen we see a nice clean bitmap area, right? Wrong. There is probably an interesting stripped pattern on the display in place of all our nice FORTH words.The problem is that more SUPER-FORTH words must be used to initialize the bitmap area before we can work with it. Lets create another definition which will initialize the graphics screen properly for us:

```
: B-INIT
   GRAPHICS           ( MAKE SURE WE ARE IN BITMAP AREA )
   RED CYAN B-COLOR-FILL ( INITIALIZE BITMAP COLORS )
   BLUE BORDER        ( SEE THE PRETTY BLUE )
   0 B-FILL           ( CLEAR BITMAP AREA )
   B-DRAW ;           ( SPECIFY DRAW MODE )
```

Now type: B-INIT. This initializes the bitmap area.

I use B-INIT in the graphics examples under the descriptions of each word to insure that you are in bitmap mode before executing. I've chosen RED and CYAN as the bitmap colors, and BLUE for the border, but you can choose your own colors.

B-FILL fills each bitmap character position with a byte value. Try 63 B-FILL. You will see a striped pattern. If you are not in bitmap mode you must be sure to use GRAPHICS first to insure that you are in the correct bitmap area, otherwise B-FILL is likely to fill the first 8000 bytes in memory, and guess what's there? That's right- your SUPER-FORTH system! Performing hi-res graphics commands on your system is a sure way to crash it (crash is a term sometimes used to describe a computer system which acts like it has run into a brick wall at 300 m.p.h.). However, I digress...

B-DRAW specifies to the graphics system that we want to "draw" on the screen (as opposed to "erasing" things from the screen). After entering 0 B-FILL we have a nice, clean CYAN colored screen again. First one more definitions to help clarify the drawing:

```
  ( PUT X & Y COORD. OF THE SCREEN CENTER ON STACK )
  : CENTER  160 100 ;
```

Since we will be working around the center of the screen, CENTER will help us to remember what we are doing.

We will now proceed to plot several curves and lines which will impress your friends, neighbors and make you the envy of your BASIC programmer acquaintances. You can enter the following command lines as a definition and sit back and watch it draw, but I think it's more interesting typing them in interactively while in graphics mode to see the effects of each line:

```
  CENTER B-PLOT          ( SHOULD GET A DOT IN CENTER OF SCREEN )
  CENTER 80 CIRCLE       ( CIRCLE OF RADIUS 80 AROUND CENTER )
  CENTER 20 4 ELLIPSE    ( SMALL HORIZ. ELLIPSE )
  CENTER 80 16 ELLIPSE   ( LARGE HORIZ. ELLIPSE )
  CENTER 20 60 ELLIPSE   ( LARGE VERTICAL ELLIPSE )
  0 199  M-ORIGIN        ( DEFINE CENTER OF ARC AT LOWER LEFT )
  152 76 270 360 ARC     ( ARCS "HOLDING UP" CIRCLE )
  319 199 M-ORIGIN       ( DEFINE CENTER OF ARC AT LOWER RIGHT )
  152 76 180 270 ARC
  0 0 B-PLOT             ( START LINE AT UPPER LEFT CORNER )
  CENTER B-LINE          ( LINE TO CENTER )
  319 0 B-LINE           ( LINE TO UPPER RIGHT CORNER )
```

Just for fun, let's get a sprite involved in this. We have two sprites defined for the DEMO program, so we can just borrow one of them for this example:

```
  NORMAL                 ( MUST BE NORMAL TO USE DISK )
  91 LOAD                ( LOADS IN SPRITE SCREEN )
  GRAPHICS               ( RETURNS TO GRAPHICS SCREEN )
  49152 DRAGON1          ( MOVE SPRITE TO BANK 3, SPRITE 0 )
  S1 0 S-POINTER         ( SPECIFY SPRITE POINTER 0 FOR SPRITE S1 )
  ON S-ENABLE            ( ENABLE THE SPRITE )
  170 70 S-POSITION      ( MAKE IT VISIBLE )
  ON S-XEXP              ( EXPAND THE CRITTER HORIZONTALLY )
  YELLOW S-COLOR         ( TURN IT YELLOW )
  ON S-MULTI             ( A MULTI-COLORED DRAGON! )
```

By experimenting with repetitively calling the graphics words, while changing the values, interesting patterns may be discovered. Here is one such pattern. It is simply a series of concentric circles being drawn:

```
: SHIELD
   60 1 DO
    160 100 I CIRCLE
   LOOP ;

B-INIT
SHIELD
NORMAL
```

Well, thats the basic idea. I recommend just experimenting around, interactively changing things for a while to get the hang of the various graphics words. Also, go through the examples given with each graphics word definition. If you're curious you might want to take a look at the screens that implement DEMO and see if you can follow how the animation is done (if you can follow the recursive algorithm you can explain it to me). Refer to Appendix I for other program examples.

5.7.1 'BANK : Get Address of 16K Bank

(--- ADDR)

Leaves the address of the 16k bank which the VIC-II chip has access to (this will be one of the following : 0, 16384, 32768, 49152). When the system first comes up the VIC-II chip is set to look at the lowest 16k of RAM (0-16383).

Example: 'BANK . <cr> will print out the current address of the 16k bank.

5.7.2 'BITMAP : Get Address of Bitmap Area

(--- ADDR)

Leaves the address of the 8k Bitmap area which is used when the VIC-II chip is put into high resolution graphics (bitmap) mode (this will be one of the following : 0, 8192, 16384, 24576, 32768, 40960, 49152, 57344). Also see BITMAP.

Example: 'BITMAP . <cr> will print out the current address of the high resolution bitmap area.

5.7.3 'CHARBASE : Get Address of Character Memory

(--- ADDR)

Leaves the base address of the Character Memory area which the VIC-II chip looks at to get its character set information. There are 32 possible areas where the character set may reside, from 0 to 63488 in 2k (2048 byte) intervals. Initially, the character memory is set to 4096 (or 6144 depending on which character set is being used).

Example: 'CHARBASE . <cr> will print out the current address of the character memory area.

5.7.4 'SCREEN : Get Address of Screen Memory

(--- ADDR)

Leaves the base address of the Screen Memory area which the VIC-II chip looks at to perform character mappings for output to the screen in character mode, or gets color information from in bitmap (hi-res) mode. There are 64 possible areas in which Screen Memory may reside, from 0 to 64512 in 1k (1024 byte) intervals.

Example: 'SCREEN . <cr> will print out the current address of the screen memory area.

5.7.5 ARC : Plot A Hi-res ARC

(HR VR START END ---)

This word invokes M-PLOT to draw an arc (part of an ellipse) on a hi-res screen. The arc is defined in terms of the ellipse which it is a part of: the center coordinates are taken from the values of M-X and M-Y. HR is the horizontal radius, VR is the vertical radius. START,END are the start and end points of the arc in degrees where 0, 90, 180, 270 degrees would point east, south, west and north in compass points.

Example:

```
B-INIT                ( SEE GRAPHICS INTRO FOR DEFINITION)
200 110 M-ORIGIN      ( DEFINES CENTER OF ARC )
90 90 150 300 ARC     ( DRAW ARC )
NORMAL                ( SEE INTRO )
```

This example draws a 150 degree arc from 150 to 300 degrees of an ellipse whose X and Y radii are both 90 and whose center is at 200,110.

5.7.6 B-CLINE : Plot A Color Line On the Bitmap

(X-NEW Y-NEW ---)

Plots a color line in bitmap mode. The line is plotted from the last plotted point to the coordinates specified by X-NEW and Y-NEW, where X-NEW ranges from 0 to 319 and Y-NEW ranges from 0 to 199.

The color of the line is set by the Turtle Graphics word, PENCOLOR. Draw/erase mode determines whether the points are turned on or off (line drawn or erased). To erase a drawn line the erasing line must follow the same path as the original drawn line.

5.7.7 B-CPLOT : Plot A Color Point On the Bitmap

(X Y ---)

Plots a point in the bitmap area. The color of the point is set by the Turtle Graphics word PENCOLOR. See B-PLOT for details.

5.7.8 B-COLOR : Select Bitmap Character Colors

(POS HI-COLOR LO-COLOR ---)

Set the high and low nibble colors for a particular character position in the bitmap character color memory (screen memory) area. POS is the screen position the colors will control (0-999), HI-COLOR is the color which will be used for bits which are on (1) in the character position, LO-COLOR is the color which will be used for bits which are off (0) in the character position.

Example:

```
B-INIT              ( SEE GRAPHICS INTRO FOR DEFINITION )
40 CYAN RED B-COLOR
NORMAL              ( SEE INTRO )
```

This example will set the first character position on line 2 (character 40) so that bits in that position that are off will be red and bits which are on will be CYAN.

5.7.9 B-COLOR-FILL : Fill Bitmap Color Area

(HI-COLOR LO-COLOR ---)

Sets the Screen Memory area (which is used to determine color usage in bitmap mode) to the given colors for all 1000 background locations. The upper 4-bits of all Screen Memory locations is set to the value of HI-COLOR. The lower 4-bits of all Screen Memory locations is set to the value of LO-COLOR. A particular bits screen color will be set to HI-COLOR if the bit is turned on (a 1 value) or LO-COLOR if the bit is turned off (a 0 value).

Example:

```
B-INIT              ( SEE GRAPHICS INTRO FOR DEFINITION )
160 100 40 CIRCLE
NORMAL              ( SEE INTRO )
```

This example will fill the bitmap screen with red where bits are on and yellow where bits are off.

5.7.10 B-DRAW : Set the System to Draw Mode

(---)

Stores a 1 into B-PEN, designating draw mode (turn on bits when plotting). B-PEN is examined by B-PLOT to determine whether the point to be plotted should be turned on (draw mode) or turned off (erase mode). B-DRAW need only be invoked once for each set of points which will be drawn. It need only be invoked again after B-ERASE has been invoked.

5.7.11 B-ERASE : Set the System to Erase Mode

(---)

Stores a 0 into B-PEN, designating erase mode (turn off bits when plotting). B-PEN is examined by B-PLOT to determine whether the point to be plotted should be turned on (draw mode) or turned off (erase mode). B-ERASE need only be invoked once for each set of points which will be erased. It need only be invoked again after B-DRAW has been invoked.

5.7.12 B-FILL : Fill Bitmap with Byte Pattern

(VALUE ---)

Fills the complete Bitmap area, character by character, with the byte VALUE. Any value can be used, but generally it is expected that either 0 (which will set the bitmap area to all zeroes) or 255 (which will set the bitmap area to all ones) will be used. Other values will cause the bitmap to assume various striped patterns.

Note : Even with the Bitmap completely cleared or set there may appear to be random color patterns in the area unless B-COLOR-FILL is used to specify the colors of character positions within the area.

WARNING: The bitmap area must have been set up using BITMAP and bitmap graphics should be turned on prior to using this word.

Example:

```
    B-INIT           ( SEE GRAPHICS INTO FOR DEFINTION )
    255 B-FILL       ( TURNS ON ALL BITS )
    15 B-FILL        ( EVEN STRIPED PATTERN )
    1 B-FILL         ( UNEVEN STRIPED PATTERN )
    0 B-FILL         ( TURNS OFF ALL BITS )
    NORMAL           ( SEE INTRO )
```

5.7.13 B-GRAPHICS : Turn Bitmap Graphics On/Off

(FLAG ---)

If FLAG is ON (1) the C64 is set to Bitmap Graphics mode. The Bitmap area should be set up by using BITMAP prior to attempting to actually change any values in the area. If FLAG is OFF (0) the C64 is set to Normal Graphics mode. BITMAP will probably have to be used to reset to area 0 where the screen normally resides.

Example:

```
7 BITMAP  ON B-GRAPHICS
```

will put the C64 into bitmap graphics mode and set the bitmap area to area 7 (57344 [$E000]).

```
0 BITMAP OFF B-GRAPHICS
```

will reset the C64 to its normal screen mode.

5.7.14 B-LINE : Plot A Line On the Bitmap

(X-NEW Y-NEW ---)

This word is used to plot a line in bitmap mode. The line is plotted starting from the last plotted point (set by B-PLOT or words such as B-LINE, ARC, CIRCLE or ELLIPSE, which use B-PLOT) to the coordinates specified by X-NEW and Y-NEW, where X-NEW ranges from 0 to 319 and Y-NEW ranges from 0 to 199. Draw/erase mode determines whether the points are turned on or off (line drawn or erased). To erase a drawn line the erasing line must follow the same path as the original drawn line.

WARNING: The bitmap area must have been set up using BITMAP and bitmap graphics should be turned on prior to using this word.

Example: Draw a line from the origin to the center of the screen and then erase it:

```
B-INIT          ( SEE GRAPHICS INTRO FOR DEFINITION )
B-DRAW          ( INVOKE DRAW MODE )
0 0 B-PLOT      ( PLOT ORIGIN POINT )
160 100 B-LINE  ( DRAW A LINE TO THE CENTER )
B-ERASE         ( INVOKE ERASE MODE )
0 0 B-PLOT      ( ERASE POINT AT THE ORIGIN )
160 100 B-LINE  ( ERASE LINE TO THE CENTER )
NORMAL          ( SEE INTRO )
```

5.7.15 B-PEN : Draw/Erase Mode Variable

(--- ADDR)

This variable is used by the graphics system to determine the mode of hi-res drawing to be invoked by B-PLOT. If B-PEN is ON (1) DRAW mode will be invoked. If B-PEN is OFF (0) ERASE mode will be invoked.

B-PEN is not normally directly invoked by the user since B-DRAW and B-ERASE have been provided to set the value of B-PEN

Example:

```
: B-DRAW 1 B-PEN ! ;
```

is the definition of the SUPER FORTH 64 word which puts the system into DRAW mode.

5.7.16 B-PLOT : Plot a Point In the Bitmap

(X Y ---)

This word is used to plot a point in the Bitmap area. If B-PEN is ON (see B-DRAW) the bit for the point is set to a 1 and the color of the screen becomes the HI-COLOR for the character position which the point resides in. If B-PEN is OFF (see B-ERASE) the bit is set to a 0 and LO-COLOR is used to color the screen at that point.

X and Y are the coordinates of the point on a 320 by 200 grid where point 0,0 is at the uppermost left-hand corner of the screen. X ranges from 0 to 319. Y ranges from 0 to 199. Values for X and Y are returned by invoking words B-X and B-Y. B-PLOT performs a bounds check on X and Y. B-X and B-Y are always updated, but the point is only plotted if X falls within 0 to 319 and Y falls within 0 to 199.

If MULTI-COLOR mode has been set then every two points along the X-axis will be used to determine which of four colors the two points will be (see the Commodore Reference Manual for determining colors in Multi-color mode).

The kernel is swapped out when plotting a point. Thus, bitmap 7 ($E000 - $FFFF can be utilized for bitmap graphics. This area does not interfere with the dictionary area and is recommended for use.

Note: If bitmap area 7 is to be used, the default screen area ($C400) may interfere with the disk buffer area. Therefore, it is recommended that The top of memory be moved below the screen area. The following code will effect the change:

```
HEX C380 ' LIMIT ! CHANGE
```

WARNING: The bitmap area must have been set up using BITMAP and bitmap graphics should be turned on prior to using this word.

Example:

```
( TURN ON CENTER AND ORIGIN POINTS )
B-INIT                   ( SEE GRAPHICS INTRO FOR DEFINITION )
B-DRAW                   ( SET DRAW MODE )
160 100 B-PLOT           ( PLOT POINT AT CENTER OF SCREEN )
0 0 B-PLOT               ( PLOT "ORIGIN"- UPPER LEFT CORNER POINT )
( TURN OFF CENTER AND ORIGIN POINTS )
B-ERASE                  ( SET ERASE MODE )
160 100 B-PLOT           ( PLOT POINT AT CENTER OF SCREEN )
0 0 B-PLOT               ( PLOT "ORIGIN" )
NORMAL                   ( SEE INTRO )
```

5.7.17 B-X : Return X Coordinate Value

(--- VALUE)

This word returns the value of the X coordinate which is passed to B-PLOT whenever a point is plotted. In this way other routines can perform calculations based on the last plotted point.

5.7.18 B-Y : Return Y Coordinate Value

(--- VALUE)

This word returns the value of the Y coordinate which is passed to B-PLOT whenever a point is plotted. In this way other routines can perform calculations based on the last plotted point.

5.7.19 BANK : Set VIC-II Bank

(BANK# ---)

Sets up which 16k bank of memory the VIC-II chip will look at for its Bitmap, Screen and Character Memory areas. Bank addresses are as follows:

Bank	Decimal	Hex
0	0	0000
1	16384	4000
2	32768	8000
3	49152	C000

This word is used by other graphics routines to set up the video bank.

Example:

```
2 BANK sets the 16k area to 32768
0 BANK restores the normal video setting
```

5.7.20 BITMAP : Set BITMAP Area

(BITMAP-AREA# ---)

Sets up the proper registers (CIA Bank and VIC-II Control) to put the Bitmap in one of 8 areas as follows:

Bitmap	Decimal	Hex
0	0	0000
1	8192	2000
2	16384	4000
3	24576	6000
4	32768	8000
5	40960	A000
6	49152	C000
7	57344	E000

Example:

```
23 D-SPLIT
7 BITMAP
0 D-SPLIT
```

will split the screen, set the bitmap area to 57344 and restore the normal screen.

5.7.21 BORDER : Set Border Color

(COLOR ---)

Word used to set the screen border color. COLOR is one of the C64 colors as defined by the Commodore User's Manual (see Color Constants).

Example: RED BORDER turns the border red.

5.7.22 BKGND : Set A Background Register Color

([REG#] COLOR ---)

Word used to set the screen background register color. COLOR is a C64 color as defined by the Commodore User's Manual (see Color Constants). REG# is 0 to 3 for the particular background register. If a single parameter is on the stack, background register 0 is assumed and the parameter is used as the COLOR value.

Example: PURPLE BKGND will set the background to purple.

5.7.23 CHARBASE : Set Character Base Area

(CHARBASE-AREA# ---)

Sets the base address of the Character Memory area which the VIC-II chip looks at to get its character set information. There are 32 possible areas where the character set may reside, from 0 to 63488 in 2k (2048 byte) intervals. Initially, the character memory is set to 4096 (or 6144 depending on which character set is being used).

CHARBASE-AREA# may range from 0 to 7 to span the 16k area accessible from a particular bank (see BANK).

5.7.24 CIRCLE : Draw A Hi-res Circle

(X Y R ---)

This word invokes B-PLOT to draw a true circle of radius R and center at X,Y. Circle is really a special case of ELLIPSE.

WARNING: The bitmap area must have been set up using BITMAP and bitmap graphics should be turned on prior to using this word.

Example:

```
B-INIT            ( SEE GRAPHICS INTRO FOR DEFINITION )
B-DRAW 160 100 60 CIRCLE
NORMAL            ( SEE INTRO )
```

draws a circle of radius 60 around the center of the screen.

5.7.25 Color Constants

Each color is defined as a constant to ease using the colors. These can be used in various commands requiring a color constant as input. The following colors and their constant values (see Commodore User's Manual) are defined:

BLACK	0	PURPLE	4	ORANGE	8	MED.GRAY	12
WHITE	1	GREEN	5	BROWN	9	LT.GREEN	13
RED	2	BLUE	6	LT.RED	10	LT.BLUE	14
CYAN	3	YELLOW	7	DK.GRAY	11	LT.GRAY	15

Either the constant name or its value may be used. Since naming the constant causes its value to be compiled into the definition, execution speed is not affected by using one or the other.

Example:

```
: SINIT  PURPLE BKGND  7 BORDER ;
```

would define a word called SINIT which when invoked would set the screen background to purple and the screen border to yellow.

5.7.26 COLOR-MEM : Address of Color Memory Area

(--- ADDR)

A constant which leaves the address of the color memory area on the stack.

Example: COLOR-MEM U. will print 55296.

5.7.27 ELLIPSE : Plot A Hi-res Ellipse

(X Y HR VR ---)

This word invokes ARC to plot an elliptical shape on a hi-res screen. X and Y define the center of the ellipse and are stored in C-X and C-Y before calling ARC. HR and VR define the horizontal and vertical radii respectively.

WARNING: The bitmap area must have been set up using BITMAP and bitmap graphics should be turned on prior to using this word.

5.7.28 B-MFLAG : Turn Mirror Function On/Off

(--- ADDR)

Variable containing the flag which determines whether M-PLOT should mirror the point which it is plotting. If B-MFLAG contains a TRUE (1) value, mirroring is on. If FALSE (0), mirroring is off.

Example:

```
ON B-MFLAG !
```

Turns on the mirror function.

5.7.29 M-ORIGIN : Set the Mirror Origin

(X Y ---)

Sets M-X and M-Y to return X and Y respectively when called by mirroring routines.

5.7.30 M-PLOT : Plot A Four Point Mirror Image

(X Y ---)

If B-MFLAG is true, then X and Y are offsets from a center coordinate defined by the values of M-X and M-Y. The following four points are plotted on the bitmap area:

M-X + X	M-Y - Y
M-X - X	M-Y + Y
M-X - X	M-Y - Y
M-X + X	M-Y + Y

In this way, fast mirror images may be plotted.

If B-MFLAG is false, then R-PLOT is called directly. ELLIPSE uses M-PLOT to plot four quadrants simultaneously.

Example:

```
B-INIT              ( SEE GRAPHICS INTRO FOR DEFINITION )
ON B-MFLAG !        ( SET MIRROR MODE )
160 100 M-ORIGIN
20 40 M-PLOT        ( MIRROR A POINT )
40 20 M-PLOT        ( DO ANOTHER )
: TEST 100 0 DO I I M-PLOT LOOP ;
TEST                ( "X" MARKS THE SPOT )
OFF B-MFLAG !       ( TURN OFF MIRROR )
NORMAL              ( SEE INTRO )
```

5.7.31 M-X : X Coordinate of Mirror Center

(--- X-COORD)

Returns the value of the X coordinate of the point which is used by M-PLOT as a relative origin for plotting four mirrored points around it (see M-ORIGIN).

5.7.32 M-Y : Y Coordinate of Mirror Center

(--- Y-COORD)

Returns the value of the Y coordinate of the point which is used by M-PLOT as a relative origin for plotting four mirrored points around it (see M-ORIGIN).

5.7.33 MULTI-COLOR : Set/Clear Multi-color Mode

(FLAG ---)

Sets/clears multi-color graphics mode based on the value of flag. An ON (1) value sets multi-color. An OFF (0) value clears multi-color.

Example:

```
     ON MULTI-COLOR
Turns multi-color mode on.
```

5.7.34 R-PLOT : Plot A Point Relative To Center

(X Y ---)

Plots a point relative to the center coordinates, M-X,M-Y. M-X is added to X and M-Y is added to Y before calling B-PLOT to plot the point. This routine is used by M-PLOT in order to implement plotting four points relative to a center point.

Example:

```
B-INIT          ( SEE GRAPHICS INTRO FOR DEFINITION )
160 100 M-ORIGIN
20 40 R-PLOT
NORMAL          ( SEE INTRO )
```

5.7.35 S1, S2, S3, S4, S5, S6, S7, S8 : Set Active Sprite

Sets the active sprite as one of eight sprites.

Example:

```
S5 280 180 S-POSITION
ON S-ENABLE
```

turns on sprite 5.

5.7.36 S-B-COLLISION : Get Spr-Bkgnd Collision Reg.

(--- VALUE)

Returns the value of the Sprite to Background Collision register. This value can be checked to determine if any sprites have collided with the background. Use of this word automatically clears the register. Refer to Commodore Programmers Reference guide p.144,180 for usage of the collision registers.

Example:

```
     S-B-COLLISION .
```

displays the value of the register and clears it.

5.7.37 S-COLOR : Set Sprite Color

(COLOR ---)

Set the color value of the active sprite to COLOR.

Example:

```
     S1 280 180 S-POSITION
     ON S-ENABLE
     RED S-COLOR
```

enables sprite 1 and makes it appear red.

5.7.38 S-DEF : Sprite Definition Structure

(---) [63 sprite byte values]
(ADDR ---)

This structure is provided as an aid in handling sprite definitions. The 63 bytes of sprite data follow the definition of the word. These are compiled into the dictionary. At execution time the 63 bytes are moved to the area located at ADDR (16-bit address).

Example:

```
  HEX
  S-DEF DRAGON1
    01  01  00       00  81  80
    00  C1  C0       00  E1  C0
    00  F1  E0       00  F9  80
    00  FF  00       00  FF  FF
    00  7F  FE       00  3F  F8
    00  3F  E0       00  3F  80
    00  3C  00       00  78  00
    00  F0  00       01  E0  00
    06  60  00       1C  70  00
    38  38  00       70  1C  00
    A8  2A  00
  8000 DRAGON1     DECIMAL
```

This sprite example, taken from the DEMO program, compiles the data for a sprite named DRAGON1 into the dictionary. When DRAGON1 is invoked the sprite data is moved to 32768 ($8000), the 0 sprite area in BANK 2.

The data could be entered in decimal, but HEX notation is a closer representation of the on/off bit patterns which make up a sprite. If your screen listing editing format allows at least 23 lines (see Editor section) the sprite data could be entered in binary. This may give a more visual representation of the sprite itself.

Example:

```
    40 ' C/L !       ( CHANGE FORMAT TO 40 X 25 )
    100 LIST
    SCREEN #100
     0) ( BINARY DRAGON )
     1) 2 BASE !     ( SET BASE TO BINARY )
     2) S-DEF DRAGON3
     3) 00000001 00000001 00000000
     4) 00000000 10000001 10000000
     5) 00000000 11000001 11000000
     6) 00000000 11100001 11000000
     7) 00000000 11110001 11100000
     8) 00000000 11111001 10000000
     9) 00000000 11111111 00000000
    10) 00000000 11111111 11111111
    11) 00000000 01111111 11111110
    12) 00000000 00111111 11111000
    13) 00000000 00111111 11100000
    14) 00000000 00111111 10000000
    15) 00000000 00111100 00000000
    16) 00000000 01111000 00000000
    17) 00000000 11110000 00000000
    18) 00000001 11100000 00000000
    19) 00000110 01100000 00000000
    20) 00011100 01110000 00000000
    21) 00111000 00111000 00000000
    22) 01110000 00011100 00000000
    23) 10101000 00101010 00000000
    24) DECIMAL
```

Since S-DEF requires 63 numbers to be entered in the input stream it is expected that usage of S-DEF will be within an editing screen.

5.7.39 S-EDITOR : Sprite Editor

(---) spritename

This word is provided as a simple, single color sprite editor. When "S-EDITOR spritename" is typed, a dictionary entry named "spritename" is created, and a grid of dots appears on the screen.

To create a sprite, use the cursor movement keys and the space bar to change the dots to blanks. A carriage return signals SUPER FORTH that you are finished. The image drawn in the grid is transferred both to the dictionary area defined by "spritename", and to sprite pointer area 13 (location 832 on the first video bank).

The actual sprite will appear in blue next to the grid. The sprite can be turned off by typing OFF S-ENABLE.

Example:

```
S-EDITOR PICTURE ( AN EASEL IS DISPLAYED )
( DRAW SPRITE THEN CR )
B-INIT          ( SEE GRAPHICS INTRO FOR DEFINITION )
( PLACE THE SPRITE, "PICTURE", INTO SPRITE AREA 0 OF BANK 2 )
49152 PICTURE
NORMAL          ( SEE INTRO )
OFF S-ENABLE    ( TURN SPRITE OFF )
```

5.7.40 S-ENABLE : Turn Sprite On/Off

(FLAG ---)

If FLAG is ON (1), turn active sprite on. If FLAG is OFF (0) turn active sprite off.

Example:

```
S1 280 180 S-POSITION
ON S-ENABLE
OFF S-ENABLE
```

Positions, enables and disables sprite 1.

5.7.41 S-FSET : Set/Clear Bit in Sprite Register on Flag

(FLAG ADDR ---)

Sets or clears a bit in register at ADDR based on the active sprite number. If FLAG is TRUE (1) the sprite bit is set. If FLAG is FALSE (0) the sprite bit is cleared. This is a utility routine used by the system to handle sprite registers in the VIC-II chip.

5.7.42 S-MULTI : Set/Clear Multi-Color Mode for A Sprite

(FLAG ---)

Sets or clears the bit for the active sprite in the Sprite Multi-color Mode register of the VIC-II chip. If FLAG is ON (1) the sprite bit is set. If FLAG is OFF (0) the sprite bit is cleared. S-MULTIR should be used to set the sprite multi color registers before using this word.

Example:

```
ON S-MULTI
```

turns on multi-color mode for the active sprite.

5.7.43 S-MULTIR : Set Multi-Color Sprite Register Color

(COLOR REG# ---)

Sets one of the two Sprite Multi-color register to the given COLOR. REG# is 0 or 1.

Example:

```
RED 0 S-MULTIR
```

sets multi-color register 0 for the active sprite to red.

5.7.44 S-POINTER : Set Sprite Pointer Number

(SPRITE-ADDR ---)

Sets the SPRITE-ADDR for the active sprite. Sprite addresses range from 0 to 255. Each sprite address covers a 64 byte range, therefore the 256 sprite addresses will allow sprites to be defined anywhere in the 16k bank which the VIC-II chip is looking at.

Example:

```
0 S-POINTER
```

will set the sprite pointer for the active sprite to location 0 relative to the start of the current 16k bank.

5.7.45 S-POSITION : Set Sprite Position

(X Y ---)

Sets the position of the active sprite to the specified X,Y coordinates.

Example:

```
S1  ON S-ENABLE
100 100 S-POSITION
```

will position the active sprite to sprite coordinates 100,100.

5.7.46 S-PRIORITY : Set Sprite-Background Priority

(FLAG ---)

Sets priority of active sprite according to FLAG. If FLAG is FALSE (0) the active sprite takes priority over the background. If FLAG is TRUE (1) the background takes priority over the sprite.

Example:

```
ON S-PRIORITY
```

gives the active sprite priority over the background.

5.7.47 S-S-Collision : Get Spr-Spr Collision Reg.

(--- VALUE)

Returns the value of the Sprite to Sprite Collision register. This value can be examined to determine which, if any, sprites have collided. Use of this word automatically clears the register.

Example:

```
S-S-COLLISION .
```

prints the value of the register and clears it.

5.7.48 S-XEXP : Expand Sprite In X-Direction

(FLAG ---)

Sets up the VIC-II chip to set the active sprite to expand/normal in the X direction depending of the given FLAG. If FLAG is TRUE (1) the sprite will be expanded. If FLAG is FALSE (0) the sprite will be unexpanded.

Example:

```
ON S-XEXP
```

expands the active sprite in the X direction.

5.7.49 S-YEXP : Expand Sprite In Y-Direction

(FLAG ---)

Sets up the VIC-II chip to set the active sprite to expand/normal in the Y direction depending of the given FLAG. If FLAG is TRUE (1) the sprite will be expanded. If FLAG is FALSE (0) the sprite will be unexpanded.

Example:

```
ON S-YEXP
```

expands the active sprite in the Y direction.

5.7.50 SCREEN : Set Screen Display Area

```
( SCREEN-AREA# --- )
```

Sets the Screen Memory area which the VIC-II chip looks at to perform character mappings for output to the screen in character mode, or gets color information from in bitmap (hi-res) mode.

SCREEN-AREA# may range from 0 to 15, putting the display screen in one of the 16 1k (1024 byte) areas accessible by a particular 16k bank. Initially, the screen is located at area 1 (1024).

Note: If the screen area is changed, the screen editor must be notified of the change. The following accomplishes this:

```
NEW-ADDR  256 / 648 C!
```

5.8 Turtle Graphics

These words follow the turtle graphics definitions of the Logo language. The "turtle" in this system is a "virtual turtle", that is, the turtle is not actually displayed, but the turtle words move and change the direction of the invisible turtle. The Turtle Graphics words may be intermixed with other graphics words, such as B-LINE, B-CIRCLE or the various sprite control words.

The examples for each word should be followed to get the basic idea of how to move the turtle. A more elaborate example of how to build turtle word definitions follows. This example demonstates how to create a complex design based on simple definitions.

First, let us create a definition for drawing a hexagon. A hexagon may be created by drawing six equal sides, turning 60 degrees after drawing a side. Thus, the definition:

```
: HEXSIDE  40 FORWARD 60 RIGHT ;
```

will draw one side and prepare for the next. Executing the following:

```
DRAW HEXSIDE HEXSIDE HEXSIDE HEXSIDE HEXSIDE HEXSIDE
```

will verify that HEXSIDE can be used to draw a hexagon. The definition for hexagon follows directly from HEXSIDE:

```
    : HEXAGON  6 0 DO  HEXSIDE  LOOP ;
```

and DRAW HEXAGON can be used to verify HEXAGON.

Next, we can draw a geometric figure based on rotated hexagons. If we leave the number of degrees to rotate as a parameter, we can create many figures from the same definition:

```
    : HEXFIGURE  ( #DEGREES --- )
        360 0 DO         ( ROTATE FOR 360 DEGREES )
         HEXAGON         ( DRAW A HEXAGON )
         DUP RIGHT       ( ROTATE BY #DEGREES )
        DUP +LOOP DROP ;
```

Now try the following figures:

```
    DRAW 120 HEXFIGURE
    DRAW 60 HEXFIGURE
    DRAW 30 HEXFIGURE
    DRAW 15 HEXFIGURE
    DRAW 3 HEXFIGURE
```

The same procedure may be used with other types of shapes to create different geometric figures.

5.8.1 BACK : Move Turtle Backward

(N ---)

The turtle is moved N units in the direction opposite of HEADING, drawing if the pen is down. May be abbreviated BK.

Example:

```
        DRAW              ( INITIALIZE TURTLE SCREEN )
        40 BACK           ( DRAW LINE SOUTH )
        45 LEFT           ( TURN TURTLE TO FACE NORTHWEST )
        40 BACK           ( DRAW LINE SOUTHEAST )
```

5.8.2 BACKGROUND : Set Background Color

(COLOR ---)

Changes the turtle screen background to COLOR. Also changes any drawing on screen to present turtle pen color.

Example:

```
DRAW                 ( INITIALIZE TURTLE SCREEN )
160 100 60 CIRCLE ( DRAW A CIRCLE )
YELLOW PENCOLOR    ( CHANGE THE PEN TO YELLOW )
120 FORWARD         ( DRAW YELLOW LINE )
PURPLE PENCOLOR    ( CHANGE PEN TO PURPLE )
YELLOW BACKGROUND ( CHANGE BACKGND YELLOW & DRAWING PURPLE )
```

5.8.3 BG : Set Background Color

(COLOR ---)

Abbreviation for BACKGROUND. May be abbreviated BG.

5.8.4 BK : Move Backward

(N ---)

Abbreviation for BACK.

5.8.5 CLEARSCREEN : Clear the Graphics Area

(---)

Erases anything drawn on the turtle screen. Leaves turtle in its last position. May be abbreviated CS.

Example:

```
DRAW               ( INITIALIZE TURTLE SCREEN )
160 100 50 CIRCLE ( DRAW A CIRCLE )
CLEARSCREEN       ( ERASES THE SCREEN )
20 BACK           ( DRAW A LINE FROM LAST TURTLE POSITION )
```

5.8.6 CS : Clear the Graphics Area

(---)

Abbreviation for CLEARSCREEN.

5.8.7 DRAW : Initialize Turtle Screen

(---)

Sets up the C64 to perform turtle graphics. DRAW performs the following functions:

- Splits the screen into hi-res/text
- Clears the bitmap area
- Sets the turtle pen red
- Sets the hi-res background color to cyan
- Moves the turtle to its "home" position
- Sets the turtle pen to "draw" mode (down)

5.8.8 FD : Move Forward

(N ---)

Abbreviation for FORWARD.

5.8.9 FORWARD : Move Turtle Forward

(N ---)

The turtle is moved N units in the direction of HEADING, drawing if the pen is down. May be abbreviated FD.

Example:

```
DRAW                ( INITIALIZE TURTLE SCREEN )
40 FORWARD          ( DRAW LINE 40 UNITS NORTH )
45 RIGHT            ( SET HEADING NORTHEAST )
40 FORWARD          ( DRAW LINE 40 UNITS NORTHEAST )
```

5.8.10 FS : Set Graphics Screen

(---)

Abbreviation for FULLSCREEN.

5.8.11 FULLSCREEN : Set Graphics Screen

(---)

Enter turtle full screen mode- the complete screen is used for turtle graphics. Does not affect any drawing on turtle screen. May be abbreviated FS.

Example:

```
DRAW                ( INITIALIZE TURTLE SCREEN )
134 BACK            ( DRAW LINE TO BOTTOM OF SCREEN )
FULLSCREEN          ( ENTER FULL SCREEN MODE )
```

5.8.12 HEADING : Heading Variable

(--- ADDR)

Variable which contains the current heading of the turtle (direction in which it is pointed) in degrees. Headings of 0, 90, 180 and 270 represent north, east, south and west, respectively. HEADING is automatically updated by rotation commands- RT, RIGHT, LT and LEFT.

Example:

```
DRAW               ( SETS UP TURTLE SCREEN )
HEADING ?          ( DISPLAYS "0", INITIAL HEADING )
20 RT HEADING ? ( CURRENT HEADING IS 20 )
 5 RT HEADING ? ( CURRENT HEADING IS 25 )
```

5.8.13 HOME : Position To Center of Screen

(---)

Puts turtle in its "home" position (in the center of the hi-res screen).

Example:

```
DRAW               ( INITIALIZE TURTLE SCREEN )
160 100 60 CIRCLE ( DRAW A CIRCLE )
HOME               ( PUT TURTLE BACK IN CENTER )
60 BACK            ( DRAW LINE TO CIRCLE FROM CENTER )
```

5.8.14 LEFT : Turn Left

(N ---)

Causes turtle to turn counterclockwise by N degrees. May be abbreviated LT.

Example:

```
DRAW               ( SET UP TURTLE SCREEN )
20 FORWARD         ( MOVE TURTLE FORWARD )
90 LEFT            ( TURN LEFT 90 DEGREES )
20 FORWARD         ( MOVE TURTLE FORWARD )
```

5.8.15 LT : Turn Left

(N ---)

Abbreviation for LEFT.

5.8.16 TS : Set Text Screen

(---)

Abbreviation for NODRAW.

5.8.17 TEXTSCREEN : Set Text Screen

(---)

Exit turtle screen- enter text mode. May be abbreviated ND.

Example:

```
DRAW               ( INITIALIZE TURTLE SCREEN )
80 FORWARD         ( DRAW A LINE )
TEXTSCREEN         ( RETURN TO TEXT SCREEN )
```

5.8.18 PC : Set Color of Pen

(COLOR ---)

Abbreviation for PENCOLOR.

5.8.19 PENCOLOR : Set Color of Pen

(COLOR ---)

Set the color of the turtle's "pen" to the value of COLOR. DRAW sets the pen color to RED. May be abbreviated PC.

Example:

```
DRAW               ( INITIALIZE TURTLE SCREEN )
90 RIGHT 40 FORWARD ( DRAW A RED LINE )
YELLOW PENCOLOR ( CHANGE THE PEN TO YELLOW )
40 FORWARD         ( DRAW A YELLOW LINE )
```

5.8.20 PENFLG : Pen Variable

(--- ADDR)

Variable used to determine whether the pen is up (turtle does not draw) or down (turtle draws foreground color). Set by PENUP and PENDOWN.

5.8.21 PD : Set Pen Down

(---)

Abbreviation for PENDOWN.

5.8.22 PENDOWN : Set Pen Down

(---)

Sets the turtle pen to draw when the turtle is moved. May be abbreviated PD.

Example:

```
DRAW              ( SETS UP TURTLE SCREEN )
20 FORWARD        ( DRAW LINE )
PENUP             ( SET PEN TO UP POSITION )
10 FORWARD        ( MOVE TURTLE BUT DON'T DRAW )
PENDOWN           ( SET PEN TO DOWN POSITION )
10 FORWARD        ( DRAW WHILE MOVING TURTLE )
```

5.8.23 PENUP : Set Pen Up

(---)

Sets the turtle pen to not draw when the turtle is moved. May be abbreviated PU.

Example:

```
DRAW              ( SETS UP TURTLE SCREEN )
90 RIGHT          ( TURN EAST )
20 FORWARD        ( DRAW LINE )
PENUP             ( SET PEN TO UP POSITION )
10 FORWARD        ( MOVE TURTLE BUT DON'T DRAW )
PENDOWN           ( SET PEN TO DOWN POSITION )
10 FORWARD        ( DRAW WHILE MOVING TURTLE )
```

5.8.24 PU : Set Pen Up

(---)

Abbreviation for PENUP.

5.8.25 RIGHT : Turn Right

(N ---)

Causes turtle to turn clockwise by N degrees. May be abbreviated RT.

```
DRAW              ( SET UP TURTLE SCREEN )
40 FORWARD        ( MOVE TURTLE FORWARD )
60 RIGHT          ( TURN RIGHT 60 DEGREES )
40 FORWARD        ( MOVE FORWARD )
```

5.8.26 RT : Turn Right

(N ---)

Abbreviation for RIGHT.

5.8.27 SETH : Set Turtle Heading

(ANGLE ---)

Abbreviation for SETHEADING.

5.8.28 SETHEADING : Set Turtle Heading

(ANGLE ---)

The ANGLE entered becomes the new HEADING of the turtle. May be abbreviated SETH.

Example:

```
DRAW              ( SET UP TURTLE SCREEN )
30 SETHEADING     ( POINT THE TURTLE TO 30 DEGREES )
20 FORWARD        ( DRAW A LINE OF 20 UNITS )
```

5.8.29 SETX : Move To New X Coordinate

(XVALUE ---)

The turtle is moved horizontally to new X coordinate. A line is drawn based on whether the pen is up (PENUP) or down (PENDOWN).

Example:

```
DRAW              ( SET UP TURTLE SCREEN )
20 SETX           ( DRAW HORIZ LINE TO X COORD. 20 )
```

5.8.30 SETXY : Set X,Y Coordinate

(X Y ---)

The turtle is moved to coordinate X,Y. Nothing is drawn.

Example:

```
DRAW              ( SET UP TURTLE SCREEN )
50 50 SETXY       ( MOVE TURTLE TO NEW COORDINATE )
20 FD             ( DRAW A LINE )
```

5.8.31 SETY : Move To New Y Coordinate

(YVALUE ---)

The turtle is moved vertically to new Y coordinate. A line is drawn based on whether the pen is up (PENUP) or down (PENDOWN).

Example:

```
DRAW              ( SET UP TURTLE SCREEN )
20 SETY           ( DRAW VERTICAL LINE TO Y COORD. 20 )
```

5.8.32 SPLITSCREEN : Set Split Graphics/Text Screen

(---)

Enter turtle split screen mode. Does not affect turtle screen. May be abbreviated SS.

Example:

```
DRAW                ( INITIALIZE TURTLE SCREEN )
60 FORWARD          ( DRAW A LINE )
YELLOW BACKGROUND   ( CHANGE BACKGROUND COLOR )
TEXTSCREEN          ( EXIT TURTLE SCREEN )
SPLITSCREEN         ( RE-ENTER TURTLE SCREEN )
```

5.8.33 SS : Set Split Graphics/Text Screen

(---)

Abbreviation for SPLITSCREEN.

5.9 C64 Sound Related Words

As with the graphics extensions, these words are designed to ease programming the SID (sound synthesizer) chip on the C64. The SID is a fairly sophisticated sound generation device. In order to fully explore its potential the user will need to learn something about modern music synthesizers (upon which the chip is based). Refer to one of the books on C64 Sound listed in Appendix VIII for detailed information on sound synthesis and the SID chip.

These word descriptions assume a basic understanding of sound terminology. The voice referenced by the contents of VOICE is referred to as the "active voice". The definitions V1, V2 and V3 are provided to automatically set the active voice.

The system has been designed to enable programming of additional SID chips if the user should interface them to his system. If a multiple chip system is available the user simple puts the base address of the currently active chip into variable SID and the routines will use this address to properly set up the registers in that chip.

Refer to the Appendix I for tutorial examples of sound word usage. v

5.9.1 ENV3@ : Fetch Envelope Value

(--- VALUE)

Returns the VALUE (0 to 255) of the oscillator envelope 3 output register. This value can be used for envelope modulation effects by adding it to input registers, such as voice freqency, filter frequency or pulse width.

Example: ENV3@ . will print the value of the envelope 3 output register.

5.9.2 F-FREQ : Set Filter Frequency

(VALUE ---)

Set the filter frequency. VALUE ranges from 0 to 2047 (this is NOT a frequency in Hz). The word makes the correct alignment adjustments sets both the lo and high portions of the filter freqency from VALUE. The filter must be turned on in order to affect any voices.

Example: 800 F-FREQ sets the filter frequency value to 800.

5.9.3 MODEVOL : Set Mode/Volume

(MODE VOLUME ---)

Sets the mode/volume register in the SID chip from the given values. Both MODE and VOLUME range from 0 to 15.

Example: 3OFF HIGHFPASS OR 15 MODEVOL sets the mode to 12 (voice 3 off filter set to high pass) and the volume to 15.

5.9.4 NOTE@ : Fetch Note from NOTE-VALUES

(NOTE ---)

A primitive used by PLAY.NOTE to fetch a value from the NOTE-VALUES table. NOTE ranges from 0 to 11 (C to B).

5.9.5 NOTE-VALUES : Table of Chromatic Note Values

(--- ADDR)

Table of 12 SID frequencies corresponding to the highest octave playable by the SID chip. These values may be divided down to produce the note in a different octave (see PLAY.NOTE).

5.9.6 OSC3@ : Fetch Oscillator 3 Value

(--- VALUE)

Returns the VALUE (0 to 255) of the oscillator 3 frequency output register. This value can be used for frequency modulation effects by adding it to input registers, such as voice freqency, filter frequency or pulse width.

OSC3@ can also be used to generate random numbers by setting the waveform of oscillator 3 to NOISE and using OSC3@ to read the digitized noise output.

Example:

```
SOUND.INIT              ( INITIALIZE CHIP )
V3 NOISE V-CTRL         ( SET V3 FOR NOISE )
65535 V-FREQ            ( HIGHER FREQENCIES ARE BETTER )
: RANDOM-TEST           ( PRINT 100 RANDOM #'S)
   100 0 DO
     OSC3@  .
   LOOP ;
```

5.9.7 PLAY.NOTE : Play A Chromatic Note

(NOTE ---)

Plays a note from the chromatic scale. NOTE is a value from 0 to 95 corresponding to the chromatic scale as follows:

Decimal Value	Note	Base 12 Value
0 - 11	C0 - B0	00 - 0B
12 - 23	C1 - B1	10 - 1B
24 - 35	C2 - B2	20 - 2B
36 - 47	C3 - B3	30 - 3B
48 - 63	C4 - B4	40 - 4B
60 - 71	C5 - B5	50 - 5B
72 - 83	C6 - B6	60 - 6B
84 - 95	C7 - B7	70 - 7B

As can be seen from the table, the values given in base 12 are much more suited to playing notes than decimal- the "twelves" digit corresponds to the octave which the note is in and the ones digit corresponds to the note within the octave as follows:

0	1	2	3	4	5	6	7	8	9	A	B
C	C#	D	D#	E	F	F#	G	G#	A	A#	B

Thus, to play a G5 you would use 57 PLAY.NOTE. This is one of the handy features of FORTH- you can set the number base to suit your needs. To set base twelve use the following: 12 BASE !. to get back to decimal mode use DECIMAL.

The waveform to be played is taken from variable WAVE, which must be set before using NOTE. SID control information must also have been set up. SOUND.INIT will set default values for all three voices. Refer to the sound example in the appendix.

Example:

```
        SOUND.INIT       ( INITIALIZE SOUND SYSTEM )
        12 BASE !        ( SET TO BASE 12 )
        48 PLAY.NOTE     ( PLAY A G# )
        DECIMAL
```

This example initializes the sound system, sets the system number base to 12, plays the G in octave 4, and resets the base to decimal.

5.9.8 RESFILT : Set Resonance/Filter

(RESONANCE FLAGS ---)

Sets the resonance value and filter flags bits in the resonance/filter SID register. Both RESONANCE and FLAGS range from 0 to 15.

Example: 13 FILT1 RESFILT sets the resonance to 13 and filtering active for voice 1.

5.9.9 SID : SID Address Variable

(--- ADDR)

Variable in which is stored the address of the SID chip in current usage. All SID register references are made as offsets into the area addressed by this variable. Thus, by changing the SID address stored here, a multiple SID chip system can be programmed using the same FORTH SID words. It is initialized by SOUND.INIT to be the address of the SID chip which comes in the C64 initially.

Example: SID @ . will print 54272 in the initial system after SOUND.INIT has been invoked.

5.9.10 SID! : Store Value into SID Register

(VALUE OFFSET ---)

Stores the byte VALUE (range 0 to 255) into the SID register OFFSET (range 0 to 24) within the current SID area (address stored in variable SID). This utility, used by other sound routines, is generally not used directly, but is available for use if desired.

Example: TRIANGLE 4 SID! sets voice 1 to a triangle waveform.

5.9.11 SID@ : Fetch Value From SID Register

(OFFSET --- VALUE)

Gets a byte value from the register OFFSET (range 0 to 24) in the SID area referenced by the address stored in SID. This is a sound system utility word.

5.9.12 SOUND.INIT : Initialize Sound System

(---)

Initializes the sound system for the SID chip whose address is stored in the variable SID (initially 54272). Sets defaults for all voices and leaves V1 as the active voice.

Example:

```
SOUND.INIT
48 PLAY.NOTE
```

This example initializes the sound system and plays a fourth octave C.

5.9.13 Sound Constants

The following constants are supplied (both with the system and as supplemental screens) to ease use setting up the SID chip.

Voice Control Register Constants:

TRIANGLE	17	SYNC	3
SAWTOOTH	33	RESET	8
PULSE	65	RING	21
NOISE	129		

Filter Constants:

LOWPASS	1	FILT1	1
BANDPASS	2	FILT2	2
HIGHPASS	4	FILT3	4
NOTCH	5	FILTEX	8
3OFF	8		

Misc:

OFF	0
ON	1

Calling the constant places its value on the stack. As with other FORTH constants calling the constant by name will cause its value to be compiled into the definition.

Example: TRIANGLE V-CTRL sets the active voice to a triangle waveform.

5.9.14 V-AD : Set Voice Attack/Decay

(ATTACK DECAY ---)

Sets the attack and decay characteristics of the active voice. ATTACK and DECAY values both range from 0 to 15.

Example: 12 5 V-AD sets a fairly long attack and a short decay for the active voice.

5.9.15 V-CTRL : Set Voice Control Register

(MASK ---)

Sets the control register for the active voice.

Example: TRIANGLE V-CTRL will set the control register with the value for a gated on triangle wave (17). OFF V-CTRL will clear the register.

5.9.16 V-DEFAULT : Default Settings of the SID Chip

(---)

This word activates default settings of the active voice. It is provided mainly as an example of how to initialize the SID chip so that a voice will "play" when a waveform is gated on.

Source code is provided (see source screens, Appendix II) and it is expected that once the user understands the components of the word he will probably want to use different settings of the various parameters. Meanwhile, it is an easy way to set up the chip to play notes using PLAY.NOTE. V-DEFAULT is invoked for all three voices by SOUND.INIT.

5.9.17 V-FREQ : Set Voice Freqency

(VALUE ---)

Sets the VALUE (ranging from 0 to 65535) into the frequency registers of the active voice. Note, the split between hi and low byte is made automatically. The frequency value is NOT a value in hertz. The frequency range of each oscillator is between 0 and 4khz.

Example: V2 8000 V-FREQ sets the frequency of voice 2 to 8000.

5.9.18 V-PW : Set Voice Pulse Width

(VALUE ---)

Sets the pulse width value for the active voice. VALUE ranges from 0 to 4095. The value is automatically split up and put into the two pulse width registers.

Example: 2048 V-PW sets the pulse width for the active voice to a square wave.

5.9.19 V-SR : Set Voice Sustain/Release

(SUSTAIN RELEASE ---)

Sets the sustain and release characteristics of the active voice. SUSTAIN and RELEASE values both range from 0 to 15.

Example: 15 5 V-SR sets the active voice sustain to 15 and release to 5.

5.9.20 V! : Put Value in Active Voice Register

(VALUE OFFSET ---)

Sound system primitive which puts the given byte VALUE (0 to 255) into the active voice register determined by adding OFFSET (0-6) to the base address of the active voice in the SID area. This word is used by other sound system words and is generally not directly invoked by the user but is available for use if desired.

Example: If SID is 53272 and the active voice is V2 then the FORTH statement: 100 2 V! would put the value 100 into the pulse width register at 53281.

5.9.21 V1,V2,V3 : Set Active Voice

(---)

Sets the active voice to voice 1, voice 2 or voice 3.

Example: V3 400 V-PW sets the pulse width of voice three to 400.

5.9.22 VOICE : Active Voice Variable

(--- ADDR)

Variable which contains the number of the active voice - 1. V1, V2 or V3 are provided to set the active voice, therefore this word is not generally used directly.

Example: VOICE ? can be used to display the currently active voice, where 0=V1, 1=V2 and 2=V3.

5.9.23 WAVE : Waveform Variable

(--- ADDR)

Variable used by PLAY.NOTE to determine what type of waveform to use to play a note. It must be initialized by the user- either directly or through use of V-DEFAULT.

5.10 Music Editor Words

Included in the system is a Music Editor. The Music Editor enables users to enter music in up to three parts, either interactively or as compiled FORTH definitions. The lower level sound extension words may be intermixed with Music Editor definitions. Thus, the full capabilities of the SID chip may be realized while using the Music Editor.

The tempo to be used is specified by storing a value in beats per minute into the variable TEMPO. Prior to playing any notes of a piece of music the song initialization word SONG.INIT must be entered to reset the clocks for each voice. A voice is chosen by specifying one of the voice words, V1, V2 or V3. A note is chosed by specifying what octave it is in, what its duration will be and the name of the note (C, C# or D, D, D# or E, etc.).

Example:

```
SOUND.INIT        ( INITIALIZE SID CHIP )
60 TEMPO !        ( SET TO 60 BEATS PER MINUTE )
: MARY
   SONG.INIT      ( INITIALIZE SONG )
   O4             ( FOURTH OCTAVE )
   1/8 E D C D E E 1/4 E
   1/8 D D 1/4 D
   1/8 E G 1/4 G ;
MARY
```

A simple one voice melody is played.

To play multiple voices, enter the voices note in the sequence in which it should be played. Notes for three separate voices will be heard as being played simultaneously since separate note durations are kept for each voice.

Example:

```
     SOUND.INIT          ( INITIALIZE SID CHIP )
     60 TEMPO !          ( SET TO 60 BEATS PER MINUTE )
     : MARY1
        SONG.INIT        ( INITIALIZE SONG )
        O4               ( SET OCTAVE )
        1/8 V1 E V2 G V3 B  V1 D V2 F V3 A  V1 C V2 E V3 G
            V1 D V2 F V3 A  V1 E V2 G V3 B  V1 E V2 G V3 B
        1/4 V1 E V2 G V3 B
        1/8 V1 D V2 F V3 A  V1 D V2 F V3 A  1/4 V1 D V2 F V3 A
        1/8 V1 E V2 G V3 B  V1 G V2 B O5 V3 D
        1/4 O4 V1 G V2 B V3 O5 D ;
     MARY1
```

The simple melody is played in three voice harmony.

Music definitions can be "built" and programmed using FORTH. Phrases can be repeated or transposed to other keys. In the next example I will define a bass line then repeat it and transpose it to play a blues bass:

```
     SOUND.INIT
     : ROOT 0 T! ;      ( SET  ROOT )
     : 4TH 5 T! ;       ( SET FORTH )
     : 5TH 7 T! ;       ( SET FIFTH )
     : BLUES            ( DEFINE 1 MEASURE )
        O2 1/8
        C E G A A# A G E ;
     : C-BLUES
        100 TEMPO !
        V1 0 9 V-AD  TRIANGLE WAVE !
        SONG.INIT ROOT          ( START OUT IN C )
        4 0 DO BLUES LOOP       ( 4 MEASURES IN C )
        4TH BLUES BLUES         ( 2 MEASURES IN F )
        ROOT BLUES BLUES        ( 2 MEASURES IN C )
        5TH BLUES               ( 1 MEASURE IN G )
        4TH BLUES               ( 1 MEASURE IN F )
        ROOT BLUES BLUES        ( END IT IN C )
        C ;
```

This example displays several features of the Music Editor- it uses program control over repeating a phrase (BLUES-RUN), transposing, and sets sound parameters upon initialization of the song.

A harmonized melody can be programmed by defining an array which specifies the scale, and calculating the harmony based on the scale. For instance, let's say we want to harmonize a major scale. We can enter the following definitions:

```
: SCALE-DEF      ( DATA STRUCTURE FOR DEFINING SCALES )
   CREATE        ( EXPECTS 8 BYTES IN INPUT STREAM )
    8 0 DO
     BL TEXT PAD NUMBER DROP C,
    LOOP
   DOES> + C@ ;
SCALE-DEF MAJOR 0 2 4 5 7 9 11 12

( PLAY A HARMONIZED CHORD GIVEN BASE NOTE )
: HARMONY  ( NOTE --- )
   DUP V1 PLAY.WAIT     ( PLAY ROOT & WAIT GIVEN DURATION)
   DUP 4 + V2 PLAY.WAIT ( PLAY THE THIRD )
   7 + V3 PLAY.WAIT ;   ( PLAY THE FIFTH )

( NOW SET UP A HARMONIZED MAJOR SCALE )
: MAJ-SCALE
   SONG.INIT
   8 0 DO I MAJOR HARMONY LOOP ;
O3 MAJ-SCALE     ( PLAY THIRD OCTAVE MAJOR SCALE )
O4 MAJ-SCALE     ( PLAY FOURTH OCTAVE MAJOR SCALE )
```

This may appear to be an elaborate setup, but using SCALE-DEF we can set up any type of scale and play harmonies from it!

These examples should give you some ideas on the sorts of things which can be accomplished using the SUPER FORTH 64 Music Editor. You can program sound effects, musical progressions, or just plain old three part music, and the SID chip can be set to sound the same throughout a song or re-programmed for every note in the song!

The source code for the Music Editor is provided as an example of how to set up a system such as this using FORTH. The definitions are all fairly simple and it is recommended hat the user examine the source code both to gain an understanding of how a Music Editor is implemented and to get ideas for extending the Music Editor even further. This is one of the powers of FORTH- what would be a fixed utility in another system is a growing entity under SUPER FORTH- if you want it to do more, you MAKE it do more! A description of each of the Music Editor words follows.

5.10.1 DURATION : Note Duration Variable

(--- ADDR)

This variable is used to control the duration of a note. The duration value is set by one of the note duration words (see table below). How long in time a note plays is determined both by the note duration and the setting of TEMPO. The word TRIPLET converts the duration value to triplets of the last duration set by multiplying the duration value by 2/3. The last duration entered will be used for all notes until a new duration is entered.

Musical Duration	FORTH Word
Whole note	WHOLE
Dotted half note	.1/2
Half note	1/2
Dotted quarter note	.1/4
Quarter note	1/4
Dotted eighth note	.1/8
Eighth note	1/8
Dotted sixteenth note	.1/16
Sixteenth note	1/16
Thirtysecond note	1/32
Sixtyfourth note	1/64

The 60 cycle clock is used to determine durations, therefore, the minimum duration of a note is 1/60 of a second.

Example:

```
SONG.INIT O4 1/4 C D E F 1/8 TRIPLET C D E F G A B A G F E C
```

This example plays four quarter notes followed by four sets of eighth note triplets.

5.10.2 NCALC : Calculate the Time for A Note Duration

(---)

This word calculates the time of the 60 cycle clock which would signal the start of a new note for a particlar voice. The timer in NEXT.NOTE for the active voice is updated. The new time is based on the values in DURATION and TEMPO. This word is called by PLAY.WAIT.

5.10.3 NEXT.NOTE : Timing For Next Note For Each Voice

(--- ADDR)

This array contains three double precision timers, one for each voice. The timer for a voice is compared with the 60 cycle clock by READY to determine whether it is time to play a note for a particular voice. The timers are set by NCALC. When NEXT.NOTE is called it returns the address of the timing variable for the active voice.

5.10.4 Notes : Play a Chromatic Note

These words are the main words used to play chromatic notes using this Music Editor. The words are named for their musical equivalents, with the exception that flats (Bb, Eb, etc.) are represented by the note name suffixed with the British pound sign (B\, E\, etc.). The defined notes are as follows:

C C# D\ D D# E\ E F F# G\ G G# A\ A A B\ B

A word, TIE, has been defined to designate a note which should be "tied" over the value in duration.

5.10.5 O@ : Fetch the Current Octave Value

(--- VALUE)

This word returns the value of the current octave. The value may be used in computations.

5.10.6 O! : Set New Octave Value

(---)

This word is used by the octave setting words to set the value of the current octave. An octave setting word (O0, O1 ... O7) sets the octave value to the PLAY.NOTE value of the first note in the octave. The following table lists the values:

Octave	Value
O0	0
O1	12
O2	24
O3	36
O4	48
O5	60
O6	72
O7	84

5.10.7 PLAY.WAIT : Wait Until Ready and Play Note

(VALUE ---)

PLAY.WAIT waits until it is time to play the note for the active voice. It then takes the note value which is on the stack, adds in the current octave value and the current transposition value, plays the note, and calls NCALC to calculate when the next note should be played.

5.10.8 SONG.INIT : Initialize Timers for Song

This word synchronizes the timers for all voices with the 60-cycle clock and must be used once at the start of each piece of music which is to be played.

Example:

```
SONG.INIT V1 WHOLE O3 C  V2 O4 1/4 C E G E
```

5.10.9 T@ : Fetch Transposition Value

(--- VALUE)

This word fetches the transposition value. It is called by PLAY.WAIT to add the transposition value to the note value. It may be called by the user for computations involving the transposition value (see T!).

5.10.10 T! : Set New Transposition Value

(VALUE ---)

This word sets the transposition value. This value is added to each note prior to playing the note. Thus, a phrase may be transposed in key by changing the transposition value before it is played.

Example:

```
: SCALE C D E F G A B C ;
O4 1/8
0 T! SONG.INIT SCALE     ( PLAYS C SCALE )
1 T! SONG.INIT SCALE     ( PLAYS C# SCALE )
2 T! SONG.INIT SCALE     ( PLAYS D SCALE )
```

5.10.11 TEMPO : Music Tempo Variable

(--- ADDR)

This variable is used to control the tempo of the music. TEMPO is set in standard metronone "beats per minute", where a beat is the length of time a quarter note will play.

Example:

```
60 TEMPO !
```

sets a tempo of 60 beats per minute (one quarter note per second).

5.11 String Extension Words

The words in this section have been designed to enable the user to manipulate strings using words which are similar to BASIC string manipulation functions. All string extensions (except for immediate strings) start with a dollar sign ("$") to enable instant recognition as string creation or manipulation words.

Strings in SUPER FORTH are data areas which are of the form:

```
length : 1 byte
characters : length bytes
```

A string area is allocated in the dictionary by either using the $CONSTANT or $VARIABLE defining words. A string is created through use of $CONSTANT (which adds a string to the dictionary) or " (which moves a string to the PAD area for further use).

Strings are usually referenced by placing their starting address (the address of the length byte) on the stack. Invoking a created string data structure (one which has been defined using $CONSTANT or $VARIABLE) leaves the string's starting address on the stack. The immediate string word " leaves the starting address of the string on the stack.

Here is an example of building a string from various substrings using the string manipulation words:

```
100 $VARIABLE WORK      ( SET UP A 100 BYTE STRING WORK AREA )
$CONSTANT STRING1 "TESTING"
WORK STRING1 $CONCAT    ( MOVE "TESTING" INTO WORK AREA )
WORK "  ONE, TWO, THREE. "  $CONCAT
WORK STRING1 4  $LEFT   ( GET "TEST" )
WORK WORK 8 4  $MID     ( GET " ONE" )
WORK " ." $CONCAT       ( ADD "." )
WORK $.                 ( PRINT OUT STRING IN WORK AREA )
```

TESTING ONE, TWO, THREE. TEST ONE. will print out on the screen.

The string words can be grouped into several categories: $VARIABLE $CONSTANT and " are used to create strings or allocate room for strings. $SCONCAT $CONCAT $LEFT $MID and $RIGHT are used to manipulate substrings. $CMP $< $> and $= are used for string comparisons. $FIND is used to determine the occurance of one string in another. $. $LEN $VAL and $CLR are utility functions which enable printing, finding the length, converting a numeric string to a 16-bit value and clearing a string.

5.11.1 " : Create An Immediate String

(--- S1)

Allows user to create an immediate string (one with no permanent name attached to it). The string is put into the PAD area and its starting address is left on the stack. The string is entered using a quote as the ending delimiter. Note, there must be a blank space after the initial quote since the quote is part of the FORTH word.

This word can be used prior to any words which require a string address to be left on the stack (such as DOS or OPEN).

Example: " THIS IS AN EXAMPLE" puts the string in the PAD leaving its starting address on the stack.

5.11.2 "" : Create A Null String

(--- S1)

Creates a "null" string (string of length 0) in the PAD area and leaves its address on the stack. This word may be used as an argument to another word which requires a string address on the stack, but for which a null string is suitable.

Example:

```
127 4 0 "" OPEN
```

opens a channel to a printer.

5.11.3 $. : Display A String

(S1 ---)

Sends the string whose starting address is on the stack to the current output device.

Example: " TEST!" $. will cause TEST! to be printed.

5.11.4 $< : Test Strings For <

(S1 S2 --- F)

The string at address S1 is compared character by character with the string at address S2. If string S1 is less than string S2 then a true flag (1) is left on the stack, otherwise a false flag (0) is left. See $CMP for string comparisons.

Example:

```
$CONSTANT STR1 "TEST"
$CONSTANT STR2 "ABCDE"
STR1 STR2 $<
```

Results in a 0 left on the stack since STR1 is not < STR2.

5.11.5 $= : Test Strings For =

(S1 S2 --- F)

The string at address S1 is compared character by character with the string at address S2. If string S1 is equivalent to string S2 then a true flag (1) is left on the stack, otherwise a false flag (0) is left. See $CMP for string comparisons.

Example:

```
$CONSTANT STR1 "TEST"
$CONSTANT STR2 "ABCDE"
STR1 STR2 $=
```

Results in a 0 left on the stack since STR1 is not = STR2.

5.11.6 $> : Test Strings For >

(S1 S2 --- F)

The string at address S1 is compared character by character with the string at address S2. If string S1 is greater than string S2 then a true flag (1) is left on the stack, otherwise a false flag (0) is left. See $CMP for string comparisons.

Example:

```
$CONSTANT STR1 "TEST"
$CONSTANT STR2 "ABCDE"
STR1 STR2 $>
```

Results in a 1 left on the stack since STR1 is > STR2.

5.11.7 $CLR : Clear A String

(S1 ---)

The string starting at S1 is cleared by setting its character count to 0. This is useful for clearing out string variable areas or the PAD work area.

Example: PAD $CLR clears the PAD area.

5.11.8 $CMP : String Comparison

(S1 S2 --- F)

This word is used by other string comparison words ($<, $= and $>) to perform a comparison of the two strings starting at addresses S1 and S2. String comparisons are made by taking one character at a time (left-to-right) from each string and evaluating each character code position from the C64 character set. If the character codes are the same, the characters are equal. The comparison stops when the end of either string is reached. All other things being equal, the shorter string is considered less than the longer string. Leading or trailing blanks ARE significant.

Upon completion of the comparison a result is left on the stack as follows: -1 if S1 is less than S2, 0 if S1 equals S2 or 1 if S1 > S2.

Example:

```
$CONSTANT STR1 "TEST"
$CONSTANT STR2 "ABCDE"
STR2 STR1 $CMP
```

Results in a -1 left on the stack since STR2 is < STR1.

5.11.9 $CONCAT : Concatenate Strings

(S1 S2 ---)

The string starting at S2 is appended onto the end of the string starting at S1. S2 is left untouched.

Example:

```
$CONSTANT STR1 "TEST"
$CONSTANT STR2 "ABCDE"
40 $VARIABLE WORK
WORK STR1 $CONCAT
WORK STR2 $CONCAT
WORK $.
```

Results in the string "TESTABCDE" printing out.

5.11.10 $CONSTANT : Define String Constant

```
( --- )      Compile-time
( --- S1 )   Execution-time
```

Defining word which is used to create a string constant. The string is entered in quotes (") after its name. The length of the string is compiled into the dictionary as a byte in front of the string itself. After invoking the defined name, the starting address of the string is left on the stack.

Example:

```
$CONSTANT STR1 "THIS IS A TEST"
STR1 $.
```

Results in the constant STR1 being defined and output to the display.

5.11.11 $FIND : Locate A String Within Another String

```
( S1 A2 C2 --- [ADDR] FLAG )
```

The string whose characters start at address A2 is searched for a count of C2 for the string whose starting address is at S1. If the string is found, ADDR is the address of its starting character in A2 and FLAG is TRUE (1). If the string is not found, no address is placed on the stack and FLAG is FALSE (0).

This word can be used to set up higher level string search words (search for Nth occurence, etc.).

Example:

```
$CONSTANT STR1 "THIS IS A TEST"
" IS" STR1 COUNT $FIND      ( LOCATES "IS" IN STR1)
. .
```

Results in a 1 (TRUE FLAG value) and the address of the located string displayed.

5.11.12 $INPUT : Input A String

```
( --- S1 )
```

After execution of $INPUT the system will wait for a line to be entered by the user. A carriage return signals the end of the line. The input line is transferred to the PAD and its address is left on the stack.

Example:

```
$INPUT  (carriage-return) THIS IS A STRING
$.
```

Results in the string "THIS IS A STRING" being transferred to the PAD and then displayed.

5.11.13 $LEFT : Concatenate Leftmost Substring

(S1 S2 NUM ---)

Concatenates the leftmost NUM characters of the string starting at S2 onto the end of the string starting at S1. S2 is left untouched.

Example:

```
40 $VARIABLE WORK
$CONSTANT STR1 "TESTING ONE, TWO, THREE"
$CONSTANT STR2 "STILL AT IT"
WORK STR1 8 $LEFT
WORK STR2 5 $LEFT
WORK $.
```

Results in WORK being set to "TESTING STILL" and output to the display.

5.11.14 $LEN : Get Length of String

(S1 --- LEN)

Leaves the byte count of the string starting at S1 on the stack.

Example:

```
$CONSTANT STR1 "TESTING ONE, TWO, THREE "
STR1 $LEN .
```

Results in a 24 being output to the display.

5.11.15 $MID : Concatenate Middle Substring

(S1 S2 POS NUM ---)

Concatenates NUM characters of the string starting at the POSth character of S2 onto the end of the string starting at S1. S2 is left untouched.

Example:

```
40 $VARIABLE WORK
$CONSTANT STR1 "TESTING ONE, TWO, THREE"
$CONSTANT STR2 "STILL AT IT"
WORK STR1 9 3 $MID
WORK STR2 6 3 $MID
WORK $.
```

Results in WORK being set to "ONE AT" and output to the display.

5.11.16 $RIGHT : Concatenate Right Substring

(S1 S2 NUM ---)

Concatenates the rightmost NUM characters of the string starting at S2 onto the end of the string starting at S1. S2 is left untouched.

Example:

```
40 $VARIABLE WORK
$CONSTANT STR1 "TESTING ONE, TWO, THREE "
$CONSTANT STR2 "STILL AT IT"
WORK STR1 6 $RIGHT
WORK STR2 2 $RIGHT
WORK $.
```

Results in WORK being set to "THREE IT" and output to the display.

5.11.17 $VAL : Get Numeric Value of String

(S1 --- NUM)

Converts a numeric string starting at address S1 to a 16-bit number and leaves it on the stack. If the string contains a non-numeric character ABORT" is called and the error message "NOT RECOGNIZED" is output to the display.

NOTE: If a string variable area is used to contain the numeric string, the length of the string area must be at least 1 greater than the length of the string.

Example:

```
" 12345" PAD $VAL .
```

will convert the immediate string "12345" to its binary equivalent and print it out under the current base.

5.11.18 $VARIABLE : Create String Variable

```
( N --- )      Compile-time
( --- ADDR )   Run-time
```

Defining word used to allocate dictionary space for a string variable of maximum N bytes in length. The variable is cleared initially. Upon invoking the variable name, the starting address of the variable is left on the stack.

Example:

```
40 $VARIABLE WORK
WORK  " TESTING "   $CONCAT
WORK  " ONE, TWO "  $CONCAT
WORK $.
```

Results in a 40 byte space allocated for the string variable WORK and WORK set to " TESTING ONE, TWO " which is output to the display.

5.11.19 <"> : Immediate String Run-time Routine

```
( --- S1 )
```

Used by " for immediate strings within definitions to move the string to the PAD and leave its address on the stack. Although available, this word is not meant for general use.

5.11.20 <$CONCAT> : Perform Concatenation

```
( S1 A2 C2 --- )
```

This word is used by various other words to perform a concatenation function. C2 characters starting at A2 are appended to the end of the string starting at S1.

Example:

```
$CONSTANT STR1 "TEST"
$CONSTANT STR2 "ABCDE"
40 $VARIABLE WORK
WORK STR1 COUNT <$CONCAT>
WORK STR2 COUNT <$CONCAT>
WORK $.
```

Results in WORK being set to the string "TESTABCDE" and output to the display. Note that this could more easily be done using $CONCAT.

5.12 Interrupt Extension Words

Among the many uses of the FORTH language are applications which require an action to be performed upon receiving an interrupt from a peripheral device. High level languages typically do not have the hardware control to provide high level support of interrupts. SUPER FORTH, however, supports both low level (machine language) and high level (FORTH) handling of IRQ interrupts. The concept of interrupt handling is an advanced programming topic which the user should fully understand before attempting to use the words in this section.

Whether an interrupt routine can be handled in high level is determined by how often the interrupt occurs and how much processing must happen upon detection of the interrupt. An interrupt which occurs once a second, for instance, and signals the reading of an analog-to-digital converter can easily be handled in high level FORTH. An interrupt which occurs 50,000 times a second (which may be the frequency for digitizing an audio signal, for example) could not be handled in high level FORTH. Once method of determining whether the interrupt could be handled in high level is to write the interrupt routine in high level code and call it in a test loop to determine how long it takes to execute. If the routine takes longer to execute than the expected time between interrupts then it will have to be coded in machine code (this could be done using the FORTH 6510 assembler).

For example, if we define an interrupt routine:

```
        : I-ROUTINE  1 32768 +! ;
```

and a test loop which uses it, along with a dummy test loop:

```
        : TEST1 30000 0 DO LOOP ;
        : TEST2 30000 0 DO I-ROUTINE LOOP ;
        0. SETTIM TEST1 RDTIM DROP 6 / .
        0. SETTIM TEST2 RDTIM DROP 6 / .
```

We find that execution of TEST1 takes 3.8 seconds (38 1/10 seconds) and execution of TEST2 takes 16.3 seconds. Thus I-ROUTINE executes 30000 times in 12.5 seconds, or 2400 times a second. An interrupt which occurred more than 2400 times a second would have to be handled by an equivalent machine code routine. Note that high level routines MUST BE KEPT SHORT! Otherwise they will take too long to execute. The following paragraphs give a description of the words which are used to implement the SUPER FORTH interrupt system.

I-INIT is used to initialize the SUPER FORTH interrupt system. Upon being called by the user, I-INIT saves the system IRQ interrupt vector (at location $314) and replaces it with the address of the interrupt executive routine. The executive is used by SUPER FORTH as a replacement for the standard system interrupt routine. It is called on at the occurance of an IRQ interrupt (such as the 60 cycle clock interrupt). Upon entry, it checks to see if there is a user defined interrupt routine to execute. If there is, the FORTH system context is saved and the system is set up to execute the user interrupt routine. I-INIT should be called only once after any cold or warm starts.

I-SET is used to set up the user interrupt routine and interrupt routine stack size. I-CLEAR is used to clear the user interrupt routine and resume using the initial system IRQ vector.

Since there is only a single IRQ interrupt signal, the user interrupt routine must verify that the interrupt is meant for it. Before exiting, the user interrupt routine must set the byte length interrupt return flag at location $7F either to 0, designating that the interrupt has been handled and a normal return should occur, or 1, designating that this interrupt is not a user interrupt and execution should pass to the system interrupt handler. The SUPER FORTH words I-USER and I-SYSTEM can be called to clear or set the return flag location ($7F) quickly.

After exiting the user interrupt routine execution go to an interrupt return routine which handles restoring the regular FORTH context and based on the return flag either performs the return from interrupt sequence or jumps to the system interrupt routine.

Thus, the interrupt sequence when using the SUPER FORTH system is as follows:

```
    INTERRUPT -> INTERRUPT EXEC -> USER WORD -> INTERRUPT RETURN
      [ -> system default routine ]
```

The following is a complete example of an interrupt routine implementation. Both the high level and the assembly language versions are shown. The routine will increment the 16-bit value at location 32768 on every 60-cycle interrupt and will give control to the system interrupt routine when finished:

```
I-INIT      ( INITIALIZES FORTH INTERRUPT SYSTEM )
( HIGH LEVEL VERSION )               ( ASSEMBLER  VERSION )

: INCREMENT                          CODE INCREMENT
   1 32768 +!                         1 # LDA,
   I-SYSTEM ;                         CLC,
                                      32768 ADC,
                                      32768 STA,
                                      0= IF,
                                       0 # LDA,
                                       32769 ADC,
                                       32769 STA,
                                      THEN,
                                      ' I-SYSTEM JMP,
                                     END-CODE

0 32768 !       ( CLEAR THE LOCATION INITIALLY )

( WORD TO TEST INCREMENT OF LOCATION 32768 )
: TEST  100 0 DO 32768 ? LOOP ;
' INCREMENT 5 I-SET
TEST            ( PRINTS OUT INCREMENTING VALUES )
I-CLEAR         ( STOPS INTERRUPT ROUTINE )
```

Note that the assembly language version, while much more complicated to read and slightly larger in memory usage, would run somewhat faster than the high level version. Note also that the assembler version uses no stack space while the high level version uses two stack locations (we allocate 5 locations just to be "safe").

5.12.1 I-CLEAR : Clear User Interrupt Routine Address

(---)

Sets the SUPER FORTH interrupt system to go to the system interrupt routine and not call a user interrupt routine. Used to stop calling a routine which had been set up by using I-SET.

Example: I-CLEAR clears any previously set user interrupt routine.

5.12.2 $7F : Interrupt Return Flag

(--- 7F)

Returns the address of a byte which is used to the interrupt system to determine whether or not to execute the default system interrupt routine (generally, the 60 cycle clock routine) after user interrupt processing is completed. If 0, a standard return from interrupt sequence is executed (PLA TAY PLA TAX PLA RTI). If 1, the system interrupt routine is executed. The words I-USER or I-SYSTEM should be used to set I-FLAG.

5.12.3 I-INIT : Initialize Interrupt System

(---)

Initializes the SUPER FORTH interrupt system to allow user defined interrupts. Replaces the system default interrupt vector with the address of the SUPER FORTH interrupt executive routine. Since system defaults are reset upon warm starting, this word must be invoked on initial startup and warms starts after the initial startup.

Example: I-INIT sets up system to enable user defined interrupts.

5.12.4 I-SET : Set User Interrupt Routine Address

(PFA N ---)

Sets the SUPER FORTH interrupt system to go to the interrupt routine whose parameter field address is PFA. N is the number of stack values to allocate to the interrupt stack (stack usage must be determined for each interrupt routine).

Example: ' ROUTINE 10 I-SET sets up the word ROUTINE to be executed upon receiving an IRQ interrupt. Ten values are allocated for ROUTINE to use as its system stack.

5.12.5 I-SYSTEM : Set I-FLAG to System Routine Exit

(---)

Word which sets the interrupt return flag to 1 so that the interrupt return routine will call the system default interrupt routine upon completion. No stack usage is performed by this word. I-SYSTEM is implemented in machine language for efficiency and can be called by a machine language interrupt routine to exit to the system.

Examples:
I-SYSTEM sets the interrupt return flag to 1.

In assembler: ' I-SYSTEM JMP, sets the interrupt return flag to 1 and returns control to the system.

5.12.6 I-USER : Set Interrupt Return Flag to Exit

Word which sets the interrupt return flag to 0 so that the interrupt return routine will perform a return from interrupt upon completion. No stack usage is performed by this word. I-USER is implemented in machine language for efficiency and can be called by a machine language interrupt routine to exit to the system.

Examples:

I-USER sets the interrupt return flag to 0.
In assembler: ' I-USER JMP, sets the interrupt return flag to 0 and returns control to the system.

5.13 Display Screen Words

These extensions give the user control over the output display screen. All display related extensions are prefixed with "D-" to enable the user to remember the "D"isplay subset of SUPER FORTH commands. D-SPLIT allows text and hi-res graphics to be mixed on the output display. D-CLEAR allows the user to paritally clear a display, leaving selected data untouched. D-READ and D-POSITION are used to control placement of output on the display screen.

5.13.1 D-CLEAR : Clear Screen From Cursor Position

(---)

This word reads the position of the cursor on the screen and clears from that line to the bottom of the screen, leaving the cursor at the start of the line to which it had been positioned at the start of execution of the word. The cursor should be positioned to the starting line to be cleared before entering the word D-CLEAR. D-CLEAR could be used to prevent the screen from scrolling off the top if there were data on the top of the screen which needed to be examined while commands were being entered.

Example: User positions cursor to proper line and types D-CLEAR. The screen is cleared from that line to the bottom of the screen.

5.13.2 D-READ : Return Position of Cursor on Screen

(--- COLUMN ROW)

Returns the COLUMN number (0-39) and the ROW number (0-24) of the current cursor position on the screen.

Example: D-READ ." ROW=" . ." COLUMN=" . will get the position of the cursor and display its row and column positions.

5.13.3 D-POSITION : Position Cursor On Screen

(COLUMN ROW ---)

Positions the cursor at the given COLUMN number (0-39) and ROW number (0-24) on the screen. Output to the screen will proceed from the position the cursor is left at.

Example: 10 5 D-POSITION ." THIS IS A TEST" will position the cursor to column 10, row 5 and display the string THIS IS A TEST starting there.

5.13.4 D-SPLIT : Split Screen Into Hi-res/Text

(LINE# ---)

The screen splits into two parts, the upper part will be a hi-res (bitmap) display of the currently selected bitmap area. The lower part will be the normal 0 bank text screen. In this way, bitmap graphics commands can be seen executing on the top while being entered in on the bottom of the screen. LINE# is a value which determines at what line the screen will be split.

A value of 0 clears the split-screen mode and resets the 'BANK and 'BITMAP to 0. Thus, 0 D-SPLIT can be used to return to a normal screen when working with hi-res graphics.

Example:

```
20 D-SPLIT
7 BITMAP
```

splits the screen so that 5 text lines remain at the bottom of the screen. The top of the screen displays bitmap area 7 (57344) which resides under the Commodore Kernel ROM.

WARNING: Timing problems in the graphics chip may arise if D-SPLIT is used to change the line split while split screen is in effect. The screen will appear to lock up in bitmap mode. If this occurs, a warm start (RUN-STOP/RESTORE sequence) should bring things back to normal.

5.14 C64 High RAM Access Words

The following words are used to access the memory which is located underneath the C64 Kernel ROM and I/O Memory Map area. This effectively give the user an extra 12k of RAM to use for variable or array storage. The FORTH dictionary itself cannot extend into this area, but constants can be set up and arrays referenced through the constant.

Example: If we want a 4k array at $D000 and an 6k array at $E000 we could define the arrays as follows :

```
HEX  D000 CONSTANT ARRAY1
     E000 CONSTANT ARRAY2
```

Data could be stored at ARRAY1+100 as follows:

```
100 ARRAY1 + H!
```

Since a write will "go through" the Kernel ROM into the RAM below, H! and HC! need not be used to write into that RAM, but H@ and HC@ must be used to fetch the written values.

Note : The last 8 bytes in the 6510 address space, $FFF8-$FFFF, should not be disturbed by the user. They are set up by the system and insure that interrupts (IRQ or NMI) will not cause the system to hang up when the C64 Kernel is swapped out.

5.14.1 H! : Store Value At High RAM Address

(VALUE ADDR ---)

Turns off the 60 cycle clock interrupt, swaps out the C64 Kernel ROM and I/O Memory Map area and stores the 16-bit VALUE at location ADDR. After the store is completed swaps the ROMs back in and turns the clock interrupt back on.

Note: H! need not be used to store into the C64 Kernel area ($E000-$FFFF), however H@ must be used to fetch values which have been stored in RAM under the Kernel and I/O Memory Map areas.

Example: HEX 12AB D500 H! will store the 16-bit hex value 12AB at high memory locations D500 and D501.

5.14.2 H@ : Fetch Value From High RAM Address

(ADDR --- VALUE)

Turns off the 60 cycle clock interrupt, swaps out the C64 Kernel ROM and I/O Memory Map area and fetches the 16-bit VALUE from location ADDR. After the fetch is completed swap the ROMs back in and turn the clock interrupt back on.

Example: HEX D500 H@ . will fetch and display the 16-bit value stored in the RAM located at D500.

5.14.3 HC! : Store Byte At High RAM Address

(BYTE-VALUE ADDR ---)

Turns off the 60 cycle clock interrupt, swaps out the C64 Kernel ROM and I/O Memory Map area and stores the 8-bit VALUE at location ADDR. After the store is completed swap the ROMs back in and turn the clock interrupt back on.

Note: HC! need not be used to store into the C64 Kernel area ($E000-$FFFF), however HC@ must be used to fetch values which have been stored in RAM underneath the Kernel ROM.

Example: HEX 1B D500 HC! will store the 8-bit hex value 1B at high memory location D500.

5.14.4 HC@ : Fetch Byte From High RAM Address

(ADDR --- BYTE-VALUE)

Turns off the 60 cycle clock interrupt, swaps out the C64 Kernel ROM and I/O Memory Map area and fetches the 8-bit VALUE from location ADDR. After the fetch is completed swap the ROMs back in and turn the clock interrupt back on.

Example: HEX D500 HC@ . will fetch the 8-bit value from D500 and display it.

5.15 Data Structure Words

The following words are included not so much as an aid to declaring data (which they are), but to provide the beginning FORTH user with examples of how to define "defining words". As such, the source screens are provided (see Screens, Appendix II).

These words contain no run-time bounds checks although such data validation could be written into the run-time part of the structure at the expense of the extra execution time involved in performing the checks.

As with other defining words, these are invoked by setting up the stack, naming the defining word, and naming the word to be defined.

Example: 15 1ARRAY NEWWORD uses the defining word 1ARRAY to define a 15 element array called NEWWORD.

5.15.1 1ARRAY : One Dimensional Array Structure

```
( #ELEMENTS --- ) Compilation
( ELEMENT# --- ADDR ) Execution
```

This structure allows users to declare an N element one dimensional array. N is the number of 16-bit elements to be allocated during compilation. During execution, the number of the required element is put on the stack and the ADDR of the element is returned. ELEMENT# ranges from 0 to #ELEMENTS-1.

Example:
```
16 1ARRAY SINGLE    ( DECLARES A 16 ELEMENT ARRAY NAMED SINGLE)
-1 8 SINGLE !       ( STORES A -1 IN ELEMENT 8 )
8 SINGLE @ .        ( WILL PRINT A -1 )
```

5.15.2 2ARRAY : Two Dimensional Array Structure

```
( X Y --- ) Compilation
( X# Y# --- ADDR ) Execution
```

This structure allows users to declare an X by Y element two dimensional array. X determines the number of columns and Y determines the number of rows in the matrix. X times Y 16-bit values are allocated during compilation. During execution, the X# and Y# determine the column and row of the element whose ADDRess is returned. X# and Y# range from 0 to X#-1 and 0 to Y#-1 respectively.

Example:
```
4 5 2ARRAY DOUBLE    ( DECLARES A 4 x 5 ARRAY NAMED DOUBLE )
-1 3 2 DOUBLE !      ( STORES -1 IN ELEMENT 3,2 )
3 2 DOUBLE @ .       ( PRINTS A -1 )
```

5.16 Math Extension Words

The math routines were adapted from several articles published in FORTH Dimensions. See the articles (in Appendix VII) for a basic description of usage. Then refer to the appropriate area in this section for extensions which were added for efficiency and utility.

5.16 Math Extension Words

In addition to the words defined in the published article, several words have been added for easier utilization of the floating point system. Probably the greatest change was in allowing direct entry of floating point numbers into the system by redefinition of <NUMBER>. The new definition, <FNUM> is set up by calling FINIT. This allows entry of floating point numbers instead of the integer double precision numbers usually accepted by <NUMBER>. If standard double precision arithmetic is required, FEXIT should be used to exit the floating point system.

Other additions include integer to floating and floating to integer conversion words and floating point sine, cosine and square root words. The word !EXPONENT as described in the article was changed to E! to enable a more familiar entry of scientific notation floating numbers and @EXPONENT was changed to E@ for consistancy. In the examples, FINIT is used only as a reminder. It need only be entered once during floating point arithmetic.

5.16.1.1 <FNUM> : Floating Number Conversion Routine

(ADDR --- F)

This routine replaces the standard system number conversion routine (<NUMBER>) when FINIT is invoked. Single precision input is unaffected, however double precision numbers are automatically converted into floating point, with the decimal point marking what the exponent will be. This provides an easy way to enter floating numbers whose exponent range fits within the standard double precision number format.

Positive double precision integers are identical in either integer or floating point format, however negative numbers differ, therefore this mode cannot be considered compatible with the standard double precision entry of numbers.

Example:

```
FINIT 123.456  23.7892 F+ F.
```

will cause 147.2452 to be output to the display.

5.16.1.2 DFIX : Convert Floating to Double Integer

(F# --- D#)

Converts the floating point number F# to a double precision number D# truncating any fraction.

Example: `FINIT 1234.567 DFIX D.`

Results in "1234" being displayed.

5.16.1.3 DFLOAT : Convert Double Integer to Floating

```
( D# --- F# )
```

Converts the double precision number D# to a floating point number F#.

Example:

```
FINIT 1234. DFLOAT F.
```

Results in "1234." being displayed.

5.16.1.4 E! : Enter Floating Number in Scientific

```
( M E --- )
```

Allows the user to enter a number of the form MANTISSA EXPONENT and converts it to floating point format. M is the double precision signed mantissa. E is the single precision exponent. Regular double precision input must be in effect in order to enter negative numbers.

Example:

```
12345. -25 E!  E.
```

Results in 12345 E-25 being displayed.

5.16.1.5 FCOS : Return Floating Point Cosine

```
( F# --- FCOS )
```

Returns the cosine of the floating point number F#.

Example:

```
FINIT 120. FCOS F.
```

Results in "-0.5000" (the cosine of 120 degrees) to be displayed.

5.16.1.6 FEXIT : Exit Floating Point Mode

```
( --- )
```

Restores the standard system routine (<NUMBER>) to provide numeric conversion. After execution of FEXIT, double precision numbers entered will be kept in the standard double number format.

5.16.1.7 FLOAT : Convert Integer to Floating

(N --- F#)

Converts the single precision number N into a floating point number F#.

Example:

```
FINIT 1234 FLOAT F.
```

Results in "1234." being displayed.

5.16.1.8 FSIN : Return Floating Point Sin of An Angle

(F# --- FSIN)

Returns the floating point sin of a floating point number F#.

Example:

```
FINIT 120. FSIN F.
```

Results in "0.8660" being displayed.

5.16.1.9 FSQRT : Return Floating Square Root

(F# --- FSQRT)

Calculates the floating point square root of a floating point number F#. F# must be in the range 0 <= F# <= 65535.

Example:

```
54321. FSQRT F.
```

Results in "233.06" being displayed.

5.16.2 Trig Extensions

Three levels of integer sine/cosine routines are provided for different requirements of speed vs. flexibility. <SIN> and <COS> are fastest (completely machine language), but can only be used in the range of 0 to 90 degrees. QSIN and QCOS as not as fast but accept values of 0 through 360 degrees. SIN and COS are slowest, but accept negative degrees and degrees > 360.

All sine/cosine values are scaled by 16384. The word TSCALE must be used to bring values back to normal. See the examples for TSCALE usage.

5.16.2.1 <SIN>

(N --- SIN)

Returns the sine of an angle N. N must be in the range of 0 to 90 degrees. The sine is in the range of 0 to 16383. This is a very fast machine language table lookup routine.

Example:

```
45 <SIN> 10000 U* TSCALE .
```

Displays the sine of 45 degrees multiplied by 10000.

5.16.2.2 <COS>

(N --- COS)

Returns the cosine of an angle N. N must be in the range of 0 to 90 degrees. The cosine is in the range of 0 to 16383. This is a very fast machine language table lookup routine.

Example:

```
60 <COS> 10000 U* TSCALE .
```

Displays the cosine of 60 degrees multiplied by 10000.

5.16.2.3 QSIN : Quick Sin Routine

(N --- SIN)

Returns the sine of an angle N. N must be in the range of 0 to 360 degrees. The sine is in the range of 0 to 16383.

Example:

```
240 QSIN 10000 M* TSCALE .
```

Displays the sine of 240 degrees multiplied by 10000.

5.16.2.4 QCOS : Quick Cosine Routine

(N --- COSINE)

Returns the cosine of an angle N. The cosine falls in the range of 0 to 16384. N must be in the range of 0 to 360 degrees.

Example:

```
135 QCOS 10000 M* TSCALE .
```

Displays the cosine of 135 degrees multiplied by 10000.

5.16.2.5 SIN : Sin Routine

(N --- SIN)

Returns the sine of an angle N. N must be in the range of -32768 to 32767 degrees. The sine is in the range of 0 to 16383.

Example:

```
600 SIN 10000 M* TSCALE .
```

Displays the sine of 600 degrees multiplied by 10000.

5.16.2.6 COS : Cosine Routine

(N --- COSINE)

Returns the cosine of an angle N. N must be in the range of -32768 to 32767 degrees. The cosine falls in the range of 0 to 16384.

Example:

```
495 COS 10000 M* TSCALE .
```

Displays the cosine of 495 degrees multiplied by 10000.

6. MVP Standard Word Set Glossary

This glossary contains definitions of the words in the MVP FORTH Standard Word Set. Non-standard MVP words and Commodore 64 extensions are defined in the section titled "Implementation Specific Words".

Since it is expected that a 1541 type drive will be used with a C64, the disk system has been simplified to eliminate the various words, tables and calculations dealing with drive densities (there is only one drive density possible) present in ALL ABOUT FORTH.

The word LIT has been changed to <LIT> to avoid accidently typing LIT interactively (by mistyping LIST for example) thereby crashing the system.

!	(n addr ---) Store n at address.
#	(ud1 --- ud2) Generate the next ASCII character which is placed in an output string. The result ud2 is the quotient after division by BASE and is maintained for further processing. Used between <# and #>.
#>	(ud --- addr n) End pictured numeric output conversion. Drop ud, leaving the text address, and character count, suitable for TYPE.
#BUFF	(--- n) A constant returning the number of disk buffers allocated. For the disk I-O routines to work correctly #BUFF must be greater than 1.
#S	(ud --- 0 0) Convert digits of unsigned 32-bit number ud, adding each to the pictured numeric output text, until remainder is zero. A single zero is added to the output string if the number was initially zero. Use only between <# and #>.
' (tick)	(--- addr) If executing, leave PFA of next word accepted from input stream. If compiling, compile address as a literal; later execution will place this value on the stack.
'-FIND	(--- addr) A user variable containing the address to be executed by -FIND.
'?TERMINAL	(--- f) A user variable containing the compilation address to be executed by ?TERMINAL.
'ABORT	(--- addr) A user variable containing the compilation address to be executed by ABORT.

'BLOCK	(--- addr) A user variable containing the compilation address to be executed by BLOCK.
'CR	(--- addr) A user variable containing the compilation address to be executed by CR.
'EMIT	(--- addr) A user variable containing the compilation address to be executed by EMIT.
'EXPECT	(--- addr) A user variable containing the compilation address to be executed by EXPECT.
'INTERPRET	(--- addr) A user variable containing the compilation address to be executed by INTERPRET.
'KEY	(--- addr) A user variable containing the compilation address to be executed by KEY.
'LOAD	(--- addr) A user variable containing the compilation address to be executed by LOAD.
'NUMBER	(--- addr) A user variable containing the compilation address to be executed by NUMBER.
'PAGE	(--- addr) A user variable containing the compilation address to be executed by PAGE.
'R/W	(--- addr) A user variable containing the compilation address to be executed by R/W.
'S	(--- addr) Place the address of the top of the stack on the top of the stack.
'STREAM	(--- addr) Returns the address of the next character in the input stream.
'T&SCALC	(--- addr) A user variable containing the compilation address to be executed by T&SCALC.
'TITLE	(--- addr) A variable holding the compilation address executed by TRIAD to place a message at the bottom of each page (UTILITY).
'VOCABULARY	(--- addr) A user variable containing the compilation address to be executed by VOCABULARY.
'WORD	(--- addr) A user variable containing the compilation address to be executed by WORD.
(	(---) Accept and ignore comment characters from the input stream until the next right parenthesis. Used to comment FORTH screens and definitions.

*	(n1 n2 --- n3) Leave the arithmetic product of n1 times n2.
*/	(n1 n2 n3 --- n4) Multiply n1 by n2, divide the result by n3 and leave the quotient n4. n4 is rounded toward zero.
*/MOD	(n1 n2 n3 --- n4 n5) Multiply n1 by n2, divide the result by n3 and leave the remainder n4 and quotient n5. The remainder has the same sign as n1. Used for scaling and rounding.
+	(n1 n2 --- n3) Leave the arithmetic sum of n1 plus n2.
+!	(n addr ---) Add n to the 16-bit value at he address, by the convention given for +.
+-	(n1 n2 --- n3) Apply the sign of n2 to n1, which is left as n3.
+BUF	(addr1 --- addr2 f) Advance the disk buffer address addr1 to the address of the next buffer addr2. Boolean f is false when addr2 is the buffer presently pointed to by the variable PREV.
+LOOP	(n ---) Add the signed increment n to the loop index and compare the total to the limit. Return execution to the corresponding DO until the new index is equal to or greater than the limit (n>0) or until the new index is less than the limit (n<0). Upon exiting from the loop, discard the loop control parameters.
, (comma)	(n ---) Allot two bytes in the dictionary, storing n there.
-	(n1 n2 --- n3) Subtract n2 from n1 and leave the difference n3.
-->	(---) Continues interpretation (LOADing) from the next sequential screen number. This word can be used for a word definition which will not fit on a single screen. All information following --> on the screen will be skipped in the loading.

Note: This word should NOT be used on a screen which is being loaded using THRU.

-FIND	(--- pfa b tf) if found (--- ff) if not found-Accepts the next word (delimited by blanks) in the input stream to HERE and searches the CONTEXT and then the FORTH vocabularies for a matching entry. If found, the dictionary entry's parameter field address, length byte and a boolean true is left. Otherwise, a boolean false is left.

-TEXT	(addrl nl addr2 --- f) Compare two strings over the length nl beginning at addrl and addr2. Return zero if the strings are equal. If unequal, return 1 if the string at addrl > the string at addr2, or -1 if the string at addrl < the string at addr2. The comparison is performed a byte at a time.
-TRAILING	(addr nl --- addr n2) Adjust character count nl of a text string beginning at addr to exclude trailing blanks, i.e., the characters at the addr+n2 to addr+nl-1 are blanks. An error exists if nl is negative.
.	(n ---) Display n converted according to BASE in a free-field format with one trailing blank. Display only a negative sign.
."	(---) Interpreted or used in a colon-definition. Accept the following text from the input stream, terminated by " (double quote). If executing, transmit this text to the selected output device. If compiling, compile so that later execution will transmit the text to the selected output device. At least 127 characters are allowed in the text. If the input stream is exhausted before the terminating double-quote, an error condition exists.
.INDEX	(n ---) Print line 0 on screen n.
.LINE	(line scr ---) Print on the terminal device, a line of text from the disk by its line and screen number. Trailing blanks are suppressed.
.R	(nl n2 ---) Print nl right aligned in a field of n2 characters, according to BASE. If n2 is less than 1, no leading blanks are supplied.
.S .SL .SR	(---)
.SS	(--- FLAG) These words work in concert to implement nondestructive stack display. .S will print the values on the stack in ascending or descending order according to the flag in the constant .SS. The flag is set by .SL and .SR. .SL causes the stack to be displayed from the most recent entry through the oldest entry, while .SR causes the stack to be displayed from the oldest to the most recent entry (default).
.R	(nl n2 ---) Print nl right aligned in a field of n2 characters, according to BASE. If n2 is less than 1, no leading blanks are supplied.
/	(nl n2 --- n3) Divide nl by n2 and leave the quotient n3. n3 is rounded toward zero.
/LOOP	(n---) A DO-LOOP terminating word. The loop index is incremented by the unsigned magnitude of n. Until the

	resultant index exceeds the limit, execution returns to just after the corresponding DO, otherwise, the index and limit are discarded. Magnitude Logic is used.
/MOD	(n1 n2 --- n3 n4) Divide n1 by n2 and leave the remainder n3 and quotient n4. n3 has the same sign as n1.
0	(--- 0) The value is defined as an ideogram.
0<	(n --- flag) True if n is less than zero (negative).
0=	(n --- flag) True is n is zero.
0>	(n --- flag) True is n is greater than zero.
0BRANCH	(f ---) The run-time procedure to conditionally branch. If f is false (zero), the following in-line parameter is added to the interpretive pointer to branch ahead or back. Compiled by IF, Until, and While.
1	(--- 1) A common integer defined as a constant.
1+	(n --- n+1) Increment n by one, according to the operation for +.
1-	(n --- n-1) Decrement n by one, according to the operation -.
2	(--- 2) A common integer value defined as a constant.
2!	(d addr ---) Stored in 4 consecutive bytes beginning at addr, as for a double number.
2*	(n1 -- n2) Leave 2*(n1).
2+	(n --- n+2) Increment n by two, according to the operation for +.
2-	(n --- n-2) Decrement n by two, according to the operation for -.
2/	(n1 --- n2) Leave (n1)/2.
2@	(addr --- d) Leave on the stack the contents of the four consecutive bytes beginning at addr, as for a double number.
2CONSTANT	(d ---) A defining word used to create a dictionary entry for <name> is later executed, d will be left on the stack.
2DROP	(d ---) Drop the top double number on the stack.
2DUP	(d --- d d) Duplicate the top double number on the stack.

2OVER	(d1 d2 --- d1 d2 d1) Leave a copy of the second double number the stack.
2SWAP	(d1 d2 --- d2 d1) Exchange the top double number on the stack.
2VARIABLE	(---) A defining word used to create a dictonary entry of <name> and assign 4 bytes for storage in the parameter field. When <name> is later executed, it will leave the address of the first byte of its parameter field on hte stack.
79-STANDARD	(---) Execute assuring that a FORTH-79 Standard system is available, otherwise an error condition exists.
:	(---) A defining word which selects the CONTEXT vocabulary to be identical to CURRENT. Create a dictionary entry for <name> in CURRENT, and set compile mode. Words thus defined are called 'colon-definitions' The compilation addresses of subsequent words from the input stream which are not immediate words are stored into the dictionary to be executed when <name> is later executed. IMMEDIATE words are executed as encountered. If a word is not found after a search of the CONTEXT and FORTH vocabularies, conversion compilation of a lateral number is attempted, with regard to the current BASE; that failing, an error condition exists.
;	(---) Terminate a colon-definition and stop compilation. If compiling from mass storage and the input stream is exhausted before encountering ; an error condition exists.
;CODE	(---) Stop compilation and terminate a defining word. ASSEMBLER becomes the CONTEXT vocabulary.
<	(n1 n2 --- flag) True if n1 is less than n2.
<#	(d1 --- d1) Initialized pictured numeric output. The ideograms <#, #, #S, HOLD, SIGN, and #> can be used to specify the conversion of a double-precision number into an ASCII character string stored in right-to-left order.
<+LOOP>	(n ---) The run-time procedure compiled by +LOOP.
<-FIND>	(--- pfa b tf) if found (--- ff) if not found: A run-time procedure compiled by <-FIND>.
<.">	(---) A run-time procedure, compiled by ." which transmits the following in-line text to the selected output device.
</LOOP>	(u ---) The run-time procedure compiled by /LOOP.
<;CODE>	(---) The run-time procedure compiled by ;CODE.

<<CMOVE>	(addr1 addr2 u ---) The run-time procedure compiled by <CMOVE.
<?TERMINAL>	(--- f) The run-time procedure compiled by ?TERMINAL.
<ABORT">	(f ---) The run-time procedure compiled by ABORT".
<ABORT>	(---) The run-time procedure compiled by ABORT.
<BLOCK>	(n --- addr) The run-time procedure compiled by BLOCK.
<CMOVE>	(addr1 addr2 n ---) The primitive code routine for CMOVE and MOVE. Up to 65,535 bytes may be moved. Nothing is moved when u=0.
<CR>	(---) The run-time procedure compiled by CR.
<DO>	(n1 n2 ---) The run-time procedure compiled by DO.
<EMIT>	(c ---) The run-time procedure compiled by EMIT.
<EXPECT>	(addr n ---) The run-time procedure compiled by EXPECT.
<FILL>	(addr n b ---) The run-time procedure compiled by FILL.
<FIND>	(addr1 addr2 --- pfa b tf) if found (addr1 addr2 --- ff) if not found: Searches the dictionary starting at the name field address addr2, matching to the text at addr1. Returns parameter field address, length byte of name field and boolean true for a good match. If no match is found, only a boolean false is left.
<INTERPRET>	(---) The run-time procedure compiled by INTERPRET.
<KEY>	(--- char) The run-time procedure compiled by KEY.
<LINE>	(n1 n2 --- addr count) The run-time procedure compiled by LINE.
<LIT>	(--- n) Within a colon-definition, <LIT> is automatically compiled before each 16-bit literal number encountered in input text. Later execution of LIT causes the contents of the next dictionary address to be pushed to the stack.
<LOAD>	(n ---) The run-time procedure compiled by LOAD.
<LOOP>	(---) The run-time procedure compiled by LOOP.
<NUMBER>	(addr --- d) The run-time procedure compiled by NUMBER.
<PAGE>	(---) The run-time procedure compiled by PAGE.
<R/W>	(addr blk f ---) The run-time procedure compiled by R/W.

<T&SCALC>	(n --- d s t) Track & Sector and drive calculation for disk I/O. n is the total sector displacement. The corresponding drive (d), track (t) and sector numbers are calculated. The track number is stored in TRACK; the sector number is stored is SEC.
<VOCABULARY79>	(---) The run-time routine for a defining word to create (in the CURRENT vocabulary) a dictionary entry for <name>, which specifies a new ordered list of word definitions. Subsequent execution of <name> will make it the CONTEXT vocabulary. When <name> becomes the CURRENT vocabulary (see DEFINITIONS), new definitions will be created in that list. In lieu of any further specifications, new vocabularies 'chain' to FORTH. That is, when a dictionary search through vocabulary is exhausted, FORTH will be searched. When VOCABULARY is vectored toward this word (the system default) the implementation produces the correct run-time procedure according to the function described in the FORTH-79 STANDARD.
<VOCABULARYFIG>	(---) Vectoring VOCABULARY to this word will produce the run-time procedure for VOCABULARY according to the FIG-FORTH definition.
<WORD>	(char --- addr) The run-time procedure compiled by WORD.
=	(n1 n2 --- flag) True if n1 is equal to n2.
>	(n1 n2 --- flag) True if n1 is greater than n2.
>BINARY	(d1 addr1 --- d2 addr2) Same as CONVERT.
>IN	(--- addr) Leave the address of a variable which contains the present character offset within the input stream.
?	(addr ---) Display the number at address, using the format of ".".
>R	(n ---) Transfer n to the return stack. Every >R must be balanced by a R> in the same control structure nesting level of a colon-definition.
?COMP	(---) Issue an error message if not compiling.
?CONFIGURE	(---) Display the current configuration for all available disk drives.
?CSP	(---) Issue an error message if stack position differs from value saved in CSP.
?DUP	(n --- n n) Duplicate n if it is non-zero.

?PAIRS	(n1 n2 ---) Issue an error message if n1 does not equal n2. Indicates that compiled conditionals do not match.
?STACK	(---) Issue an error message if the stack is out of bounds.
?STREAM	(f ---) Issue an error message if the flag is true, indicating that the input stream is exhausted.
@	(addr --- n) Leave on the stack the number contained at addr.
ABORT	(---) Clear the data and return stacks, setting execution mode. Return control to the terminal.
ABORT"	(flag ---) Used in a colon-definition. If the flag is true, print the following text, till ". Then execute ABORT.
ABS	(n1 --- n2) Leave the absolute value of a number.
AGAIN	(addr n ---) Compiling (---) Run-time-Effect an unconditional jump back to the start of a BEGIN-AGAIN loop.
ALLOT	(n ---) Add n bytes to the parameter field of the most recently defined word.
AND	(n1 n2 --- n3) Leave the bitwise logical 'and' of n1 and n2.
ASSEMBLER	(---) Sets CONTEXT vocabulary to ASSEMBLER.
BASE	(--- addr) Leave the address of a variable containing the current input-output numeric conversion base.
BEGIN	(--- addr n) Used in a colon definition. BEGIN marks the start of a word sequence for repetitive execution. A BEGIN-UNTIL loop will be repeated until flag is true. A BEGIN-WHILE-REPEAT loop will be repeated until flag is false. The ideograms after UNTIL or REPEAT will be executed when either loop is finished.
BL	(--- c) A constant that leaves the ASCII value for "blank" on the stack.
BLANK	(addr n ---) Fill in an area of memory over n bytes with the value for ASCII blank, starting at addr . If n is less than or equal to zero, take no action.
BLK	(--- addr) Leave the address of a variable containing the number of the mass storage block being interpreted as the input stream. If the content is zero, the input stream is

	taken from the terminal. The value of the variable is an unsigned number.
BLOCK	(n --- addr) Leave the address of the first byte in block n. If the block is not already in memory, it is transferred from mass storage into whichever memory buffer has been least recently accessed. If the block occupying that buffer has been UPDATED (i.e., modified), it is rewritten onto mass storage before block n is read into the buffer. n is an unsigned number. If correct mass storage read or write is not possible, an error condition exists. Only data, within the latest block referenced by BLOCK is valid by byte address, due to sharing of the block buffers.
BMOVE	(addr1 addr2 n ---) Move n bytes beginning at address addr1 to addr2. Perform the operation correctly even if the ranges involved overlap.
BRANCH	(---) The run-time procedure to unconditionally branch. An in-line offset is added to the interpretive pointer IP to branch ahead or back. BRANCH is compiled by ELSE, AGAIN, REPEAT.
BUFFER	(n --- addr) Obtain the next block buffer, assigning it to block n. The block is not read from mass storage. In standard mode, if the previous contents of the buffer has been marked as UPDATED, it is written to mass storage. If correct writing to mass storage is not possible, an error condition exists. In file mode, if no empty buffers exist, the operation aborts with the message "BUFFERS FULL". The address left is the first byte within the buffer for data storage. n is an unsigned number.
BYE	(---) save FORTH and perform a system cold start, returning to the C-64 kernal and restoring the BASIC ROM.
C!	(n addr ---) Store the least significant 8-bits of n at addr .
C,	(n ---) Store the low order 8 bits of n at the next byte in the dictionary, advancing the dictionary pointer.
C/L	(--- n) Constant leaving the number of characters per line; used by the editor.
C@	(addr --- byte) Leave on the stack in the contents of the byte at addr (with higher bits zero, in a 16-bit field.)
CFA	(pfa --- cfa) Convert the parameter field address of a definition to its code field address.
CHANGE	(---) Modify the size of your FORTH image and the number

of buffers in use according to the current values of LIMIT and #BUFF. This word can be used to move the FORTH User area. LIMIT should be set to the new User area.

CHANGE executes a cold start, so buffers will be emptied and stack values will be lost.

CLEAR	(n ---) Clear screen n to all blanks.
CMOVE	(addr1 addr2 n ---) Move n bytes beginning at address addr1 to addr2. The contents of addr1 is moved first proceeding toward high memory. If n is zero or negative nothing is moved.
COLD	(---) The cold start procedure to adjust the dictionary pointer to the minimum standard and restart via ABORT. May be called from the terminal to remove application programs and restart.
COMPILE	(---) When a word containing COMPILE executes, the 16-bit value following the compilation address of COMPILE is copied (compiled) into the dictionary. i.e., COMPILE DUP will copy the compilation address of DUP.
CONFIGURE	(---) Lets you change the number of drives available on the system. The configuration is initially set for one drive.
CONSTANT	(n ---) A defining word to create a dictionary entry for <name>, leaving n in its parameter field. When <name> is later executed, n will be left on the stack.
CONTEXT	(--- addr) Leave the address of a variable specifying the vocabulary in which dictionary searches are to be made, during interpretation of the input stream.
CONVERT	(d1 addr1 --- d2 addr2) Convert to the equivalent stack number the text beginning at addr1+1 with regard to BASE. The new value is accumulated into double number d1, being left as d2. addr2 is the address of the first non-convertable character.
COPY	(n1 n2 ---) Copy the contents of screen n1 to screen n2.
COUNT	(addr --- addr+1 n) Leave the address addr+1 and the character count of text beginning at addr. The first byte at addr must contain the character count n. Range of n is 0..255.
CR	(---) Cause a carriage-return and line-feed to occur on the current output device.

CREATE	(---) A defining word used to create a dictionary entry for <name> without allocating any parameter field memory. When <name> is subsequently executed, the address of the first byte of <name>'s parameter field is left on the stack.
CSP	(--- addr) A user variable temporarily storing the stack pointer position, for compilation error checking.
CURRENT	(--- addr) Leave the address of a variable specifying the vocabulary into which new word definitions are to be entered.
D!	(d addr ---) Same as 2!.
D+	(d1 d2 --- d3) Leave the arithmetic sum of d1 plus d2.
D+-	(d1 n --- d2) Apply the sign of n to the double number d1, leaving it as d2.
D-	(d1 d2 --- d3) Subtract d2 from d1 and leave the difference d3.
D.	(d ---) Display d converted according to BASE in a free-field format, with one trailing blank. Display the sign only if negative.
D.R	(d n ---) Display d converted according to BASE, right aligned in an n character field. Display the sign only if negative.
D0=	(d --- flag) Leave true if d is zero.
D<	(d1 d2 --- flag) True if d1 is less than d2.
D=	(d1 d2 --- flag) True if d1 equals d2.
D>	(d1 d2 --- f) True if d1 is less than d2.
D@	(addr --- d) Same as 2@.
DABS	(d1 --- d2) Leave as a positive double number d2, the absolute value of a double number, d1. Range 0..2,147,483,647.
DCONSTANT	(---) Same as 2CONSTANT.
DDROP	(d ---) Same as 2DROP.
DDUP	(d --- d d) Same as 2DUP.
DECIMAL	(---) Set the input-output numeric conversion base to ten.
DEFINITIONS	(---) Set CURRENT to the CONTEXT vocabulary so that

	subsequent defintions will be created in the vocabulary previously selected as CONTEXT.
DEPTH	(--- n) Leave the number of the quantity of 16-bit values contained in the data stack, before n was added.
DIGIT	(c n1 --- n2 tf) (c n1 --- ff) bad-Converts the ASCII character c (using base n1) to its binary equivalent n2, accompanied by a true flag. If the conversion is invalid, leaves only a false flag.
DLITERAL	(d --- d) [executing] (d ---) [compiling] : If compiling, compile a stack double number into a literal. Later execution of the defintion containing this literal will push it to the stack. If executing, the number will remain on the stack.
DMAX	(d1 d2 --- d3) Leave the larger of two double numbers.
DMIN	(d1 d2 --- d3) Leave the smaller of two double numbers.
DNEGATE	(d1 --- -d1) Leave the two's complement of a double number.
DO	(n1 n2 ---) Use only in a colon-definition. Begin a loop which will terminate based on control parameters. The loop index begins at n2, and terminates based on the limit n1. At LOOP or +LOOP, the index is modified by a positive or negative value. The range of a DO-LOOP is determined by the terminating word. DO-LOOP may be nested. Capacity for three levels of nesting is specified as a minimum for standard systems.
DOES>	(---) Define the run-time action of a word created by a high-level defining word. It marks the termination of the defining part of the defining word <name> and begins the definition of the run time action for words that will later be defined by <name>. On execution of <namex> the sequence of words between DOES> and ; will be executed, with the address of <name>'s parameter field on the stack.
DOVER	(d1 d2 --- d2 d1) Leave a copy fo the second double number on the stack.
DP	(--- addr) A user variable, the dictionary pointer, which contains the address of the next free memory above the dictionary. The value may be read by HERE and altered by ALLOT.
DPL	(--- addr) A user variable containing the number of digits to the right of the decimal on double integer input. Defaults to -1 on single number input.

DR0 , DR1 , DR2 , DR3 , DR4	(---) Commands to select disk drives, by presetting OFFSET. The contents of OFFSET is added to the block number in BLOCK to allow for this selection.
DROP	(n ---) Drop the top number from the stack.
DSWAP	(d1 d2 --- d2 d1) Exchange the top two double numbers on the stack.
DU<	(ud1 ud2 --- flag) True if ud1 is less than ud2. Both numbers are unsigned.
DUMP	(addr n ---) List the contents of n addresses starting at addr. Each line of values may be preceded by the address of the first value.
DUP	(n --- n n) Leave a copy of the top stack number.
DVARIABLE	(---) A defining word used to create a dictionary entry of <name> and assign 4 bytes for storage in the parameter field. When <name> is later executed, it will leave the address of the first byte of its parameter field on the stack.
ELSE	(---) Used in a colon-definition and executes after the true part following IF. ELSE forces execution to skip till just after THEN. It has no effect on the stack. (See IF).
EMIT	(c ---) Transmit a charcter to the current output device.
EMPTY	(---) Forget all new words added to the dictionary by the user. This is useful for correctly adjusting the dictionary pointer after an aborted entry of a definition, or after warm starting without using FORGET to drop unwanted definitions.
EMPTY-BUFFERS	(---) Mark all block buffers as empty. The block buffers are set to blanks. UPDATED blocks are not written to mass storage.
ENCLOSE	(addr1 c --- addr1 n1 n2 n3) The text scanning primitive used by WORD. From the text address addr1 and an ASCII delimiting character c, is determined the byte offset to the first non-delimiter character n1, the offset to the first delimiter after the text n2, and the offset to the first character not included n3.
ERASE	(addr n ---) Clear a region of memory to zero from addr over n addresses.
EXECUTE	(addr ---) Execute the dictionary entry whose compilation address is on the stack.
EXIT	(---) When compiled within a colon-definition, terminate

	execution of that definition, at that point. May not be used within a DO...LOOP.
EXPECT	(addr n ---) Transfer characters from the terminal beginning at addr, upward, until a "return" or the count of n has been received. Take no action for n less than or equal to zero. One or two nulls are added at the end of the text.
FENCE	(--- addr) A user variable containing an address below which FORGETting is trapped. To forget below this point the user must alter the contents of FENCE.
FILL	(addr n byte ---) Fill memory beginning at address with a sequence of n copies of byte. If the quantity is less than or equal to zero, take no action.
FIND	(--- addr) Leave the compilation address of the next word name which is accepted from the input stream. Leave zero if the word cannot be found.
FIRST	(--- n) A constant that leaves the address of the first (lowest) block buffer.
FLD	(--- addr) A variable pointing to the field length reserved for a number during output conversion.
FLUSH	(---) A synonym for SAVE-BUFFERS.
FORGET	(---) Delete from the dictionary <name> (which is in the CURRENT vocabulary) and all words added to the dictionary after <name>, regardless of their vocabulary. Failure to find <name> in CURRENT or FORTH is an error condition.
FORTH	(---) The name of the primary vocabulary. Execution makes FORTH the CONTEXT vocabulary. New defintions become a part of FORTH until a differing CURRENT vocabulary is established. User vocabularies conclude by 'chaining' to FORTH, so it should be considered that FORTH is 'contained' within each users' vocabulary.
FREEZE	(---) Save the current values of the user variables and the top of the dictionary in low memory in place of the original values.
GO	(addr ---) Makes the address on the stack the next address in the hardware program counter.
H	(--- addr) A synonym for DP, the dictionary pointer.
HERE	(--- addr) Return the address of the next available dictionary location.

HEX	(---) Set the numeric conversion base to sixteen (hexadecimal).
HLD	(--- addr) A user variable that holds the address of the latest character of text during numeric output conversion.
HOLD	(char ---) Insert char into a pictured numeric output string. May only be used between <# and #>.
I	(--- n) Copy the loop index onto the data stack. May be only used in the DO-LOOP control structure.
I'	(--- n) Used within a colon-definition executed only from within a DO-LOOP to return the corresponding loop index.
ID.	(nfa ---) Print a definition's name from its name field address.
IF	(flag ---) Used only in a colon-definition. If flag is true, the words following IF are executed and the words following ELSE are skipped. The ELSE part is optional. If flag is false, words between IF and ELSE , or between IF and THEN (when no ELSE is used), are skipped. IF-ELSE-THEN conditionals may be nested.
IMMEDIATE	(---) Mark the most recently made dictionary entry as a word which will be executed when encountered during compilation rather than compiled.
INDEX	(from to ---) Print the first line of each screen over the range from, to. This is used to view the comment lines of an area of text on disk screens.

This word incorporates a PAUSE feature, which holds the display still when any key is pressed. Once suspended, the INDEX may be resumed by striking any key once, or aborted by striking any two keys in rapid succession.

INIT-FORTH	(--- addr) A constant locating the bootup parameter used to initialize the FORTH vocabulary.
INIT-USER	(--- addr) A constant returning a pointer to the start of the bootup parameter area in low memory. This area is an array containing cold-start values for the user variables.
INTERPRET	(---) Begin interpretation at the character indexed by the contents of >IN relative to the block number contained in BLK, continuing until the input stream is exhausted. If BLK contains zero, interpret characters from the terminal input buffer.
J	(--- n) Return the index of the next outer loop. May be used only within a nested DO-LOOP.

KEY	(--- char) Leave the ASCII value of the next available character from the current input device.
LATEST	(--- addr) Leave the name field address of the topmost word in the CURRENT vocabulary.
LEAVE	(---) Force termination of a DO-LOOP at the next LOOP or +LOOP by sertting the loop limit equal to the current value of the index. The index itself remains unchanged, and execution proceeds normally until the loop terminating word is encountered.
LFA	(pfa --- lfa) Convert the parameter field address of a dictionary definition to its link field address.
LIMIT	(--- n) A constant leaving the address just above the highest memory available for a disk buffer.
LIST	(n ---) List the ASCII symbolic contents of screen n on the current output device, setting SCR to contain n and setting FLAST if n is greater than FLAST. A single keystroke while the screen is LISTing will terminate the LIST.
LIT	Changed to <LIT> to avoid crashing the system by accidental typing of LIT from the keyboard (this can easily happen while typing LIST, for instance).
LITERAL	(n ---) If compiling, then compile the stack value n as a 16-bit literal, which when later executed, will leave n on the stack.
LOAD	(n ---) Begin interpretation of screen n by making it the input stream; preserve the locators of the present input stream (from >IN and BLK). If interpretation is not terminated explicitly it will be terminated when the input stream is exhausted. Control then returns to the input stream containing LOAD, determined by the input stream locators >IN and BLK. Note that screen 0 is unloadable.
LOOP	(---) Increment the DO-LOOP index by one, terminating the loop if the new index is equal to or greater than the limit. The limit and index are signed numbers in the range -32,768..32,767.
M*	(n1 n2 --- d) A mixed magnitude math operation which leaves the double number signed product to two signed numbers.
M*/	(d1 n1 n2 --- d2) Multipies d1 by n1 and divides the triple precision product by n2 leaving the quotient d2. All values are signed.

M+	(d1 n --- d2) Add d1 to n and return d2. Note all values are signed.
M/	(d n1 --- n2 n3) A mixed magnitude math operation which leaves the signed remainder n2 and signed quotient n3, from a double number dividend and divisor n1. The remainder takes its sign from the dividend.
M/MOD	(ud1 u2 --- u3 ud4) An unsigned mixed magnitude math operation which leaves a double quotient ud4 and remainder u3, from a double dividend ud1 and single precision divisor u2.
MAX	(n1 n2 --- n3) Leave the greater of two numbers.
MAX-DRV	(--- n) A constant which returns the current maximum number of drives.
MIN	(n1 n2 --- n3) Leave the lesser of two numbers.
MOD	(n1 n2 --- n3) Divide n1 by n2, leaving the remainder n3, with the same sign as n1.
MOVE	(addr1 addr2 n ---) Move the specified quantity n of 16-bit memory cells beginning at addr1 into addr2. The contents of addr1 is moved first. If n is negative or zero, nothing is moved.
NEGATE	(n --- -n) Leave the two's complement of a number, i.e., the difference of 0 less n.
NEXT	(---) An assembler constant pointing to the machine code entry point of the inner interpreter.
NFA	(pfa --- nfa) Convert the parameter field address of a definition to its name field.
NOT	(flag1 --- flag2) Reverse the boolean value of flag1. This is identical to 0=.
NUMBER	(addr --- d) Convert the count and character string at addr, to a signed 32-bit integer, using the current base. If numeric conversion is not possible, an error condition exists.

This implementation follows the Starting Forth version. It allows the following characters to be used as punctuation signalling entry of a double precision number: '.' ',' '/' '-' and ':'.

OFFSET	(--- addr) A variable that contains the offset added to the block number on the stack by BLOCK to determine the actual physical block number. The user must add any desired offset when utilizing BUFFER.

OR	(n1 n2 --- n3) Leave the bitwise inclusive-or of two numbers.
OUT	(--- addr) A user variable that contains a value incremented by EMIT. The user may alter and examine OUT to control display formatting.
OVER	(n1 n2 --- n1 n2 n1) Leave a copy of the second number on the stack.
PAD	(--- addr) The address of a scratch area used to hold character strings for imtermediate processing. The minimal capacity of PAD is 64 characters. (addr through addr+63)
PAGE	(---) Clear the terminal screen or perform an action suitable to the output device currently active.
PAUSE	(---) Test the terminal keyboard for actuation of any key. If true, wait until a key has been pressed again.
PFA	(nfa --- pfa) Convert the name field address of a compiled definition to its parameter field address.
PICK	(n1 --- n2) Return the contents of the n1-th stack value, not counting n1 itself. An error condition results for n less than one.
PP	(n --- <text>) On the latest screen listed, put <text> on line n. This ideogram makes it possible to enter new source text on a screen without use of an EDITOR.
PREV	(--- addr) A variable containing the address of the disk buffer most recently referenced.
QUERY	(---) Accept input of up to 80 characters (or until a carriage return) from the operator's terminal, into the terminal input buffer.
QUIT	(---) Clear the return stack, setting execution mode, and return control to the terminal. No message is given.
R#	(--- addr) A user variable which contains the current editing line in the current screen.
R/W	(ADDR BLK F ---) The fig-FORTH standard disk read-write linkage. addr specifies the source or destination buffer, blk is the sequential number of the referenced block; and f is a flag for f=0: write and f=1: read.
R0	(--- addr) A user variable containing the initial location of the return stack.

R>	(--- n) Transfer n from the return stack to the data stack.
R@	(--- n) Copy the number on the top of the return stack to the data stack.
REPEAT	(---) Used in a colon-definition. At run-time, REPEAT returns to just after the corresponding BEGIN.
ROLL	(n ---) Extract the n-th stack value to the top of the stack, not counting n itself, moving the remaining values into the vacated position. An error condition results for n less than one.
ROT	(n1 n2 n3 --- n2 n3 n1) Rotate the top three values, bringing the deepest to the top.
RP!	(---) A procedure to initialize the return stack pointer from the variable R0.
RP@	(--- addr) Leaves the current value in the return stack pointer register.
S->D	(n --- d) Sign extend a single number to form a double number.
S0	(--- addr) Returns the address of the bottom of the stack, when empty.
SAVE-BUFFERS	(---) Write all blocks to mass-storage that have been flagged as UPDATEd. An error condition results if mass-storage writing is not completed.
SCR	(--- addr) Leave the address of a variable containing the number of the screen most recently listed. The value of the variable is unsigned.
SEC/BLK	(--- addr) A variable containing the number of sectors in a block.
SET-DRX	(n ---) For drive number n, calculates and adds the necessary value to OFFSET.
SIGN	(n ---) Insert the ASCII "-" (minus sign) into the pictured numeric output string, if n is negative.
SMUDGE	(---) Used during word definition to toggle the "smudge bit" in a definition's name field.
SP!	(---) A procedure to initialize the stack pointer from S0.
SP0	(--- addr) A user variable that contains the initial value of the stack pointer.

SP@	(--- addr) Return the address of the top of the stack, just before SP@ was executed.
SPACE	(---) Transmit an ASCII blank to the current output device.
SPACES	(n ---) Transmit n spaces to the current output device. Take no action for n of zero or less.
STATE	(--- addr) Leave the address of the variable containing the compilation state. A non-zero content indicates compilation is occurring.
SWAP	(n1 n2 --- n2 n1) Exchange the top two stack values.
T&SCALC	(rel.sect --- drive sector track) Performs physical drive, sector and track calculations given a relative sector number. rel.sect is the total sector displacement from the first logical drive to the desired sector.
TEXT	(c ---) Accept characters from the input stream, as for WORD, into PAD, blank filling the remainder of PAD to 64 characters.
THEN	(---) Used in a colon-definition. THEN is the point where execution resumes after ELSE or IF (when no ELSE is present).
THRU	(n1 n2 ---) Load consecutively the blocks from n1 through n2.
TIB	(--- addr) A user variable containing the address of the terminal input buffer.
TITLE	(---) Print a fixed message followed by a carriage return.
TOGGLE	(addr b ---) Complement the contents of addr by the bit pattern b.
TRAVERSE	(addr1 n --- addr2) Move across the field of a fig-FORTH variable length dictionary header. addr1 is the address of either the length byte or the last letter. If n=1, the motion is toward high memory; if n = -1, the motion is toward low memory. The addr2 resulting is the address of the other end of the name.
TRIAD	(scr ---) Display on the selected output device the three screens which include that numbered scr, beginning with a screen evenly divisible by three.
TYPE	(addr n ---) Transmit n characters beginning at address to the current output device. No action takes place for n less than or equal to zero.

U*	(un1 un2 --- un3) Perform an unsigned multiplication of un1 by un2, leaving the double number product un3. All values are unsigned.
U.	(un ---) Display un converted according to BASE as an unsigned number, in a free-field format, with one trailing blank.
U.R	(un1 n2 ---) Output un1 as an unsigned number right justified in a field n2 characters wide. If n2 is smaller than the characters required for n1, no leading spaces are given.
U/MOD	(ud1 un2 --- un3 un4) Perform the unsigned division of double number ud1 by un2, leaving the remainder un3, and quotient un4. All values are unsigned.
U<	(un1 un2 --- flag) Leave the flag representing the magnitude comparison of un1 < un2 where un1 and un2 are treated as 16-bit unsigned integers.
UNTIL	(addr n ---) [compiling] (f1 ---) [executing] : Within a colon-definition, mark the end of a BEGIN-UNTIL loop, which will terminate based on a flag. If flag is true, the loop is terminated. If flag is false, execution returns to the first word after BEGIN. BEGIN-UNTIL structures may be nested.
UP	(--- addr) A constant returning a pointer to the cell in low memory which holds the pointer to the user area.
UPDATE	(---) Mark the most recently referenced block is modified. The block will subsequently be automatically transferred to mass storage of a different block, or upon execution of SAVE-BUFFERS.
USE	(--- addr) A variable containing the address of the block buffer to use next, as the least recently written.
USER	(n ---) A defining word which creates a user variable <name>. n is the cell offset within the user area where the value for <name> is stored. Execution of <name> leaves its absolute user area storage address.
VARIABLE	(---) A defining word to create a dictionary entry for <name> and allot two bytes for storage in the parameter field. The application must initialize the stored value. When <name> is later executed, it will place the storage addresses on the stack.
VLIST	(---) List the word names of the CONTEXT vocabulary starting with the most recent definition. This word

	incorporates the PAUSE feature- pressing any key will freeze the display. Once suspended, the VLIST may be resumed by a single keystroke, or aborted by striking any two keys in rapid succession.
VOC-LINK	(--- addr) A user variable containing the address of a field in the definition of the most recently created vocabulary. All vocabulary names are linked by these fields to allow control for FORGETting through multiple vocabularies.
VOCABULARY	(---) A defining word to create (in the CURRENT vocabulary) a dictionary entry for <name>, which specifies a new ordered list of word definitions. Subsequent executions of <name> will make it the CONTEXT vocabulary. When <name> becomes the CURRENT vocabulary (see DEFINITIONS), new defintions will be created in that list. In lieu of any further specifications, new vocabularies 'chain' to FORTH. That is, when a dictionary search through a vocabulary is exhausted, FORTH will be searched.
WARNING	(--- addr) A user variable containing a flag which enables the output of selected non-fatal error messages.
WHERE	(---) Display the last character string parsed by the text interpreter, along with the line containing it. If loading, the screen and line numbers are printed.
WHILE	(flag ---) Used in a colon-definition to select conditional execution based on the flag. On a true flag, continue execution through to REPEAT, which then returns back to just after BEGIN. On a false flag, skip execution to just after REPEAT, exiting the structure.
WIDTH	(--- addr) In fig-FORTH, a user variable containing the maximum number of letters saved in the compilation of a definition's name. It must be 1 through 31, with a default value of 31.
WORD	(char --- addr) Receive characters from the input stream until the non-zero delimiting character is encountered or the input stream is exhausted, ignoring leading delimiters. The characters are stored as a packed string with the character count in the first character position. The actual delimiter encountered (char or null) is stored at the end of the text but not included in the count. If the input stream was exhausted as WORD is called, then zero length will result. The address of the beginning of this packed string is left on the stack.
X	(---) This is a pseudonym for the "null" or dictionary entry for a name of one character of ASCII null. It is the

	execution procedure to terminate interpretation of a line of text from the terminal or within a disk buffer, as both buffers always have a null at the end.
XOR	(n1 n2 --- n3) Leave the bitwise exclusive-or of two numbers.
[	(---) End the compilation mode. The text from the input stream is subsequently executed. See]
[']	(---) Used in a colon-definition to compile the parameter field address of the next word in the input stream as a literal.
[COMPILE]	(---) Used in a colon-definition to force compilation of the following word. This allows compilation of an IMMEDIATE word when it would otherwise be executed.
]	(---) Set the compilation mode. The text from the input stream is subsequently compiled. See [

```
  *    ****  ****  ***** *   * ****  ***** *   *
 * *   *   * *   * *     **  * *   *   *    * *
***** ****  ****  ****  * * * *   *   *     *
*   * *     *     *     *  ** *   *   *    * *
*   * *     *     ***** *   * ****  ***** *   *
```

I. Example Programs

An extensive program example (also included in the system) is the CURVES demo (see source screens in Appendix II). This demo uses hi-res graphics, sprites, and a sound oscillator at same time. To load and run the demo enter the following:

```
99 110 THRU      ( LOADS THE DEMO SCREENS )
DEMO             ( STARTS THE DEMO )
```

To stop it perform a warm start (RUN/STOP RESTORE sequence. After running DEMO, trying changing the LEVEL and LINE-LENGTH values which DEMO uses to call D-CURVE and C-CURVE (see screens) for variations of those curves.

```
    Examples:
        23 D-SPLIT  0 23 D-POSITION
        4 8 D-CURVE  ( 4 LEVEL DRAGON CURVE- RESOLUTION 8)
        8 6 C-CURVE  ( LARGE 8 LEVEL C-CURVE )
        12 2 D-CURVE ( THIS WILL TAKE A WHILE TO DISPLAY )
        SOUND.INIT   ( GET RID OF SOUND )
        OFF D-SPLIT  ( GET BACK TO NORMAL SCREEN )
```

Comments in the examples (anything which is delimited by parenthesis-"(...)") need not be entered when entering definitions interactively (since the comments will not be stored anyway). They are provided to add clarity to the example. If the examples are entered as source files, however, I recommend inclusion of the comments.

A brief pep talk on the use of comments in source code: My particular technique is to put a comment line above each definition giving a general description of the definition, a "stack notation" comment of the form "(input --- output)" after the name of the definition (this describes the inputs and outputs of the word), and other comments as needed to clarify the definition. Unlike BASIC where everything you might do to clarify the program slows it down even more, FORTH comments are NOT compiled into the definition, so they DON'T slow down execution! Therefore, I highly recommend the use, overuse and abuse of comments- believe me, comments are just dying to be seen! Well, so much for my comments on comments.

The examples included in this section are intended to provide the user with some concrete programming examples. The first example describes how to design a complete application from the "top down" to the bottom. The second example provides an example of how the language itself can be extended to provide very high level functions.

I have used music for these examples since that is my particular interest, but the concepts demonstrated apply to any application area.

I-1 Designing A Program- Sound Synthesizer Example

The SID chip takes lots of parameters- I mean LOTS of parameters. A certain number of these must be set up in order to get the chip to do anything at all! The SUPER-FORTH word SOUND.INIT has been provided to perform a basic initialization of the SID chip. SOUND.INIT clears all SID registers and then performs resets on each of the three oscillators, leaving V1 as the active oscillator.

In order to specify a sound you must set up what is known as the amplitude envelope, or ADSR. Since the Commodore Reference Manual (or one of the other recommended books on sound) gives a description of ADSR I won't go into that here. Words V-AD and V-SR are used to set up the attack/decay parameters and the sustain/release parameters.

So, you need to understand all about ADSR in order to use the SID chip, right? Wrong! There is a word called V-DEFAULT which will set up the active voice with default parameters for the ADSR and the waveform. SOUND.INIT calls V-DEFAULT for each of the three voices as part of its SID initialization. Using SOUND.INIT should get you started, but try experimenting with different parameters to get your own sounds. The source code (see screen #47) is provided for V-DEFAULT and SOUND.INIT, so if you desire a different default the appropriate words in V-DEFAULT should be replaced and screen 47 re-loaded. All the user really need know to use the SID chip, however, is that SOUND.INIT must be invoked in order for the sound system to do anything. It is only necessary to use SOUND.INIT once after each FORTH startup.

Note: Words which are prefixed with "V-" affect one of the three oscillators, V1, V2 or V3, whichever was last invoked. That voice is referred to as the "active" voice.

You can play a note from the chromatic scale using PLAY.NOTE (see PLAY.NOTE description). If you type the following, for example:

```
    SOUND.INIT             ( INITIALIZES SOUND SYSTEM )
    48 PLAY.NOTE           ( PLAY A C4 )
    60 PLAY.NOTE           ( PLAY A C5 )
```

you will hear two C notes played one octave apart.

But who wants to remember all those numbers? Certainly not I! The following are two different approaches to letting the computer keep track of the numbers while you, the user, tell the computer what to play by using a more familiar format.

The first example describes how to write a SUPER FORTH program which will allow the user the "play" the C64 keyboard like a piano (or synthesizer) keyboard. It could be the basis of a real-time synthesizer program, that is, a user plays notes and instantly gets the musical response.

The second example describes ways of creating useful music composition tools which could be used as the basis of a music editing program.

I-1.1 Sound Synthesizer Example

This example is intended both as an example of using the SID chip AND as an example of what is known in the programming world as "structured top down design", a method of programming which FORTH is particularly suited for. Unlike the previous graphics example, this is NOT an interactive example, that is, you won't be able to enter anything until the complete program has been designed by us. If you are particularly itchy to hear your C64 make music using SUPER-FORTH, the complete program example is given at the end of this section. It may be useful to enter the program to see the end result of what we are trying to accomplish, and then come back here to read the description of how we went about creating the program. Okay, on with the program!

Lets say we want to write a program which will allow the user to "play" the C64 keyboard as if it is a piano keyboard (similar to the program in the Commodore 64 User's Manual). The following keys:

```
 2  3     5  6  7     9  0     -
Q  W  E  R  T  Y  U  I  O  P  @  *
```

would map into a piano keyboard:

```
 C# D#    F# G# A#    C# D#    F#
C  D  E  F  G  A  B  C  D  E  F  G
```

First we need to set up a method by which a keyboard key can be used by the program to designate a key on a piano keyboard (for instance, pressing a Q would look to the program like a C on the piano). The method we will use is called a translation table. This table will be used to translate the keys entered (Q, W etc.) into the numbers required by PLAY.NOTE to play the proper note. The table requires up to 128 values since that is the maximum number of ASCII characters we may want to translate. Since the values used will be less than 256 (a character value ranges from 0-255) we can use a byte table to contain the values.

```
CREATE TRANSLATE 128 ALLOT
```

defines a word named TRANSLATE in the dictionary and allocates 128 bytes of dictionary storage.

Lets set up a string which describes each character in chromatic scale order. If we use the SUPER-FORTH data structure STRING, when NOTE.KEYS is invoked the length of NOTE.KEYS and the address of the beginning of the string will be left on the stack.

```
$CONSTANT NOTE.KEYS  "Q2W3ER5T6Y7UI9O0P@-*"
```

Okay, our data structures have been defined. I will describe the actual program using the method "top down design, bottom up testing", that is, we first determine at a high level what we need to do and then implement the words to perform our high level requirements. Testing, however proceeds from the lowest level words up to the high level words. Don't worry about it if this is not very clear yet.

Note: Since we are going to design high level words first, you will not be able to enter the words interactively until we are ready to test, the low level definitions being the last to be defined.

Let's call our highest level word PLAY, since that is what we are trying to do. What we'll do is fill in the words which make up PLAY as we determine what they are. So far we have:

```
( PLAY THE C64 KEYS LIKE A PIANO KEYBOARD )
: PLAY   ( --- )  ;
```

The first thing we must do is initialize all those things that need initializing. What's a good name for our initialize routine? How about INITIALIZE?

```
( PLAY THE C64 KEYS LIKE A PIANO KEYBOARD )
: PLAY ( --- )
   INITIALIZE       ( INITIALIZE VARIOUS THINGS )
;
```

The main function of PLAY is to get a note from the keyboard and to play it:

```
( PLAY THE C64 KEYS LIKE A PIANO KEYBOARD )
: PLAY ( --- )
   INITIALIZE       ( INITIALIZE VARIOUS THINGS )
   GET.NOTE         ( GETS A NOTE FROM THE C64 KEYBOARD )
   PLAY.NOTE        ( TELLS THE SID CHIP TO PLAY THE NOTE )
;
```

What happens after we've played the note? We fall off the edge of the program, that's what! Well, this is fine if we want to play a single note, but it's a heck of a lot of work to go through for one little note. So, why don't we put in a way of playing lots of notes?

```
( PLAY THE C64 KEYS LIKE A PIANO KEYBOARD )
: PLAY ( --- )
   INITIALIZE       ( INITIALIZE VARIOUS THINGS )
   BEGIN
    GET.NOTE        ( GETS A NOTE FROM THE C64 KEYBOARD )
    PLAY.NOTE       ( TELLS THE SID CHIP TO PLAY THE NOTE )
   AGAIN
;
```

Okay, now we've got LOTS of notes, but no way to stop! We can always warm start, but that's really tacky. Let's specify a way to stop and call it MORE?. We can worry about the definition of MORE? later.

```
    ( PLAY THE C64 KEYS LIKE A PIANO KEYBOARD )
    : PLAY ( --- )
      INITIALIZE        ( INITIALIZE VARIOUS THINGS )
      BEGIN
       GET.NOTE         ( GETS A NOTE FROM THE C64 KEYBOARD )
       MORE? WHILE      ( PLAY ANOTHER NOTE? )
        ?DUP IF         ( IF ZERO, DON'T PLAY )
         PLAY.NOTE      ( TELLS THE SID CHIP TO PLAY THE NOTE )
        THEN
      REPEAT ;          ( GO GET ANOTHER NOTE )
```

That's our highest level definition. That was really the hard part. Now all we have to do is fill in the lower level words, but we already have some idea of what they do.

Notice how the highest level word is really a description of what we want the program to do. There are NO FORTH level primitives in this word, just broad, descriptive words and the necessary control structures required (in this case, BEGIN...WHILE...REPEAT).

First let's define INITIALIZE. INITIALIZE must handle the initialization of the TRANSLATE table and of the SID chip.

```
    ( INITIALIZATION PROCEDURE )
    : INITIALIZE
      SOUND.INIT          ( INITIALIZE SID CHIP )
      INIT.TRANSLATE ;  ( INITIALIZE THE TRANSLATE TABLE )
```

A good thing to remember is to keep definitions short. Short definitions will make it much easier later on to understand previous work. SOUND.INIT has already been discussed. INIT.TRANSLATE will initialize the character translation table, TRANSLATE. Unused places in the table will contain zero. The character which flags the end of execution ("F" for "finished") is set to 255. Used character places will contain the number of that characters equivalent PLAY.NOTE parameter. The character data will be offset by 48 to raise the notes to a more listenable octave (48 raises a note 4 octaves).

```
    ( INITIALIZE TRANSLATION TABLE )
    : INIT.TRANSLATE ( --- )
      TRANSLATE 128 0 FILL      ( ZERO OUT TABLE )
      255 70 TRANSLATE + C!     ( FLAG "F" AS FINISH CHAR )
      NOTE.KEYS COUNT 0 DO      ( DO FOR # CHARS. IN STRING )
       I 48 +                   ( SET UP NOTE VALUE )
       OVER I + C@              ( GET CHAR FROM STRING )
       TRANSLATE + C!           ( ADD CHAR TO TABLE & STORE NOTE )
      LOOP DROP ;               ( DO REST OF STRING )
```

Since this is a low-level word, we can immediately test its functionality (or testing could be postponed until ALL low level words have been defined). Type the following:

```
     INIT.TRANSLATE               ( INITIALIZE TABLE )
     ' TRANSLATE 16 DUMP          ( SHOULD DISPLAY ALL ZEROES )
     ' TRANSLATE 112 + 16 DUMP
     81 TRANSLATE + C@ .          ( "Q" SHOULD PRINT 48 [C4] )
     42 TRANSLATE + C@ .          ( "*" SHOULD PRINT 67 [G5] )
     70 TRANSLATE + C@ .          ( "F" SHOULD PRINT 255 [FINISHED])
```

The next component in our high level definition to be defined is GET.NOTE, the routine which gets input from the keyboard and translates it according to the TRANSLATE table. GET.NOTE returns 255 if the "finished" character ("F") is entered, 0 for non-defined characters and the note value for characters defined in NOTE.KEYS.

```
     ( GET A NOTE FROM THE KEYBOARD )
     : GET.NOTE   ( --- VALUE )
        KEY TRANSLATE + C@ ;
```

Test out GET.NOTE by entering some keys and displaying their values. GET.NOTE is the routine which would be changed if a different type of input device was used (a synthesizer keyboard, for instance).

MORE? simply checks to see if the depressed key is the "finished" key and leaves a FALSE flag on the stack if it is (this ends the BEGIN...WHILE...REPEAT main loop) or leaves a TRUE flag if it is not.

```
     ( CHECK FOR "FINISHED" CHARACTER [255] )
     : MORE?   ( VALUE --- VALUE FLAG )
        DUP 255 = NOT ;           ( FALSE IF "FINISHED" )
```

If execution efficiency becomes a problem this would be the first word to be eliminated and incorporated into the main definition, but it pays to define MORE? for the clarity it brings to the main definition.

This one is easy to test, enter a value, if the value was 255, 255 0 will be left on the stack, otherwise, VALUE 1 will be left. Values of 0 will be checked for and not played.

The final word to test is our high-level word, PLAY. Well, try it- type PLAY and then try playing on the keyboard. For reference, here is the complete program:

```
CREATE TRANSLATE 128 ALLOT
$CONSTANT NOTE.KEYS  "Q2W3ER5T6Y7UI9O0P@-*"

( GET A NOTE FROM THE KEYBOARD )
: GET.NOTE  ( --- VALUE )
   KEY TRANSLATE + C@ ;

( CHECK FOR "FINISHED" VALUE [255] )
: MORE?  ( VALUE --- VALUE FLAG )
   DUP 255 = NOT ;           ( FALSE IF "FINISHED" )

( INITIALIZE TRANSLATION TABLE )
: INIT.TRANSLATE ( --- )
   TRANSLATE 128 0 FILL      ( ZERO OUT TABLE )
   -1 70 TRANSLATE + C!      ( FLAG "F" AS FINISH CHAR )
   NOTE.KEYS COUNT 0 DO      ( DO FOR # CHARS. IN STRING )
    I 48 +                   ( SET UP NOTE VALUE )
    OVER I + C@              ( GET CHAR FROM STRING )
    TRANSLATE + C!           ( ADD CHAR TO TABLE  & STORE NOTE )
   LOOP DROP ;               ( DO REST OF STRING )

( INITIALIZATION PROCEDURE )
: INITIALIZE
   SOUND.INIT         ( INITIALIZE SID CHIP )
   INIT.TRANSLATE ;   ( INITIALIZE THE TRANSLATE TABLE )

( PLAY THE C64 KEYS LIKE A PIANO KEYBOARD )
: PLAY ( --- )
   INITIALIZE         ( INITIALIZE VARIOUS THINGS )
   BEGIN
    GET.NOTE          ( GETS A NOTE FROM THE C64 KEYBOARD )
    MORE? WHILE       ( PLAY ANOTHER NOTE? )
     ?DUP IF          ( IF ZERO, DON'T PLAY )
      PLAY.NOTE       ( TELLS THE SID CHIP TO PLAY THE NOTE )
     THEN
   REPEAT ;           ( GO GET ANOTHER NOTE )
```

I-1.2 Extending SUPER FORTH- Music Tools

I will describe extending the system by creating various music handling tools using the SUPER FORTH 64 Sound primitives. These words actually form the basis of the Music Editor words incorporated into SUPER FORTH. Thus, to avoid having the ISN'T UNIQUE message print out (since many of these words are already defined in the system) enter the following:

```
OFF WARNING !
```

I'm assuming the user has a basic knowledge of music theory for the understanding of the following examples.

Let's set up a way of specifying tempo and note duration:

```
    VARIABLE TEMPO                      ( TEMPO OF MUSIC )
    VARIABLE DURATION                   ( NOTE DURATION )
```

TEMPO will store our tempo value (how fast the notes are played) and DURATION will store our single note duration. The following words will be used to set duration (similar definitions exist in the system, but these are less complex)

```
    ( NOTE DURATIONS BASED ON 64TH NOTE RESOLUTION )
    ( 4/4 TIME ASSUMED )
    : WHOLE 16 DURATION ! ;     ( WHOLE NOTE )
    : .1/2  12 DURATION ! ;     ( DOTTED HALF NOTE )
    : 1/2    8 DURATION ! ;     ( HALF NOTE )
    : .1/4   6 DURATION ! ;     ( DOTTED QUARTER NOTE )
    : 1/4    4 DURATION ! ;     ( QUARTER NOTE )
    : 1/8    2 DURATION ! ;     ( EIGHTH NOTE )
        etc.
```

DURATION.PAUSE provides a pause for a given note (or chord) based on the values in DURATION and TEMPO.

```
    ( PAUSE TO ALLOW NOTE TO LAST FOR DURATION )
    : DURATION.PAUSE  ( --- )
       DURATION @ 900 TEMPO @ */  ( PAUSE = DURATION / TEMPO )
       WAIT ;
```

PLAY.WAIT plays a note of a given value, then calls waits an amount of time based on the duration of the note and the tempo of the music.

```
    ( PLAY A NOTE OF GIVEN DURATION )
    : PLAY.WAIT ( VALUE --- )
       PLAY.NOTE                   ( PLAY THE NOTE )
       DURATION.PAUSE ;            ( PAUSE TO LET NOTE COMPLETE )
```

Notes will play faster as TEMPO is increased and slower as TEMPO is decreased. First, ways of dealing with PLAY.WAIT more reasonably in interactive mode. One simple method would be to create 96 definitions, one for each chromatic note, as follows:

```
    : C0   0 PLAY.WAIT ;   : C1  12 PLAY.WAIT ;   : C2  24 PLAY.WAIT ;
    : C0#  1 PLAY.WAIT ;   : C1# 13 PLAY.WAIT ;   : C2# 25 PLAY.WAIT ;
    : D0   2 PLAY.WAIT ;   : D1  14 PLAY.WAIT ;   : D2  26 PLAY.WAIT ;
          etc.
```

These can now be combined, using the appropriate duration value to define phrases:

```
    : DOE .1/4 C4  1/8 D4  .1/4 E  1/8 D4  1/4 E4 D4 1/2 E4 ;
    : REY .1/4 D4  1/8 E4 F4 F4 E4 D4 WHOLE F ;
    DOE REY
```

A different usage of the definitions C0, C#0 etc., may be to set the 96 values as constants:

```
    0 CONSTANT C0      12 CONSTANT C1      24 CONSTANT C2
    1 CONSTANT C0#     13 CONSTANT C1#     25 CONSTANT C2#
    2 CONSTANT D0      14 CONSTANT D1      26 CONSTANT D2
          etc.
```

We can now use the constant values to specify chords, for instance. Let's define a FORTH structure to play a major chord. For those non-music-theorectical type people, I will give a brief explanation about how to define a major chord structure. A simple major chord (three notes played simultaneously), can be described by specifying a root note (let's say C4 for our example), and playing the third of the root key (E4 in our example) and the fifth of the root key (G4 in our example) simultaneously.

To set up our chord constructions we can define more constants defining key intervals which can be used to construct chords. The following table describes only the more commonly used intervals. Of course, none of these constants NEED to be defined, they are used only for clarity in defining higher level FORTH words.

```
    D0 C0 - CONSTANT SECOND
    D#0 C0 - CONSTANT MIN.THIRD
    E0 C0 - CONSTANT THIRD
    F0 C0 - CONSTANT FORTH
    G0 C0 - CONSTANT FIFTH
    A0 C0 - CONSTANT SIXTH
    A#0 C0 - CONSTANT DOM.SEVENTH
    B0 C0 - CONSTANT SEVENTH
```

Okay, now for our major chord definition:

```
    : MAJOR   ( ROOT --- )
      DUP THIRD +        ( PUT THIRD ON STACK )
      OVER FIFTH +       ( PUT FIFTH ON STACK )
      V3 PLAY.NOTE       ( PLAY FIFTH ON VOICE 3 )
      V2 PLAY.NOTE       ( PLAY THIRD ON VOICE 2 )
      V1 PLAY.NOTE       ( PLAY ROOT ON VOICE 1 )
      DURATION.PAUSE ;   ( PAUSE FOR DURATION OF CHORD )
```

Great, now we have a chord structure definition.

We Must initialize voices V2 and V3 (V1 is already initialized) before attempting to use MAJOR:

```
    V2 V-DEFAULT  V3 V-DEFAULT  ( INIT. V2 AND V3 )
```

Now try out the following chord changes:

```
    120 TEMPO !  WHOLE C4 MAJOR  1/2 F4 MAJOR  G4 MAJOR
```

You should hear a measure of C4 major followed by a half measure of F4 major and a half measure of G4 major.

Well, I think I've given you enough ideas to start you off. The same methods could be used to define other chords, arpeggios, or sequences of notes which will be repeated.

The techniques described were used to create the SUPER FORTH Music Editor which is included in the system. The Music Editor vocabulary is somewhat more complex due to having to handle synchronizing the three voices, however you should now be able to understand the code which implements the Music Editor. Refer to source screens 81-86 for the code. Also, if you haven't done so, go through the examples in the Music Editor section of this manual (however, first FORGET the definitions which you have added in this section). I hope you have as much fun as I've had bringing the system to you.

II. SUPER-FORTH 64 User Source Screens

The index and listing which follows describe the source screens which are distributed with the system. All words up to the point which is marked "Not Compiled") have been compiled into the SUPER-FORTH system. The source code of these words has been provided to allow user flexibility in configuring a FORTH system, and as an educational tool to provide to beginner with examples of implementations of FORTH words.

```
       The following screens are compiled into the system:
  0
  1  ( SF64 REVIEW SYSTEM LOADER BLOCK )
  2  ( DISK DIRECTORY/LOADER )
  3  ( UTILITIES: THRU )
  4  ( C64 UTILITIES: SYSCALL RECURSE -TEXT )
  5  ( KERNAL & I/O: SETLFS SETNAM OPEN CLOSE CLRCHN )
  6  ( KERNAL INTERFACE: LOADRAM DOS ST )
  7  ( STRINGS: $VARIABLE $CONSTANT <$CONCAT> )
  8  ( STRINGS: $CONCAT $LEFT $MID $RIGHT )
  9  ( STRINGS: $VAL  $LEN  $.  $CLR )
 10  ( STRINGS: <">  "  "" )
 11  ( STRINGS: $CMP $< )
 12  ( STRINGS: $> $= $FIND )
 13  ( FILE MODE: FNAME FOPEN FILE-MODE )
 14  ( FILE-MODE: READB WRITEB )
 15  ( FILE-MODE: F-INIT F-NEW F-APPEND )
 16  ( FILE-MODE: F-EDIT F-SAVE )
 17  ( FILE MODE: F-LOAD F-NUMBER )
 18  ( GRAPHICS: S-MULTIR MULTI-COLOR S-S-COLLISION S-B-COLLISION)
 19  ( GRAPHICS: B-GRAPHICS B-FILL B-COLOR B-COL-FILL )
 20  ( GRAPHICS: S-FSET S-ENABLE S-XEXP S-YEXP )
 21  ( GRAPHICS: S-PRIORITY S-MULTI S-POINTER S-COLOR )
 22  ( SOUND- V-FREQ V-PW V-AD V-SR )
 23  ( SOUND- V-CTRL RESFILT MODEVOL )
 24  ( UTILITIES: .S .SL .SR .SS .INDEX <ROT )
 25  ( UTILITIES:MAX-BUFFS BMOVE COPY SCOPY DSWAP D- D0= )
 26  ( UTILITIES: D= D> D@ DCONSTANT DMAX DMIN DOVER DU< DVARIABLE )
 27  ( UTILITIES:  PAUSE  )
 28  ( UTILITIES: INDEX ?LOADING --> )
 29  ( UTILITIES: DUMP )
 30  ( UTILITIES: 'TITLE TITLE TRIAD )
 31  ( UTILITIES:  <EMIT7>  ID.  )
 32  ( UTILITIES: VLEN VTAB VLIST )
 33  ( SUPPLEMENTALS:  'S "2" DOUBLE NUMBER SET )
 34  ( SUPPLEMENTALS: >BINARY ERASE FLUSH H U.R ['] )
 35  ( ASSEMBLER: CONSTANTS INDEX )
 36  ( ASSEMBLER: MODE ADDRESSING MODES BOT SEC RP> UPMODE )
 37  ( ASSEMBLER: CPU )
 38  ( ASSEMBLER: M/CPU )
 39  ( ASSEMBLER: BEGIN, UNTIL, IF, THEN, ELSE, NOT BRANCHES )
 40  ( ASSEMBLER:  AGAIN, WHILE, REPEAT )
 41  ( ASSEMBLER: END-CODE ENTERCODE ;CODE CODE )
 42  ( EDITOR: CHKLIN LINE PP C )
 43  ( EDITOR: SCREEN COMMANDS )
 44  ( EDITOR: K X O M )
 45  ( EDITOR: F L W N P SC SM LIST )
 46  ( DECOMPILER: GIN  GIN+  GCHK )
 47  ( DECOMPILER: <DECOM>  )
 48  ( DECOMPILER: DECOMPILE )
 49  ( TRIG: TSCALE )
 50  ( TRIG: SIN/COS VALUES TABLE - SCALED BY 32768 )
 51  ( TRIG: QSIN QCOS SIN COS )
 52  ( MATH: SQUARE ROOT  ROUTINES )
 53  ( MATH: SQUARE ROOT ROUTINES )
 54  ( F.P. MATH: FPSW FRESET FER FZE FNE FOV )
 55  ( F.P. MATH: SFZ SFN E@ )
 56  ( F.P. MATH: E! E. )
 57  ( F.P. MATH: F. F* F/ )
 58  ( F.P. MATH: ALIGN F+ F- )
 59  ( F.P. MATH: RSCALE LSCALE DFIX )
 60  ( F.P. MATH: FIX DFLOAT FLOAT FSIN FCOS FSQRT FINIT FEXIT )
 61  ( F.P. MATH: FABS FNEGATE FMIN F> FMAX )
 62  ( C64 DATA: COLORS SPRITE-DEFS )
 63  ( C64 DATA: SOUND )
 64  ( C64 DATA: SOUND MISC I/O )
 65  ( DATA STRUCTURES: S-DEF )
 66  ( DATA STRUCTURES: 1ARRAY 2ARRAY )
 67  ( GRAPHICS: M-X M-Y B-MFLAG S.DIST L.DIST )
 68  ( GRAPHICS: BORDER BKGND M-ORIGIN )
 69  ( GRAPHICS: R-PLOT )
 70  ( GRAPHICS: M-PLOT )
 71  ( GRAPHICS: CHAR )
 72  ( GRAPHICS: B-CPLOT <B-LINE> )
 73  ( GRAPHICS: <B-LINE> )
 74  ( GRAPHICS: LSETUP )
 75  ( GRAPHICS: B-LINE )
 76  ( GRAPHICS: ELLIPSE CIRCLE )
 77  ( GRAPHICS: ARC ELLIPSE CIRCLE )
 78  ( I/O EXTENSIONS: CMD CMDI INPUT INPUT# PRINT# $INPUT PRINTER )
 79  ( I/O EXTENSIONS: GET# PUT# RS232 FRE RDTIM )
 80  ( I/O EXTENSIONS: SETTIM WAIT )
 81  ( UTILITIES: CASE OF ;; ENDCASE )
 82  ( UTILITIES: DIR )
 83  ( UTILITIES: PATCH )
 84  ( UTILITIES: H@ HC@ H! HC! )
 85  ( MUSIC EDIT: WAVE PLAY.NOTE V-DEFAULT SOUND.INIT )
 86  ( MUSIC EDIT: T@ T! )
 87  ( MUSIC EDIT: O@ O! OX )
 88  ( MUSIC EDIT: TEMPO DURATION TIMINGS )
 89  ( MUSIC EDIT: NDEF NEXT.NOTE READY NCALC )
 90  ( MUSIC EDIT: PLAY.WAIT )
 91  ( MUSIC EDIT: NOTE DEFS )
 92  ( S-EDITOR: S-EDITOR )
 93  ( S-EDITOR:  )
 94  ( TURTLE: HEADING SETH SETXY PENFLG PU PD PM SETX SETY RT LT )
 95  ( TURTLE: CS HOME PC BG TS DRAW )
 96  ( TURTLE: FD BK )
 97  ( TURTLE: LONG NAMES )
```

The following screens must be compiled to be executed:

```
 98  ( TRACE COLON WORDS )
 99  ( CURVES: CONSTANTS DIRECTIONS )
100  ( CURVES: DRAGON1 )  HEX
101  ( CURVES: DRAGON2 )    HEX
102  ( CURVES: VARIABLES  NOTE NEXT-POINTER MOVE-SPRITE )
103  ( CURVES: DRAWLINE  NEW-POINT DRAW MOVE-DIRECTION )
104  ( CURVES: C-DRAW )
105  ( CURVES: SPRITES-INIT  HI-RES-INIT ) HEX
106  ( CURVES: DIRECTIONS-INIT )
107  ( CURVES: CURVE-INIT  C-CURVE )
108  ( CURVES: CALL-RDRAGON  LDRAGON )
109  ( CURVES: RDRAGON )
110  ( CURVES: D-CURVE WAIT-5-SEC DEMO )
111  ( BACKUP: DUAL DRIVE BUFFER COPY )
112  ( BACKUP: COPYBUF )
113  ( BACKUP: PCOPY BACKUP SOURCE-BACKUP )
114  ( ASSEMBLER: CONSTANTS INDEX )
115  ( ASSEMBLER: MODE ADDRESSING MODES BOT SEC RP> UPMODE )
116  ( ASSEMBLER: CPU )
117  ( ASSEMBLER: M/CPU )
118  ( ASSEMBLER: BEGIN, UNTIL, IF, THEN, ELSE, NOT BRANCHES )
119  ( ASSEMBLER:  AGAIN, WHILE, REPEAT )
120  ( ASSEMBLER: END-CODE ENTERCODE ;CODE CODE )
121  ( ASSEMBLER: A-REMOVE )
122
123  ( SPRITES DEMO )
124  ( SPRITES DEMO )
125  ( GRAPHICS DEMOS )
126  ( GRAPHICS DEMOS )
127  ( JESU )
128  ( JESU )
129  ( TURTLE DEMOS )
130  ( DEMO EXECUTIVE )
```

SCREEN #0
```


           SUPER-FORTH 64
              FOR THE
        COMMODORE 64 COMPUTER

                V2.2R


         COPYRIGHT 1983,1984
                 BY
         ELLIOT B. SCHNEIDER

         ALL RIGHTS RESERVED


```

SCREEN #1
```
( SF64 REVIEW SYSTEM LOADER BLOCK )
      3 ." THRU "                 LOAD CR
      4 ." C64 UTILITIES "        LOAD CR
  5   6 ." C64 KERNEL WORDS "     THRU CR
  7  12 ." STRINGS "              THRU CR
 13  17 ." FILE-MODE "            THRU CR
 18  21 ." GRAPHICS "             THRU CR
 22  23 ." SOUND "                THRU CR
 24  32 ." UTILITIES "            THRU CR
 33  34 ." SUPPLEMENTALS "        THRU CR
 35  41 ." LOCAL ASSEMBLER "      THRU CR
 42  45 ." EDITOR "               THRU CR
 46  49 ." DECOMPILER UTIL. "     THRU CR
 50  51 ." TRIG ROUTINES "        THRU CR
 52  53 ." SQUARE ROOT "          THRU CR
  -->
```

SCREEN #2
```
( DISK DIRECTORY/LOADER )
54 61 ." FLOATING POINT "      THRU CR
62 66 ." C64 DATA/CONST. "     THRU CR
67 77 ." GRAPHICS "            THRU CR
78 80 ." I/O EXTENSIONS "      THRU CR
81 84 ." UTILITIES "           THRU CR
85 91 ." MUSIC EDITOR "        THRU CR
   92 ." SPRITE EDITOR "       LOAD CR
94 97 ." TURTLE GRAPHICS "     THRU CR
 EXIT
   98 ." TRACE UTILITY "       LOAD
99 110 ." GRAPH/SOUND DEMO"    THRU CR
111 113 ." BACKUP UTILITIES"   THRU CR
114 121 ." REMOTE ASSEMBLER"   THRU CR
123 130 ." DEMOS "             THRU CR

```

SCREEN #3
```
( UTILITIES: THRU )
FORTH DEFINITIONS

: THRU  1+ SWAP
   DO I U. I LOAD
    ?TERMINAL IF LEAVE THEN LOOP ;










```

SCREEN #4
```
( C64 UTILITIES: SYSCALL RECURSE -TEXT )

: SYSCALL  ( A X Y ADDR --- )
   SYS DDROP DDROP ;

( IMPLEMENT RECURSIVE DEFINITIONS )
: RECURSE  ( IMPLEMENT RECURSIVE PROCEDURES )
   LATEST  PFA  CFA  , ;  IMMEDIATE

: -TEXT   ( ADR1 C ADR2 --- FLAG )
   DDUP + SWAP
   DO DROP 1+ DUP 1- C@ I C@ - DUP
    IF 1 SWAP +- LEAVE THEN
   LOOP SWAP DROP ;


```

SCREEN #5
```
( KERNAL & I/O: SETLFS SETNAM OPEN CLOSE CLRCHN )
HEX
: SETLFS ( LFN DEV CMD --- )
   FFBA SYSCALL ;

: SETNAM ( STRINGADDR --- )
   COUNT SWAP SPLIT FFBD SYSCALL ;

: OPEN  ( LFN DEV CMD ADDR --- )
   SETNAM  SETLFS
   FFC0 GO ;

: CLOSE  ( LFN --- )
   0 0 FFC3 SYSCALL ;
: CLALL  ( --- )  FFE7 GO ;
: CLRCHN  ( --- )  FFCC GO ; DECIMAL
```

SCREEN #6
```
( KERNAL INTERFACE: LOADRAM DOS ST )
HEX
: LOADRAM ( LOADADDR FILEADDR --- )
   SETNAM
   >R FF SYSDEV @ R@ 0=  SETLFS
   0 R> SPLIT FFD5 SYSCALL ;


: DOS   ( STRINGADDR --- )
   >R F SYSDEV @ F R> OPEN
   F CLOSE ;

: ST   ( --- STATUS ) 0090 C@ ;
DECIMAL


```

SCREEN #7
```
( STRINGS: $VARIABLE $CONSTANT <$CONCAT> )
: $VARIABLE
   CREATE   ( # --- )
    0 C, ALLOT ;

: $CONSTANT
   CREATE   ( --- )
    HERE 1+   34 TEXT  PAD COUNT
    DUP C,   DUP ALLOT  CMOVE ;

( CONCATENATION PRIMITIVE- CONCAT C2 CHARS FROM A2 ONTO END S1 )
: <$CONCAT>   ( S1 A2 C2 --- )
   >R OVER COUNT +
   R@ CMOVE DUP C@ R> +
   SWAP C! ;

```

SCREEN #8
```
( STRINGS: $CONCAT $LEFT $MID $RIGHT )
( CONCAT S2 ONTO END OF S1 )
: $CONCAT   ( S1 S2 ---   )
   COUNT <$CONCAT> ;

( CONCAT N LEFTMOST CHARS OF S2 TO S1 )
: $LEFT   ( S1 S2 NUM --- )
   SWAP 1+ SWAP <$CONCAT> ;

( CONCAT N CHARS STARTING FROM S IN S2 TO S1 )
: $MID    ( S1 S2 S N --- )
   >R + R> <$CONCAT> ;

( CONCAT N RIGHTMOST CHARS OF S2 TO S1 )
: $RIGHT ( S1 S2 NUM --- )
   >R COUNT + R@ - R> <$CONCAT> ;
```

SCREEN #9

```
( STRINGS: $VAL  $LEN  $.  $CLR )
( CONVERT STRING S1 INTO 16 BIT # )
: $VAL    ( S1 --- N )
   NUMBER DROP ;

( GET LENGTH OF STRING S1 )
: $LEN    ( S1 --- N )
   C@ ;

( PRINT A STRING )
: $.     ( S1 --- )
   COUNT TYPE ;

( CLEAR A STRING )
: $CLR   ( S1 --- )
   0 SWAP C! ;
```

SCREEN #10

```
( STRINGS: <">  "  "" )
( RUN-TIME ROUTINE FOR " )
: <">  ( --- S1 )
   0 PAD C! PAD
   R@ COUNT DUP 1+ R> + >R
   <$CONCAT> PAD ;

( ENTER A STRING INTO THE PAD )
: "   ( --- S1 )
   34 STATE @  ( CHECK FOR COMPILE)
   IF COMPILE <">  ( COMPILE ADDR OF RUN-TIME ROUTINE )
    WORD C@ 1+ ALLOT
   ELSE TEXT PAD
   THEN ; IMMEDIATE

: ""  ( --- S1 ) 0 PAD C! PAD ;
```

SCREEN #11

```
( STRINGS: $CMP $< )
( COMPARE TWO STRINGS- RESULTS:
   0: S1=S2;  1: S1>S2;  2: S1<S2 )

: $CMP  ( S1 A2 C2 --- FLAG )
   COUNT ROT COUNT ROT DDUP - >R R@
   0> IF SWAP THEN
   DROP ROT -TEXT
    ?DUP IF  R> DROP
     ELSE R> DUP IF 1 SWAP +- THEN
     THEN ;

( CHECK IF S1 < S2 )
: $<   ( S1 S2 --- F )
   $CMP 0< ;
```

```
SCREEN #12
( STRINGS: $> $= $FIND )
( CHECK IF S1 > S2 )
: $>    ( S1 S2 --- F )
   $CMP 0> ;

( CHECK IF S1 = S2 )
: $=    ( S1 S2 --- F )
   $CMP 0= ;

( FIND OCCURENCE OF S1 IN STRING )
: $FIND ( S1 A2 C2 --- [ADDR] FLAG)
   OVER + SWAP
    DO DUP COUNT I -TEXT DUP 0=
     IF SWAP DROP I SWAP LEAVE ELSE DROP THEN
    LOOP NOT ;

SCREEN #13
( FILE MODE: FNAME FOPEN FILE-MODE )
23 $VARIABLE FNAME

( OPEN A FILE )
: F-OPEN  ( R/W  --- )
   9 SYSDEV @ ROT FNAME OPEN ;
( DUMMY R/W ROUTINE )
: FR/W  DDROP DROP ;
( SET UP FILE-MODE )
: FILE-MODE  ( "FILENAME" --- )
   DR0  1 MODE !  ' FR/W CFA 'R/W !
   34 TEXT PAD $LEN
    IF FNAME $CLR FNAME PAD $CONCAT
    THEN ;

: F-EXIT 0 MODE !   ' <R/W> CFA 'R/W ! ;

SCREEN #14
( FILE-MODE: READB WRITEB )
( READ A BLOCK FROM AN OPENED FILE INTO A BUFFER )
: READB  ( ADDR --- )
   9 INPLFN !
   1024 0 DO
     KEY OVER I + C!
    LOOP
   2 INPLFN !  DROP ;

( WRITE A BLOCK FROM A BUFFER INTO AN OPENED FILE )
: WRITEB  ( ADDR --- )
   9 OUTLFN !
   1024 0 DO
     DUP I + C@ EMIT
    LOOP
   3 OUTLFN ! DROP ;
```

```
SCREEN #15
( FILE-MODE: F-INIT F-NEW F-APPEND )
( INITIALIZE TO EDIT A FILE )
: F-NEW   ( --- )
   EMPTY-BUFFERS  FILE-MODE
   LIMIT 1028 - PREV !  FIRST USE !
   0 FLAST !  1 SCR ! ;

( APPEND FILE TO BUFFERS )
: F-APPEND  ( [FILENAME] )
   FILE-MODE  0 F-OPEN
   #BUFF 1+ FLAST @ 1+ DUP SCR ! DO
    I BLOCK READB ST
     IF I FLAST ! LEAVE THEN
    LOOP
   9 CLOSE ;


SCREEN #16
( FILE-MODE: F-EDIT F-SAVE )
( INIT SYSTEM & READ IN FILE )
: F-EDIT  ( [FILENAME] )
   F-NEW  F-APPEND ;

HEX
( SAVE BUFFERS TO A FILE )
: F-SAVE  ( [FILENAME] )
   FILE-MODE 1 F-OPEN
   #BUFF 0 DO
    I 404 * FIRST + DUP @
    7FFF AND 7FFF XOR
     IF 2+ WRITEB ELSE DROP THEN
    LOOP
   9 CLOSE ;
DECIMAL

SCREEN #17
( FILE MODE: F-LOAD F-NUMBER )
: F-LOAD  ( [FILENAME] ; --- )
   F-NEW   0 F-OPEN
   1 BLOCK  ( LOAD INTO BLOCK 1 )
   170 0 DO
    DUP READB ST
    1 LOAD
    IF LEAVE THEN  ( CHECK STATUS FOR EOF )
   LOOP   9 CLOSE ;   HEX
: F-NUMBER  ( START --- )
   DEPTH 0=   IF 1 THEN
   8000 OR  LIMIT 1- FIRST DO
     I @ 7FFF = NOT
      IF DUP I ! 1+ THEN
   404 /LOOP DROP ;
DECIMAL
```

```
SCREEN #18
 0) ( GRAPHICS: S-MULTIR MULTI-COLOR S-S-COLLISION S-B-COLLISION)
 1) HEX
 2) : S-MULTIR ( COLOR NUM --- )
 3)    D025 + C! ;
 4)
 5) : MULTI-COLOR   ( F --- )
 6)    D016 10 FBIT ;
 7)
 8) : S-S-COLLISION   ( --- VALUE )
 9)    D01E C@ ;
10)
11) : S-B-COLLISION   ( --- VALUE )
12)    D01F C@ ;
13) DECIMAL   EXIT
14)
15)

SCREEN #19
 0) ( GRAPHICS: B-GRAPHICS B-FILL B-COLOR B-COL-FILL )
 1) HEX
 2) : B-GRAPHICS    ( FLAG --- )
 3)    D011 20 FBIT ;
 4)
 5) : B-FILL    ( CHAR --- )
 6)    'BITMAP 1F40 ROT FILL ;
 7)
 8) : B-COLOR     ( POS HCOL LCOL --- )
 9)    CATNIB SWAP 'SCREEN + C! ;
10)
11) : B-COLOR-FILL    ( HCOL LCOL --- )
12)    CATNIB 'SCREEN 3E8 ROT FILL ;
13) DECIMAL
14)
15)

SCREEN #20
 0) ( GRAPHICS: S-FSET S-ENABLE S-XEXP S-YEXP )
 1) HEX
 2) : S-FSET   ( FLAG ADDR --- )
 3)    SPRITE @ MASK FBIT ;
 4)
 5) : S-ENABLE   ( FLAG --- )
 6)    D015 S-FSET ;
 7)
 8) : S-XEXP   ( FLAG --- )
 9)    D01D S-FSET ;
10)
11) : S-YEXP    ( FLAG --- )
12)    D017 S-FSET ;
13) DECIMAL  EXIT
14)
15)
```

SCREEN #21
```
( GRAPHICS: S-PRIORITY S-MULTI S-POINTER S-COLOR )
HEX
: S-PRIORITY   ( FLAG --- )
        D01B S-FSET ;

: S-MULTI   ( FLAG --- )
   D01C S-FSET ;

: S-POINTER   ( SPR# --- )
   'SCREEN 3F8 + SPRITE @ + C! ;

: S-COLOR   ( COLOR --- )
   D027 SPRITE @ + C! ;
DECIMAL EXIT


```

SCREEN #22
```
( SOUND- V-FREQ V-PW V-AD V-SR )
( SET FREQUENCY OF VOICE )
: V-FREQ  ( VALUE --- )
   SPLIT  1  V!  0  V! ;

( SET PULSE WIDTH OF VOICE )
: V-PW  ( VALUE12 --- )
   SPLIT  3  V!  2  V! ;

( SET ATTACK & DECAY OF VOICE )
: V-AD  ( ATTACK4 DECAY4 --- )
   CATNIB  5  V! ;

( SET SUSTAIN & RELEASE OF VOICE )
: V-SR  ( SUST4 REL4 --- )
   CATNIB  6  V! ;
```

SCREEN #23
```
( SOUND- V-CTRL RESFILT MODEVOL )
( SET CONTROL VALUES OF VOICE )
: V-CTRL  ( VALUE --- )
   4  V! ;

( SET FILTER FREQUENCY )
( SET RESONANCE & FILTER SWITCHES )
: RESFILT  ( RES FSWS --- )
   CATNIB 23 SID! ;

( SET FILTER MODE & SID VOLUME )
: MODEVOL  ( MODE VOLUME --- )
   CATNIB 24 SID! ;



```

SCREEN #24
```
( UTILITIES: .S .SL .SR .SS .INDEX )
-1 CONSTANT .SS
: .SL 0 ' .SS ! ;
: .SR -1 ' .SS  ! ;
: .S CR DEPTH
    IF .SS  IF SP@ S0 2-
     ELSE SP@ S0 SWAP THEN
     DO I @ 0 D. 2 .SS +- +LOOP
    ELSE ." EMPTY STACK " THEN CR ;

: .INDEX
   DUP PAD 1 ROT T&SCALC 1 RWTS DROP
   CR OFFSET @ - 4 .R
   2 SPACES PAD C/L -TRAILING TYPE ;


```

SCREEN #25
```
( UTILITIES: <ROT BMOVE COPY SCOPY D- D0= D= MAX-BUFFS )
: <ROT ROT ROT '
: BMOVE <ROT DDUP U<
   IF ROT <CMOVE
   ELSE ROT CMOVE THEN ;

: COPY OFFSET @ + SWAP BLOCK 2- ! UPDATE ;
: SCOPY ( FSTART FEND TSTART --- )
   <ROT 1+ SWAP   DO I OVER COPY 1+ LOOP DROP ;
: D- DNEGATE D+ ;
: D0= OR 0= ;
: D= D- D0= ;

: MAX-BUFFS  ( --- )
   LIMIT HERE - 0 1028 U/MOD
   ' #BUFF ! DROP CHANGE ;
```

SCREEN #26
```
( UTILITIES:DSWAP D> D@ DU< DCONSTANT DOVER DMAX DMIN DVARIABLE)
: DSWAP 4 ROLL 4 ROLL '
: D> DSWAP D< ;
: D@ DUP 2+ @ SWAP @ ;
: DU< >R >R 32768 +
     R> R> 32768 + D< ;
: DCONSTANT CREATE SWAP , ,
   DOES> DUP @ SWAP 2+ @ ;
: DOVER 4 PICK 4 PICK ;
: DMAX DOVER DOVER D< IF DSWAP THEN DDROP ;
: DMIN DOVER DOVER D< NOT IF DSWAP THEN DDROP ;
: DVARIABLE CREATE 4 ALLOT ;




```

```
SCREEN #27
 0) ( UTILITIES:  PAUSE  )
 1) HEX
 2) : PAUSE  ?TERMINAL
 3)    IF 1000 0 DO LOOP
 4)      BEGIN ?TERMINAL UNTIL
 5)      1000 0 DO LOOP
 6)    THEN ;
 7) DECIMAL
 8)
 9)
10)
11)
12)
13)
14)
15)
```

```
SCREEN #28
 0) ( UTILITIES: INDEX ?LOADING --> )
 1)
 2) : INDEX     ( FROM TO --- , )
 3)   CR OFFSET @ DUP ROT + 1+ <ROT +
 4)   OVER MAX-BLKS 1+ >
 5)   ABORT" BLK NO. ERROR"
 6)   DO I .INDEX PAUSE ?TERMINAL
 7)     IF LEAVE THEN
 8)   LOOP ;
 9)
10) : ?LOADING
11)   BLK @ NOT ABORT" LOADING ONLY " ;
12)
13) : -->  ( --- )
14)    ?LOADING 0 >IN ! 1 BLK +! ;
15) IMMEDIATE
```

```
SCREEN #29
 0) ( UTILITIES: DUMP )
 1) HEX   ( ADDR N --- )
 2) : DUMP 0 BASE @ >R HEX
 3)   DO CR DUP I + DUP 0 6
 4)    D.R 2 SPACES DUP 8 0
 5)     DO DUP I + C@ 3 .R LOOP
 6)    DROP SPACE DUP 8 + 8 0
 7)     DO DUP I + C@ 3 .R LOOP
 8)    DROP 3 SPACES 10 0
 9)     DO DUP I + C@ DUP 20 < OVER 7E > OR
10)      IF DROP 2E THEN EMIT
11)    LOOP DROP 10
12)   PAUSE ?TERMINAL IF LEAVE THEN
13)  /LOOP
14)  DROP CR R> BASE ! ;
15) DECIMAL
```

```
SCREEN #30
( UTILITIES: 'TITLE TITLE TRIAD )
VARIABLE 'TITLE
: TITLE  CR 11 SPACES
   ." SUPER FORTH 64 VERSION 2.2R" CR ;

' TITLE CFA 'TITLE !

: TRIAD 0 3 U/MOD SWAP DROP
   3 * 3 OVER + SWAP
   DO CR I LIST ?TERMINAL
    IF LEAVE THEN  1  /LOOP
   'TITLE @ EXECUTE
   12 EMIT ;




SCREEN #31
( UTILITIES:  <EMIT7>  ID.  )
HEX

: <EMIT7>  7F AND <EMIT> ;

( ID. : PRINT NAME FROM NFA )
: ID.   ( ADDR --- )
   ' <EMIT7> CFA 'EMIT !
   COUNT 1F AND TYPE
   ' <EMIT> CFA 'EMIT ! ;

DECIMAL





SCREEN #32
( UTILITIES: VLEN VTAB VLIST )
VARIABLE VLEN     40 VLEN !
VARIABLE VTAB     13 VTAB !

: VLIST 32767 OUT !  CONTEXT @ @
   BEGIN VLEN @ 1- OUT @ - OVER C@ 31 AND
    < IF CR 0 OUT ! THEN
    DUP ID.
    VTAB @ OUT @ OVER MOD - SPACES
    PFA LFA @ DUP
    0= PAUSE ?TERMINAL OR
   UNTIL DROP ;
```

SCREEN #33
```
( SUPPLEMENTALS:  'S "2" DOUBLE NUMBER SET )
: 'S SP@ ;

: 2! D! ;
: 2@ D@ ;
: 2CONSTANT DCONSTANT ;
: 2DROP DDROP ;
: 2DUP DDUP ;
: 2OVER DOVER ;
: 2SWAP DSWAP ;
: 2VARIABLE DVARIABLE ;





```

SCREEN #34
```
( SUPPLEMENTALS: >BINARY ERASE FLUSH H U.R ['] )
: >BINARY CONVERT ;

: EMPTY INIT-FORTH @ ' FORTH 2+ !
   INIT-USER UP @ 6 + 48 CMOVE ;

: ERASE 0 FILL ;

: FLUSH SAVE-BUFFERS ;

: H DP ;

: U.R 0 SWAP D.R ;

: ['] ?COMP [COMPILE] ' ; IMMEDIATE

```

SCREEN #35
```
( ASSEMBLER: CONSTANTS INDEX )
VOCABULARY ASSEMBLER IMMEDIATE
HEX ASSEMBLER DEFINITIONS
( REGISTER ASSIGNMENTS SPECIFIC TO THIS IMPLEMENTATION )
83 CONSTANT XSAVE   81 CONSTANT W
04 CONSTANT IP      87 CONSTANT N

( NUCLEUS LOCATIONS SPECIFIC TO THIS IMPLEMENTATION)
0884 CONSTANT POP    0882 CONSTANT POPTWO
088B CONSTANT PUT    0889 CONSTANT PUSH
0890 CONSTANT NEXT   0871 CONSTANT SETUPN

VARIABLE INDEX -2  ALLOT
0909 , 1505 , 0115 , 8011 , 8009 , 1D0D , 8019 , 8080 ,
0080 , 1404 , 8014 , 8080 , 8080 , 1C0C , 801C , 2C80 , DECIMAL

```

SCREEN #36
```
( ASSEMBLER: MODE ADDRESSING MODES BOT SEC RP> UPMODE )
HEX
VARIABLE MODE 2 MODE !
: .A 0 MODE ! ; : # 1 MODE ! ;   : MEM 2 MODE ! ;
: ,X 3 MODE ! ; : ,Y 4 MODE ! ; : ,X) 5 MODE ! ;
: ),Y 6 MODE ! ; : ) F MODE ! ;

: BOT ,X 0 ;       ( ADDRESS THE BOTTOM OF DATA STACK )
: SEC ,X 2 ;       ( ADDRESS SECOND ITEM ON DATA STACK )
: RP> ,X 101    ; ( ADDRESS BOTTOM OF RETURN STACK )

: UPMODE IF MODE @   8 AND 0= IF 8 MODE +! THEN THEN
   1 MODE @ 0F AND ?DUP  IF 0 DO DUP + LOOP THEN
   OVER 1+ @ AND 0= ;

DECIMAL
```

SCREEN #37
```
( ASSEMBLER: CPU )

HEX
: CPU CREATE C, DOES>  C@ C, MEM ;
    00 CPU BRK,    18 CPU CLC,    DE CPU CLD,    58 CPU CLI,
    B8 CPU CLV,    CA CPU DEX,    88 CPU DEY,    E8 CPU INX,
    C8 CPU INY,    EA CPU NOP,    48 CPU PHA,    08 CPU PHP,
    68 CPU PLA,    28 CPU PLP,    40 CPU RTI,    60 CPU RTS,
    38 CPU SEC,    F8 CPU SED,    78 CPU SEI,    AA CPU TAX,
    A8 CPU TAY,    BA CPU TSX,    8A CPU TXA,    9A CPU TXS,
    98 CPU TYA,

DECIMAL



```

SCREEN #38
```
( ASSEMBLER: M/CPU )
HEX
: M/CPU CREATE C, , DOES>
   DUP 1+ @ 80 AND IF 10 MODE +! THEN OVER
   FF00 AND UPMODE UPMODE IF MEM CR LATEST ID.
   ABORT" INCORRECT ADDRESSING" THEN C@ MODE C@
   INDEX + C@ + C, MODE C@ 7 AND IF MODE C@
   0F AND 7 < IF C, ELSE , THEN THEN MEM ;
1C6E 60 M/CPU ADC, 1C6E 20 M/CPU AND, 1C6E C0 M/CPU CMP,
1C6E 40 M/CPU EOR, 1C6E A0 M/CPU LDA, 1C6E  0 M/CPU ORA,
1C6E E0 M/CPU SBC, 1C6C 80 M/CPU STA, 0D0D  1 M/CPU ASL,
0C0C C1 M/CPU DEC, 0C0C E1 M/CPU INC, 0D0D 41 M/CPU LSR,
0D0D 21 M/CPU ROL, 0D0D 61 M/CPU ROR, 0414 81 M/CPU STX,
0486 E0 M/CPU CPX, 0486 C0 M/CPU CPY, 1496 A2 M/CPU LDX,
0C8E A0 M/CPU LDY, 048C 80 M/CPU STY, 0480 14 M/CPU JSR,
8480 40 M/CPU JMP, 0484 20 M/CPU BIT, DECIMAL
```

SCREEN #39
```
( ASSEMBLER: BEGIN, UNTIL, IF, THEN, ELSE, NOT BRANCHES )
: BEGIN, HERE 1 ;
: UNTIL,  >R 1 ?PAIRS R> C, HERE 1+ - C, ;
: IF, C, HERE 0 C, 2 ;
: THEN,  2 ?PAIRS HERE OVER C@
   IF SWAP !  ELSE OVER 1+ - SWAP C! THEN ;
: ELSE, 2 ?PAIRS HERE 1+ 1 JMP,
   SWAP HERE OVER 1+ - SWAP C! 2 ;
 HEX
: NOT  20 + ;    ( REVERSE ASSEMBLY TEST )
90 CONSTANT CS  ( ASSEMBLE TEST FOR CARRY SET )
D0 CONSTANT 0=  ( ASSEMBLER TEST FOR EQUAL ZERO )
10 CONSTANT 0<  ( ASSEMBLE TEST FOR LESS THAN OR EQUAL ZERO )
90 CONSTANT >=  ( ASSEMBLE TEST FOR GREATER OR EQUAL ZERO )
                ( >= IS ONLY CORRECT AFTER SUB, OR CMP, )
50 CONSTANT VS  DECIMAL
```

SCREEN #40
```
( ASSEMBLER:  AGAIN, WHILE, REPEAT )

: AGAIN, 1 ?PAIRS JMP, ;

: WHILE, >R DUP 1 ?PAIRS R> IF, 2+ ;

: REPEAT, >R >R 1 ?PAIRS JMP, R> R> 2 - THEN, ;









```

SCREEN #41
```
( ASSEMBLER: END-CODE ENTERCODE ;CODE CODE )
: END-CODE   CURRENT @ CONTEXT ! SP@ 2+ =
   IF SMUDGE
   ELSE ." CODE ERROR, STACK DEPTH CHANGE"
   THEN ;
FORTH DEFINITIONS
: ENTERCODE [COMPILE] ASSEMBLER SP@ ;
: CODE CREATE SMUDGE HERE DUP 2- !
   ASSEMBLER MEM ENTERCODE ;   IMMEDIATE
: ;CODE ?CSP COMPILE <;CODE> [COMPILE]  [ ENTERCODE ; IMMEDIATE
   EXIT



( THIS 6510 FORTH ASSEMBLER WAS WRITTEN BY WILLIAM F. RAGSDALE )
( IT WAS PUBLISHED IN "FORTH DIMENSIONS", VOL. III # 5         )
SCREEN #42
 0) ( EDITOR: CHKLIN LINE PP C )
 1) : CHKLIN     ( LINE# --- LINE# )
 2)    DUP C/L * 1023 > ABORT" OFF SCREEN" ;
 3)
 4) : LINE ( LINE# --- BUF-ADDR COUNT )
 5)    CHKLIN DUP R# ! SCR @ <LINE> UPDATE ;
 6)
 7) : PP    ( LINE# --- )
 8)    PAD 1+ SWAP LINE 1 TEXT CMOVE ;
 9)
10) VOCABULARY EDITOR IMMEDIATE
11)
12) EDITOR DEFINITIONS
13) : C     ( FROM# TO# --- )
14)    SWAP LINE DROP SWAP LINE CMOVE ;
15)

SCREEN #43
 0) ( EDITOR: SCREEN COMMANDS )
 1) : SE DUP R# ! PP QUIT ;
 2) :  0) 0 SE ;     :  1)  1 SE ;
 3) :  2) 2 SE ;     :  3)  3 SE ;
 4) :  4) 4 SE ;     :  5)  5 SE ;
 5) :  6) 6 SE ;     :  7)  7 SE ;
 6) :  8) 8 SE ;     :  9)  9 SE ;
 7) : 10) 10 SE ;    : 11) 11 SE ;
 8) : 12) 12 SE ;    : 13) 13 SE ;
 9) : 14) 14 SE ;    : 15) 15 SE ;
10) EXIT  ( EXT. FOR > 15 LINES/SCR )
11) : 16) 16 SE ;    : 17) 17 SE ;
12) : 18) 18 SE ;    : 19) 19 SE ;
13) : 20) 20 SE ;    : 21) 21 SE ;
14) : 22) 22 SE ;    : 23) 23 SE ;
15) : 24) 24 SE ;    : 25) 25 SE ;

SCREEN #44
 0) ( EDITOR: K X O M )
 1) ( KILL - REPLACE LINE WITH BLANKS )
 2) : K       ( LINE# --- ) LINE BLANK ;
 3) : LL      ( --- LAST.LINE.+1.ADDR )
 4)    SCR @ BLOCK 1024 + ;
 5) ( X-TRACT A LINE FROM SCREEN )
 6) : X       ( LINE# --- )
 7)    LINE OVER + SWAP OVER LL SWAP -
 8)    CMOVE LL C/L - C/L BLANK ;
 9) ( OPEN A LINE FOR INPUT )
10) : O       ( LINE# --- )
11)    DUP LINE OVER + LL
12)    OVER -  <CMOVE K ;
13) ( MOVE LINE )
14) : M       ( FROM TO --- )
15)    OVER SWAP C X ;
```

SCREEN #45
```
( EDITOR: F L W N P SC SM LIST )
: F  PREV @ DUP @ DUP 0<
   IF 32767 AND SWAP DDUP ! 2+ SWAP 0 R/W ELSE DDROP THEN ;
: L  ( [SCR] --- ) DEPTH IF F ELSE SCR @ THEN PAGE LIST ;
: W   SCR @ BLOCK 1024 BLANK UPDATE L ;
: N   F  1 SCR +! ;
: P   F -1 SCR +! ;
( COPY LINE FROM DIFF SCREEN )
: SC   ( FR-SCR FR-LINE --- )
   CHKLIN  SWAP  <LINE>  DROP
   R# @  DUP  O LINE  CMOVE ;
( MOVE LINE FROM DIFF SCREEN )
: SM   ( FR-SCR FR-LINE --- )
   SCR @  R# @  DSWAP  DDUP  SC
   SWAP SCR !  X   R# !   SCR ! ;
FORTH DEFINITIONS : LIST LIST [COMPILE] EDITOR ;
```

SCREEN #46
```
( DECOMPILER: GIN  GIN+  GCHK )
VARIABLE GIN  ( # TO INDENT )
: GIN+ CR GIN @ 2+ DUP GIN ! SPACES ;
: GCHK  DUP @ 2+ ' COMPILE =
   IF SPACE 2+ DUP @ 2+ NFA ID. 2+
   ELSE DUP @ 2+ DUP ' <LIT> =
        OVER ' BRANCH = OR
        OVER ' 0BRANCH = OR
        OVER ' <LOOP> = OR
        OVER ' </LOOP> = OR
        SWAP ' <+LOOP> = OR
     IF 2+ DUP @ SPACE . 2+
     ELSE DUP @ 2+ DUP ' <."> = SWAP ' <ABORT"> = OR
       IF SPACE 2+ DUP COUNT TYPE DUP C@ 1+  +
       ELSE 2+ THEN  THEN
   THEN -2 GIN +! ;
```

SCREEN #47
```
( DECOMPILER: <DECOM>  )
: <DECOM>  ( PFA --- )
   DUP CFA @ ' : CFA @ =
  IF ( COLON DEF. )
    BEGIN DUP @ DUP ' EXIT CFA =
      OVER ' <;CODE> CFA = OR 0=
    WHILE ( HIGH LEVEL & NOT END OF COLON DEF )
      2+ DUP GIN+ NFA ID. KEY DUP 81 =
     IF ( 'Q' ) SP! QUIT
     ELSE 13 =  ( RETURN )
       IF RECURSE ( GO DOWN ONE LEVEL )
       ELSE  DROP
       THEN
     THEN  GCHK  REPEAT ( SHOW LAST WORD )
   2+ CR GIN @ SPACES NFA ID.
  THEN DROP ;
```

SCREEN #48

```
( DECOMPILER: DECOMPILE )
: DECOMPILE -FIND IF DROP 0 GIN !
  <DECOM> ELSE ." NOT FOUND" THEN ;
```

SCREEN #49

```
( TRIG: TSCALE )
CODE TSCALE  ( D --- N )
 BOT ASL, BOT 1+ ROL,
 SEC 1+ LDA,
 .A LSR, .A LSR, .A LSR, .A LSR, .A LSR, .A LSR, .A LSR,
 BOT ORA, SEC STA,
 BOT 1+ LDA, SEC 1+ STA,
 0< IF, SEC INC,
  0= IF, SEC 1+ INC, THEN,
 THEN,
POP JMP,  END-CODE
```

SCREEN #50

```
( TRIG: SIN/COS VALUES TABLE - SCALED BY 32768 )
CREATE SINTABLE
 00000 , 00572 , 01144 , 01715 , 02286 , 02856 , 03425 , 03993 ,
 04560 , 05126 , 05690 , 06252 , 06813 , 07371 , 07927 , 08481 ,
 09032 , 09580 , 10126 , 10668 , 11207 , 11743 , 12275 , 12803 ,
 13328 , 13848 , 14365 , 14876 , 15384 , 15886 , 16384 , 16877 ,
 17364 , 17847 , 18324 , 18795 , 19261 , 19720 , 20174 , 20622 ,
 21063 , 21498 , 21926 , 22348 , 22763 , 23170 , 23571 , 23965 ,
 24351 , 24730 , 25102 , 25466 , 25822 , 26170 , 26510 , 26842 ,
 27166 , 27482 , 27789 , 28088 , 28378 , 28660 , 28932 , 29197 ,
 29452 , 29698 , 29935 , 30163 , 30382 , 30592 , 30792 , 30983 ,
 31164 , 31336 , 31499 , 31651 , 31795 , 31928 , 32052 , 32166 ,
 32270 , 32365 , 32449 , 32524 , 32588 , 32643 , 32688 , 32723 ,
 32748 , 32763 , 32767 ,
```

SCREEN #51
```
( TRIG: <SIN> <COS> QSIN QCOS SIN COS )
CODE <SIN> BOT ASL, BOT LDY, SINTABLE ,Y LDA, BOT STA,
 SINTABLE 1+ ,Y LDA, BOT 1+ STA, NEXT JMP, END-CODE
CODE <COS> SEC, 90 # LDA, BOT SBC, BOT STA,
 ' <SIN> JMP, END-CODE
: S180 DUP 90 >  IF 180 SWAP - THEN <SIN> ;
: QSIN  DUP 180 > IF ( 181-359 DEG)
   180 - S180 NEGATE ELSE S180 THEN ;
: QCOS  DUP 270 > IF 270 - ELSE 90 + THEN QSIN ;
: SIN ( DEGREES --- SINE*10000 )
   DUP ABS 359 >
    IF 360 MOD THEN ( DOESN'T CHANGE SIN VALUE )
   DUP 0< IF 360 + THEN ( HANDLE NEGATIVE ARGUMENT )
   QSIN ;
: COS ( DEGREES --- COSINE*10000 )
   90 + SIN ;
```

SCREEN #52
```
( MATH: SQUARE ROOT  ROUTINES )
CODE D2* SEC ASL, SEC 1+ ROL,
  BOT ROL, BOT 1+ ROL, NEXT JMP, END-CODE
: EASY-BITS ( DREM1 PARTIALROOT1 COUNT --- DREM2 PARTIALROOT2 )
   0 DO
    >R D2* D2*               ( SHIFT DREM TWICE )
    R@ - DUP                 ( SUBR. PARTIAL ROOT )
    0< IF R@ +   R> 2* 1-    ( RESTORE DREM & SET 0 )
      ELSE  R> 2* 3 +          ( OR SET 1 )
      THEN ( PROOT SHIFTED FOR NEXT GO-ROUND ) LOOP ;
: 2'S-BIT   ( DREM2 PROOT2 --- DREM3 PROOT3 ; GET PENULT. BIT )
   >R D2*  DUP 0< IF    D2* R@ -    R> 1+
                  ELSE  D2* R@ DDUP
                        U< IF DROP R> 1-     ( SET 0 )
                           ELSE - R>  1+     ( SET 1 )
                  THEN THEN ;
```

SCREEN #53
```
( MATH: SQUARE ROOT ROUTINES )
: 1'S-BIT   ( DREM3 PROOT3 --- FULLROOT ; REMAINDER LOST )
   >R DUP 0< IF  DDROP R> 1+
             ELSE D2* 32768 R@ DU< 0= R> + THEN ;

: SQRT ( UD1 --- U2 ; 32-BIT UNSIGNED RADICAND--> 16-BIT ROOT)
   0 1 8 EASY-BITS  ROT DROP 6 EASY-BITS
   2'S-BIT  1'S-BIT ;
EXIT

( THIS WAS WRITTEN BY KLAXON SURALIS AND PUBLISHED IN
  FORTH DIMENSIONS VOL. IV NO. 1 )




```

SCREEN #54
```
( F.P. MATH: FPSW FRESET FER FZE FNE FOV )
CREATE FPSW 0 C,          ( FLOATING POINT STATUS WORD )
CREATE FBASE 10 C,        ( BASE )
: FRESET ( --- )          ( CLEAR CONDITION CODES )
   0 FPSW C! ;

: FER ( --- N )           ( RETURNS SUM OF CONDITION CODES )
   FPSW C@ ;
: FZE ( --- N )           ( TRUE IF LAST F# WAS ZERO )
   FER 1 AND 0= NOT ;
: FNE ( --- N )           ( TRUE IF LAST F# WAS < ZERO )
   FER 2 AND 0= NOT ;
: FOV ( --- N )           ( TRUE IF LAST OPERATION OVERFLOWED )
   FER 4 AND 0= NOT ;


```

SCREEN #55
```
( F.P. MATH: SFZ SFN E@ )
( CC'S 8, 16, 32, 64 AND 128 ARE AVAILABLE FOR USE )
HEX
: SFZ  ( F# --- F# ; Z )   ( SETS Z ACCORDING TO F# )
   FER FFFE AND FPSW C!    ( RESET Z )
   DDUP 00FF AND D0= FER OR FPSW C! ;
: SFN  ( F# --- F# ; N )   ( SETS N ACCORDING TO F# )
   FER FFFD AND FPSW C!    ( RESET N )
   DUP 0080 AND 40 / FER OR FPSW C! ;
: E@    ( F# --- M E ; Z N )  ( REMOVE EXPONENT )

   FRESET  SFZ  SFN        ( SET FLAGS )
   DUP  FF00 AND 100 /  >R  ( OBTAIN EXPONENT )
   FNE IF FF00 OR          ( SIGN EXTEND MANTISSA )
       ELSE 00FF AND THEN   R> ;
DECIMAL
```

SCREEN #56
```
( F.P. MATH: E! E. )
HEX

: E! ( M E --- F# ; V Z N )  (  RESTORE EXPONENT )
   DUP 100 *  DUP 100 / ROT = NOT
    IF 4 FPSW C! THEN         ( EXPONENT OVERFLOW )
   SWAP DUP FF00 AND DUP
    IF DUP FF00 = NOT
     IF 4 FPSW C! THEN        ( MANTISSA OVERFLOW )
    THEN
   DROP 00FF AND  OR  SFZ SFN ;

: E.   ( F# --- ; Z N )
   E@  <ROT D. ." . E" . ;

DECIMAL
```

SCREEN #57
```
( F.P. MATH: F. F* F/ )
: F.   ( F# --- ; Z N )
   E@ >R   SWAP OVER DABS
   <#  R@   0<
    IF I ABS 0 DO  #  LOOP  46 HOLD
    ELSE 46 HOLD  R@
      IF  R@ 0 DO 48 HOLD LOOP THEN
    THEN  R> DROP  #S ROT SIGN #> TYPE SPACE ;

: F*  ( F#1 F#2 --- F# ; N Z V )  ( MULTIPLY )
   DSWAP E@ >R   DSWAP E@  >R
   DROP 1 M*/ R>  R> +  E! ;

: F/  ( F#1 F#2 --- F# ; N Z V )  ( DIVIDE )
   DSWAP E@ >R   DSWAP E@ >R
   DROP 1 SWAP M*/ R> R> SWAP - E! ;
```

SCREEN #58
```
( F.P. MATH: ALIGN F+ F- )
: ALIGN ( M1 E1  M2 E2 --- M1 M2 E )
   4 ROLL
   BEGIN   DDUP = NOT
   WHILE  DDUP >   ( E2 > E1? )
    IF >R >R FBASE C@ 1 M*/ R> 1- R>
    ELSE >R >R DSWAP FBASE C@ 1 M*/ DSWAP R> R> 1- THEN
   REPEAT DROP ;

: F+  ( F#1 F#2 --- FSUM ; N V Z )
   E@  >R  DSWAP R> <ROT E@
   ALIGN  >R  D+  R>  E! ;

: F-  ( F#1 F#2 --- FDIFF ; N V Z )
   DSWAP E@  >R  DSWAP R> <ROT E@
   ALIGN >R  D-  R>  E! ;
```

SCREEN #59
```
( F.P. MATH: RSCALE LSCALE DFIX )
: RSCALE   ( F# --- F# ; N Z V )
   E@  1- <ROT   FBASE C@ 1 M*/ ROT   E! ;

: LSCALE   ( F# --- F# ; N Z V )
   E@  1+ <ROT   1 FBASE C@ M*/ ROT   E! ;

: DFIX   ( F# --- D ; V Z N )
   E@
    BEGIN ?DUP
    WHILE  DUP 0>
     IF 1- <ROT FBASE C@ 1 M*/
     ELSE 1+ <ROT  1 FBASE C@ M*/ DDUP D0=
      IF 5 FPSW C! ROT DROP 0 <ROT THEN ( UNDERFLOW )
     THEN ROT
    REPEAT ;
```

SCREEN #60

```
( F.P. MATH: FIX DFLOAT FLOAT FSIN FCOS FSQRT FINIT FEXIT )
: FIX   ( F# --- N )  DFIX DROP ;
: DFLOAT ( D --- F# ) 0 E! ;
: FLOAT  ( N --- F# ) S->D DFLOAT ;
: FSIN   ( FDEG --- FSINE )
   FIX SIN 10000 M* TSCALE S->D -4 E! ;
: FCOS   ( FDEG --- FCOSINE )
   90. F+ FSIN ;
: FSQRT ( F# --- FSQRT )
   FIX 10000 U* SQRT S->D -2 E! ;
: <FNUM> ( ADDR --- D )
   <NUMBER> DPL @ NEGATE E! ;
: FINIT  ( --- )
   FRESET BASE @ FBASE C!
   ' <FNUM> CFA 'NUMBER ! ;
: FEXIT  ( --- )  ' <NUMBER> CFA 'NUMBER ! ;
```

SCREEN #61

```
( F.P. MATH: FABS FNEGATE FMIN F> FMAX )
: FABS   ( F# --- ABS[F#] : N Z V )
   E@   <ROT DABS ROT E! ;
: FNEGATE  ( F# --- -F# : N Z V )
   E@  <ROT  DNEGATE  ROT  E! ;
: FMIN   ( F#1 F#2 --- MIN[F#S] ; N Z V )
   DSWAP  E@  >R   DSWAP R> <ROT E@
   ALIGN >R DMIN R> E! ;
: F>  ( F#1 F#2 --- B ; N Z V )
   F- DDROP FNE ;
: FMAX ( F#1 F#2 --- MAX[F#S] ; N Z V )
   DOVER DOVER FMIN F- F+ ;

( THESE ROUTINES WERE ADAPTED FROM THE ARTICLE BY MICHAEL JESCH
  PUBLISHED IN FORTH DIMENSIONS, VOL. IV NO. 1 )

```

SCREEN #62

```
( C64 DATA: COLORS SPRITE-DEFS )
0 CONSTANT BLACK    1 CONSTANT WHITE     2 CONSTANT RED
3 CONSTANT CYAN     4 CONSTANT PURPLE    5 CONSTANT GREEN
6 CONSTANT BLUE     7 CONSTANT YELLOW    8 CONSTANT ORANGE
9 CONSTANT BROWN   10 CONSTANT LT.RED   11 CONSTANT DK.GRAY
12 CONSTANT MED.GRAY 13 CONSTANT LT.GREEN
14 CONSTANT LT.BLUE  15 CONSTANT LT.GRAY

CODE S1 SPRITE STY, NEXT JMP, END-CODE
CODE S2 INY, ' S1 JMP, END-CODE
CODE S3 2 # LDY, ' S1 JMP, END-CODE
CODE S4 3 # LDY, ' S1 JMP, END-CODE
CODE S5 4 # LDY, ' S1 JMP, END-CODE
CODE S6 5 # LDY, ' S1 JMP, END-CODE
CODE S7 6 # LDY, ' S1 JMP, END-CODE
CODE S8 7 # LDY, ' S1 JMP, END-CODE
```

SCREEN #63
```
( C64 DATA: SOUND )
( CTRL REG CONSTANTS )    HEX
11 CONSTANT TRIANGLE
21 CONSTANT SAWTOOTH
41 CONSTANT PULSE
81 CONSTANT NOISE
15 CONSTANT RING
 8 CONSTANT RESET
 3 CONSTANT SYNC
    DECIMAL
( FILTER CONSTANTS )
 1 CONSTANT FILT1  2 CONSTANT FILT2
 4 CONSTANT FILT3  8 CONSTANT FILTEX
 1 CONSTANT LOWPASS 2 CONSTANT BANDPASS
 4 CONSTANT HIGHPASS
 LOWPASS HIGHPASS OR CONSTANT NOTCH
```

SCREEN #64
```
( C64 DATA: SOUND MISC I/O )
CREATE NOTE-VALUES 34334 , 36376 , 38539 , 40830 , 43258 ,
45830 , 48556 , 51443 , 54502 , 57743 , 61176 , 64814 ,
: NOTE@ ( # --- FREQ ) 2* NOTE-VALUES + @ ;

8 CONSTANT 3OFF
   HEX
CODE V1 VOICE STY, NEXT JMP, END-CODE
CODE V2 INY, ' V1 JMP, END-CODE
CODE V3 2 # LDY, ' V1 JMP, END-CODE

0 CONSTANT OFF     1 CONSTANT ON
DC01 CONSTANT JOY1
DC00 CONSTANT JOY2
DD01 CONSTANT UPORT
D800 CONSTANT COLOR-MEM     DECIMAL
```

SCREEN #65
```
( DATA STRUCTURES: S-DEF )
( COMPILE: ALLOCATES ROOM & READS SPRITE DATA FROM INPUT STREAM)
( EXEC: MOVES SPRITE DATA TO ADDR ON STACK )
: S-DEF
   CREATE   ( [63 BYTE VALUES IN INPUT STREAM] ; --- )
    63 0 DO
     BL TEXT PAD NUMBER DROP C,
    LOOP
   DOES>    ( ADDR --- )
    SWAP 63 CMOVE ;
```

SCREEN #66

```
( DATA STRUCTURES: 1ARRAY 2ARRAY )
( CREATES & GETS ELEMENT FROM 16-BIT 1 DIMENSIONAL ARRAY )
: 1ARRAY
   CREATE     ( #ELEM --- )
    2* ALLOT
   DOES>      ( ELEMENT --- ADDR )
    SWAP 2* + ;

( CREATES & GETS ELEMENT FROM 16-BIT 2 DIMENSIONAL ARRAY )
: 2ARRAY
   CREATE     ( #Y-ELEM #X-ELEM --- )
    DUP , * 2* ALLOT
   DOES>      ( Y-ELEM X-ELEM --- ADDR )
    >R R@ @ ROT * + ( GET X-MAX * Y + X )
    2* R> + 2+      ( MAKE 2 BYTE & ADD ARRAY BASE )
 ;
```

SCREEN #67

```
( GRAPHICS: M-X M-Y B-MFLAG S.DIST L.DIST )

( MIRROR CENTER COORDINATES )
0 CONSTANT M-X
0 CONSTANT M-Y

( MIRROR FLAG )
VARIABLE B-MFLAG  0 B-MFLAG !

( LINE VARIABLES )
VARIABLE S.DIST    VARIABLE L.DIST
```

SCREEN #68

```
( GRAPHICS: BORDER BKGND M-ORIGIN )
( CHANGE BORDER COLOR )
: BORDER ( C --- ) 53280 C! ;

( CHANGE A BACKGROUND REGISTER )
: BKGND ( [REG] C --- )
   53281 DEPTH 2-
    IF ROT + THEN C! ;

CODE M-ORIGIN  ( X Y --- )
 SEC LDA,   ' M-X STA,
 SEC 1+ LDA, ' M-X 1+ STA,
 BOT LDA,   ' M-Y STA,
 POPTWO JMP,
END-CODE
```

SCREEN #69
```
( GRAPHICS: R-PLOT )
( PLOT A POINT X,Y RELATIVE TO M-X,M-Y )
CODE R-PLOT ( X Y --- )
 CLC,
 ' M-X LDA,  SEC ADC,  SEC STA,
 ' M-X 1+ LDA,  SEC 1+ ADC,  SEC 1+ STA,
 CLC,
 ' M-Y LDA,  BOT ADC,  BOT STA,
 ' M-Y 1+ LDA,  BOT 1+ ADC,  BOT 1+ STA,
 ' B-PLOT JMP,
END-CODE
```

SCREEN #70
```
( GRAPHICS: M-PLOT )
( M-PLOT 4 POINTS AROUND M-X,M-Y )
: M-PLOT  ( X Y --- )
   B-MFLAG @ IF
    DDUP NEGATE R-PLOT (  X -Y )
    OVER NEGATE OVER DDUP R-PLOT ( -X Y )
    NEGATE R-PLOT ( -X -Y )
    R-PLOT ( X Y )
   ELSE
    R-PLOT
   THEN ;
```

SCREEN #71
```
( GRAPHICS: CHAR )
CODE CHAR  ( 'SCREEN --- )
 HEX  ( B-Y F8 AND 5 * B-X 3 RSHIFT + + 85 C@ SWAP C! )
 ' B-Y LDA,  F8 # AND,  N STA, N 1+ STY,
 CLC, .A ASL, N 1+ ROL, .A ASL, N 1+ ROL, N ADC, N STA,
 TYA, N 1+ ADC, N 1+ STA,
 ' B-X 1+ LDA, .A LSR,
 ' B-X LDA, .A ROR, .A LSR, .A LSR,
 CLC, N ADC, BOT STA,
  BOT 1+ LDA, N 1+ ADC, BOT 1+ STA,
 85 LDA, BOT ,X) STA,
 POP JMP,
END-CODE

DECIMAL
```

SCREEN #72

```
( GRAPHICS: B-CPLOT <B-LINE> )

: B-CPLOT B-PLOT 'SCREEN CHAR ;

CODE <B-LINE>
 CLC,
 S.DIST LDA,  BOT ADC,  BOT STA,
 S.DIST 1+ LDA,  BOT 1+ ADC, BOT 1+ STA,
 0< IF,    CLC,
  4 ,X LDA,  ' B-X ADC,  -2 ,X STA,
  5 ,X LDA,  ' B-X 1+ ADC,  -1 ,X STA,
  CLC,
  2 ,X LDA,  ' B-Y ADC,  -4 ,X STA,
  3 ,X LDA,  ' B-Y 1+ ADC,  -3 ,X STA,
 ELSE,

```

SCREEN #73

```
( GRAPHICS: <B-LINE> )
  SEC,
  BOT LDA,  L.DIST SBC,  BOT STA,
  BOT 1+ LDA,  L.DIST 1+ SBC,  BOT 1+ STA,
  CLC,
  8 ,X LDA,  ' B-X ADC,  -2 ,X STA,
  9 ,X LDA,  ' B-X 1+ ADC,  -1 ,X STA,
  CLC,
  6 ,X LDA,  ' B-Y ADC,  -4 ,X STA,
  7 ,X LDA,  ' B-Y 1+ ADC,  -3 ,X STA,
 THEN,
 DEX, DEX, DEX, DEX,
 NEXT JMP,
END-CODE


```

SCREEN #74

```
( GRAPHICS: LSETUP )
: LSETUP ( X Y --- )
   B-Y - ( DY )  1 OVER +- ( SIGN OF DY )
   >R  ABS SWAP
   B-X - ( DX )  1 OVER +- ( SIGN OF DX )  >R ABS
   DDUP MIN S.DIST !  DDUP MAX L.DIST !  > ( DY > DX? )
   R> R> DDUP 5 ROLL
   IF   SWAP DROP 0 SWAP B-Y
   ELSE DROP 0 B-X THEN
   L.DIST @ MOD L.DIST @ -
   L.DIST @ 0 ;





```

SCREEN #75
```
( GRAPHICS: B-LINE )
: B-LINE   ( X Y --- )
   LSETUP DO
    <B-LINE> B-PLOT
   LOOP
  DDROP DDROP DROP ;

: B-CLINE   ( X Y --- )
   LSETUP DO
    <B-LINE> B-CPLOT
   LOOP
   DDROP DDROP DROP ;




```

SCREEN #76
```
( GRAPHICS: ELLIPSE CIRCLE )
: ELLIPSE ( X Y HR VR --- )
   DSWAP  M-ORIGIN   ON B-MFLAG !
   DDUP MAX 360. ROT U/MOD SWAP DROP >R
   364 0 DO
    OVER I 2 RSHIFT <COS> U* TSCALE
    OVER I 2 RSHIFT <SIN> U* TSCALE
    M-PLOT
   J +LOOP
  R> DDROP DROP OFF B-MFLAG ! ;


: CIRCLE ( X Y R --- ) DUP 3 4 */ ELLIPSE ;



```

SCREEN #77
```
( GRAPHICS: ARC )
: ARC ( HR VR STRT END --- )
   >R >R DDUP MAX 360 SWAP /
   R> 2* 2* R> 1+ 2* 2* SWAP ROT >R
   DO
    OVER I 2 RSHIFT QCOS M* TSCALE
    OVER I 2 RSHIFT QSIN M* TSCALE
    M-PLOT
   J +LOOP
  R> DDROP DROP ;






```

SCREEN #78
```
( I/O EXTENSIONS: CMD CMDI INPUT INPUT# PRINT# $INPUT PRINTER )
: CMD    ( LFN --- ) OUTLFN ! ;
: CMDI   ( LFN --- ) INPLFN ! ;

: INPUT ( --- N )
   QUERY BL WORD NUMBER DROP ;
: INPUT# ( LFN --- N )
   CMDI INPUT ;
: PRINT#    ( N LFN --- )
   CMD .  ;
: $INPUT   ( --- ADDR )
   QUERY 1 TEXT PAD ;

: PRINTER ( FLAG --- )
   IF 127 4 0 "" OPEN  127 CMD
   ELSE CR 127 CLOSE   0 CMD THEN ;
```

SCREEN #79
```
( I/O EXTENSIONS: GET# PUT# RS232 FRE RDTIM )
: GET#   ( LFN --- N ) CMDI ?TERMINAL ;
: PUT#   ( N LFN --- ) CMD EMIT ;
: RS232   ( LFN ADDR --- )
   2 0 ROT OPEN ;

: FRE   ( --- N )
   FIRST HERE - U. ;

CODE RDTIM   ( --- D )
 DEX, DEX, SEI,
 162 LDA,  BOT STA,
 161 LDA,  BOT 1+ STA,
 160 LDA,  PHA,  TYA,
 CLI,  PUSH JMP,
END-CODE
```

SCREEN #80
```
( I/O EXTENSIONS: SETTIM WAIT )
( SET THE 60 CYCLE CLOCK VALUE )
CODE SETTIM ( D --- )
 SEI,  BOT LDA,   160 STA,
 SEC 1+ LDA, 161 STA,
 SEC LDA,   162 STA,
 CLI,  POPTWO JMP,
END-CODE

( WAIT N TICKS [1/60 SECONDS] )
: WAIT   ( N --- )
   S->D RDTIM D+
   BEGIN
    DDUP RDTIM
    D> NOT
   UNTIL DDROP ;
```

SCREEN #81
```
( UTILITIES: CASE OF ;; ENDCASE )
( EXECUTE CODE BASED ON STACK VALUE )
: CASE ?COMP CSP @ SP@ CSP ! 4 ;
IMMEDIATE

: OF
   4 ?PAIRS COMPILE OVER COMPILE
   = COMPILE 0BRANCH HERE 0 ,
   COMPILE DROP 5 ; IMMEDIATE

: ;;
   5 ?PAIRS  COMPILE BRANCH  HERE 0 ,
   SWAP 2 [COMPILE] THEN 4 ; IMMEDIATE
: ENDCASE 4 ?PAIRS COMPILE DROP
   BEGIN SP@ CSP @ = 0=
   WHILE 2 [COMPILE] THEN REPEAT CSP ! ; IMMEDIATE
```

SCREEN #82
```
( UTILITIES: DIR )
: DIR ( --- )
   PAD " $" LOADRAM   CR  PAD 2+
( MAIN LOOP - PRINT ENTRY )
 BEGIN
  DUP @ . 2+   ( PRINT #PAGES )
  BEGIN        ( PRINT TEXT )
   DUP C@ ?DUP
    WHILE EMIT 1+
  REPEAT   1+ CR
( CHECK FOR USER INTERVENTION )
  PAUSE ?TERMINAL IF QUIT THEN
( MORE TO DO? )
  DUP 2+ SWAP @
  NOT UNTIL  DROP ;

```

SCREEN #83
```
( UTILITIES: PATCH )
: PATCH  ( PATCH-ADDR --- NEXT-PATCH-ADDR )
   CR QUERY                 ( GET PATCH INPUT LINE )
   BEGIN
    DUP BL WORD             ( GET AN ENTRY )
    DUP COUNT SWAP DROP     ( ENTRY NOT NULL? )
     WHILE NUMBER DROP      ( CONVERT TO 16 BIT NUMBER )
      SWAP C! 1+            ( PUT AT PATCH ADDR & UPDATE ADDR )
   REPEAT  DDROP ;
```

SCREEN #84
```
( UTILITIES: H@ HC@ H! HC! )
( HI-RAM ACCESS ROUTINES )
: H@   ( --- N )
    SWAPOUT @ SWAPIN ;

: HC@ ( --- N )
    SWAPOUT C@ SWAPIN ;

: H!  ( N --- )
    SWAPOUT ! SWAPIN ;

: HC! ( N --- )
    SWAPOUT C! SWAPIN ;



```

SCREEN #85
```
( MUSIC EDIT: WAVE PLAY.NOTE V-DEFAULT SOUND.INIT )
VARIABLE WAVE
: PLAY.NOTE ( NOTE# --- )
    12 /MOD     ( DETERMINE OCTAVE & NOTE )
    7 XOR      ( GET VALUE TO DIVIDE BY )
    SWAP NOTE@ ( GET NOTE VALUE )
    SWAP RSHIFT ( DIVIDE FOR OCTAVE)
    V-FREQ      ( SET FREQUENCY )
    OFF V-CTRL ( CLEAR PREVIOUS NOTE )
    WAVE @ V-CTRL ; ( PLAY NEW NOTE )

: V-DEFAULT  ( --- )
    RESET V-CTRL  9 12 V-AD  SAWTOOTH WAVE ! ;
: SOUND.INIT ( --- )
    SID @ 25 0 FILL  ( CLEAR SID )
    V3 V-DEFAULT  V2 V-DEFAULT  V1 V-DEFAULT 0 15 MODEVOL ;
```

SCREEN #86
```
( MUSIC EDIT: T@ T! )
0 CONSTANT T@

CODE T!
 BOT LDA, ' T@ STA, POP JMP,
END-CODE










```

SCREEN #87
```
( MUSIC EDIT: O@ O! OX )
( RETURN OCTAVE VALUE )
0 CONSTANT O@

CODE O!
 BOT LDA, ' O@ STA, POP JMP,
END-CODE

: O0 0 O! ;   : O1 12 O! ;
: O2 24 O! ;   : O3 36 O! ;
: O4 48 O! ;   : O5 60 O! ;
: O6 72 O! ;   : O7 84 O! ;




```

SCREEN #88
```
( MUSIC EDIT: TEMPO DURATION TIMINGS )
VARIABLE TEMPO   VARIABLE DURATION
( DURATIONS BASED ON 64TH NOTE RESOLUTION - 4/4 TIME ASSUMED )
: WHOLE 192 DURATION ! ;
: .1/2 144 DURATION ! ;
: 1/2 96 DURATION ! ;
: .1/4 72 DURATION ! ;
: 1/4 48 DURATION ! ;
: .1/8 36 DURATION ! ;
: 1/8 24 DURATION ! ;
: .1/16 18 DURATION ! ;
: 1/16 12 DURATION ! ;
: 1/32 6 DURATION ! ;
: 1/64 3 DURATION ! ;
: TRIPLET DURATION @ 2* 3 / DURATION ! ;
60 TEMPO ! 1/4
```

SCREEN #89
```
( MUSIC EDIT: NDEF NEXT.NOTE READY NCALC )
: NDEF CREATE ( --- ) 12 ALLOT
   DOES> ( --- ADDR )
    VOICE @ 2* 2* + ;

NDEF NEXT.NOTE

: READY ( --- FLAG )
   NEXT.NOTE D@  RDTIM D< ;

( CALCULATE TICKS TILL NEXT NOTE )
: NCALC DURATION @ 75 TEMPO @ */ 0
   NEXT.NOTE D@ D+ NEXT.NOTE D! ;

: SONG.INIT  RDTIM DDUP DDUP
   V3 NEXT.NOTE D! V2 NEXT.NOTE D! V1 NEXT.NOTE D! ;
```

SCREEN #90
```
( MUSIC EDIT: PLAY.WAIT )

: PLAY.WAIT  ( VALUE --- )
   BEGIN READY UNTIL
   O@ +          ( GET OCTAVE )
   T@ +          ( ADD TRANSPOSE )
   PLAY.NOTE
   NCALC ;

EXIT






```

SCREEN #91
```
( MUSIC EDIT: NOTE DEFS )
: C   0 PLAY.WAIT ;
: C#  1 PLAY.WAIT ;   : D\  C#  ;
: D   2 PLAY.WAIT ;
: D#  3 PLAY.WAIT ;   : E\  D#  ;
: E   4 PLAY.WAIT ;
: F   5 PLAY.WAIT ;
: F#  6 PLAY.WAIT ;   : G\  F#  ;
: G   7 PLAY.WAIT ;
: G#  8 PLAY.WAIT ;   : A\  G#  ;
: A   9 PLAY.WAIT ;
: A# 10 PLAY.WAIT ; : B\  A#  ;
: B  11 PLAY.WAIT ;
: R BEGIN READY UNTIL OFF V-CTRL NCALC ;
: TIE NCALC ;
FORTH DEFINITIONS
```

SCREEN #92
```
( S-EDITOR: S-EDITOR )
: S-EDITOR ( --- )
   CREATE 63 ALLOT PAGE
    ( DRAW SPRITE EASEL )
     21 0 DO SPACE SPACE
      24 0 DO 209 EMIT LOOP CR
     LOOP
     26 0 DO 184 EMIT LOOP
    ( CLEAR SPRITE AREA #13 )
     832 63 0 FILL
    ( ACTIVATE S1 AT AREA #13 )
     S1  ON S-ENABLE  13 S-POINTER BLUE S-COLOR
     270 130 S-POSITION ON S-XEXP ON S-YEXP
    CHARIN DROP ( SIGNAL END OF INPUT )
    0 22 D-POSITION
      -->
```

SCREEN #93
```
( S-EDITOR:  )
   832  ( SPRITE AREA # 13 )
   21 0 DO  ( ROWS )
    24 0 DO I J  ( ADDR COL ROW )
     40 * 2+ + 'SCREEN +
     8 0 DO  ( BIT# )
      DUP C@  81 = NOT
      3 PICK I 7 XOR MASK FBIT
     1+ LOOP
     DROP 1+
    8 +LOOP
   LOOP DROP
   832 LATEST PFA 63 CMOVE

   DOES> SWAP 63 CMOVE ;

```

SCREEN #94
```
( TURTLE: HEADING SETH SETXY PENFLG PU PD PM SETX SETY RT LT )
VARIABLE HEADING
: SETH ( ANGLE --- ) HEADING ! ;
: SETXY ( X Y --- )  ' B-Y !  ' B-X ! ;
VARIABLE PENFLG
: PU 0 PENFLG ! ;
: PD 1 PENFLG ! ;
: PM ( X Y --- )  PENFLG @ IF B-CLINE ELSE SETXY THEN ;
: SETX ( X --- ) B-Y PM ;
: SETY ( Y --- ) B-X SWAP PM ;

: RT ( N --- )
   HEADING @ + DUP 359 >
    IF 360 MOD THEN
    DUP 0< IF  360 + THEN SETH ;
: LT ( N --- ) NEGATE RT ;
```

SCREEN #95
```
( TURTLE: CS HOME PC BG TS DRAW )
: CS ( --- ) 0 B-FILL ;
: HOME ( --- ) 160 100 PM  0 SETH ;
: PC  ( COLOR --- )
   133 C@ 15 AND CATNIB 133 ! ;
: BG  ( COLOR --- )
   133 C@ 4 RSHIFT SWAP DDUP

   CATNIB 133 C!  B-COLOR-FILL ;
: TS ( --- ) OFF D-SPLIT ;
: SS ( --- ) TS 21 D-SPLIT 7 BITMAP
   PAGE  0 20 D-POSITION ;
: FS ( --- ) TS 7 BITMAP ON B-GRAPHICS ;
: DRAW ( --- )
   SS  CS   RED PC  CYAN BG
   PU HOME  PD ;
```

SCREEN #96
```
( TURTLE: FD BK )
: FD  ( N --- )
   DUP HEADING @ QSIN M* TSCALE B-X +
   SWAP HEADING @ QCOS M* TSCALE
   3 4 */  NEGATE  B-Y +
   PM ;

: BK  ( N --- )
   NEGATE FD ;
```

SCREEN #97
```
( TURTLE: LONG NAMES )
: BACK         BK ;
: BACKGROUND   BG ;
: CLEARSCREEN  CS ;
: FORWARD      FD ;
: FULLSCREEN   FS ;
: LEFT         LT ;
: PENCOLOR     PC ;
: PENDOWN      PD ;
: PENUP        PU ;
: RIGHT        RT ;
: SETHEADING   SETH ;
: SPLITSCREEN  SS ;
: TEXTSCREEN   TS ;
```

SCREEN #98
```
( TRACE COLON WORDS )
FORTH DEFINITIONS
CREATE TFLAG 1 ,
: TRACE TFLAG ! ;

: <TRACE>
   TFLAG @ IF
    CR R@ 2- NFA  ID. .S KEY DROP
   THEN ;
: : SP@ CSP ! CURRENT @ CONTEXT ! CREATE TFLAG @ IF
   ' <TRACE> CFA DUP @ HERE 2- ! , THEN SMUDGE ]
  ;CODE
   IP 1+ LDA, PHA, IP LDA, PHA, CLC, W LDA, 2 # ADC,
   IP STA, TYA, W 1+ ADC, IP 1+ STA,
   NEXT JMP,
  END-CODE
```

SCREEN #99
```
( CURVES: CONSTANTS DIRECTIONS )

4 CONSTANT +90DEG
+90DEG NEGATE CONSTANT -90DEG

( DIRECTIONS: TABLE OF 2 CELL RECORDS- XDIR*XLEN , YDIR*YLEN )
(  EACH RECORD DESCRIBES A 90 DEG CHANGE IN DIR FROM LAST REC )

CREATE DIRECTIONS
   0 , -1 ,  (   0 DEG )
   1 ,  0 ,  (  90 )
   0 ,  1 ,  ( 180 )
  -1 ,  0 ,  ( 270 )



```

SCREEN #100
```
( CURVES: DRAGON1 )  HEX
S-DEF DRAGON1
01  01  00      00  81  80
00  C1  C0      00  E1  C0
00  F1  E0      00  F9  80
00  FF  00      00  FF  FF
00  7F  FE      00  3F  F8
00  3F  E0      00  3F  80
00  3C  00      00  78  00
00  F0  00      01  E0  00
06  60  00      1C  70  00
38  38  00      70  1C  00
A8  2A  00
DECIMAL


```

SCREEN #101
```
( CURVES: DRAGON2 )     HEX
S-DEF DRAGON2
00  01  00     00  01  80
00  01  C0     00  01  C0
00  01  E0     00  01  80
03  FF  80     07  FF  E0
0F  FF  F0     0F  FF  F8
0F  FF  FC     0E  7D  FE
0C  F8  7E     09  F0  3E
03  E0  1E     07  60  1E
0E  60  00     1C  70  00
38  70  00     54  70  00
00  A8  00
DECIMAL


```

SCREEN #102
```
( CURVES: VARIABLES  NOTE NEXT-POINTER MOVE-SPRITE )
VARIABLE LENGTH
VARIABLE DIR-INDX

( SPRITE POINTER ALTERNATES BETWEEN SPRITE AREAS 0 & 1 )
: NEXT-POINTER   ( TOGGLE SPRITE POINTER )
   'SCREEN 1016 + SPRITE @ + DUP C@
    1 XOR SWAP C! ;

: MOVE-SPRITE    ( MOVE NEXT SPRITE TO NEW X,Y POSITION )
   NEXT-POINTER  B-X B-Y S-POSITION  ;





```

SCREEN #103
```
( CURVES: DRAWLINE  NEW-POINT DRAW MOVE-DIRECTION )

: NEW-POINT ( CALC. NEW X,Y COORD )
   DIR-INDX @  DIRECTIONS +
   DUP  @  B-X   +  SWAP  2 +
        @  B-Y   + ;

: LDRAW   ( DRAW A LINE )
   NEW-POINT B-LINE MOVE-SPRITE ;

: MOVE-DIRECTION  ( DIR --- )
   DIR-INDX @  + 15 AND  DIR-INDX ! ;




```

SCREEN #104
```
( CURVES: C-DRAW )
( DRAW A C-CURVE )
: C-DRAW  ( LEVEL --- )
   DUP 0=
   IF LDRAW
   ELSE
     DUP 1- RECURSE
     +90DEG MOVE-DIRECTION
     DUP 1- RECURSE
     -90DEG MOVE-DIRECTION
   THEN DROP ;





```

SCREEN #105
```
( CURVES: SPRITES-INIT  HI-RES-INIT ) HEX
( MOVE SPRITE-DATA TO 0 & 1 SPRITES IN BANK 2 )
: SPRITES-INIT  S1   ( SET SPRITE 1 AS ACTIVE SPRITE )
   C000 DRAGON1    ( MOVE DATA TO SPRITE AREAS )
   C040 DRAGON2    ( 2 POSITIONS OF DRAGON FOR ANIMATION )
   0 S-POINTER     ( SET POINTER TO SPRITE AREA 0 )
   RED S-COLOR     ( SET SPRITE COLOR TO RED )
   ON S-ENABLE     ( TURN ON SPRITE )
   ON S-XEXP ;     ( EXPAND IN X DIRECTION )

: HI-RES-INIT
   7 BITMAP  ON B-GRAPHICS  ( SET BITMAP AREA )
   0 B-FILL        ( CLEAR BITMAP AREA )
   YELLOW BLACK B-COLOR-FILL  ( SET BITMAP COLOR SCHEME )
   B-DRAW  A0 90 B-PLOT ; ( SPECIFY PEN DOWN & PLOT START PT )
DECIMAL
```

SCREEN #106
```
( CURVES: DIRECTIONS-INIT )
: DIRECTIONS-INIT
   LENGTH @
   DIRECTIONS DUP 16 + SWAP
    DO I @ IF DUP 1 I @ +-   *  I ! THEN
    2 /LOOP DROP ;
```

SCREEN #107
```
( CURVES: CURVE-INIT  C-CURVE )

: CURVE-INIT  ( LEVELS LENGTH DIR --- )
   DIR-INDX !
   LENGTH !
   DIRECTIONS-INIT
   HI-RES-INIT
   SPRITES-INIT ;


: C-CURVE ( LEVELS LENGTH --- )
   0 CURVE-INIT
   C-DRAW ;
```

SCREEN #108

```
( CURVES: CALL-RDRAGON  LDRAGON )

( ADDRESS OF RDRAGON GETS STUFFED INTO HERE ENABLING )
(    LDRAGON TO CALL RDRAGON BY CALLING THIS WORD )
: CALL-RDRAGON  0 ;

( MAKE A -90 DEGREE TURN )
: LDRAGON  ( LEVEL --- )
   DUP 0=
   IF LDRAW
   ELSE
     DUP 1- RECURSE
     -90DEG MOVE-DIRECTION
     DUP 1- CALL-RDRAGON
   THEN DROP ;

```

SCREEN #109

```
( CURVES: RDRAGON )

( MAKE A +90 DEGREE TURN )
: RDRAGON  ( LEVEL --- )
   DUP 0=
   IF LDRAW
     ELSE
       DUP 1- LDRAGON
       +90DEG MOVE-DIRECTION
       DUP 1- RECURSE
     THEN DROP ;

( SET UP SO LDRAGON CAN CALL RDRAGON )
' RDRAGON CFA ' CALL-RDRAGON !


```

SCREEN #110

```
( CURVES: D-CURVE WAIT-5-SEC DEMO )
: D-CURVE ( LEVELS LENGTH --- )
   +90DEG 2* CURVE-INIT
   LDRAGON ;

: DDEMO ( --- )
  24 D-SPLIT PAGE 3 24 D-POSITION
  ." *** FRACTALS BY SUPER-FORTH 64 *** "
   BEGIN
    7 1 DO
     I 2* I 7 XOR MASK D-CURVE
     SOUND.INIT 120 WAIT LOOP
    6 1 DO
     I 2* I 7 XOR 1- MASK C-CURVE
     SOUND.INIT 120 WAIT LOOP
   AGAIN ;
```

```
SCREEN #111
( BACKUP: TWO DRIVE BUFFER COPY )
( COPY BUFFERS FROM DR0 TO DR1 )
: COPYBUF ( FLAG END START --- )
   ( READ SOURCE SCREENS)
   DO I . I I BPDRV + COPY LOOP
    IF 53248 1 0 20 17 3 RWTS
    53248 0 1 20 17 3 RWTS DDROP
    THEN
   SAVE-BUFFERS CR ;

 113 LOAD






SCREEN #112
( BACKUP: COPYBUF )
( SINGLE DRIVE BUFFER COPY )
: COPYBUF ( FLAG END START --- )
   CR ." INSERT SOURCE-HIT KEY " KEY DROP
   ( READ SOURCE SCREENS)
    DO I . I BLOCK UPDATE DROP LOOP
   DUP IF 53248 1 0 20 17 3 RWTS DROP THEN
   CR ." INSERT DEST-HIT KEY " KEY DROP  SAVE-BUFFERS CR
   IF 53248 0 0 20 17 3 RWTS DROP THEN ;
  -->






SCREEN #113
( BACKUP: PCOPY BACKUP SOURCE-BACKUP )
( PERFORM A PARTIAL BACKUP COPY )
: PCOPY  ( FLAG START END --- )
   EMPTY-BUFFERS 1+ DDUP SWAP -
   #BUFF /MOD DROP OVER SWAP -
   DUP 1- 4 ROLL ( E+1 E+1-R E-R S --- )
   DDUP > IF ( CHECK IF SCREENS <= #BUFF )
    DO 0 I #BUFF + I COPYBUF
     #BUFF +LOOP ELSE DDROP THEN
   COPYBUF ;

: BACKUP  ( --- )
   1 0 169 PCOPY ;

: SOURCE-BACKUP  ( --- )
   0 0 130 PCOPY ;
```

SCREEN #114

```
( ASSEMBLER: CONSTANTS INDEX )
  LATEST HERE 36864 DP !   ( SET UP FOR ASSEMBLER REMOVAL )
VOCABULARY ASSEMBLER IMMEDIATE
HEX ASSEMBLER DEFINITIONS
( REGISTER ASSIGNMENTS SPECIFIC TO THIS IMPLEMENTATION )
83 CONSTANT XSAVE    81 CONSTANT W
04 CONSTANT IP       87 CONSTANT N

( NUCLEUS LOCATIONS SPECIFIC TO THIS IMPLEMENTATION)
0884 CONSTANT POP     0882 CONSTANT POPTWO
088B CONSTANT PUT     0889 CONSTANT PUSH
0890 CONSTANT NEXT    0871 CONSTANT SETUPN

VARIABLE INDEX -2  ALLOT
0909 , 1505 , 0115 , 8011 , 8009 , 1D0D , 8019 , 8080 ,
0080 , 1404 , 8014 , 8080 , 8080 , 1C0C , 801C , 2C80 , DECIMAL
```

SCREEN #115

```
( ASSEMBLER: MODE ADDRESSING MODES BOT SEC RP> UPMODE )
HEX
VARIABLE MODE 2 MODE !
: .A 0 MODE ! ; : # 1 MODE ! ;   : MEM 2 MODE ! ;
: ,X 3 MODE ! ; : ,Y 4 MODE ! ; : ,X) 5 MODE ! ;
: ),Y 6 MODE ! ; : ) F MODE ! ;

: BOT ,X 0 ;       ( ADDRESS THE BOTTOM OF DATA STACK )
: SEC ,X 2 ;       ( ADDRESS SECOND ITEM ON DATA STACK )
: RP> ,X 101    ; ( ADDRESS BOTTOM OF RETURN STACK )

: UPMODE IF MODE @   8 AND 0= IF 8 MODE +! THEN THEN
   1 MODE @ 0F AND ?DUP  IF 0 DO DUP + LOOP THEN
   OVER 1+ @ AND 0= ;

DECIMAL
```

SCREEN #116

```
( ASSEMBLER: CPU )

HEX
: CPU CREATE C, DOES>  C@ C, MEM ;
     00 CPU BRK,    18 CPU CLC,    DE CPU CLD,    58 CPU CLI,
     B8 CPU CLV,    CA CPU DEX,    88 CPU DEY,    E8 CPU INX,
     C8 CPU INY,    EA CPU NOP,    48 CPU PHA,    08 CPU PHP,
     68 CPU PLA,    28 CPU PLP,    40 CPU RTI,    60 CPU RTS,
     38 CPU SEC,    F8 CPU SED,    78 CPU SEI,    AA CPU TAX,
     A8 CPU TAY,    BA CPU TSX,    8A CPU TXA,    9A CPU TXS,
     98 CPU TYA,

DECIMAL



```

SCREEN #117
```
( ASSEMBLER: M/CPU )
HEX
: M/CPU CREATE C, , DOES>
   DUP 1+ @ 80 AND IF 10 MODE +! THEN OVER
   FF00 AND UPMODE UPMODE IF MEM CR
   ABORT" INCORRECT ADDRESSING" THEN C@ MODE C@
   INDEX + C@ + C, MODE C@ 7 AND IF MODE C@
   0F AND 7 < IF C, ELSE , THEN THEN MEM ;
1C6E 60 M/CPU ADC, 1C6E 20 M/CPU AND, 1C6E C0 M/CPU CMP,
1C6E 40 M/CPU EOR, 1C6E A0 M/CPU LDA, 1C6E  0 M/CPU ORA,
1C6E E0 M/CPU SBC, 1C6C 80 M/CPU STA, 0D0D  1 M/CPU ASL,
0C0C C1 M/CPU DEC, 0C0C E1 M/CPU INC, 0D0D 41 M/CPU LSR,
0D0D 21 M/CPU ROL, 0D0D 61 M/CPU ROR, 0414 81 M/CPU STX,
0486 E0 M/CPU CPX, 0486 C0 M/CPU CPY, 1496 A2 M/CPU LDX,
0C8E A0 M/CPU LDY, 048C 80 M/CPU STY, 0480 14 M/CPU JSR,
8480 40 M/CPU JMP, 0484 20 M/CPU BIT, DECIMAL
```

SCREEN #118
```
( ASSEMBLER: BEGIN, UNTIL, IF, THEN, ELSE, NOT BRANCHES )
: BEGIN, HERE 1 ;
: UNTIL,  >R 1 ?PAIRS R> C, HERE 1+ - C, ;
: IF, C, HERE 0 C, 2 ;
: THEN,  2 ?PAIRS HERE OVER C@
   IF SWAP !  ELSE OVER 1+ - SWAP C! THEN ;
: ELSE, 2 ?PAIRS HERE 1+ 1 JMP,
   SWAP HERE OVER 1+ - SWAP C! 2 ;
 HEX
: NOT  20 + ;    ( REVERSE ASSEMBLY TEST )
90 CONSTANT CS   ( ASSEMBLE TEST FOR CARRY SET )
D0 CONSTANT 0=   ( ASSEMBLER TEST FOR EQUAL ZERO )
10 CONSTANT 0<   ( ASSEMBLE TEST FOR LESS THAN OR EQUAL ZERO )
90 CONSTANT >=   ( ASSEMBLE TEST FOR GREATER OR EQUAL ZERO )
( .= IS ONLY CORRECT AFTER SUB, OR CMP, )
50 CONSTANT VS  DECIMAL
```

SCREEN #119
```
( ASSEMBLER:  AGAIN, WHILE, REPEAT )

: AGAIN, 1 ?PAIRS JMP, ;

: WHILE, >R DUP 1 ?PAIRS R> IF, 2+ ;

: REPEAT, >R >R 1 ?PAIRS JMP, R> R> 2 - THEN, ;
```

SCREEN #120
```
( ASSEMBLER: END-CODE ENTERCODE ;CODE CODE )
: END-CODE   CURRENT @ CONTEXT ! SP@ 2+ =
   IF SMUDGE
   ELSE ." CODE ERROR, STACK DEPTH CHANGE"
   THEN ;
FORTH DEFINITIONS
: ENTERCODE [COMPILE] ASSEMBLER SP@ ;
: CODE CREATE SMUDGE HERE DUP 2- !
   ASSEMBLER MEM ENTERCODE ;   IMMEDIATE
: ;CODE ?CSP COMPILE <;CODE> [COMPILE]  [ ENTERCODE ; IMMEDIATE
    EXIT




( THIS 6510 FORTH ASSEMBLER WAS WRITTEN BY WILLIAM F. RAGSDALE )
( IT WAS PUBLISHED IN "FORTH DIMENSIONS", VOL. III # 5         )
```

SCREEN #121
```
( ASSEMBLER: A-REMOVE )
  DP ! ( RESET DP TO PRE-ASSEMBLER AREA )
: A-REMOVE ( CAUSES REMOVAL OF ASSEMBLER FROM DICTIONARY )
   LITERAL [ 0 LATEST PFA LFA ] LITERAL ! ; DROP
```

SCREEN #122

SCREEN #123
```
( SPRITES DEMO )
: INIT-SID
  SOUND.INIT
   V3 65535 V-FREQ ( SET RANDOM )
   NOISE V-CTRL
   0 0 MODEVOL ;

: INIT-SPRITES
   8 0 DO
    I SPRITE !  ON S-ENABLE
    I 2* S-COLOR
    49152 DRAGON1  49216 DRAGON2
    I 4 / S-POINTER
   LOOP ;


```

SCREEN #124
```
( SPRITES DEMO )
: MOVE-SPRITES
    8 0 DO
     I SPRITE !
     OSC3@ OSC3@ S-POSITION
    LOOP ;

: SPRITE-OFF
   8 0 DO
    I SPRITE !
    OFF S-ENABLE
   LOOP ;




```

SCREEN #125
```
( GRAPHICS DEMOS )
: LINES   ( COLOR1 COLOR2 --- )
  DRAW FS
  B-COLOR-FILL
  320 0 DO
   0 0 B-PLOT
    I 199 B-LINE
  4 +LOOP
  0 199 DO
   0 0 B-PLOT
    319 I B-LINE
  -2 +LOOP
  180 WAIT ;



```

```
SCREEN #126
 0) ( GRAPHICS DEMOS )
 1) : SDEMO
 2)     INIT-SID  DRAW FS INIT-SPRITES
 3)     240 20 DO
 4)      I 100 U* SQRT 20 -
 5)      I 1 RSHIFT 80 + SWAP
 6)      I 3 RSHIFT DUP 15 10 */ ELLIPSE
 7)      MOVE-SPRITES
 8)     6 +LOOP
 9)     180 WAIT SPRITE-OFF ;
10)
11)
12)
13)
14)
15)

SCREEN #127
 0) ( JESU )
 1) : O+ 12 ' O@ +! ;  : O- -12 ' O@ +! ;
 2) : J1  SOUND.INIT O2 120 TEMPO !
 3)     V1 1/8 TRIPLET  0 9 V-AD  SONG.INIT
 4)     G O+ G A
 5)     B O+ D C
 6)     C E D
 7)     D G F#
 8)     G D O- B
 9)     G A B
10)     O+ C D E
11)     D C O- B
12)     A B G
13)     F# G A
14)     D F# A
15)     O+ C O- B A ;

SCREEN #128
 0) ( JESU )
 1) : JESU
 2)     J1
 3)   B G A
 4)   B O+ D C
 5)   C E D
 6)   D G F#
 7)   G D O- B
 8)   G A B
 9)   E O+ D C O-
10)   B A G
11)   D G F#
12)   WHOLE G O- ;
13)
14)
15)
```

```
SCREEN #129
 0) ( TURTLE DEMOS )
 1) : SQUARE
 2)    4 0 DO
 3)     DUP FD 90 RT LOOP DROP ;
 4)
 5) : FAN
 6)    RED CYAN CATNIB 133 C!
 7)    18 0 DO 20 SQUARE 20 RT LOOP
 8)    18 0 DO 40 SQUARE 20 RT LOOP
 9)    18 0 DO 60 SQUARE 20 RT LOOP
10)    WHITE PC
11)    168 90 100 CIRCLE ;
12)
13)
14)
15)

SCREEN #130
 0) ( DEMO EXECUTIVE )
 1) : DEMO
 2)    BLACK ORANGE LINES
 3)    JESU                180 WAIT
 4)    8 2 C-CURVE         180 WAIT
 5)    SDEMO               DRAW
 6)    ."   SUPER FORTH 64 TURTLE GRAPHICS DEMO "
 7)    FAN                 180 WAIT TS
 8)    9 2 D-CURVE         180 WAIT
 9)    SPRITE-OFF
10)    TS 2067 GO ;
11)
12)
13)
14)
15)

SCREEN #131
 0)
 1)
 2)
 3)
 4)
 5)
 6)
 7)
 8)
 9)
10)
11)
12)
13)
14)
15)
```

III. SUPER-FORTH Dictionary List

The following list contains all of the words defined in the SUPER-FORTH 64 system. As with VLIST, the order is from highest level to lowest level definitions. The list is broken up into the three vocabularies, FORTH, EDITOR and ASSEMBLER.

III-1 SUPER FORTH 64 Main Vocabulary Word Set

TEXTSCREEN	SPLITSCREEN	SETHEADING	RIGHT	PENUP
PENDOWN	PENCOLOR	LEFT	FULLSCREEN	FORWARD
CLEARSCREEN	BACKGROUND	BACK	BK	FD
DRAW	FS	SS	TS	BG
PC	HOME	CS	LT	RT
SETY	SETX	PM	PD	PU
PENFLG	SETXY	SETH	HEADING	S-EDITOR
TIE	R	B	B\	A#
A	A\	G#	G	G\
F#	F	E	E\	D#
D	D\	C#	C	PLAY.WAIT
SONG.INIT	NCALC	READY	NEXT.NOTE	NDEF
TRIPLET	1/64	1/32	1/16	.1/16
1/8	.1/8	1/4	.1/4	1/2
.1/2	WHOLE	DURATION	TEMPO	O7
O6	O5	O4	O3	O2
O1	O0	O!	O@	T!
T@	SOUND.INIT	V-DEFAULT	PLAY.NOTE	WAVE
HC!	H!	HC@	H@	PATCH
DIR	ENDCASE	;;	OF	CASE
WAIT	SETTIM	RDTIM	FRE	RS232
PUT#	GET#	PRINTER	$INPUT	PRINT#
INPUT#	INPUT	CMDI	CMD	ARC
CIRCLE	ELLIPSE	B-CLINE	B-LINE	LSETUP
<B-LINE>	B-CPLOT	CHAR	M-PLOT	R-PLOT
M-ORIGIN	BKGND	BORDER	L.DIST	S.DIST
B-MFLAG	M-Y	M-X	2ARRAY	1ARRAY
S-DEF	COLOR-MEM	UPORT	JOY2	JOY1
ON	OFF	V3	V2	V1
3OFF	NOTE@	NOTE-VALUES	NOTCH	HIGHPASS
BANDPASS	LOWPASS	FILTEX	FILT3	FILT2
FILT1	SYNC	RESET	RING	NOISE
PULSE	SAWTOOTH	TRIANGLE	S8	S7
S6	S5	S4	S3	S2
S1	LT.GRAY	LT.BLUE	LT.GREEN	MED.GRAY
DK.GRAY	LT.RED	BROWN	ORANGE	YELLOW
BLUE	GREEN	PURPLE	CYAN	RED
WHITE	BLACK	FMAX	F>	FMIN

FNEGATE	FABS	FEXIT	FINIT	<FNUM>
FSQRT	FCOS	FSIN	FLOAT	DFLOAT
FIX	DFIX	LSCALE	RSCALE	F-
F+	ALIGN	F/	F*	F.
E.	E!	E@	SFN	SFZ
FOV	FNE	FZE	FER	FRESET
FBASE	FPSW	SQRT	1'S-BIT	2'S-BIT
EASY-BITS	D2*	COS	SIN	QCOS
QSIN	S180	<COS>	<SIN>	SINTABLE
TSCALE	DECOMPILE	<DECOM>	GCHK	GIN+
GIN	LIST	EDITOR	PP	LINE
CHKLIN	;CODE	CODE	ENTERCODE	ASSEMBLER
[']	U.R	H	FLUSH	ERASE
EMPTY	>BINARY	2VARIABLE	2SWAP	2OVER
2DUP	2DROP	2CONSTANT	2@	2!
'S	VLIST	VTAB	VLEN	ID.
<EMIT7>	TRIAD	TITLE	'TITLE	DUMP
-->	?LOADING	INDEX	PAUSE	DVARIABLE
DMIN	DMAX	DOVER	DCONSTANT	DU<
D@	D>	DSWAP	MAX-BUFFS	D=
D0=	D-	SCOPY	COPY	BMOVE
<ROT	.INDEX	.S	.SR	.SL
.SS	MODEVOL	RESFILT	V-CTRL	V-SR
V-AD	V-PW	V-FREQ	S-COLOR	S-POINTER
S-MULTI	S-PRIORITY	S-YEXP	S-XEXP	S-ENABLE
S-FSET	B-COLOR-FILL	B-COLOR	B-FILL	B-GRAPHICS
S-B-COLLISION		S-S-COLLISION		MULTI-COLOR
S-MULTIR	F-NUMBER	F-LOAD	F-SAVE	F-EDIT
F-APPEND	F-NEW	WRITEB	READB	F-EXIT
FILE-MODE	FR/W	F-OPEN	FNAME	$FIND
$=	$>	$<	$CMP	""
"	<">	$CLR	$.	$LEN
$VAL	$RIGHT	$MID	$LEFT	$CONCAT
<$CONCAT>	$CONSTANT	$VARIABLE	ST	DOS
LOADRAM	CLRCHN	CLALL	CLOSE	OPEN
SETNAM	SETLFS	-TEXT	RECURSE	SYSCALL
THRU	I-CLEAR	I-SET	I-SYSTEM	I-USER
I-INIT	D-CLEAR	D-READ	D-POSITION	D-SPLIT
ENV3@	OSC3@	PADDLE@	F-FREQ	V!
SID!	VOICE	SID	S-POSITION	SPRITE
B-PLOT	B-DRAW	B-ERASE	B-PEN	B-Y
B-X	CHARBASE	BITMAP	SCREEN	BANK
'CHARBASE	'BITMAP	'SCREEN	'BANK	SWAPIN
SWAPOUT	FBIT	CBIT	SBIT	MASK
APPLICATION	SAVE-FORTH	SAVENAME	SYS	SAVE
SPLIT	CATNIB	8	4	3
]	[COMPILE]	[	XOR	
WORD	WIDTH	WHERE	WHILE	WARNING
VOCABULARY	VOC-LINK	VARIABLE	USER	USE
UPDATE	UP	UNTIL	U<	U/MOD
U.	U*	TYPE	TRAVERSE	TOGGLE
T&SCALC	TIB	THEN	TEXT	SWAP
SYSDEV	STATE	SPACES	SPACE	SP@

SUPER-FORTH Dictionary List

SP0	SP!	SMUDGE	SIGN	SECTRKT
SET-DRX	SEC/BLK	SCR	SAVE-BUFFERS	S0
S->D	RWTS	RP!	RP@	ROT
ROLL	REPEAT	R@	R>	R0
R/W	RSHIFT	R#	QUIT	QUERY
PREV	PICK	PFA	PAGE	PAD
OVER	OUTLFN	OUT	OR	OFFSET
NUMBER	NOT	NFA	NEGATE	MOVE
MODE	MOD	MIN	MAX-DRV	MAX-BLKS
MAX	M/MOD	M/	M+	M*/
M*	LSHIFT	LOOP	LOAD	LITERAL
<LIT>	LIST	LIMIT	LFA	LEAVE
LATEST	KEY	J	INTERPRET	INIT-USER
INIT-FORTH	INPLFN	IMMEDIATE	IF	I'
I	HOLD	HLD	HEX	HERE
GO	FREEZE	FORTH	FORGET	FLD
FLAST	FIRST	FIND	FILL	FENCE
EXPECT	EXIT	EXECUTE	ENCLOSE	
EMPTY-BUFFERS		EMIT	ELSE	DUP
DUAL	DROP	DR4	DR3	DR2
DR1	DR0	DPL	DP	DOSERR
DOES>	DO	DNEGATE	DLITERAL	DIGIT
DEPTH	DEFINITIONS	DECIMAL	DDUP	DDROP
DABS	D<	D.R	D.	D+-
D+	D!	CURRENT	CSP	CREATE
CR	COUNT	CONVERT	CONTEXT	CONSTANT
CONFIGURE	COMPILE	COLD	CMOVE	CLEAR
CHARIN	CHANGE	CFA	C@	C/L
C,	C!	BYE	BUFFER	BRANCH
BPDRV	BLOCK	BLK	BLANK	BL
BEGIN	BASE	AND	ALLOT	AGAIN
ABS	ABORT"	ABORT	@	?TERMINAL
?STREAM	?STACK	?PAIRS	?DUP	?DEPTH
?CSP	?CONFIGURE	?COMP	?	>R
>IN	>	=	<VOCABULARYFIG>	
<VOCABULARY79>		<WORD>	<T&SCALC>	<R/W>
<PAGE>	<NUMBER>	<LOOP>	<LOAD>	<LINE>
<KEY>	<INTERPRET>	<FIND>	<FILL>	<EXPECT>
<EMIT>	<DO>	<CR>	<CMOVE>	<CMOVE
<BLOCK>	<ABORT">	<ABORT>	<?TERMINAL>	<<CMOVE>
<;CODE>	</LOOP>	<.">	<-FIND>	<+LOOP>
<#	<	;	:	79-STANDARD
2/	2-	2+	2*	2
1-	1+	1	0BRANCH	0>
0=	0<	0	/MOD	/LOOP
/	.R	.LINE	."	.
-TRAILING	-FIND	-	,	+LOOP
+BUF	+-	+!	+	*/MOD
*/	*	(	'WORD	'VOCABULARY
'T&SCALC	'STREAM	'R/W	'PAGE	'NUMBER
'LOAD	'KEY	'INTERPRET	'EXPECT	'EMIT
'CR	'BLOCK	'ABORT	'?TERMINAL	'-FIND
'	#S	#BUFF	#>	#

!

III-2 Editor Vocabulary Word Set

SM	SC	P	N	W
L	F	M	O	X
LL	K	15)	14)	13)
12)	11)	10)	9)	8)
7)	6)	5)	4)	3)
2)	1)	0)	SE	C

III-3 Assembler Vocabulary Word Set

END-CODE	REPEAT,	WHILE,	AGAIN,	VS
>=	0<	0=	CS	NOT
ELSE,	THEN,	IF,	UNTIL,	BEGIN,
BIT,	JMP,	JSR,	STY,	LDY,
LDX,	CPY,	CPX,	STX,	ROR,
ROL,	LSR,	INC,	DEC,	ASL,
STA,	SBC,	ORA,	LDA,	EOR,
CMP,	AND,	ADC,	M/CPU	TYA,
TXS,	TXA,	TSX,	TAY,	TAX,
SEI,	SED,	SEC,	RTS,	RTI,
PLP,	PLA,	PHP,	PHA,	NOP,
INY,	INX,	DEY,	DEX,	CLV,
CLI,	CLD,	CLC,	BRK,	CPU
UPMODE	RP>	SEC	BOT	)
),Y	,X)	,Y	,X	MEM
#	.A	MODE	INDEX	SETUPN
NEXT	PUSH	PUT	POPTWO	POP
N	IP	W	XSAVE	

IV. Various Articles of Implemented Screens

A FORTH ASSEMBLER FOR THE 6502

by William F. Ragsdale

INTRODUCTION

This article should further polarize the attitudes of those outside the growing community of FORTH users. Some will be fascinated by a label-less, macro-assembler whose source code is only 96 lines long! Others will be repelled by reverse Polish syntax and the absence of labels.

The author immodestly claims that this is the best FORTH assembler ever distributed. It is the only such assembler that detects all errors in op-code generation and conditional structuring. It is released to the public domain as a defense mechanism. Three good 6502 assemblers were submitted to the FORTH Interest Group but each had some lack. Rather than merge and edit for publication, I chose to publish mine with all the submitted features plus several more.

Imagine having an assembler in 1300 bytes of object code with:

1. User macros (like IF, UNTIL,) definable at any time.
2. Literal values expressed in any numeric base, alterable at any time.
3. Expressions using any resident computation capability.
4. Nested control structures without labels, with error control.
5. Assembler source itself in a portable high level language.

OVERVIEW

Forth is provided with a machine language assembler to create execution procedures that would be time inefficient, if written as colon-definitions. It is intended that "code" be written similarly to high level, for clarity of expression. Functions may be written first in high-level, tested, and then re-coded into assembly, with a minimum of restructuring.

THE ASSEMBLY PROCESS

Code assembly just consists of interpreting with the ASSEMBLER vocabulary as CONTEXT. Thus, each word in the input stream will be matched according the Forth practice of searching CONTEXT first then CURRENT.

```
ASSEMBLER (now CONTEXT)
FORTH     (chained to ASSEMBLER)
user's    (CURRENT if one exits)
FORTH     (chained to user's vocab)
try for literal number
else, do error abort
```

The above sequence is the usual action of Forth's text interpreter, which remains in control during assembly.

During assembly of CODE definitions, Forth continues interpretation of each word encountered in the input stream (not in the compile mode). These assembler words specify operands, address modes, and op-codes. At the conclusion of the CODE definition a final error check verifies correct completion by "unsmudging" the definition's name, to make it available for dictionary searches.

RUN-TIME, ASSEMBLY-TIME

One must be careful to understand at what time a particular word definition executes. During assembly, each assembler word interpreted executes. Its function at that instant is called 'assembling' or 'assembly-time'. This function may involve op-code generation, address calculation, mode selection, etc.

The later execution of the generated code is called 'run-time'. This distinction is particulary important with the conditionals. At assembly time each such word (i.e., IF, UNTIL, BEGIN, etc.) itself 'runs' to produce machine code which will later execute at what is labeled 'run-time' when its named code definition is used.

AN EXAMPLE

As a practical example, here's a simple call to the system monitor, via the NMI address vector (using the BRK opcode).

```
CODE MON  ( exit to monitor  )
     BRK, NEXT JMP, END-CODE
```

The word CODE is first encountered, and executed by Forth. CODE builds the following name "MON" into a dictionary header and calls ASSEMBLER as the CONTEXT vocabularly.

The "(" is next found in FORTH and executed to skip til ")". This method skips over comments. Note that the name after CODE and the ")" after "(" must be on the same text line.

OP-CODES

BRK, is next found in the assembler as the op-code. When BRK, executes, it assembles the byte value 00 into the dictionary as the op-code for "break to monitor via "NMI".

Many assembler words names end in ",". The significance of this is:

1. The comma shows the conclusion of a logical grouping that would be one line of classical assembly source code.
2. "," compiles into the dictionary; thus a comma implies the point at which code is generated.
3. The "," distinguishes op-codes from possible hex numbers ADC and ADD.

NEXT

Forth executes your word definitions under control of the address interpreter, named NEXT. This short code routine moves execution from one definition, to the next. At the end of your code definition, you must return control to NEXT or else to code which returns to NEXT.

RETURN OF CONTROL

Most 6502 systems can resume execution after a break, since the monitor saves the CPU register contents. Therefore, we must return control to Forth after a return from the monitor. NEXT is a constant that specifies the machine address of Forth's address interpreter (say $0242). Here it is the operand for JMP,. As JMP, executes, it assembles a machine code jump to the address of NEXT from the assembly time stack value.

SECURITY

Numerous tests are made within the assembler for user errors:

1. All parameters used in CODE definitions must be removed.
2. Conditionals must be properly nested and paired.
3. Address modes and operands must be allowable for the op-codes

These tests are accomplished by checking the stack position (in CSP) at the creation of the definition name and comparing it with the position at END-CODE. Legality of address modes and operands is insured by means of a bit mask associated with each operand.

Remember that if an error occurs during assembly, END-CODE never executes. The result is that the "smudged" condition of the definition name remains in the "smudged" condition and will not be found during dictionary searches.

The user should be aware that one error not trapped is referencing a definition in the wrong vocabulary:

i.e., 0= of ASSEMBLER when you want 0= of FORTH

(Editor's note: the listing assumes that the figFORTH error messages are already available in the system, as follows:

?CSP issues the error message "DEFINITION NOT FINISHED" if the stack position differs from the value saved in the user variable CSP, which is set at the creation of teh definition name.

?PAIRS issues the error message "CONDITIONALS NOT IMPAIRED" if its two arguments do not match.

3 ERROR prints the error message "HAS INCORRECT ADDRESS MODE".)

SUMMARY

The object code of our example is:

```
305   983 4D 4F CE    CODE MON
305D 4D 30            link field
305F 61 30            code field
3061  00              BRK
3062  4C 42 02        JMP NEXT
```

OP-CODES, revisited

The bulk of the assembler consists of dictionary entries for each op-code. The 6502 one mode op-codes are:

```
BRK,  CLC,  CLD,  CLI,  CLV,
DEX,  DEY,  INX,  INY,  NOP,
PHA,  PHP,  PLA,  PLP,  RTI,
RTS,  SEC,  SED,  SEI,  TAX,
TAY,  TSX,  TXS,  TXA,  TYA,
```

When any of these are executed, the corresponding op-code byte is assembled into the dictionary.

The multi-mode op-codes are:

```
ADC,  AND,  CMP,  EOR,  LDA,
ORA,  SBC,  STA,  ASL,  DEC,
INC,  LSR,  ROL,  ROR,  STX,
CPX,  CPY,  LDX,  LDY,  STY,
JSR,  JMP,  BIT,
```

These usually take an operand, which must already be on the stack. An address mode may also be specified. If none is given, the op-code uses z-page or absolute addressing. The address modes are determined by:

Symbol	Mode	Operand
.A	accumulator	none
#	immediate	8 bits only
,X	indexed X	z-page or absolute
,Y	indexed Y	z-page or absolute
,X)	indexed indirect X	z-page only
)Y	indirect indexed Y	z-page only
)	indirect	absolute only
none	memory	z-page or absolute

EXAMPLES

Here are examples of Forth vs. conventional assembler. Note that the operand comes first, followed by any mode modifier, and then the op-code mnemonic. This makes best use of the stack at assembly time. Also, each assembler word is set off by blanks, as is required for all Forth source text.

```
         .A ROL,    ROL A
         1 # LDY,   LDY #1
    DATA ,X STA,    STA DATA,X
    DATA ,Y CMP,    CMP DATA,Y
         6 ,X) ADC, ADC (06,X)
     POINT )Y STA,  STA (POINT),Y
      VECTOR ) JMP, JMP (VECTOR)
```

(.A distinguishes from hex number 0A)

The words DATA and VECTOR specify machine addresses. In the case of "6)X ADC," the operand memory address $0006 was given directly. This is occasionally done if the usage of a value doesn't justify devoting the dictionary space to a symbolic value.

6502 CONVENTIONS

Stack Addressing

The data stack is located in z-page, usually addressed by "Z-PAGE,X". The stack starts near $009E and grows downward. The X index register is the data stack pointer. Thus, incrementing X by two removes a data stack value; decrementing X twice makes room for one new data stack value.

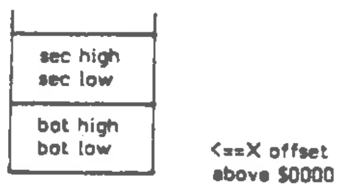
Sixteen bit values are placed on the stack according to the 6502 convention; the low byte is at low memory, with the high byte following. This allows "indexed, indirect X" directly off a stack value.

The bottom and second stack values are referenced often enough that the support words BOT and SEC are included. Using

BOT LDA, assembles LDA 0,X and
SEC ADC, assembles ADC 2,X

BOT leaves 0 on the stack and sets the address mode to ,X. SEC leaves 2 on the stack also setting the address mode to ,X.

Here is a pictorial representation of the stack in z-page.



Here is an examples of code to "or" to the accumulator four bytes on the stack:

```
BOT   LDA,       LDA 0,X
BOT 1+ ORA,      ORA 1,X
SEC   ORA,       ORA 2,X
SEC 1+ ORA,      ORA 3,X
```

To obtain the 14-th byte on the stack:
BOT 13 + LDA,

RETURN STACK

The Forth Return Stack is located in the 6502 machine stack in Page 1. It starts at $01FE and builds downward. No lower bound is set or checked as Page 1 has sufficient capacity for all (non--recursive) applications.

By 6502 convention the CPU's register points to the next free byte below the bottom of the Return Stack. The byte order follows the convention of low significance byte at the lower address.

Return stack values may be obtained by: PLA, PLA, which will pull the low byte, then the high byte from the return stack. To operate on aribitrary bytes, the method is:

1) save X in XSAVE

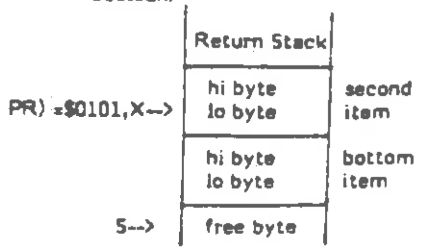
2) execute TSX, to bring the S register to X.

3) use RP) to address the lowest byte of the return stack. Offset the value to address higher bytes. (Address mode is automatically set to ,X.)

4) Restore X from XSAVE.

As an example, this definition non-destructively tests that the second item on the return stack (also the machine stack) is zero.

```
 CODE IS-IT  ( zero ? )
     XSAVE STX,  TSX,   (setup for
     return stack)
     RP) 2+ LDA,  RP)  3  +  ORA,
     (  or 2nd item's two bytes
     together)
0= IF,  INY,  THEN,  (  if zero, bump
Y to one)
     TYA,  PHA, XSAVE  LDX,  (save
     low byte, retore data stack)
     PUSH JMP,  END-CODE  (  push
     boolean)
```



FORTH REGISTERS

Several Forth registers are available only at the assembly level and have been given names that return their memory addresses. These are:

IP address of the Interpretive Pointer, specifying the next Forth address which will be interpreted by NEXT.

W address of the pointer to the code field of the dictionary definition just interpreted by NEXT. W-1 contains $6C, the op-code for indirect jump. Therefore, jumping to W-1 will indirectly jump via W to the machine code for the definition.

UP User Pointer containing address of the base of the user area.

N a utility area in z-page from N-1 thru N+7.

CPU Registers

When Forth execution leaves NEXT to execute a CODE definition, the following conventions apply:

1. The Y index register is zero. It may be freely used.

2. The X index register defines the low byte of the bottom data stack item relative to machine address $0000.

3. The CPU stack pointer S points one byte below the low byte of the bottom return stack item. Executing PLA, will pull this byte to the accumulator.

4. The accumulator may be freely used.

5. The processor is in the binary mode and must be returned in that mode.

XSAVE

XSAVE is a byte buffer in z-page, for temporary storage of the X register. Typical usage, with a call which will change X, is:

```
        CODE DEMO
            XSAVE STX,  USER'S JSR,
            ( which will change X )
            XSAVE LDX,  NEXT JMP,
            END-CODE
```

N Area

When absolute memory registers are required, use the 'N Area' in the base page. These registers may be used as pointers for indexed/indirect addressing or for temporary values. As an example of use, see CMOVE in the system source code.

The assembler word N returns the base address ~~(usually $00D1)~~. The N Area spans 9 bytes, from N-1 thru N+7. Conventionally, N-1 holds one byte and N, N+2, N+4, N+6 are pairs which may hold 16-bit values. See SETUP for help on moving values to the N Area.

It is very important to note that many Forth procedures use N. Thus, N may only be used within a single code definition. Never expect that a value will remain there, outside a single definition!

```
CODE DEMO   HEX
   6 # LDA,  N 1 - STA,
      (setup a counter)

BEGIN,  8001 BIT,
      (tickle a port)

        N 1 - DEC,
          (decrement the counter)

   0= UNTIL,  NEXT JMP, END-CODE
      (loop till negative)
```

SETUP

Often we wish to move stack values to the N area. The sub-routine SETUP has been provided for this purpose. Upon entering SETUP the accumulator specifies the quantity of 16-bit stack values to be moved to the N area. That is, A may be 1, 2, 3, or 4 only:

```
         3 # LDA,   SETUP JSR,

    stack before    N after     stack after
          H  high                    H
          G_ low             bot--> G_
          F             F
          E_            E_
          D             D
sec-->    C_            C_
          B             B
bot-->    A_       N--> A_
```

CONTROL FLOW

Forth discards the usual convention of assembler labels. Instead, two replacements are used. First, each Forth definition name is permanently included in the dictionary. This allows procedures to be located and executed by name at any time as well as be compiled within other definitions.

Secondly, within a code definition, execution flow is controlled by label-less branching according to "structured programming". This method is identical to the form used in colon-definitions. Branch calculations are done at assembly time by temporary stack values placed by the control words:

```
        BEGIN,   UNTIL,    IF,   ELSE,
        THEN,
```

Here again, the assembler words end with a comma, to indicate that code is being produced and to clearly differentiate from the high-level form.

One major difference occurs! High-level flow is controlled by run-time boolean values on the data stack. Assembly flow is instead controlled by processor status bits. The programmer must indicate which status bit to test, just before a conditional branching word (IF, and UNTIL,).

Examples are:

```
    PORT LDA, 0=  IF, <a>  THEN,
        (read port, if equal to zero do <a> )

    PORT LDA,  0=  NOT IF,  <a>  THEN,
        (read port, if not equal to zero
        do <a> )
```

The conditional specifiers for 6502 are:

CS	test carry set processor status	C=1 in
0<	byte less than zero	N=1
0=	equal to zero	Z=1
CS NOT	test carry clear	C=0
0 <NOT	test positive	N=0
0= NOT	test not equal zero	Z=0

The overflow status bit is so rarely used, that it is not included. If it is desired, compile:

```
    ASSEMBLER  DEFINITIONS   HEX
    50 CONSTANT   VS      (test overflow
    set)
```

CONDITIONAL LOOPING

A conditional loop is formed at assembler level by placing the portion to be repeated between BEGIN, and UNTIL,:

```
     6 # LDA,    N STA,
    (define loop counter in N)
BEGIN,   PORT DEC,
    (repeated action)
      N DEC,        0= UNTIL,
    (N reaches zero)
```

First, the byte at address N is loaded with the value 6. The beginning of the loop is marked (at assembly time) by BEGIN,. Memory at PORT is decremented, then the loop counter in N is decremented. Of course, the CPU updates its status register as N is decremented. Finally, a test for Z=1 is made; if N hasn't reached zero, execution returns to BEGIN,. When N reaches zero (after executing PORT DEC, 6 times) execution continues ahead after UNTIL,. Note that

BEGIN, generates no machine code, but is only an assembly time locator.

CONDITIONAL EXECUTION

Paths of execution may be chosen at assembly in a similar fashion and done in colon-definitions. In this case, the branch is chosen based on a processor status condition code.

```
PORT LDA,    0=  IF,    (for zero set)
THEN, (continuing code)
```

In this example, the accumulator is loaded from PORT. The zero status is tested if set (Z=1). If so, the code (for zero set) is executed. Whether the zero status is set or not, execution will resume at THEN,.

The conditional branching also allows a specific action for the false case. Here we see the addition of the ELSE, part.

```
PORT LDA, 0= IF,   < for  zero  set>
         ELSE, <for zero clear>
         THEN, <continuing code>
```

The test of PORT will select one of two execution paths, before resuming execution after THEN,. The next example increments N based on bit D7 of a port:

```
PORT LDA,            ( fetch one byte )
 0< IF,  N  DEC,     ( if D7=1, decrement
                     N )
   ELSE,   N INC,    ( if D7=0, increment
                     N )
   THEN,             ( continue ahead  )
```

CONDITIONAL NESTING

Conditionals may be nested, according to the conventions of structured programming. That is, each conditional sequence begun (IF, BEGIN,) must be terminated (THEN, UNTIL,) before the next earlier conditional is terminated. An ELSE, must pair with the immediately preceding IF,.

```
BEGIN, < code always executed>
   CS IF, <code if carry set>
        ELSE,    <code  if  carry  clear>
        THEN,
   0= NOT UNTIL,   (  loop till condition
                      flag is non-zero)
        <code that continues onward>
```

Next is an error that the assembler security will reveal.

```
BEGIN,  PORT  LDA,
   0= IF,  BOT INC,
        0= UNTIL,  THEN,
```

The UNTIL, will not complete the pending BEGIN, since the immediately preceding IF, is not completed. An error trap will occur at UNTIL, saying "conditionals not paired".

RETURN OF CONTROL, revisited

When concluding a code definition, several common stack manipulations often are needed. These functions are already in the nucleus, so we may share their use just by knowing their return points. Each of these returns control to NEXT.

POP	remove one 16-bit stack values.
POPTWO	remove two 16-bit stack values.
PUSH	add two bytes to the data stack.
PUT	write two bytes to the data stack, over the present bottom of the stack.

Our next example complements a byte in memory. The bytes' address is on the stack when INVERT is executed.

```
CODE INVERT  ( a memory byte )  HEX
  BOT X) LDA,     (fetch byte addressed
                   by stack)
    FF  #  EOR,   (complement accumu-
                   lator)
  BOT  X) STA,  ( replace in memory )
  POP JMP,  END-CODE  ( discard
                   pointer  from stack,
                   return to NEXT )
```

A new stack value may result from a code definition. We could program placing it on the stack by:

```
CODE ONE  ( put 1 on the stack )
      DEX,  DEX,  ( make room on the
                    data stack)
      1 # LDA, BOT STA,  (store low byte)
      BOT 1+ STY, ( hi byte stored from Y
                    since = zero)
      NEXT JMP,  END-CODE
```

A simpler version could use PUSH:

```
CODE ONE
  1 # LDA,  PHA,  ( push low byte to
  machine stack )
  TYA,    PUSH  JMP,  ( high byte to
  accumulator, push to data stack )
  END-CODE
```

The convention for PUSH and PUT is:

1. push the low byte onto the machine stack.
2. leave the high byte in the accumulator.
3. jump to PUSH or PUT.

PUSH will place the two bytes as the new bottom of the data stack. PUT will over-write the present bottom of the stack with the two bytes. Failure to push exactly one byte on the machine stack will disrupt execution upon usage!

FOOLING SECURITY

Occasionally we wish to generate unstructured code. To accomplish this, we can control the assembly time security checks, to our purpose. First, we must note the parameters utilized by the control structures at assembly time. The notation below is taken from the assembler glossary. The --- indicates assembly time execution, and separate input stack values from the output stack values of the words execution.

```
BEGIN, ==>                    --- addrB  1
UNTIL, ==>   addrB  1   cc    ---

IF,     ==>             cc    --- addrI  2
ELSE,  ==>   addrI  2         --- addrE  2
THEN,  ==>   addrI  2         ---
        or   addrE  2         ---
```

The address values indicate the machine location of the corresponding 'B'EGIN, 'I'F, or 'E'LSE,. cc represents the condition code to select the processor status bit referenced. The digit 1 or 2 is tested for conditional pairing.

The general method of security control is to drop off the check digit and manipulate the addresses at assembly time. The security against errors is less, but the programmer is usually paying intense attention to detail during this effort.

To generate the equivalent of the high level:

```
        BEGIN <a> WHILE <b>  REPEAT
```

we write in assembly:

```
        BEGIN,  DROP ( the check digit
                      1, leaving addrB)
                 <a>
         CS IF,  ( leaves addrI and digit
                 2)
                     <b>
           ROT ( bring addrB to bottom)
           JMP, (to addrB  of BEGIN, )
        THEN,  ( complete false for-
                 ward branch from IF, )
```

It is essential to write the assembly time stack on paper, and run through the assembly steps, to be sure that the check digits are dropped and re-inserted at the correct points and addresses are correctly available.

ASSEMBLER GLOSSARY

#	Specify 'immediate' addressing mode for the next op-code generated.
)Y	Specify 'indirect indexed Y' addressing mode for the next op-code generated.

,X Specify "indexed X" addressing mode for the next op-code generated.

,Y Specify "indexed Y" addressing mode for the next op-code generated.

.A Specify accumulator addressing mode for the next op-code generated.

0< — cc (assembling)
Specify that the immediately following conditional will branch based on the processor status bit being negative ([illegible] i.e., less than zero. The flag cc is left at assembly time; there is no run-time effect on the stack.

0= — cc (assembling)
Specify that the immediately following conditional will branch based on the processor status bit being equal to zero (Z=1). The flag cc is left at assembly time; there is no run-time effect on the stack.

;CODE Used to conclude a colon-definition in the form:
: <name> . . . ;CODE <assembly code> END-CODE
Stop compilation and terminate a new defining word <name> . Set the CONTEXT vocabulary to ASSEMBLER, assembling to machine code the following nmenonics. An existing defining word must exist in name prior to ;CODE.

When <name> later executes in the form:
<name> <namex>
the definition <namex> will be created with its execution procedure given by the machine code following <name> . That is, when <namex> is executed, the address interpreter jumps to the code following ;CODE in <name> .

ASSEMBLER in FORTH
Make ASSEMBLER the CONTEXT vocabulary. It will be searched first when the input stream in interpreted.

BEGIN, — addr 1 (assembling)
— (run-time)
Occurs in a CODE definition in the form:
BEGIN, . . . cc UNTIL,
At run-time, BEGIN, marks the start of an assembly sequence repeatedly executed. It serves as the return point for the corresponding UNTIL,. When reaching UNTIL, a branch to BEGIN, will occur if the processor status bit given by cc is false; otherwise execution continues ahead.

At assembly time, BEGIN, leaves the dictionary pointer address addr and the value 1 for later testing of conditionary pairing by UNTIL,.

BOT — n (assembling)
Used during code assembly in the form:

BOT LDA, or BOT 1+ X) STA,

Addresses the bottom of the data stack (containing the low byte) by selecting the ,X mode and leaving n=0, at assembly time. This value of n may be modified to another byte offset into the data stack. Must be followed by a multi-mode op-code mnemonic.

CODE A defining word used in the form:

CODE <name> . . . . END-CODE

to create a dictionary entry for <name> in the CURRENT vocabulary. Name's code field contains the address of its parameter field. When <name> is later executed, the machine code in this parameter field will execute. The CONTEXT vocabulary is made ASSEMBLER, to make available the op-code mnemonics.

CPU n — (compiling assembler)
An assembler defining word used to crete assembler mnemonics that have only one addressing mode:

EA CPU NOP,

CPU creates the work NOP, with its op-code EA as a parameter. When NOP, later executes, it assembles EA as a one byte op-code.

CS — cc (assembling)
Specify that the immediately following conditional will branch based on the processor carry is set (C=1). The flag cc is left at assembly time; there is no run-time effect on the stack.

ELSE, — (run-time)
addr1 2 — addr2 2
(assembling)
Occurs within a code definition in the form:
cc IF, <true part> ELSE,
<false part> THEN,
At run-time, if the condition code specified by cc is false, execution will skip to the machine code following ELSE,. At assembly time ELSE, assembles a forward jump to just after THEN, and resolves a pending forward branch from IF. The values 2 are used for error checking of conditional pairing.

END-CODE
An error check word marking the end of a CODE definition. Successful execution to and including END-CODE will unsmudge the most recent CURRENT vocabulary definition, making it available for execution. END-CODE also exits the ASSEMBLER making CONTEXT the same as CURRENT. This word previously was named C;

IF, cc --- addr 2 (assembly time)
--- addr 2 (assembly-time)

Occurs within a code definition in the form:
cc IF, <true part> ELSE,
false part THEN,
At run time, IF, branches based on the condition code cc, (0< or 0= or CS). If the specified processor status is true, execution continues ahead, otherwise branching occurs to just after ELSE, (or THEN, when ELSE, is not present). At ELSE, execution resumes at the corresponding THEN,.

When assembling, IF, creates an unresolved forward branch based on the condition code cc, and leaves addr and 2 for resolution of the branch by the corresponding ELSE, or THEN,. Conditionals may be nested.

INDEX --- addr (assembling)
An array used within the assembler, which holds bit patterns of allowable addressing modes.

IP --- addr (assembling)
Used in a code definition in the form:

IP STA, or IP)Y LDA,

A constant which leaves at assembly time the address of the pointer to the next FORTH execution address in a colon-definition to be interpreted.

At run-time, NEXT moves IP ahead within a colon-definition. Therefore, IP points just after the execution address being interpreted. If an in-line data structure has been compiled (i.e., a character string), indexing ahead by IP can access this data:

IP STA, or IP)Y LDA,

loads the third byte ahead in the colon-definition being interpreted.

M/CPU n1 n2 --- (compiling assembler)
An assembler defining word used to create assembler mnemonics that have multiple address modes:

1C6E 60 M/CU ADC,

M/CPU creates the word ADC, with two parameters. When ADC, later executes, it uses these parameters, along with stack values and the contents of MODE to calculate and assemble the correct op-code and operand.

MEM Used within the assembler to set MODE to the default value for direct memory addressing, z-page.

MODE --- addr
A variable used within the assembler, which holds a flag indicating the addressing mode of the op-code being generated.

N --- addr (assembling)
Used in a code definition in the form:

N 1 - STA, or N 2+)Y ADC,

A constant which leaves the address of a 9 byte workspace in z-page. Within a single code definition, free use may be made over the range N-1 thru N+7. See SETUP.

NEXT --- addr (assembling)
A constant which leaves the machine address of the Forth address interpreter. All code definitions must return execution to NEXT, or code that returns to NEXT (i.e., PUSH, PUT, POP, POPTWO).

NOT cc1 --- cc1 (assembly-time)
When assembling, reverse the condition code for the following conditional. For example:

0= NOT IF, <true part> THEN,

will branch based on 'not equal to zero'.

POP --- addr (assembling)
n --- (run-time)

A constant which leaves (during assembly) the machine address of the return point which, at run-time, will pop a 16-bit value from the data stack and continue interpretation.

POPTWO --- addr (assembling)
n1 n2 --- (run-time)
A constant which leaves (during assembly) the machine address of the return point which, at run-time, will pop two 16-bit values from the data stack and continue interpretation.

PUSH --- addr (assembling)
--- n (run-time)
A constant which leaves (during assembly) the machine address of the return point which, at run-time, will add the accumulator (as high-byte) and the bottom machine stack byte (as low-byte) to the data stack.

PUT --- addr (assembling)
n1 --- n2 (run-time)
A constant which leaves (during assembly) the machine address of the return point which, at run-time, will write the accumulator (as high-byte) and the bottom machine stack byte (as low-byte) over the existing data stack 16-bit value (n1).

RP) --- (assembly-time)
Used in a code definition in the form:

RP) LDA, or RP) 3+ STA,

Address the bottom byte of the return stack (containing the low byte) by selecting the ,X mode and leaving n=$101. n may be modified to another byte offset. Before operating on the return stack the X register must be saved in XSAVE and TSX, be executed; before returning to NEXT, the X register must be restored.

SEC --- n (assembling)
Identical to BOT, except that n=2. Addresses the low byte of the second 16-bit data stack value (third byte on the data stack).

THEN, --- (run-time)
addr 2 --- (assembly-time)
Occurs in a code definition in the form:

cc IF, <true part> ELSE,
<false part> THEN,

At run-time THEN, marks the conclusion of a conditional structure. Execution of either the true part or false part resumes following THEN,. When assembling addr and 2 are used to resolve the pending forward branch to THEN,.

UNTIL, --- (run-time)
addr 1 cc --- (assembling)
Occurs in a CODE definition in the form:

BEGIN, . . . cc UNTIL,

At run-time, UNTIL, controls the conditional branching back to BEGIN,. If the processor status bit specified by cc is false, execution returns to BEGIN,; otherwise execution continues ahead.

At assembly time, UNTIL, assembles a conditional relative branch to addr based on the condition code cc. The number 1 is used for error checking.

UP --- addr (assembling)
Used in a code definition in the form:

UP LDA, or UP)Y STA,

A constant leaving at assembly time the address of the pointer to the base of the user area. i.e.,

12 # LDY, UP)Y LDA,

load the low byte of the sixth user variable, DP.

W --- addr (assembling)
Used in a code definition in the form:

W 1+ STA, or W 1 - JMP, or W)Y ADC,

A constant which leaves at assembly time the address of the pointer to the code field (execution address) of the Forth dictionary word being executed. Indexing relative to W can yield any byte in the definition's parameter field. i.e.,

2 # LDY, W)Y LDA,

fetches the first byte of the parameter field.

,X) Specify 'indexed indirect X' addressing mode for the next op-code generated.

XSAVE --- addr (assembling)
Used in a code definition in the form:

XSAVE STX, or XSAVE LDX,

A constant which leaves the address at assembly time of a temporary buffer for saving the X register. Since the X register indexes to the data stack in z-page, it must be saved and restored when used for other purposes.

Floating Point FORTH

By MICHAEL JESCH
Aregon Systems, Inc.

```
 cell 1                                 cell 2
|eeee eeee smmm mmmm|                  |mmmm mmmm mmmm mmmm|
|scaler and sign|            mantissa and sign
```

Figure 1: Floating Point Representation. "s" is the sign bit; "e" is a scaler, or exponent, bit; "m" is a mantissa bit.

One of the first things most programmers find missing in FORTH is floating point arithmetic. While most implementers of FORTH probably weigh the advantages and adversities of floating point for their version, they usually decide to forego it for various reasons. On the other hand, some excellent floating point systems have been developed in FORTH. This article overviews some major problems with floating point numbers, and examines a very rudimentary floating point system, written entirely in high level FORTH.

The first major problem with floating point numbers is that no computer can work with numbers that are truly floating point. Instead, quite often they are stored as two separate numbers, mantissa and exponent. This leads to

The greatest advantage of a floating point math system comes in ease of use

the second major problem, that of speed. Because each floating point number is stored as two separate numbers, each function requires that two numbers be dealt with, costing quite a bit in speed. One of the most common solutions for speed problems is to buy a dedicated processor chip to do all the arithmetic. This is impossible on some computers, and costly on others.

Another problem, that of accuracy, is a prime consideration. While floating point numbers can have a greater range, their precision suffers a little. The system outlined in this article has precision to only six full decimal digits, but the decimal point can be moved 127 places in either direction. This compared to a normal 32 bit double length number, where the range and accuracy are +/−2,147,483,647.

The greatest advantage of a floating point math system comes in ease of use: there is no need for the programmer to worry about where the decimal point should be, as it is handled internally by the floating point operators themselves. This saves time programming and debugging, and usually saves some memory too.

This floating point system is written in high level FORTH as an educational tool. Once the machine language of the target machine is understood, it should be rewritten in low level to capitalize on speed. One important side effect/advantage of this approach is transportability: It has since been implemented on two other computers, under different FORTH systems (polyFORTH and MMSFORTH); one with a different CPU (an LSI-11/23). This approach does, however, cost a lot of execution time.

As can be seen in Figure One, the floating point number is represented in two 16-bit cells on the stack. The high cell (cell 1) contains the 8-bit exponent (containing one sign bit) and the high 8 bits of the mantissa, including the mantissa sign, while the low cell (cell 2) contains the lower 16 bits of the mantissa. In this manner, the existing double length stack and memory operators can be used to manipulate the values.

The error detection is handled by the system, but the recovery is left up to the programmer. A special condition code 'register' is used to return information about the last operation. Currently, three of these bits are used: One each to indicate the occurrence of a zero value, a negative value, and most overflow/underflow conditions. These flags can be tested, as in a FORTH **IF . . . THEN** structure, with words defined in the package (**FZE FNE** and **FOV**). The word **FER** will return a true if any error existed.

The scaler is used to tell where the decimal point is, relative to the ones' column of the mantissa, during all math operations and output. A positive value indicates that the radix point is actually to the right of the ones' column and by how many digits, while a negative value means to move it to the left. It could be considered a 'times **BASE** to the **SCALER**' type of suffix to a number. For addition and subtraction, the scalers must be made equal (the word **ALIGN** does this). This means shifting the mantissa the number of places equal to the difference of the exponents, which causes most of the imprecision problems present in this system. Furthermore, if one number was entered in hexadecimal (base 16) and the other in decimal, the scalers would be incompatible. To help circumvent this, a floating point base value is kept separate from the FORTH base value, and all internal scaling oprations use this value for the number base. The floating point base is set to the current FORTH base when the floating point system is initialized (with **FINIT**). It can also be explicitly set by the programmer, but be careful with this: if you output a number in a different base than you did arithmetic, the results will not be correct.

Number formatting is left up to the programmer, as it is in most FORTH systems. A double length number may be converted to floating point by inserting the desired scaler (number scaler F!). To change a number from floating point to integer, the word **FIX** will scale the mantissa to zero. The scaler of the top floating point number can be extracted with the word E@ which leaves the double precision mantissa below, unmodified. **F.** and **E.** are used to output the top floating point number. **F.** prints in floating point format (i.e., 123.45), while **E.** prints in scientific notation (i.e., 12345 E −2).

The four basic arithmetic functions, add, subtract, multiply and divide, are called **F+**, **F−**, **F*** and **F/**, respectively.

RSCALE and **LSCALE** are used to change the position of the least significant digit in the mantissa. **RSCALE** decrements the scaler and multiplies the mantissa by base (changes 12.3 to 12.30), while **LSCALE** increments the scaler and divides the mantissa by base (changes 12.34 to 12.3). Be careful with these words, however. If the number is close to the limit of precision, the number will probably lose accuracy.

Other miscellaneous words are **FABS**, **FNEGATE**, **FMIN**, **F>**, and **FMAX**; these are the floating point counterparts to **ABS**, **NEGATE**, **MIN**, **>**, and **MAX**, respectively.

A Recursive Decompiler

Robert Dudley Ackerman

Editor's Note: A FORTH "decompiler" is a tool that scans through a compiled dictionary entry and tells you what has been compiled. In the case of a colon definition, it prints the names of the words that are pointed to inside the definition. In an ideal programming environment, in which you have the source for your system right on your disk, you may not need a decompiler. But otherwise, it beats all the hit and miss "ticking" and dumping you would have to do. Decompilers can also be useful learning tools.

A very thorough decompiler was written by Ray Duncan of Laboratory Microsystems and published in Doctor Dobbs, September 1981. The following decompiler, while not as complete as Ray's (and not as elegantly written — beware of long definitions), introduces a clever feature: recursive descent. In this version, pressing the space bar steps you through each name used in a colon defintion, but pressing carriage return instead causes the word whose name was just printed to be itself decompiled. This allows you to weave your way through the threaded interpretive code down to any level you want.

On occasion it is desirable to know what words a given word is made up of and what words those words are made up of in turn. Thus the word DECOMPILE which naturally calls for recursion.

GIN keeps track of indentation (Goes IN). **DIN** does an indentation (Does INdent). **GCHK** does special cases, particulary where a word is followed by a literal

The main word, <DECOMP> is straight-forward For a colon definition, it goes through each code field printing a name and waiting for a key.

A 'Q' ends execution; a carriage return calls <DECOM> recursively, printing out the names in the last word shown; any other key continues until EXIT signals the end of a colon definition, or <:CODE> signals a drop into machine language from high level.

One improvement I envision is being able to back up one level, rather than quiting altogether. This would avoid the problem of having to avoid 'error' and other words which use words which use themselves. You could back up one level rather than quiting, not being able to finish the original word. Another improvement would be to use a fence to avoid seeing low level words of no immediate interest.

To use this utility with a Starting FORTH system, change the ticks to bracket-ticks. ' —> ['] .

Robert Dudley Ackerman is head of the San Francisco Apple Core FORTH Users. □

TRACING COLON-DEFINITIONS

Paul van der Eijk
5480 Wisconsin Avenue, #1128
Chevy Chase, MD 20015
(301) 656-2772

This short article describes a few simple words to trace colon definitions. When I am completely lost trying to find a bug in a FORTH program, I use colon tracing to get a print-out of all words executed together with a few parameters on the data-stack. Such a print-out is often enough to spot the bug; in addition, it gives some insight how many times certain words are executed which can help to improve the execution time of a program.

How it works:

A technique to trace colon definitions is to insert a tracing word directly after the colon.

i.e., : TEST T1 T2 ; TEST can be traced by having a definition compiled as if it were:

```
: TEST <TRACE> T1 T2;
```

When <TRACE> executes, the address of the word following it is on the return stack. Subtracting two from this address will give the parameter field address, from which we can reach the name field address using the word NFA. In order to enable/disable the trace ouput, the variable TFLAG is used; a non-zero value will enable the output and a zero value will suppress the trace output.

The insertion of the <TRACE> word can be automated if we redefine the definition of the colon.

The colon is redefined to insert the runtime procedure for the colon followed by the address of <TRACE>.

Note that the address of the colon runtime procedure is obtained by taking it from the code field address of the word <TRACE>.

Improvements:

1. If we save in (TRACE) the value of the variable OUT and direct output to the line-printer, words doing formatted terminal output can be debugged effectively.

2. A variable TRACE is introduced to control the insertion of the word (TRACE) in the new definition for the colon.

If the value of TRACE equals zero, (TRACE) is not inserted, if the value is non-zero (TRACE) will be inserted.

This enables tracing code to be inserted in a selective way by changing the value of TRACE preceding a colon definition.

i.e.:

```
OFF TRACE ! : TEST1  T11  T12 ; ( TEST1 will
 not be traced )

ON TRACE ! : TEST2  T21  T22 ; ( TEST 2 can
 be traced )
```

Fixed Point Square Roots

By KLAXON SURALIS

As you learn the FORTH approach to problem solving, you are sold on the virtues of fixed-point arithmetic. Confidently armed with the neat little techniques for adapting integers to nearly any application, you set out to write a numeric-type program.

While translating the requisite formulae into postfix notation, however, you run into a radical sign — a square root. In vain, you try to sneak around it or tunnel underneath; there's no way to avoid it. What can you do? Pile K upon K of floating point routines into your dictionary, just so you can use the SQRT function? Give up and go back to BASIC?

This can happen to you, whether your field is statistics, electronics, graphics, or special relativity. Square roots are everywhere, in all kinds of famous equations. Of all the "irrational" functions in mathematics, this is the most common — and the simplest to implement.

Loading the Screens.
The program is listed as screens 33 through 34 . Of course, you may put them anywhere your mass storage allows. All three screens load in base ten; all are FORTH-79 Standard.

If your FORTH system is nonstandard, the source code will require only minor modification. The FORTH-79 word **R@** must be replaced by **R** (in fig-FORTH) or **I** (in polyFORTH). You may have to code **1—** as a separate **1** and **—**.

Additionally, some systems may omit the FORTH-79 required words **D<** and **U<**. If yours is one of them, just define (with **BASE** set to **DECIMAL**):

```
: D<  ROT 2DUP =
 IF ROT ROT DNEGATE D+ 0<
 ELSE SWAP < SWAP DROP
 THEN SWAP DROP ;

: U<  32768 + SWAP 32768 + SWAP < ;
```

Make SURE 32000 −32000 < returns 0 on your system. If not, demand a refund.

Trying It Out.
With all screens loaded, enter:

```
169 0 SQRT .
```

This should print 13, which is the square root of 169. Note that we had to put an 0 on top of 169, since SQRT demands a double-precision operand. Most systems would have accepted 169. , taking the decimal point as a signal to extend the value to 32 bits. Thus, you could try:

```
480249. SQRT .
```

and see the result: 693. Omitting the decimal point from 480249 , however, would print a garbage value and possibly a "stack empty" message.

SQRT works only for integers, dropping any fractional component in the root. So, if you type:

```
3. SQRT .
```

the system will respond with 1 , even though you know the answer should be something like 1.7320508... . But after all, this is fixed point arithmetic.

However, you can take the square root of 3 million:

```
3000000. SQRT .
```

and sneak a peek at three more root digits: 1732 . This is the principle embodied in the word XX defined on screen 224. Type:

```
3 XX
```

and there you have 1.732 . XX will type the square root of any integer up to 4095 . Since it takes a single-precision argument, you don't have to type a decimal point. Try a few roots of your own.

If you compare XX's results to a scientific calculator's, you'll see that XX is always accurate to three decimal places, although it never rounds upward. Round-to-nearest, while possible, usually doesn't justify the added complexity.

SQRT's 32-bit radicand makes it perfect for finding hypotenuses, standard deviations, RMS values, and lots of other "root-of-sum-of-squares" procedures.

Now look at the definition of XX. Right off, it multiplies n by one million. Then, after calling SQRT, it performs a special numeric conversion, which puts the three low-order root digits on the fraction side of a decimal point (ASCII code 46).

This combination of scaling and output formatting is the standard FORTH technique for ersatz floating point. The only wrinkle with SQRT is:

> If you multiply SQRT's radicand by a scale factor s, the root will emerge scaled by the square root of s.

More concretely, you've seen how XX shifts the radicand left six decimal digits, while the root is offset by only three places. Square root of 1,000,000 = 1,000. Get it?

Using SQRT in Your Application.
XX is included merely for demonstration purposes. In normal use, SQRT is the only word the rest of your application will need to know. It will work for any double number you feed it, and harbors no surprises or special cases (as far as I can tell).

SQRT's 32-bit radicand makes it perfect for finding hypotenuses, standard deviations, RMS values, and lots of other "root-of-sum-of-squares" procedures. If the input data are all like-scaled 16-bit integers, simply square them with **M*** or **U*** and accumulate them using **D+** (or **D−** if you need subtractions). Apply SQRT to this double-precision sum, and the root will be a 16-bit integer with the same scale factor as your original data.

Of course, it's up to you to ensure that **D+** won't overflow and that **D−** doesn't hand SQRT a negative number.

Since SQRT takes an unsigned radicand, your application is free to handle imaginary roots as appropriate. If you like, you may install simple error checking for negative radicands, or integrate SQRT into a complete fixed-point (!) complex numbers package.

But How Does It Work?
On big machines, square roots are extracted by a technique from calculus called "Newton's Method." It is best suited to CPUs with full floating-point arithmetic hardware.

The alternative approach, used in this SQRT, works by addition, subtraction, and shifting. The result is constructed bit by bit, in a fashion quite similar to classical binary long division.

As in quotient generation, we set up

a partial remainder (initially equal to the radicand) and proceed to chip away at it. If we take away too much, we put it back and shift a "O" into the root; otherwise, the root gets a "1". Sixteen such trials take place, one for each bit of the root.

What's different is the quantity subtracted/added to that remainder. Instead of an unchanging divisor, we must use the root itself — as many bits of it as are already determined — with a binary "01" stuck on the end. As the calculation advances, this subtrahend gets wider and wider.

For all but the last two trials, 16-bit addition and subtraction are wide enough to cover the action. This 87.5% share of the job is done by EASY-BITS (screen 33). As it churns along, EASY-BITS uses left shifts to keep root and remainder aligned under the optimal 16-bit window.

Within EASY-BITS, the phrases " 2* 1- " and " 2* 3+ " may bewilder you. They are sneaky, optimized equivalents for " 1- 2* 1+ " and " 1+ 2* 1+ ", respectively. In two brief steps, they shift a new bit into the root and reestablish the "01" suffix.

Now, if you look at the definition of SQRT (screen 34), you'll see the 14 easy bits extracted in two chunks: first eight bits, then six more. There is a very good reason for this: it saves us from defining and using a 48-bit shifting operator, which would run slow as molasses.

You see, we needn't look at the radicand's low-order 16 bits until the root is half finished. At that point, the "ROT DROP" throws away a cell cleared to zeroes by "8 EASY-BITS". With the rest of the radicand thus exposed, we're ready to crank out six more root bits.

The last two bits are coaxed out by 2'S-BIT (screen 33) and 1'S-BIT (screen 34). As usual in computer arithmetic, it's the down-to-the-last-bit accuracy that really costs. The problem here is that the remainder and root become too wide for plain old + , - , and 0<. We have to worry about overflow.

Both 2'S-BIT and 1'S-BIT use " DUP 0<IF " to test a high bit before D2* shifts it into oblivion. Moreover, since " - 0<" can no longer be trusted, we must resort to unsigned comparisons (U< and DU<). These definitions wouldn't be so ugly if FORTH had the "carry bit" found on most microprocessors.

The principles of SQRT may be applied to roots of greater precision — 32 bits, frinstance. In fact, you could use this technique in a program to extract the square roots of floating point numbers!

You Demand Proof?

A formal derivation of SQRT's algorithm would waste a lot of FIG's paper on something nobody would read. So, I'll supply a hint, if you supply your own paper. Let:

N = the 32-bit radicand
r = the 16-bit root under construction, initially zero
b = the binary place value of the root bit to be found (initially 32768, always a power of 2)

Then, for each value of b from 32768 down to 1, DO:

```
IF  $(r+b)^2$  <=  N
THEN  add b to r ;
```

In this approach, b is the only quantity that gets shifted. Now, simple algebra rewrites the condition as:

```
b (2r+b)  <=  N - $r^2$
```

The changing value $N - r^2$ is our partial remainder. Note that the comparison is 32 bits wide, remember that multiplying by 2 and b are simple binary left shifts, and the rest is mere optimization.

Alternatively, you may find this algorithm described in any thorough treatment of computer arithmetic.

Not Fast Enough?

If SQRT runs too slow for your application, there's not much you can do without delving into machine CODE. If you don't need full 16-bit accuracy in your roots, you can substitute:

```
>R 2DROP R> 1-
```

for the final " 2'S-BIT 1'S-BIT " in the definition of SQRT. The two low-order root bits will then always be zero. The payoff: roughly 10% faster execution, depending on your system. Perhaps more importantly, you can throw out DU<, 2'S-BIT , and 1'S-BIT, cutting the program size in half.

If you have a FORTH assembler, the first thing you should try is defining 2* and D2* in low level. On my homebrew 6809 FORTH, this, by itself, nearly doubles execution speed. Actual mileage may vary, yours will probably be less.

Beyond this, all you can do is translate the whole mess into CODE. This task is straightforward, but, of course, CPU dependent.

Of those high-level programming languages which give you a choice, most make you accept a lot of unnecessary garbage just to get one function you really want. Floating point arithmetic and function packages serve as cases in point.

FORTH exhibits the opposite attitude. It lets you order a la carte, as this fixed-point square root program demonstrates. Such freedom and efficiency, however, cannot be divorced from the added responsibility of knowing exactly what you're doing and exactly what you want.

Fixed-Point Trig by Table-Lookup

By JOHN S. JAMES
Colon Systems San Jose, California

Here is an easy way to get sine and cosine on your FORTH system, even if it does not have floating point. The precision is good enough for graphics and many other applications, and the routines are fast.

The principle is to use table lookup to return the sine of 0 through 90 degrees. Simple compuations can express the sine or cosine of any integer number of degrees, positive or negative, in terms of the sine of 0-90. Since only integers are available, the values returned are sine or cosine multiplied by 10,000.

Because the results are scaled up, you can use the FORTH scaling operation `*/` (multiply and then divide) to get the results you want. For example, to multiply a number on the stack by the sine of 15 degrees, use

```
15 SIN 10000 */
```

JUST IN CASE

Dr. Charles E. Eaker

Even though FORTH provides a variety of program control structures, a CASE structure typically has not been one of them. There is no particular reason for this since, as we shall soon see, it is not difficult to implement one.

There are two different approaches one can take to implementing a CASE structure: vectored jumps and nested IF...ELSE...THEN structures. Vectored jumps provide the greatest speed at run-time but produce enormous compiling complications. So, taking the path of least resistance, here is a proposal for implementing a CASE structure for FORTH which is really just a substitute for nested IF structures. But, even though the proposal is logically redundant, there are a number of practical benefits which make it worthy of consideration.

To help this discussion, consider a word which might appear in an assembler vocabulary with a glossary entry as follows:

GEN operand, opcode, mode selector ---

Used by the ASSEMBLER vocabulary to generate opcodes. 'Mode selector' is the value which indicates which addressing mode has been specified. 'Opcode' is the value placed on the stack by the preceding mnemonic, and 'operand' is the value to be used as the argument of the opcode.

Here is one way of coding GEN.

```
: GEN   0 OVER =
   IF DROP IMMEDIATE
   ELSE 10 OVER =
    IF DROP DIRECT
    ELSE 20 OVER =
     IF DROP INDEXED
     ELSE 30 OVER =
      IF DROP EXTENDED
      ELSE DROP MODE-ERROR
      ENDIF
     ENDIF
    ENDIF
   ENDIF   RESET ;
```

GEN is defined to expect a 16-bit number on top of the stack. For each IF, this number, the "select value," is copied and tested against a constant, the "case value." If the select value equals the case value the appropriate code is executed. If all tests fail, MODE-ERROR is executed. Notice that GEN meticulously keeps the stack clean.

Depending on the select value, some action is performed on the opcode and operand, and GEN removes them from the stack. Consequently, before each test, GEN must copy (OVER) the select value, and if the test is successful, the select value must be dropped from the stack to expose the data values prior to the appropriate routine being called.

But wouldn't you rather code this thing this way?

```
: GEN  CASE
            0 OF   IMMEDIATE     ENDOF
           10 OF   DIRECT        ENDOF
           20 OF   INDEXED       ENDOF
           30 OF   EXTENDED      ENDOF
           MODE-ERROR
      ENDCASE    RESET ;
```

It is certainly easier to see what this routine is doing, so comments are not as necessary, and changes and repairs are far easier to do. Here are the required colon definitions of CASE, OF, ENDOF, and ENDCASE.

```
: CASE      ?COMP   CSP @  !CSP    4  ;  IMMEDIATE

: OF   4 ?PAIRS  COMPILE OVER  COMPILE =  COMPILE 0BRANCH
          HERE 0 ,   COMPILE DROP   5  ;  IMMEDIATE

: ENDOF     5 ?PAIRS   COMPILE BRANCH   HERE 0 ,
          SWAP  2 [COMPILE] ENDIF   4  ;  IMMEDIATE

: ENDCASE   4 ?PAIRS    COMPILE DROP
          BEGIN   SP@ CSP @ = 0=
            WHILE  2 [COMPILE] ENDIF   REPEAT
          CSP !  ;  IMMEDIATE
```

It so happens that with these definitions both versions of GEN compile the identical code into the dictionary. Let's look at the compiling details.

CASE makes sure that it is in a colon definition. Then it saves the value of CSP (which contains the position of the stack at the beginning of this case structure) and sets CSP equal to the present position of the stack. The new value of CSP will be used later by ENDCASE to resolve forward references. Finally, it throws a four onto the stack which will be used for checking syntax. CASE compiles no code into the dictionary.

OF first checks that it has been preceded either by CASE or an ENDOF. If the syntax is in order, then code is compiled into the dictionary to duplicate the select value (OVER) and test its equality to the current case value (=). Next, code for a conditional branch is compiled into the dictionary followed by code for DROP. Notice that at run-time the DROP is executed only if the select value equals the constant for this OF...ENDOF pair.

ENDOF first checks that an OF has gone before. If so, then it compiles an absolute branch to whatever code follows ENDCASE. However, the address to branch to is not yet known, so a dummy null is compiled into the address and its location is left on the stack so ENDCASE will know where to stick the address once it is known. But there is already an address on the stack just under the one which ENDOF just pushed. This address was left by OF and it points to an address that should hold a branch address to the code which follows the code generated by ENDOF. So, ENDOF swaps the addresses and calls ENDIF to resolve the address at the address left by OF. Finally, ENDOF leaves a four on the stack for syntax checking.

ENDCASE makes sure it has been preceded by either a CASE or ENDOF. Otherwise an error message is issued and compilation is aborted. Code for a DROP is compiled into the dictionary, then all the unresolved forward branches left by each ENDOF are resolved. Since there may be any number of them, including none, ENDCASE checks the current stack position against what it was when CASE was executed, and performs a fixup by calling ENDIF until the stack no longer contains addresses left by previous ENDOF's. Notice that all of these branches are resolved to point to the code after the DROP generated by ENDCASE. In the case of GEN this is RESET.

It doesn't take long to notice that OF generates an enormous amount of code (10 bytes). This is a classic example of a situation that cries out for a machine language primitive. If a run-time word could be defined, let's call it (OF), then each OF would generate just 4 bytes two to point to (OF) and two for the branch address. What (OF) would have to do is pull the top stack item (the current case value) and test it for equality with the new top stack item (the select value) If the test for equality is true then the next item on the stack the select value is also popped and execution continues after the (OF) If the test is false execution branches using the branch value following the pointer to (OF), and the select value is left on the stack.

```
CODE  (OF)  A PUL   B PUL   TSX
            1.X B SUB  0.X A SBC  ABA  0=
            IF  INS INS   ' BRANCH CFA 9 ( HEX ) 11 + JMP
            THEN   ' BRANCH CFA 2 JMP
: OF  4 ?PAIRS  COMPILE (OF) HERE 0 , 5 ; IMMEDIATE
```

The M6800 code listed above is straightforward except that is uses code in BRANCH and 0BRANCH. (OF) should work in any FIG 6800 installation provided BRANCH and 0BRANCH have not been altered (it doesn't matter where they are located). Non-6800 users will have to roll their own, but the high-level OF should make it clear what has to be done.

The disadvantages of this CASE proposal are that execution is not as fast as a vectored implementation, and in some versions of FORTH, ENDOF and ENDIF cannot be distinguished. These seem minor compared to the advantages - and there are several.

First, a CASE statement may contain any number of OF...ENDOF pairs, and the constants may be arranged in any order whatever. Actually the constants need not be constants. Between an ENDOF and the next OF the programmer may insert as much code as he or she likes including code which will compute the value of the "constant." CASE statements may be nested; a CASE...ENDCASE pair may appear between an OF...ENDOF pair. Furthermore, there need not be any code between CASE and ENDCASE, nor must there be code between OF and ENDOF. There must be code which pushes a 16-bit number to the stack prior to each OF. Finally, this proposal follows the fig-FORTH style of handling control structures.

fig-FORTH GLOSSARY

CASE --- addr n (compiling)

Used in a colon definition in the form: CASE...OF...ENDOF...ENDCASE. Note that OF...ENDOF pairs may be repeated as necessary.

At compile-time CASE saves the current value of CSP and resets it to the current position of the stack. This information is used by ENDCASE to resolve forward references left on the stack by any ENDOF's which precede it. n is left for subsequent error checking.

CASE has no run-time effects.

```
OF --- addr n        (compiling)
       n1 n2 --- n1  (if no match)
       n1 n2 ---     (if there is a match)
```

Used in a colon definition in the form: CASE...OF...ENDOF...ENDCASE. Note that OF...ENDOF pairs may be repeated as necessary.

At run-time, OF checks n1 and n2 for equality. If equal, n1 and n2 are both dropped from the stack, and execution continues to the next ENDOF. If not equal, only n2 is dropped, and execution jumps to whatever follows the next ENDOF.

At compile-time, OF emplaces (OF) and reserves space for an offset at addr. addr is used by ENDOF to resolve the offset. n is used for error checking.

ENDOF addr1 n1 --- addr2 n2 (compiling)

Used in a colon definition in the form: CASE...OF...ENDOF...ENDCASE. Note that OF...ENDOF pairs may be repeated as necessary.

At run-time, ENDOF transfers control to the code following the next ENDCASE provided there was a match at the last

OF. If there was not a match at the last OF, ENDOF is the location to which execution will branch.

At compile-time ENDOF emplaces BRANCH reserving a branch offset, leaves the address addr2 and n2 for error checking. ENDOF also resolves the pending forward branch from OF by calculating the offset from addr1 to HERE and storing it at addr1.

```
ENDCASE addr1...addrn n --- (compiling)
        n ---      (if no match)
          ---      (if match was found)
```

Used in a colon definition in the form: CASE...OF...ENDOF...ENDCASE. Note that OF...ENDOF pairs may be repeated as necessary.

At run-time, ENDCASE drops the select value if it does not equal any case values. ENDCASE then serves as the destination of forward branches from all previous ENDOF's.

At compile-time, ENDCASE compiles a DROP then computes forward branch offsets until all addresses left by previous ENDOF's have been resolved. Finally, the value of CSP saved by CASE is restored. n is used for error checking.

```
(OF) n1 n2 --- n1 (if no match)
     n1 n2 ---    (if there is a match)
```

The run-time procedure compiled by OF. See the description of the run-time behavior of OF.

This is an excellent development and presentation of a key case statement with single integer keys. The following features make it immediately useful:

1. The reader can easily understand what the statement does and how to use it. There are only four words to learn, their functions are immediately clear from the example presented and their names are not confused with each other. (The ENDOF - ENDIF similarity will go away when the FIG model drops ENDIF in favor of the Standards Team decision to use THEN.)

2. One form of the statement can be entered entirely in higher-level fig-FORTH, and run immediately on any FIG system. An optional code word (for 6800) with redefinition of one of the four higher-level words saves run-time memory and time. Either way, the whole statement fits easily on one screen, including compile-time checking.

3. The narrative documentation is excellent. The glossary definitions are detailed (appropriate for this forum). For general distribution they could be condensed to user-only information.

This entry presents one kind of case statement out of several that are desired. We hope that this competent and straightforward work will serve as a model to future development.

THE : CASE

Steve Munson

Having grown up on an ancient version of FORTH Inc. micro FORTH, I can appreciate the improvements rendered by fig-FORTH's renames and redefinitions. I was particularly impressed by the source equivalence of HERE NUMBER DROP which functions the same although in one case one is dropping the address of the first non-numeric delimiter, and in the other case one is dropping the most significant half of a double precision number!

My one beef is why was : made IMMEDIATE? Surely nobody wants a header in the middle of a colon definition. By the way, as you probably already know, this tends to mask an error in the definition of ; on the listing I have for the 6502 fig - FORTH. There is no [COMPILE] before the [which means compile mode is never terminated. In fact, I am not sure I see the point of the E property in your glossary. All words ought to be designed, at great pains if necessary, so that they can be compiled. My definition of CASE denies the E property of :, and I would be rash to assume no one would ever want to compile CASE.

Please find enclosed a listing, documentation, glossary entries, and a diskette. The diskette also contains the assembler used to generate the code, as it may be nonstandard If the fig-FORTH does not run on your system as it does on mine, feel free to edit my ideas into polished fig-FORTH (I am a novice figger) and re-list the screens; however I believe they will require no modification.

GLOSSARY ENTRIES

2BYTECASE

Keycase defining word, used in the form:

2BYTECASE cccc key$_0$ case$_0$ key$_1$ case$_1$ . . . key$_n$ case$_n$ (default case) END-CASE. Defines cccc as a caseword which expects a 2-byte key on the stack at run-time. If the key equals key$_0$ (a 2-byte key), case$_0$ (a previously defined word) will execute; if it matches key$_1$, case$_1$ will execute, and so on. The default case will execute on no match; if no default is specified, NOOP is assumed. Cases may be IMMEDIATE words, but no compile-time execution will occur within the case structure. The structure must be terminated by END-CASE. (See END-CASE, BYTE-CASE).

BYTECASE

Keycase defining word, used in the form:

BYTECASE cccc key$_0$ case$_0$ key$_1$ case$_1$ . . . key$_n$ case$_n$ END-CASE. Defines cccc as a caseword which expects a 1-byte key (most significant byte is ignored) on the stack at run-time. If the key equals key$_0$ (a 1-byte key), case$_0$ (a previously defined word) will execute; if it equals key$_1$, case$_1$ will execute, and so on. The default case will execute on no match; if no default is specified, NOOP is assumed. Cases may be IMMEDIATE words, but no compile-time execution will occur within the case structure. The structure must be terminated by END-CASE. (See END-CASE, CASE).

V. Recommended Books (available from Parsec)

V-1 FORTH Programming

Starting FORTH by Leo Brodie, Prentice Hall pub.- A tutorial introduction to FORTH.

The Complete FORTH by Alan Winfield, Wiley Press.-Easy-to-follow examples, exercises and frequent comparisons to BASIC.

BEGINNING FORTH by Paul Chirlian, Matrix Publishers, A self-teaching guide to FORTH, compares FORTH to other popular languages.

Discover FORTH by Thom Hogan, Osborne/McGraw-Hill- Introdutory learning and programming the FORTH language.

FORTH Programming by Leo J. Scanlon, Howard W. Sams & Co.- Based on both FORTH-79 and fig-FORTH. Programming is introductory to advanced.

Understanding FORTH by Joseph Reymann, Alfred Publishing Co.- An introduction and overview.

V-2 FORTH Reference Guides

All About FORTH by Glen B. Haydon, Mountain View Press- An annotated glossary of MVP FORTH words.

FORTH Encyclopedia by Mitch Derick & Linda Baker, Mountain View Press- The complete FORTH programmers reference manual.

Pocket Guide to FORTH by Linda Baker & Mitch Derick, Addison-Wesley Publications- Pocket reference guide for FORTH.

V-3 Commodore 64 Reference

Commodore 64 Programmer's Reference Guide, published by Commodore- A complete technical reference guide to the Commodore 64 computer.

Your Commodore 64 by John Heilborn & Ran Talbot, Osborne/McGraw-Hill- ' An easy-to-understand, fully illustrated teaching guide to the Commodore 64 computer.

What's Really Inside the Commodore 64 by Milton Bathurst, Datacap- Commented listings of the BASIC and Kernel ROMs in the C64.

Indispensible to the machine language programmer who wants to know when the ROM's routines are REALLY doing.

The Master Memory Map for the C-64 by Paul Pavelko, Reston Pub. Co.- Friendly guide to the inner workings of the C-64.

V-4 Commodore 64 Graphics/Sound Related

The Commodore 64 Music Book by James Vogel, Birkhauser Pub.- A guide to programming music and sound on the Commodore 64.

Commodore 64 Graphics & Sound Programming by Stan Krute, TAB books- Hands on learn-by-doing approach to mastering graphics and sound on the C-64.

V-5 Assembly Language Programming

6502 Assembly Language Programming by Lance Leventhal, Osborne/McGraw Hill- Provides a comprehensive coverage of the 6502 microprocessor assembly language. General assembly programming techniques covered as well as specific examples for the 6502.

6502 Assembly Language Subroutines by Leventhal and Saville, Osborne/McGraw-Hill, code and descriptions for common 6502 subroutines. Programming the 6502, by Rodney Zaks, Sybex Inc.- A guide to programming the 6502 in assembly language. Slightly more hardware oriented than Leventhal (above).

VI. Error Messages

The following is a list of the more common error messages which occur in the system. The word which generates the message is given along with a possible explanation of why the error occured.

Message (Word) : Explanation

ATTEMPTED TO REDEFINE NULL (CREATE) : Probably entered a defining word (such as ":" or "CREATE") with no word name following it on the line.

BLK NO. ERROR (<R/W>) : Block number passed to <R/W> was either less than 0 or greater than the maximum number of blocks the system is configured for.

BUFFERS FULL (BUFFER) : In File Mode the screen buffers are full. No more can be added to the file being edited until more buffers are allocated (the file must be saved before allocating new buffers).

COMPILE ONLY (?COMP) : A word which can only be entered within a definition was entered interactively. Examples of "COMPILE ONLY" words are DO, LOOP, IF, ELSE, THEN, etc.

CONDITIONALS NOT PAIRED (?PAIRS) : An ending word of a program structure (such as LOOP, ELSE, THEN, REPEAT) was detected with no matching beginning word.

DEFINITION NOT FINISHED (?CSP) : The ending word in a program structure (such as DO...LOOP or IF...ELSE...THEN) was not entered before ending the definition.

EMPTY STACK (?STACK) : The stack had no parameter for a word which required one.

EMPTY STACK (D.R) : Tried to print out a number but none was on the stack.

FULL STACK (?STACK) : The stack is full. Probably caused by multiple calls to a word which leaves too many values on the stack.

IN PROTECTED DICTIONARY (FORGET) : The word which the system was requested to forget is in the protected part of the dictionary (initially the FORTH Kernel)and cannot be forgotten.

ISN'T UNIQUE (CREATE) : Warning message to let the user know that the word being created has a name which already exists in the dictionary. If the definition is successful, any further references to the name (within the current vocabulary) will be to the word being defined.

The variable WARNING is used to control printing of this message. 0 WARNING ! will suppress the message while 1 WARNING ! will cause it to be displayed.

INCORRECT ADDRESSING (ASSEMBLER) : An invalid addressing mode was attempted during assembler word compilation.

INPUT > 255 (<WORD>) : The input stream being scanned by <WORD> was larger than 255 characters without a delimiter being detected.

INPUT STREAM EXHAUSTED (?STREAM) : Probably caused by a comment which was left undelimited when finished (no right parenthesis).

NOT FOUND (') : Word address not found by '. Possibly defined in different vocabulary.

NOT FOUND ([COMPILE]) : Tried to [COMPILE] a word whose name was not found in the dictionary.

NOT IN CURRENT VOCABULARY (FORGET) : The word which the system was requested to forget is not in the current vocabulary. It may, however, be in another vocabulary.

NOT RECOGNIZED (<NUMBER>) : Non-numeric character passed to <NUMBER>. Usually means system tried to interpret a word which is not in the dictionary.

OFF SCREEN (CHKLIN) : Tried to edit a line which was not on the editing screen.

OUT OF RANGE (CONFIGURE) : Tried to configure more than five drives.

PICK ARGUMENT < 1 (PICK) : The argument passed to PICK was less than one and therefore undefined.

ROLL ARGUMENT < 1 (ROLL) : The argument passed to ROLL was less than one and therefore undefined.

UNLOADABLE (LOAD) : Tried to load block 0. Block 0 is defined as unloadable.

INDEX

INDEX

INDEX

INDEX

FORTH-79 HANDY REFERENCE

Stack inputs and outputs are shown; top of stack on right. See operand key at bottom.

PARSEC RESEARCH
Drawer 1766
Fremont, CA 94538

STACK MANIPULATION

DUP	(n → n n)	Duplicate top of stack.
DROP	(n →)	Discard top of stack.
SWAP	(n1 n2 → n2 n1)	Exchange top two stack items.
OVER	(n1 n2 → n1 n2 n1)	Make copy of second item on top.
ROT	(n1 n2 n3 → n2 n3 n1)	Rotate third item to top. "rote"
PICK	(n1 → n2)	Copy n1-th item to top. (Thus 1 PICK = DUP , 2 PICK = OVER)
ROLL	(n →)	Rotate n-th item to top. (Thus 2 ROLL = SWAP , 3 ROLL = ROT)
?DUP	(n → n (n))	Duplicate only if non-zero. "query-dup"
>R	(n →)	Move top item to "return stack" for temporary storage (use caution). "to-r"
R>	(→ n)	Retrieve item from return stack. "r-from"
R@	(→ n)	Copy top of return stack onto stack. "r-fetch"
DEPTH	(→ n)	Count number of items on stack.

COMPARISON

<	(n1 n2 → flag)	True if n1 less than n2. "less-than"
=	(n1 n2 → flag)	True if top two numbers are equal. "equals"
>	(n1 n2 → flag)	True if n1 greater than n2. "greater-than"
0<	(n → flag)	True if top number negative. "zero-less"
0=	(n → flag)	True if top number zero. (Equivalent to NOT) "zero-equals"
0>	(n → flag)	True if top number greater than zero. "zero-greater"
D<	(d1 d2 → flag)	True if d1 less than d2. "d-less-than"
U<	(un1 un2 → flag)	Compare top two items as unsigned integers. "u-less-than"
NOT	(flag → ¬flag)	Reverse truth value (Equivalent to 0=)

ARITHMETIC AND LOGICAL

+	(n1 n2 → sum)	Add. "plus"
D+	(d1 d2 → sum)	Add double-precision numbers. "d-plus"
−	(n1 n2 → diff)	Subtract (n1−n2). "minus"
1+	(n → n+1)	Add 1 to top number. "one-plus"
1−	(n → n−)	Subtract 1 from top number. "one-minus"
2+	(n → n+2)	Add 2 to top number. "two-plus"
2−	(n → n−2)	Subtract 2 from top number. "two-minus"
*	(n1 n2 → prod)	Multiply. "times"
/	(n1 n2 → quot)	Divide (n1/n2). (Quotient rounded toward zero) "divide"
MOD	(n1 n2 → rem)	Modulo (*i.e.*, remainder from division n1/n2). Remainder has same sign as n1. "mod"
/MOD	(n1 n2 → rem quot)	Divide, giving remainder and quotient. "divide-mod"
*/MOD	(n1 n2 n3 → rem quot)	Multiply, then divide (n1*n2/n3), with double-precision intermediate. "times-divide-mod"
*/	(n1 n2 n3 → quot)	Like */MOD , but give quotient only, rounded toward zero. "times-divide"
U*	(un1 un2 → ud)	Multiply unsigned numbers, leaving unsigned double-precision result. "u-times"
U/MOD	(ud un → urem uquot)	Divide double number by single, giving remainder and quotient, all unsigned "u-divide-mod"
MAX	(n1 n2 → max)	Leave greater of two numbers. "max"
MIN	(n1 n2 → min)	Leave lesser of two numbers. "min"
ABS	(n → |n|)	Absolute value. "absolute"
NEGATE	(n → −n)	Leave two's complement
DNEGATE	(d → −d)	Leave two's complement of double-precision number "d-negate"
AND	(n1 n2 → and)	Bitwise logical AND
OR	(n1 n2 → or)	Bitwise logical OR.
XOR	(n1 n2 → xor)	Bitwise logical exclusive-OR. "x-or"

MEMORY

@	(addr → n)	Replace address by number at address. "fetch"
!	(n addr →)	Store n at addr "store"
C@	(addr → byte)	Fetch least significant byte only. "c-fetch"
C!	(n addr →)	Store least significant byte only. "c-store"
?	(addr →)	Display number at address. "question-mark"
+!	(n addr →)	Add n to number at addr. "plus-store"
MOVE	(addr1 addr2 n →)	Move n numbers starting at addr1 to memory starting at addr2, if n>0.
CMOVE	(addr1 addr2 n →)	Move n bytes starting at addr1 to memory starting at addr2, if n>0. "c-move"
FILL	(addr n byte →)	Fill n bytes in memory with byte beginning at addr, if n>0.

CONTROL STRUCTURES

DO ... LOOP	do: (end+1 start →)	Set up loop, given index range
I	(→ index)	Place current loop index on data stack
J	(→ index)	Return index of next outer loop in same definition.
LEAVE	(→)	Terminate loop at next LOOP or +LOOP , by setting limit equal to index
DO ... +LOOP	do: (limit start →) +loop: (n →)	Like DO ... LOOP , but adds stack value (instead of always 1) to index. Loop terminates when index is greater than or equal to limit (n>0), or when index is less than limit (n<0). "plus-loop"
IF ... (true) ... THEN	if: (flag →)	If top of stack true, execute
IF ... (true) ... ELSE ... (false) ... THEN	if: (flag →)	Same, but if false, execute ELSE clause.
BEGIN ... UNTIL	until: (flag →)	Loop back to BEGIN until true at UNTIL
BEGIN ... WHILE ... REPEAT	while: (flag →)	Loop while true at WHILE , REPEAT loops unconditionally to BEGIN. When false, continue after REPEAT
EXIT	(→)	Terminate execution of colon definition. (May not be used within DO ... LOOP)
EXECUTE	(addr →)	Execute dictionary entry at compilation address on stack (*e.g.*, address returned by FIND).

Operand key:

n, n1, ...	16-bit signed numbers	d, d1,	32-bit signed numbers	addr, addr1,	addresses	char	7-bit ascii character value
		u	unsigned	byte	8-bit byte	flag	boolean flag

TERMINAL INPUT-OUTPUT

CR	(—)	Do a carriage return and line feed "c-r"
EMIT	(char —)	Type ascii value from stack
SPACE	(—)	Type one space
SPACES	(n —)	Type n spaces, if n>0
TYPE	(addr n —)	Type string of n characters beginning at addr if n>0
COUNT	(addr — addr+1 n)	Change address of string (prefixed by length byte at addr) to TYPE form
—TRAILING	(addr n1 — addr n2)	Reduce character count of string at addr to omit trailing blanks "dash-trailing"
KEY	(— char)	Read key and leave ascii value on stack
EXPECT	(addr n —)	Read n characters (or until carriage return) from terminal to address with null(s) at end
QUERY	(—)	Read line of up to 80 characters from terminal to input buffer
WORD	(char — addr)	Read next word from input stream using char as delimiter, or until null. Leave addr of length byte

NUMERIC CONVERSION

BASE	(— addr)	System variable containing radix for numeric conversion
DECIMAL	(—)	Set decimal number base
.	(n —)	Print number with one trailing blank and sign if negative "dot"
U.	(un —)	Print top of stack as unsigned number with one trailing blank "u-dot"
CONVERT	(d1 addr1 — d2 addr2)	Convert string at addr1+1 to double number. Add to d1 leaving sum d2 and addr2 of first non-digit
<#	(—)	Start numeric output string conversion "less-sharp"
#	(ud1 — ud2)	Convert next digit of unsigned double number and add character to output string "sharp"
#S	(ud — 0 0)	Convert all significant digits of unsigned double number to output string "sharp-s"
HOLD	(char —)	Add ascii char to output string
SIGN	(n —)	Add minus sign to output string if n<0
#>	(d — addr n)	Drop d and terminate numeric output string, leaving addr and count for TYPE "sharp-greater"

MASS STORAGE INPUT/OUTPUT

LIST	(n —)	List screen n and set SCR to contain n
LOAD	(n —)	Interpret screen n, then resume interpretation of the current input stream
SCR	(— addr)	System variable containing screen number most recently listed
BLOCK	(n — addr)	Leave memory address of block, reading from mass storage if necessary
UPDATE	(—)	Mark last block referenced as modified
BUFFER	(n — addr)	Leave addr of a free buffer, assigned to block n, write previous contents to mass storage if UPDATEd
SAVE-BUFFERS	(—)	Write all UPDATEd blocks to mass storage
EMPTY-BUFFERS	(—)	Mark all block buffers as empty, without writing UPDATEd blocks to mass storage

DEFINING WORDS

: xxx	(—)	Begin colon definition of xxx "colon"
;	(—)	End colon definition "semi-colon"
VARIABLE xxx	(—)	Create a two-byte variable named xxx, returns address when executed
	xxx: (— addr)	
CONSTANT xxx	(n —)	Create a constant named xxx with value n, returns value when executed
	xxx: (— n)	
VOCABULARY xxx	(—)	Create a vocabulary named xxx, becomes CONTEXT vocabulary when executed
CREATE . . . DOES>	does: (— addr)	Used to create a new defining word, with execution-time routine in high-level FORTH "does"

VOCABULARIES

CONTEXT	(— addr)	System variable pointing to vocabulary where word names are searched for
CURRENT	(— addr)	System variable pointing to vocabulary where new definitions are put
FORTH	(—)	Main vocabulary, contained in all other vocabularies. Execution of FORTH sets context vocabulary
DEFINITIONS	(—)	Sets CURRENT vocabulary to CONTEXT
' xxx	(— addr)	Find address of xxx in dictionary; if used in definition, compile address "tick"
FIND	(— addr)	Leave compilation address of next word in input stream. If not found in CONTEXT or FORTH, leave 0
FORGET xxx	(—)	Forget all definitions back to and including xxx, which must be in CURRENT or FORTH

COMPILER

,	(n —)	Compile a number into the dictionary "comma"
ALLOT	(n —)	Add two bytes to the parameter field of the most recently-defined word
."	(—)	Print message (terminated by "). If used in definition, print when executed "dot-quote"
IMMEDIATE	(—)	Mark last-defined word to be executed when encountered in a definition, rather than compiled
LITERAL	(n —)	If compiling, save n in dictionary, to be returned to stack when definition is executed
STATE	(— addr)	System variable whose value is non-zero when compilation is occurring
[	(—)	Stop compiling input text and begin executing "left-bracket"
]	(—)	Stop executing input text and begin compiling "right-bracket"
COMPILE	(—)	Compile the address of the next non-IMMEDIATE word into the dictionary
[COMPILE]	(—)	Compile the following word, even if IMMEDIATE "bracket-compile"

MISCELLANEOUS

(	(—)	Begin comment, terminated by) on same line or screen; space after ("paren", "close-paren"
HERE	(— addr)	Leave address of next available dictionary location
PAD	(— addr)	Leave address of a scratch area of at least 64 bytes
>IN	(— addr)	System variable containing character offset into input buffer, used e.g. by WORD "to-in"
BLK	(— addr)	System variable containing block number currently being interpreted, or 0 if from terminal "b-l-k"
ABORT	(—)	Clear data and return stacks, set execution mode, return control to terminal
QUIT	(—)	Like ABORT, except does not clear data stack or print any message
79-STANDARD	(—)	Verify that system conforms to FORTH-79 Standard

FORTH INTEREST GROUP, P.O. Box 1105, San Carlos, CA 94070, USA