COIN-OF FORTH

and

SWARTHMORE EXTENSIONS

Brock A. Miller
David E. McIntyre
Eugene A. Klotz

May 31, 1983

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

The programming tools presented here are a byproduct of a grant from the National Science Foundation - National Institute of Education Program on Mathematics Education and Information Technology, which resulted in the Swarthmore Trigonometry Project.

Steve Calfee of Atari, Inc. very kindly made available to us his fine version of the fig-FORTH programming language, which constitutes the core of this package. Mike Albaugh of Atari, Inc. was most helpful in answering our many questions, since only very rudimentary documentation existed.

In creating the Coin-Op FORTH Glossary, we borrowed liberally from the fig-FORTH Glossary by William F. Ragsdale, provided through the courtesy of the FORTH Interest Group, P.O. Box 1105, San Carlos, CA 94070. We also borrowed from All About FORTH by Dr. Glen B. Haydon, Mountain View Press, Mountain View, CA 19040.

The Assembler Glossary is based on that of William F. Ragsdale, which appeared in the September, 1981 issue of Dr. Dobb's Journal, and we gratefully acknowledge his permission to make use of this.

The following persons served as programmers for the Swarthmore Trigonometry Project at one time or another, and made a contribution to our work:

Guy Blelloch, Charles Clinton, David Fristrom, Chris Helck, Gene Klotz, Ben Lewis, David McIntyre, Brock Miller, Deborah Reichert, Jay Scott, David Szent-Gyorgyi, and Fred Thomsen.

The writing and assembly of the enclosed material was done by

David Fristrom, Gene Klotz, Ben Lewis, David McIntyre, Brock Miller, Tom O'Brien, and Fred Thomsen.

Brock Miller was in charge of the writing and assembly and David McIntyre was his straw boss.

The typing was ably handled amidst great chaos by Pamela Moench, Nancy Burton, and Bruce Mallory.

This project raced for its existence with the first issue of the Schmalzried-Ryan union. We won, and affectionately dedicate this to her or him.

Gene Klotz
Swarthmore College
May 28, 1983

Bad FORTH benedictions:

Don't forget to FLUSH
May the FORTH be with you.

I.1 WHAT THIS IS

This package consists of:
(a) the Coin-Op version of the FORTH programming language residing on the disk marked Atari Coin-Op FORTH, version 1.4v.
(b) the source code for a number of programs in FORTH, to be found on the disks marked "Swarthmore 1", "Swarthmore 2", and "Swarthmore QE".
(c) Discussion and glossaries for all the above.

All disks except QE boot to unadorned Coin-Op FORTH. QE boots to an embellished version described in the next section.

Coin-Op FORTH was developed by Steve Calfee at Atari, Inc. and the disk enclosed is, except for a very few minor changes, exactly what he has informally distributed. This is a well-wrought, tight version of FORTH. Among its notable features are an excellent compile time. The 32 line screens allow for writing clear code (see Section V: How to Write Good FORTH).

The Swarthmore disks contain programming tools which were developed over a two-year period by the Swarthmore Trigonometry Project. These range from simple aids for the beginning programmer to more sophisticated program development tools. We hope that the documentation will be of use to the corresponding range of programmers. In almost all cases, we provide a discussion of each FORTH word, along with the source code (except for much of the booted material on the Coin-Op disk).

What this isn't

This is not a complete FORTH tutorial - we recommend its use in conjunction with a standard FORTH text, such as Starting Forth.

This is not a complete Atari tutorial - see the references in the next section.

This is not a version of FORTH for all users. In particular, it lacks a floating-point arithmetic package (because floating-point arithmetic is much slower - see the discussion in Chapter 5 of Starting FORTH). In most instances, this should not be a disadvantage.

The code is probably not error free - although we have tested and used everything and know of no errors. (Until it is used for some time by a wide variety of users, most code is not error free).

The documentaton will not be perfect for everyone. At least we've worked hard to produce what we would have liked to receive ourselves.

I.2 HOW TO USE THIS MATERIAL

Remove any cartridges from your Atari 400 or 800, turn on the disk drive, insert the Atari Coin-Op disk, and turn on the computer. Unless your disk drive's speed is badly out of adjustment or magnetic disaster has struck the disk, after a short time you should get the message "fig-FORTH 1.4v." You are now in FORTH. Feel free to play. The worst you can do (if your disk is write protected) is to crash the system, in which case try the Reset button, or for a high quality crash, reboot.

Such discovery learning will not get you very far very fast or very thoroughly, and we recommend you work along with a good text. Two sections hence, we chronicle the differences between Coin-Op FORTH and the FORTH referred to in the excellent text, Starting FORTH. You'll find our Glossary of the Coin-Op Kernel, Section II.2, a handy companion.

One of the first things you'll want to do is make a backup copy. If you have more than one disk drive, see section III.2 on Disk Copy Routines, and proceed accordngly. Things are tougher if you only have one drive. See Section III.1 WTOBJ to get a copy of the booted part of any disk, but transferring source screens is lengthy. To transfer two individual screens from a source disk to a destination disk, stick in the source disk and type the incantation:

```
1st.screen.number BLOCK DROP UPDATE [RETURN]
2nd.screen.number BLOCK DROP UPDATE [RETURN]
```

Then insert the destination disk and type FLUSH (and hit [RETURN]). Unfortunately, this won't work for more than two screens at a time. To see the screens before transferring them, replace BLOCK by LIST in the above.

There are two changes in your programming environment you might want to make at first. Type

```
1 WARNING !     [RETURN]
```

to receive error messages rather than error numbers. (The error messages are on screens 2 through 5). Coin-Op FORTH comes in hexadecimal. Typing DECIMAL will get you into what is likely to be a more familiar environment. However, hitting System Reset or rebooting will return you to hex. To create a disk which will boot to decimal (and stay there with System Reset), type

```
' DECIMAL CFA ' ABORT 2+ !    [RETURN]
```

Then do a WTOBJ (see Section III.1) so that these changes will be made part of the boot.

You might want to consider some further additions to your booted FORTH system. The followng seem particularly useful: Fast INDEX, $ (which causes the next number to be interpreted as hexadecimal), and the QE editor (which is large, but worth it, we think). All are in Section IV. After such additions are compiled, it is necessary to do a WTOBJ (Section III.1) to make them part of the boot.

The Swarthmore QE disk boots with all these treats but without 1 WARNING !. However, the object code extends through screen 8, so you'll get crazy error messages with 1 WARNING ! unless you insert a disk which has them on screens 2-6.

In using the documentation, we expect you to have certain references available. For the Atari, we assume you have a 400/800 Basic Referrce Manual, which comes with the computer. For deeper hardware questions, we sometimes refer to De Re Atari, by Chris Crawford, et al, Atari, Inc., 1981,APX-90008. For matters pertaining to FORTH, we refer you to Starting FORTH, by Leo Brodie, Prentice-Hall, 1981.

Sometime, please read Section V: How to Write Good FORTH.

I.3 FURTHER REFERENCES

For the Atari:

Above and beyond De Re Atari, you might wish to consult the Atari 400/800 Hardware Manual or the Atari 400/800 Operating System User's Manual, available from Atari,Inc.

For FORTH:

The following have proved useful:

Glenn B. Haydon, All About FORTH, Mountain View Press, 1982.
(Good glossary of words from several different FORTHs.)

Mitch Derick & Linda Baker, FORTH Encyclopedia, Mountain View Press, 1982.
(More detailed than the above.)

C.H. Ting, Systems Guide to Fig FORTH, Offete Enterprises, Inc., 1980.
(Good for how FORTH works.)

William F. Ragsdale, fig-FORTH Installation Manual, Forth Interest Group, 1980.

All the above are available from Mountain View Press, P.O. Box 4656, Mountain View, Ca., 94040, (415) 961-4103.

Serious FORTH users should consider joining the FORTH Interest Group. Their journal, FORTH Dimensions, makes interesting reading. FORTH Interest Group, P.O. Box 1105, San Carlos, Ca. 94070. (415) 962-8653.

I.4 COIN-OP FORTH VS. STARTING FORTH

FORTHs based on the fig (FORTH Interest Group) model, such as the Coin-Op FORTH kernel, are different in some significant ways from the version of FORTH described in Starting FORTH (which is nonetheless an excellent introductory text for the beginning Coin-Op FORTH user). Following is a list, referenced by Starting FORTH page number, of words in Starting FORTH which either have different names or are not used at all in Coin-Op FORTH. There are also a few notes included on more general differences between the two. This list of differences is fairly complete, but it should not be regarded as comprehensive. In particular, there are a number of words included in Coin-Op FORTH's kernel which are not in Starting FORTH, and which are not mentioned in the following list. These range from words used to manipulate headers (NFA, PFA, etc.) to disk buffer words (BLOCK, BUFFER) to words for listing the dictionary (VLIST). After you are familiar with Starting FORTH, browse through the Coin-Op FORTH glossary to become familiar with these words.

pg. 19 - Coin-Op FORTH allows up to 31 characters of a name to be stored in the dictionary.

pp. 57-58 - The fig-FORTH editor is very different from that described in Starting FORTH; the Starting FORTH discussion of FORTH's screen-based system will nonetheless be quite useful to the Coin-Op FORTH user. The following words are basically the same in Starting FORTH and Coin-Op FORTH: LIST, LOAD, (, FLUSH, COPY, FORGET.

pg. 91 - The fig-FORTH word 0= takes the place of NOT in Coin-Op FORTH.

pg. 101 - The fig-FORTH word -DUP takes the place of ?DUP in Coin-Op FORTH.

pg. 101 - ABORT" does not exist in Coin-Op FORTH.

pg. 107 - The words 1- and 2* do not exist in fig-FORTH, but they are included as part of the Coin-Op FORTH boot. The words 2- and 2/ are included in neither fig-FORTH nor Coin-Op FORTH.

pg. 108 - In Coin-Op FORTH, NEGATE is called MINUS.

pg. 110 - The word I' does not exist in Coin-Op FORTH.

pg. 132 - U.R does not exist in Coin-Op FORTH.

pg. 139 - In addition to BEGIN ... UNTIL and BEGIN ... WHILE ... REPEAT, Coin-Op FORTH also supports BEGIN ... AGAIN, where AGAIN executes an unconditional branch back to BEGIN.

pg. 142 - PAGE does not exist in Coin-Op FORTH. Clearing the screen is performed by the Coin-Op FORTH word CLEAR.

pg. 161 - The words U/MOD, U., and /LOOP do not exist in Coin-Op FORTH. Coin-Op FORTH does include the word U/ (unsigned divided by), which is not included in Starting FORTH.

pg. 162 - The word OCTAL does not exist in Coin-Op FORTH. Octal can be set by the command 8 BASE ! .

pg. 164 - The only punctuation mark which may be used during input to indicate a double number is . (period).

pg. 173 - The double number operators D-, DMAX, DMIN, D=, D0=, D<, and DU< do not exist in Coin-Op FORTH. The word DNEGATE has been replaced by the fig-FORTH word DMINUS. Coin-Op FORTH includes the word D. which is not mentioned in Starting FORTH.

pg. 174 - M+ and M*/ do not exist in Coin-Op FORTH. Coin-Op FORTH does include two mixed length operators, M/MOD and D8U*, which are not included in Starting FORTH.

pg. 183 - VARIABLE takes an argument on the stack in Coin-Op FORTH, and stores this argument in the newly defined variable's parameter field.

pg. 193 - Double length variables and constants do not exist in Coin-Op FORTH; thus, 2VARIABLE, 2CONSTANT, 2!, and 2@ are not defined.

pg. 204 - DUMP does not exist in Coin-Op FORTH.

pg. 217 - EXECUTE requires the CFA, not the PFA, of a word. Using the PFA will crash the system.

pg. 218 - ['] is not defined in Coin-Op FORTH, since ' is defined as being immediate. Thus ' can be used in place of ['] in a colon definition.

pp. 220-224 - The structure of Coin-Op FORTH headers is significantly different from that described in Starting FORTH. See the discussion of Coin-Op FORTH headers, linking and vocabularies. EXECUTE expects the CFA, not the PFA, of a word.

pg. 225 - The word EXIT is called ;S in Coin-Op FORTH.

pg. 233 - H is called DP in Coin-Op FORTH.

pg. 235 - 'S is called SP@ in Coin-Op FORTH.

pg. 236 - S0 is not directly available to the user in Coin-Op FORTH. See the glossary entry for S0.

pp. 242-243 - The Starting FORTH discussion of vocabularies applies quite well to Coin-Op FORTH, as far as it extends (the one exception is that ; sets CONTEXT to CURRENT, not to FORTH). The Coin-Op FORTH linking system is significantly different from other FORTHs, however, and this effects vocabularies. See discussion of linking and vocabularies in Section II.1.

pg. 256 - FLUSH does not "forget" that a buffer is in memory in Coin-Op FORTH; it only turns off the "updated" flag.

pg. 261 - >TYPE does not exist in Coin-Op FORTH.

pg. 266 - MOVE and <CMOVE are not defined in Coin-Op FORTH.

pg. 270 - WORD does not leave the address of HERE on the stack. TEXT does not take a delimiter on the stack; instead, it reads all characters up to the next carriage return.

pg. 272 - >IN is called IN in Coin-Op FORTH.

pg. 277 - >BINARY (CONVERT) does not exist in Coin-Op FORTH.

pg. 281 - -TEXT is not defined in Coin-Op FORTH.

pg. 291 - The word CREATE is called <BUILDS in Coin-Op FORTH. The Coin-Op FORTH word CREATE only

creates a header for new dictionary entries.

pg. 301 - Coin-Op FORTH uses (DO), not 2>R.

pg. 303 - (LITERAL) is called LIT in Coin-Op FORTH.

I.5 DISK DIRECTORY

Coin-Op FORTH Disk:

Screens 2-5 Error Messages
9 Screen originally used to compile assembler
11 Disk copy routines (III.2)
13-19 The Assembler (II.5)
20 Undocumented
21-25 The fig Editor (II.4)
26 OS and Hardware constants (in Glossary, II.2)
27-29 Undocumented
30 DECUS FORTH additions (in Glossary, II.2)
31 Display list disassembler, undocumented
32-33 FORMAT and WTOBJ (III.1)
35-36 Undocumented
37-38 Line Printer words (III.1)
39-43 Formatted printer words (III.4)
44-46 Undocumented

Swarthmore Disk 1:

Screen 1 Directory
2-5 Error messages
7-17 IV.1 Miscellaneous
20 IV.2 Fast INDEX
22-24 IV.3 CASE
26-27 IV.4 Display List assembler
29 IV.5 Cheap character generator
31-34 IV.6 String search
37-48 IV.7 Print commands for graphics modes
50-51 IV.8 Screen copy routines
53-55 IV.9 Vertical scrolling

Swarthmore Disk 2:

Screen 1 Directory
2-5 Error messages
10-13 IV.10 Unshadowed Player Missile words
15-20 IV.11 Shadowed Player Missile words
22-31 IV.12 Text words for disk storage of text
33-49 IV.13 Text compression text words
52-53 IV.14 VBLANK routines
55-61 IV.15 Big WTOBJ

Swarthmore QE Disk:

Screens 0-8 Boot source code continuation
14-77 Source code for QE (IV.17)

NOTE: The QE disk boot contains QE (and therefore the Miscellaneous words, CASE, Display List assembler, and unshadowed Player Missiles); it also has Fast Index. It boots to decimal. All other disks boot to Coin-Op FORTH.

II. THE COIN-OP BOOT

INTRODUCTION

The documentation of the Coin-Op FORTH boot is divided into four parts, which correspond to the four sections of the boot itself. These four sections include:

1) a modified fig-FORTH kernel
2) Atari graphics, sound, and device control words and a few other miscellaneous words
3) a fig-FORTH editor
4) a fig-FORTH assembler.

The section on the Coin-Op FORTH kernel consists of a list of the modifications which have been made to the fig-FORTH model and brief discussions of the most important departure from that model (the vocabulary/linking system) and of the Atari-specific disk access words; these are followed by a complete glossary of the kernel. The discussion of each extension includes a brief discussion of how to use the important words in that extension, followed by a complete glossary of the extension.

II.1 THE FORTH KERNEL

COIN-OP FORTH VS. FIG-FORTH

The Coin-Op FORTH kernel (i.e., the 250 or so words which constitute the heart of FORTH) is based on fig-FORTH. However, a number of modifications have been made. These are listed below, for the convenience of those who are already familiar with fig-FORTH. (The sections on disk access and disk buffers, headers, links, and vocabularies will also be of use to those who are not familiar with fig-FORTH.)

The following fig-FORTH words are not included in Coin-Op FORTH: +BUF , BLOCK-READ , BLOCK-WRITE , DLIST (this word was used to list the vocabulary; it had nothing to do with graphics display lists), DUMP , MON , MOVE , USE .

The following words are included in the Coin-Op FORTH kernel but are not part of the fig-FORTH model (for explanations of their usage, see the Coin-Op Kernel Glossary; in addition, some of these words will be discussed in more detail in the sections on disk access and disk buffers, headers, links, and vocabularies): #LINKS , 7BITS , ALT , BFND , DBUF , DDIO , DECMAP , DIO , FLUSH , HASH , HIMEM , ICAL , ICCM , ICLL , J , PHYSOFF , PTAB , SECIO , SIO , TCIOV , UP .

Headers, Links, and Vocabularies

Coin-Op FORTH executes dictionary searches much, much faster than standard fig-FORTH; this in turn means that much less time is spent by the programmer in compiling FORTH words and screens. The vastly increased speed was obtained by modifications to the header structure of dictionary entries and by the introduction of "sub-vocabularies". Most users will not need to understand these differences in order to use FORTH to full advantage. Other users will require such an understanding, however, and still others will simply be interested in what makes FORTH tick. The following discussion should be helpful.

Headers

Normal fig-FORTH headers have the following structure:

NFA
a name field, starting at NFA, which has a variable number of bytes (up to 31).
LFA
following the name field, a two byte link field, which contains the NFA of the first word in the same vocabulary which immediately precedes this one.
CFA
following the link field, a two byte code field which points to the word's execution code.
PFA
following the code field, a variable length parameter field.

In Coin-Op FORTH, the link field comes before the name field, with the code field coming immediately after the name field. The structure is thus:

LFA NFA CFA PFA

This improves the speed of dictionary searches, since the LFA of a word will have a fixed offset of -2 from the NFA rather than a variable offset.

NOTE: in fig-FORTH, the last character in a name field has its high bit set ($80). Because only this character can have this bit set, names cannot have ATASCII inverse characters in them. To make sure that this is the case, all words in the input stream are operated on by the word 7BITS, which changes any inverse characters in a word to normal characters by setting the high bit to 0. Thus, to FORTH inverse and normal ATASCII characters in a name are identical.

Linking and "Sub-vocabularies"

Normal fig-FORTH has all words in a given vocabulary linked together; ths most recently defined word links directly (through its LFA) to the NFA of the word defined before that, which links to the word before that, and so on, down to the very first word in the vocabulary. (See Starting FORTH, pp.242-44, for a good discussion of vocabularies and linking.) Coin-Op FORTH has modified this system through the creation of "sub-vocabularies" within vocabularies.

During compilation, each word in a vocabulary is placed in one of 8 sub-vocabularies. The number of sub-vocabularies is determined by the constant #LINKS, which is set to 8.) These sub-vocabularies may be numbered from 0 to 7. Which sub-vocabulary a word belongs in is determined by the ATASCII value of the first character of that word's name. That value is divided by 8, and the remainder (0-7) determines which sub-vocabulary the word will belong to. Thus, a word beginning with H (ATASCII value $48) will be placed in sub-vocabulary 0, as will a word beginning with X ($58), 8 ($38), or @ ($40). A word beginning with L ($4C), on the other hand, will be placed in sub-vocabulary 4, while a word beginning with W ($57) will be in sub-vocabulary 7, and so forth. Thus, the words in each vocabulary have been "hashed", according to their ATASCII value, into 8 groups.

The words in each sub-vocabulary are linked together in the normal manner; the LFA of the most recently defined word in a sub-vocabulary points to the NFA of the word in the same sub-vocabulary defined just before that, and so forth down to the first word defined in that sub-vocabulary. Words in one sub-vocabulary thus are linked only to each other, and not to those in any other sub-vocabulary.

Now, look at what happens when a dictionary search occurs. As an example, assume that the word DUP has just been encountered in the input stream. The search routine takes the first letter (D) of

this word, and from it determines which sub-vocabulary the word must be in, if its exists at all. (This task of "hashing" is performed, logically enough, by the word HASH.) In the case of DUP, sub-vocabulary #4 is duly searched, and DUP is found. The point is that the other 7 sub-vocabularies need not be searched, and thus on average only 1/8 as many words will need to be looked at in any given dictionary search in Coin-Op FORTH, as opposed to in a FORTH which uses only one "link-chain". This is the main reason why Coin-Op FORTH compiles code so much faster than fig-FORTH.

Vocabularies

A knowledge of the structure and purpose of FORTH vocabulary definitions is necessary in order to understand the following discussion. All About FORTH by Glen B. Haydon and FORTH Encyclopedia by Derick and Baker contain good discussions of vocabularies. It should be noted that new vocabularies need not chain directly to the FORTH vocabulary; any vocabulary may serve as a "parent vocabulary". This in turn allows an ever-expanding branching structure of vocabularies within the dictionary. Eventually, of course, all vocabularies will chain at least indirectly to FORTH.

The changes to the header structure and the addition of sub-vocabularies have necessitated two changes to the structure of vocabulary definitions in Coin-Op FORTH.

1. Instead of one vocabulary entry pointer (or, if you prefer, "Vocabulary link field") in each vocabulary definition, there are now eight, one for each sub-vocabulary. Since each entry pointer needs 4 bytes (two for the pseudo-name field, two for the link), a total of 32 bytes is required. Sub-vocabulary #0 uses bytes 1 to 4 as its entry pointer, sub-vocabulary #2 uses bytes 5 to 8, and so forth. In this way, sub-vocabulary #n of a child vocabulary chains to sub-vocabulary #n of its parent.

2. The pseudo-NFA of each entry pointer now follows the LFA, rather than vice versa; this is due to the change in header structure decribed in the above section on Headers.

VLIST

VLIST has been modified so that it will still print out the CONTEXT vocabulary words in the order in which they are placed in the dictionary. However, in vocabularies to which the CONTEXT vocabulary chains (unless the CONTEXT vocabulary is FORTH, of course) all of the words in sub-vocabulary #7 will be listed in order first, followed by those in sub-vocabulary 6, 5, and so on. (This is because VLIST sets up a table of the NFAs of the next word in each sub-vocabulary; the word in this table with the highest NFA is printed by VLIST, and its NFA is replaced with that of the word to which it links; this process then repeats for the next word, and so on to the bottom of the dictionary. This works fine, until all of the NFAs in the table are the "pseudo NFAs" in the parent vocabulary; at this point, sub-vocabulary #7's NFA is the highest address in the table, as are the NFAs of all of the subsequent words in the parent vocabulary's sub-vocabulary #7. Thus, all words in sub-vocabulary #7 will be listed until it is exhausted, after which all words in sub-vocabulary #6 will be listed, etc.) This will continue until the word FORTH is itself encountered, after which words will again be listed in order of their physical location in the dictionary as a whole.

Disk Access and Disk Buffers

Many users will never need to worry about Coin-Op FORTH's primitive disk access and disk buffer words; the higher level words LIST, LOAD, BLOCK, and BUFFER provide a great deal of flexibility and Coin-Op FORTH does use the standard fig-FORTH word R/W. Below the level of these words, however,

Coin-Op FORTH's disk words are tailored to the Atari and as such they differ from normal fig-FORTH usage. The following discussion is intended for those who either need to know or are interested in knowing what these differences are. The one section which may be of more general interest is that on the numbering of disk blocks.

Disk Access

Coin-Op FORTH dos not use the standard fig-FORTH disk access words, BLOCK-READ and BLOCK-WRITE. Instead, Coin-Op FORTH utilizes words which directly call the Atari SIO routines. These words are SECIO and SIO, which in turn are called by DIO (for single density disk drives) and DDIO (for double density disk drives); basically, the procedure for disk access is to set up a device control block (see the Atari OS manual and the words ICAL, ICCM, ICLL) and execute SECIO.

Screen Numbering

1. Coin-Op FORTH allows for a difference between physical and logical screen numbering. This is accomplished by storing a value into the user variable PHYSOFF (normally set to 12). This value is added by the disk access words to whatever logical block number is supplied by the user in order to obtain the physical block number. This allows the user to put the bootable object on a disk onto "negative" logical screens. Note that it is very difficult to use logical screen 0 on a disk to hold source code (see INTERPRET).

2. The logical screen supplied by the user is interpreted by Coin-Op FORTH as follows:
   Each block # is 16 bits, of which the highest bit is not used. The low 11 bits ($0 - $7FF) indicate the screen number. The next two bits (12 and 13) indicate the drive #, and the two bits above those (14 and 15) indicate the drive type. Consult the following chart to see what values to add to a given logical block number.

Add this value:	To access drive #:
$0000	1
$0800	2
$1000	3
$1800	4

Add this value:	To access drive type:
$0000	810-type 5 1/4" single density
$2000	815 or 8" double density
$4000	8" single density DEC interleave
$6000	8" single density non-interleaved

Disk Buffer Control

Coin-Op FORTH uses a table called PTAB to control disk buffers. This table contains four "entries", each corresponding to a disk block. The first entry contains information on the most recently accessed block, the second contains information on the next most recent block, and so forth. At the moment, only two disk buffers are used by Coin-Op FORTH, and as such only the first two entries of PTAB are used. The two buffers each consist of $400 (1K) bytes, and they begin at $500 and $900, respectively.

Each entry in the disk buffer table consists of 5 bytes. The first two bytes contain the

logical block number. The second two bytes contain the starting address of the buffer in which the block is contained. The last byte indicates the status of the buffer (0 = not in use, 1 = in use, 81 = updated).

When a new block is accessed it is placed in the buffer which was least recently used. The block number, buffer address, and status of this block are placed in the first entry in the disk buffer table, while the old contents of the first entry are placed in the second entry. (If a third buffer was in use, the contents of the third entry would indicate the least recently accessed block, and the contents of the second entry would be placed in the third entry when a new block was accessed; likewise with a fourth buffer.) If the block in the least recently accessed buffer is marked as updated, its contents are written to disk before a new block is read into that buffer.

The contents of the variable PREV point to the first byte of the most recently accessed buffer entry (entry $1) in PTAB. The contents of the variable ALT (which roughly corresponds to the fig-FORTH word USE) point to the entry for the least recently accessed buffer (entry $2). FORTH and the user can thus access the disk buffer information via these words. (NOTE: a third disk buffer could be added by pointing ALT to the third entry in PTAB (PTAB +$A) and placing a buffer address into the third and fourth bytes of this entry. The same technique could be used for a fourth disk buffer. DO NOT put the new buffer or buffers right above the old ones; FORTH starts at $D00. Put them somewhere above the dictionary.

II.2 GLOSSARY OF THE COIN_OP KERNEL

! n addr —

Store 16 bit value n at address. Pronounced "store".

!CSP —

Save the stack position in CSP. Used as part of the compiler security.

d1 — d2

Generate, from a double number d1, the next ATASCII character which is placed in an output string. Result d2 is the quotient after division by BASE and is maintained for further processing. Used between <# and #>. See #S.

#> d — addr count

Terminates numeric output conversion by dropping d, leaving the text address and character count suitable for TYPE.

#LINKS — n

Constant leaving the number of link chains in each vocabulary. For a full explanation of the linking system (which is different from that of normal fig-FORTH), see the discussion of linking which precedes this glossary.

#S d1 — d2

Convert all digits of double-number d1 to their ATASCII equivalents, adding each to the pictured numeric output text, until the remainder is 0 (this remainder is left as d2). A single 0 is added to the output string if the initial value of d1 is 0. Used between <# and #>.

' — addr

Used in the form:

' nnnn

If executing, leaves the PFA of the next word accepted from the input stream. If compiling, compiles the PFA as a literal; later execution will place this value on the stack. If the word is not found after a search of CONTEXT and CURRENT, an appropriate error message is given. This is defined as an immediate word. Pronounced "tick".

(—

Used in the form:

(cccc)

Accept and ignore comment characters from the input stream, until the next right parenthesis. Since it is a FORTH word, (must be followed by at least one blank. It may be used freely while executing or compiling. Comments may not be longer than 255 characters, and an error condition exists if the input stream is exhausted before the right parenthesis.

(+LOOP) n —

The run-time procedure compiled by +LOOP, which increments the loop index by n and tests for loop completion. See +LOOP.

(.") —

The run-time procedure, compiled by ." which transmits the following in-line text to the selected output device. See ."

(;CODE) —

The run-time procedure, compiled by ;CODE, that rewrites the code field of the most recently defined word to point to the machine code sequence which follows ;CODE. See ;CODE.

(ABORT) —

Executes after an error when WARNING is -1; also executes after a RESET or BREAK. This word normally executes ABORT, but may be altered (with care) to execute a user's alternative procedure.

(DO) —

The run-time procedure compiled by DO which moves the loop control parameters to the return stack. See DO.

(FIND) addr1 addr2 — pfa b tf (found)
addr1 addr2 —ff (not found)

Searches the dictionary starting at the NFA addr2, matching to the text at addr1. Returns parameter field address, length byte of the name field and boolean true if a match is found. If no match is found, only a boolean false is left.

(LINE) n1 n2 — addr 32

Converts the line number n1 and the screen n2 to the starting address of that line in the disk buffer. If screen n2 is not already in memory, it is read from disk to buffer. The count of 32 indicates the length of a text line.

(LOOP) —

The run-time procedure compiled by LOOP. Increments the loop index and tests for loop completion. See LOOP.

(NUMBER) d1 addr1 — d2 addr2

Convert the ATASCII 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 unconvertable digit. Used by NUMBER.

+ n1 n2 — n3

Leave the arithmetic sum n3 of n1+n2.

+! n addr —

Add n to the 16-bit value at the address. Pronounced "plus-store".

+- n1 n2 -- n3
Negate n1 if n2 is negative.

+LOOP n1 -- (run-time)
addr n2 -- (compil-time)
Used in a colon-definition in the form:
DO ... n1 +LOOP
At run-time, +LOOP selectively controls branching back to the corresponding DO based on n1, the loop index and the loop limit. The signed increment n1 is added to the index and the total is compared to the limit. The branch back to DO occurs until the new index is equal to or greater than the limit (n1>0), or until the new index is equal to or less than the limit (n1<0). Upon exiting the loop, the parameters are discarded and execution continues beyond LOOP.

At compile time, +LOOP compiles the run-time word (+LOOP) and the branch offset computed from HERE to the address left on the stack by DO. n2 is used for compile time error checking.

+ORIGIN n -- addr
Leave the memory address which is n bytes above the "origin". n is given in terms of bytes. This definition is used to access or modify the boot-up parameters at the origin area. The "origin" in Coin-Op FORTH is at $D00.

, n --
Allots 2 bytes in the dictionary and stores n there. Pronounced: "comma".

- n1 n2 -- n3
Subtract n2 from n1 and leave the difference as n3.

--> --
Used on a disk screen. Causes the next disk screen to be loaded. Interpretation continues on this screen. (Pronounced "next-screen").

-DUP n -- n (zero)
n -- n n (non-zero)
Reproduce n only if it is non-zero. This is usually used to copy a value just before IF, to eliminate the need for an ELSE part to drop it.

-FIND -- pfa b tf (found)
-- ff (not found)
Accepts the next text word (delimited by blanks) in the input stream to HERE, and searches the CONTEXT and then CURRENT vocabularies for matching entry. (Searches continue on into parent vocabularies; the CURRENT vocabulary is searched only if it is not the same as the CONTEXT vocabulary). If found, the dictionary entry's parameter field address, its length byte, and a boolean true is left. Otherwise, only a boolean false is left.

-TRAILING addr n1 -- addr n2
Adjusts the character count n1 of a text string beginning at addr to suppress the output of trailing blanks, i.e, the characters at addr+n1 to addr+n2 are blanks. Read "not trailing".

* n1 n2 -- n3
Leaves the signed arithmetic product of two signed numbers.

*/ n1 n2 n3 -- n4
Leaves the ratio n4 = n1*n2/n3 where all are signed numbers. Retention of an intermediate 31 bit product permits greater accuracy than would be available with the sequence:
n1 n2 * n3 /

*/MOD n1 n2 n3 -- n4 n5
Leaves the quotient n5 and remainder n4 of the operation n1*n2/n3. A 31 bit intermediate product is used (as for */).

. n --
Display n, converted to ATASCII according to BASE, in a fixed-field format. Prints the sign only if n is negative. A trailing blank follows. Pronounced "dot".

." --
Used in the form:
." cccc"
Accepts the following text from the input stream, delimited by " (double-quote). If executing, the text is transmitted to the output device. If compiling, compiles the text (with the count in the first byte) along with the execution procedure (.") so that later execution will transmit the text to the output device. The text may be up to 255 characters long.

.LINE n1 n2 --
Print line n1 of disk screen n2 on the terminal device. Trailing blanks are suppressed. All ATASCII control characters are printed.

.R n1 n2 --
Prints the number n1 right aligned in a field of width n2. No following blank is printed. If n2 < 1, no leading blanks are supplied.

/ n1 n2 -- n3
Divides n1 by n2 and leaves the signed quotient n3 (which is rounded toward 0).

/MOD n1 n2 -- n3 n4
Divides n1 by n2 and leaves the remainder n3 and quotient n4. The remainder n3 has the same sign as the dividend n2.

0 1 2 3 — n

These small numbers are used so often that it is attractive to define them by name in the dictionary as constants, because constants take less dictionary space and less dictionary search time than do numbers.

0< n — f

Leaves a true flag if n is less than zero (negative), otherwise leaves a false flag.

0= n — f

Leaves a true flag if n is equal to zero, otherwise leaves a false flag.

0BRANCH f —

The run-time procedure to conditionally branch. If f is false (zero), the in-line parameter which follows it is added to the interpretive pointer to branch ahead or back. Compiled by IF, UNTIL, and WHILE.

1+ n — n+1

Increments n by 1.

2+ n — n+2

Increment n by 2.

7BITS addr n —

Sets the high bit of each byte of the n bytes starting with addr to 0. This converts any characters which are in inverse representation to their normal representations; FORTH treats inverse characters which occur in word names as if they were normal characters.

: —

Used in the form:

: cccc ... ;

Creates a dictionary header for cccc in the CURRENT vocabulary, sets CONTEXT equal to CURRENT, and sets compile mode. Also, sets the CFA of cccc to point to run-time code which follows : in the dictionary. The compilaton addresses, CFA, of subsequent words in the input stream which are not immediate will be compiled into the dictonary by INTERPRET, to be executed when cccc is later executed. (Immediate words, i.e. those with their precedence bit set, will be executed when they are encountered.) Compilation of CFAs will continue until ; is encountered.

If a word which is part of a colon definition is not found after a search of both the CONTEXT and CURRENT vocabularies, an error conditon exists.

; —

Terminates a colon-definition and stops further compilaton. Compiles the run-time routine ;S. If compiling from mass storage and the input stream is exhausted before ; is encountered, the machine may crash.

;CODE —

Used in the form:

: ccc .... ;CODE assembly mnemonics

Stops compilation and terminates a new defining word cccc by compiling (;CODE). Sets the CONTEXT vocabulary to ASSEMBLER, assembling to machine code the mnemonics which follow it.

When cccc later executes in the form:

cccc nnnn

the word nnnn will be created, and its execution address will contain the address of the machine code following ;CODE in cccc. That is, when nnnn is executed, it does so by jumping to the code after nnnn.

;S —

Stops interpretation of a screen. ;S is also the run-time word compiled at the end of a colon-definition which returns execution to the calling procedure.

< n1 n2 — f

Leaves a true flag if n1 is less than n2; otherwise leaves a false flag.

<# —

Initializes pictured numeric output, which uses the words:

<# # #S SIGN #>

The conversion to ATASCII is done on a double number and produces text at PAD.

<BUILDS —

<BUILDS is used in conjunction with DOES> to created defining words. Used in the form:

: cccc <BUILDS ...
DOES> ...

When cccc is later executed in the form cccc bbbb, <BUILDS creates a dictionary entry for the new word bbbb. First, a header for bbbb is created. Then the sequence of words between <BUILDS and DOES> establishes a parameter field for bbbb. When bbbb is later executed, its PFA will be placed on the stack and the sequence of words following DOES> in cccc will be executed.

= n1 n2 — f

Leaves a true flag if n1=n2; otherwise leaves a false flag.

> n1 n2 — f

Leaves a true flag if n1 is greater than n2; otherwise leaves a false flag.

>R n —

Removes the top value from the stack and places it on the return stack. Use of >R must be balanced with R> in the same nesting level of a colon definition.

? addr —

Prints the contents of memory location addr in free-format according to the current base.

?COMP —

Issues error message if not compiling.

?CSP —
Issues error message if stack position differs from value saved in CSP.

?ERROR f n —
Issues an error message number n, if the boolean flag is true.

?EXEC —
Issues an error message if not executing.

?LOADING —
Issues an error message if not loading.

?PAIRS n1 n2 —
Issues an error message if n1 does not equal n2. The message indicates that compiled conditionals do not match.

?STACK —
Issues an error message if the stack is out of bounds.

?TERMINAL — f
Performs a test of the terminal keyboard to see if the break key has been pressed. A true flag indicates the key has been pressed. (In Coin-Op FORTH, this routine always returns a false flag; testing for BREAK occurs during TYPE and KEY.)

@ addr — n
Leaves the 16 bit contents of addr on the stack. Pronounced "fetch".

ABORT —
Clears the data and return stacks and enters execution mode. Returns control to the operators terminal, printing a message appropriate to the installation. Execution of the abort routine is vectored through ABORT to (ABORT); the actions of ABORT can thus be changed as the user desires (as long as (ABORT) is involved as the final part of the changed routine).

ABS n1 — n2
Leaves the absolute value of a number.

AGAIN —(execution)
addr n —(compiling)
Used in a colon-definitiion in the form:
BEGIN ... AGAIN
At run-time, AGAIN forces execution to jump unconditionally to the corresponding BEGIN. There is no effect on the stack. Execution cannot leave this loop (unless R> DROP is executed one level below). At compile time, AGAIN compiles BRANCH with an offset from HERE to addr. n is used for compile-time error checking.

ALLOT n --

Adds n bytes (n may be positive, negative, or zero) to the parameter field of the most recently defined word. Updates DP by n bytes. May be used to reserve or delete dictionary space.

ALT -- addr

A variable which contains the address of the entry in the disk buffer control table which corresponds to the disk buffer least recently referenced (this buffer will be used to hold the next source screen).

AND n1 n2 -- n3

Leaves the bitwise logical "and" of n1 and n2 as n3.

B/BUF -- n

This constant leaves the number of bytes per disk block buffer. This is 1024 (1K) in Coin-Op FORTH.

B/SCR -- n

This constant leaves the number of blocks per editing screen. Conventionally, an editing screen is 1024 bytes organized as 16 lines of 64 characters each. However, in Coin-Op FORTH, an editing screen is 1024 bytes organized as 32 lines of 32 characters each, due to the width of the Atari display screen.

BACK addr --

Calculate the backward branch offset from HERE to addr and compile this offset into the next available dictionary memory address.

BASE -- addr

A user variable containing the current number base used for numeric input and output conversion.

BEGIN -- addr n (compiling)

Occurs in a colon-definition in form:

```
BEGIN ... UNTIL
BEGIN ... AGAIN
BEGIN ... WHILE ... REPEAT
```

At run-time, BEGIN marks the start of a sequence that may be repetitively executed. It serves as a return point from the corresponding UNTIL, AGAIN or REPEAT. When executing UNTIL, a return to BEGIN will occur if the top of the stack is false; for AGAIN and REPEAT a return to BEGIN always occurs.

At compile time BEGIN leaves the return address for UNTIL, AGAIN, or WHILE and n for compiler error checking on the stack.

BFND n --addr (found)
n -- f (not found)
Checks to see if block n is already contained in a disk buffer which is marked as "in use". If it is, returns addr, which is the address of that buffer's entry in the disk buffer control table(PTAB). If not, returns a boolean false flag.

BL -- c
A constant that leaves the ascii value for "blank".

BLANKS addr n --
Fills an area of memory beginning at addr with an ATASCII (value = 20).

BLK -- addr
A User variable containing as the input stream the block number being interpreted. If zero, input is being taken from the terminal input buffer.

BLOCK n -- addr
Leaves the memory address of the block buffer containg block n. If the block is not already in memory and marked as being "in use", it is transferred from disk to whichever buffer was least recently written,and that disk buffer is established as the "current" buffer. If the block occupying that buffer has been marked as updated, it is rewritten to disk before block n is read into the buffer. See also BUFFER, R/W UPDATE FLUSH.

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. An in-line branch offset must follow any compiled instance of the ideogram.

BUFFER n -- addr
Obtains the next memory buffer(indicated by ALT). If the buffer is marked as updated, it is written to the disk; the buffer is then assigned to block n. The address left is the first cell within the buffer for data storage.

C! n addr --
Stores the least significant 8 bits of n at addr.

CL -- n
Constant leaving the number of characters per FORTH screen line. For the Atari, this is 32.

C, n --
Stores the lower order byte of n into the next available dictionary byte, advancing the dictionary pointer.

C@ addr -- b
Leaves on the stack the contents of the byte. (The value on the stack will be 16-bit with the high byte set to zero).

CFA pfa -- cfa
Converts the parameter field address of a definition to its code field address.

CMOVE addr1, addr2, n --
Moves n bytes beginning at address addr1 to address addr2. The contents of address addr1 is moved first, proceeding toward high memory. If n is 0, nothing is moved. n must be greater than or equal to 0 for this ideogram to function properly.

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. Cold start parameters may be altered by changing the parameters following the "origin" ($D00) in memory.

COMPILE --
When a word containing COMPILE executes, the 16-bit value following the compilation address of COMPILE is copied (compiled) into the next available position in the dictonary, i.e., COMPILE DUP will copy the CFA of DUP. COMPILE is usually used within immediate words such as IF and DOES>.

CONSTANT n --
A defining word used in the form:

n CONSTANT cccc

to create a dictonary entry for cccc, with its parameter field containing n. When cccc is later executed, n will be placed on the stack.

CONTEXT -- addr
A user variable containing a pointer to the vocabulary within which dictionary searches will begin during interpretation of the input stream.

COUNT addr -- addr+1 n
Leaves the address and the character count n of the text beginning at addr. It is presumed that the first byte at addr contains the text byte count and the actual text starts with the second byte. n ranges from 0 to 255. Typically COUNT is followed by TYPE.

CR --
Causes a carriage return and line feed to occur on selected output device.

CREATE —

A defining word used in the form:

CREATE cccc

by such words as CODE and CONSTANT to create a smudged dictionary header for a FORTH definition. The code field of the new word contains the address of the word's parameter field. The new word is created in the CURRENT vocabulary.

CSP —— addr

A user variable which can be used to store temporarily the stack pointer position, for compilation error checking.

CURRENT — addr

A user variable which specifies to which vocabulary new word definitions will be added. In dictionary searches, the CURRENT vocabulary is searched second, after the CONTEXT vocabulary.

D+ d1 d2 — d3

Leaves the double number sum d3 of two double numbers d1 and d2.

D+- d1 n — d2

Negates the double number d1 if n is negative.

D. d —

Prints the double number d, converted to ATASCII according to BASE, in a free field format, with one trailing blank. Displays the sign only if negative.

D.R d n —

Prints a signed double number d, converted to ATASCII according to BASE, right-aligned in a field n characters wide. Displays the sign only if negative. If n=0, does not right-align the value.

D8U* d1, n — d2

Multiplies an unsigned double number d1 by the unsigned 8-bit value n, leaving the product as an unsigned double number d2.

DABS d1 — d2

Leaves the absolute value of a double number.

DDIO n —

Reads or writes 256 byte sector n on a double density disk. The read or write must be set previous to entry to DDIO by setting the command byte of the device control block. The buffer address for reading or writing must also by preset; this address is incremented by 256 at the end of DDIO to prepare for reading or writing the next sequential sector.

DECIMAL —

Sets the numeric conversion base for input-output to ten.

DECMAP n1 n2 - n1 n3

Gives the logical sector number n2, returning the proper physical sector number n3 for use with a single density 8" disk drive. n1=0 indicates a non-interleaved drive; n1=8000 indicates a DEC interleave drive.

DEFINITIONS —

Sets CURRENT to the CONTEXT vocabulary, so that subsequent definitions will be added to the vocabulary previously selected as CONTEXT. (Often used in the form:

cccc DEFINITIONS

where cccc is a vocabulary name).

DIGIT c n1 — n2 tf (valid conversion)
c n1 — ff (invalid conversion)

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.

DIO n —

Reads or writes 128 byte sector n on a single density disk. The read or write must be set previous to entry to DIO by setting the command byte of the DCB. The buffer address for reading or writing must also be preset; this address is incremented by 128 at the end of DIO to prepare for reading or writing the next sequential disk sector.

DLITERAL d — d (executing)
d — (compiling)

If compiling, compiles a stack double number into a literal along with a run-time routine (LIT). Later execution of the definition containing the literal will push it onto the stack. If executing, the number will remain on the stack.

DMINUS d1 — d2

Converts d1 to its double number two's complement.

DO n1 n2 -- (execute)
addr n -- (compile)

Occurs in a colon-definition in form:

DO ... LOOP
DO ... +LOOP

At run time, DO begins a sequence with repetitive execution controlled by a loop limit n1 and an index with initial value n2. DO removes these from the stack. Upon reaching LOOP the index is incremented by one. If +LOOP is used, the index is incremented by the value on top of the stack. This value may be negative, i.e., the loop may run with its index decreasing to a lower limit; see +LOOP) Until the new index equals or exceeds the limit, execution loops back to just after DO; when the index does equal or exceed the limit the loop parameters are discarded and execution continues beyond loop. Both n1 and n2 are determined at run-time and may be the result of other operations. Both n1 and n2 are signed 16-bit values. Within a loop 'I' will copy the current value of the index to the stack. See I, LOOP, +LOOP, LEAVE. When compiling within the colon-definition, DP compiles the run-time code (DO), leaves the following address addr and n for later error checking.

DOES> --

Used in conjunction with <BUILDS to create defining words. Used in the form:

: cccc <BUILDS ...DOES> ... ;

It marks the terminaton of defining code and the start of run-time code in cccc.

When cccc is executed in the form cccc bbbb, DOES> alters the code field address and the first parameter of bbbb so that, when bbbb is executed, the sequence of words following DOES> in cccc will be executed. (The PFA of bbbb will be on the stack when this execution begins, allowing the parameters of bbbb to be used by the code following DOES).

DP -- addr

A constant which returns the address (or 0 page) of 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. It may also be used to hold output column location of a decimal point, in user generated formating. The default value on single number input is -1.

DR0
DR1 --

Commands which select disk drives, by presetting OFFSET. DR0 selects "drive 1". DR1 should select "drive 2"; however, it actually selects "drive 3". The contents of OFFSET is added to the block number in BLOCK to allow for this selection. Offset is supressed for error text so that it may always originate from the first drive.

DROP n --

Drops the top number from the stack.

DUP n -- n n

Duplicates the top value on the stack.

2 0

ELSE addr n1 -- addr2 n2 (compiling)

Occurs within a colon-definition in the form:

IF ... ELSE ... ENDIF

At run-time, ELSE executes after the true part following IF. ELSE forces execution to skip over the following false part and resumes execution after the ENDIF. It has no stack effect.

At compile-time ELSE compiles BRANCH and reserves space at addr2 for a branch offset which will be resolved by ENDIF; it then leaves the address addr2 and n2 for error testing. ELSE also resolves the pending forward branch from IF by calculating the offset from addr1 to HERE and storing it at addr1.

EMIT c --

Transmits ATASCII character c to the selected output device.

EMPTY-BUFFERS --

Marks all block-buffers as empty, without affecting the contents. Updated blocks are not written to the disc. This is also an initialization procedure before first use of the disc.

ENCLOSE addr1 c -- addr1 n1 n2 n3

The text scanning primitive used by WORD. From the text address addr1 and an ATASCII delimiting character c, the following offsets are determined:

n1 is the offset from addr1 to the first non-delimiter character
n2 is the offset from addr1 to the first delimiter c after the text
n3 is the offset from addr1 to the first character beyond addr1+n2

END --

This is a synonym for UNTIL.

ENDIF addr n -- (compile)

Occurs in a colon-definition in form:

IF ... ENDIF
IF ... ELSE ... ENDIF

At run-time, ENDIF serves only as the destination of a forward branch from IF or ELSE. It marks the conclusion of the conditional structure. THEN is another name for ENDIF. Both names are supported in fig-FORTH. See also IF and ELSE.

At compile-time, ENDIF computes the forward branch offset from addr to HERE and stores it at adr. n is used for error tests.

ERASE addr n —
Sets n bytes, starting from addr1 to 0.

ERROR line — in blk
Executes error notification and restart of system. WARNING is first examined. If 1, the text of line n, relative to line 0 of screen 4 on drive 0 is printed. This line number may be positive or negative, and may lie beyond screen 4. If WARNING=0, n is just printed as a message number. If WARNING is -1, the definition (ABORT) is executed, which executes the system ABORT. The user may cautiously modify this execution by altering (ABORT). Fig-FORTH saves the contents of IN and BLK to assist in determining the location of the error. Final action is execution of QUIT.

EXECUTE cfa —
Executes the definition whose code field address is on the stack. Different from starting FORTH, which takes pfa.

EXPECT addr n —
Transfers characters from the terminal to address, until a "return" or the count of n characters has been received. Takes no action for n less than or equal to 0. One or more nulls are added at the end of the text.

FENCE — addr
A user variable containing an address below which FORGETting is not allowed. To forget below this point the user must alter the contents of FENCE.

FILL addr n b —
Fills memory starting at addr with a sequence of n copies of byte b. n>0.

FIRST — n
A constant that leaves the address of the first (lowest) block buffer.

FLD — addr
A user variable for control of number output field width. Although defined, this ideogram is not currently used by fig-FORTH implementations.

FLUSH —
Writes all blocks which have been flagged as updated to display and mark them as "in use" rather than "updated".

FORGET —

Executed in the form:

FORGET cccc

Deletes word cccc from the dictionary along wih all words added to the dictionary after cccc, regardless of their vocabulary. If cccc cannot be found in a search of the CONTEXT and CURRENT vocabularies, or if cccc's definition lies below the address contained in FENCE, an error message will be delivered. In addition an error message will occur if the CURRENT and CONTEXT vocabularis are not currently the same.

FORTH —

The name of the primary vocabulary. Execution makes FORTH the CONTEXT vocabulary. All new definitions are added to the FORTH vocabulary until new vocabularies are created and established as the CURRENT vocabulary. Vocabularies conclude by "chaining" to FORTH, so it should be considered that FORTH is "contained" within each of its descendant vocabularies.

HASH v1, addr, — v2

Given addr, the starting address of a name string in memory, and v1 the address of the first sub-vocabulary entry pointer in a given vocabulary definition, returns v2, the address of the sub-vocabulary entry pointer in that vocabulary which points to the proper sub-vocabulary in which to search for an entry which matches the name string. In Coin-Op FORTH, each vocabulary has been hashed into 8 sub-vocabularies dependent on the ATASCII value of the first character of a word's name.

See VOCABULARY and the user manual description of Coin-Op FORTH linking.

HERE — addr

Returns the address of the next available dictionary location.

HEX —

Sets the numeric conversion base to sixteen (hexadecimal).

HIMEM — addr

A constant which returns the address of OS register HIMEN. This register contains the address of the last byte before the Atari display list (hence, the last byte usable by the dictionary).

HLD — addr

A user variable that holds the address of the latest character of text during numeric output conversion.

HOLD c —

Used between <# and #> to insert an ATASCII character into a pictured numeric output string. e.g. 2E HOLD will place a decimal point. (Output generation occurs from high memory to low memory.)

I — n

Used within a DO-LOOP to copy the loop index to the stack. Alias for R.

ICAL —— addr
A constant which returns the address of the buffer address entry in IOCB # 0 (used by keyboard and screen editor).

ICCM —— addr
A constant which returns the address of the command byte in IOCB #0 (IOCB used by the keyboard and screen editor).

ICLL —— add
A constant which returns the address of the buffer length entry in IOCB #0 (IOCB used by the keyboard and screen editor).

ID NFA ——
Prints a definition's name from its name field address.

IF f —— (run-time)
—— addr n (compile)
Occurs in a colon-definition in the form:
IF (tp) ... ENDIF
IF (tp) ... ELSE (fp) ...ENDIF
At run-time, IF elects execution based on a Boolean flag. If f is true (non-zero), execution continues ahead thru the true part. If f is false (zero), execution skips till just after ELSE to execute the false part. After either part, execution resumes after ENDIF. ELSE and its false part are optional; if missing, false execution skips to just after ENDIF.

At compile-time IF compiles OBRANCH and reserves space for an offset at addr. addr and n are used later for resolution of the offset and error testing.

IMMEDIATE ——
Mark the most recently made definition as a word which will be executed when encountered during compilation rather than compiled (i.e. the precedence bit in its header is set). The user may force compilation of an immediate definition by preceeding it with (COMPILE).

IN —— addr
A user variable containing the byte offset within the current input text buffer (terminal or disk) from which the next text will be accepted. WORD uses and moves the value of IN.

INDEX n1 n2 ——
Prints the first line of each screen over the range n1, n2. This is used to view the initial comment lines of an area of text on disk screens.

INPT —— addr
A user variable which serves as the keyboard input buffer in Coin-Op FORTH.

INTERPRET —

The outer text interpreter which sequentially executes or compiles text from the input stream (terminal or disk) depending on STATE. If a word name in the input stream cannot be found after a search of CONTEXT and then CURRENT it is converted to a number according to the current base. That also failing, an error message echoing the name with a "?" will be given. Text input will be taken according to the convention for WORD. If a decimal point is found as part of a number, a double number value will be left. The decimal point has no other purpose than to force this action. See NUMBER.

J — n

Returns the current index of the next outer loop. May be used only within a nested DO-LOOP structure.

KEY — c

Leaves the ATASCII value of the next terminal key struck. If the key was an EOL (RETURN), a 0 is left on the stack, rather than $9B. ABORTs if the BREAK key is pressed.

LATEST — addr

Leaves the name field address of the most recently defined word in the CURRENT vocabulary.

LEAVE —

Forces termination of a DO-LOOP at the next LOOP or +LOOP by setting the loop limit equal to the current value of the index. The index itself remains unchanged, and execution proceeds normally until LOOP or +LOOP is encountered.

LFA pfa — lfa

Converts 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. In Coin-Op FORTH, this is the beginning of the FORTH kernel (unlike other FORTHs, where the disk buffers are high up in RAM).

LIST n —

Displays the ATASCII text of screen n on the selected output device. SCR contains the screen number during and after this process.

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 dictonary address to be pushed to the stack.

LITERAL n -- n (executing)
n -- (compiling)

If compiling, then compile the stack value n as a 16 bit literal. This definition is immediate so that it will execute during a colon definition. The intended use is:

```
: xxxx [calculate] LITERAL ;
```

Compilation is suspended for the compile time calculation of a value. Compilation is resumed and a LITERAL compiles this value along with the run-time code LIT. If not compiling, nothing happens and n remains on the stack.

LOAD n --

Read screen n from disk and begin interpretation of it. If the word --> is encountered, screen n+1 will be loaded and interpretation will continue on it. If either the word ;S or the end of a screen is encountered, interpretation will return either to the disk screen on which LOAD was encountered (i.e., LOAD can be nested) or to the terminal.

LOOP addr n -- (compiling)

Occurs in a colon-definition in form:

```
DO ... LOOP
```

At run-time, LOOP selectively controls branching back to the corresponding DO based on the loop index and limit. The loop index is incremented by one and compared to the limit. A branch back to DO occurs until the index equals or exceeds the limit; at that time, the parameters are discarded and execution continues beyond LOOP.

At compile-time, LOOP compiles the run-time word (LOOP) and uses addr to calculate an offset to DO. n is used for error testing.

M* n1 n2 -- d

A mixed magnitude math operation which leaves the double number signed product of two signed numbers.

M/ d n1 -- n2 n3

A mixed magnitude math operator which leaves the signed remainder n2 and signed quotient n3, from a double number dividend d 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 a single divisor u2.

MAX n1 n2 -- max

Leaves the greater of two numbers.

MESSAGE n ——
Prints on the selected output device the text of line n relative to screen 4 of drive 0. n may be positive or negative. MESSAGE may be used to print incidental text such as report headers. If WARNING is zero, the message will simply be printed as a number. (If n indicates a screen beyond screen 7 (n>127), the text will actually be taken from line n-256. Thus, when n=128, text will be taken from screen 2, line 0 instead of screen 8, line 0.)

MIN n1 n2 —— min
Leaves the lesser of two numbers.

MINUS n1 —— n2
Leaves the twos complement of a number.(Negates the number).

MOD n1 n2 —— n3
Divides n1 by n2, leaving the remainder n3 (with the same sign as n2).

NFA pfa ——nfa
Converts the parameter field address of a definition to its name field address.

NUMBER addr —— d
Converts a character string left at addr (with the count in the first byte) to a signed double number, using the current numeric base. If a decimal point is encountered in the text, its position will be given in DPL, but no other effect occurs. If numeric conversion is not possible, an error message will be given. The string must end with a blank.

OFFSET —— addr
A user variable which may contain a block offset to disk drives. The contents of OFFSET is added to the stack number by BLOCK. Messages printed by MESSAGE are independent of OFFSET. See BLOCK, DR0, DR1, MESSAGE.

OR n1 n2 —— n3
Leaves the bit-wise inclusive "or" (n3) of two 16 bit values (n1,n2).

OUT —— addr
A constant which returns the address of the OS register COLCRS. This register contains the horizontal position on the screen at which the next character will be put. The user may alter and examine OUT to control display formatting. OUT is incremented by EMIT.

OVER n1 n2 —— n1 n2 n1
Copies the second stack value, n1, placing it as the new top.

PAD —— addr
Leaves the address of the text output buffer, which is a fixed number of bytes above HERE. This buffer is used as a "scratch pad" to hold characters for intermediate processing. Its capacity is 64 characters.

PFA nfa — pfa
Converts the name field address of a compiled definition to its parameter field address.

PHYSOFF — addr
A user variable containing the value added to the logical block # to obtain the physical block # of a FORTH screen. This allows for "negative screens". Used by various disk access words.

PREV — addr
A variable containing the address of the entry in the disk buffer control table (PTAB) corresponding to the disk buffer most recently referenced. The UPDATE command marks this buffer to be later written to disk.

PTAB — addr
A variable whose parameter field contains the disk buffer control table. There are 4 entries in this table, each containing five bytes. The first entry contains information on the disk buffer containing the screen most recently accessed, while the second entry contains information on the buffer containing the screen which was accessed before that. The third or fourth entries are currently unused, as there are only two disk buffers; they could be used if more than 2 disk buffers were desired. The first cell of each entry contains the number of the screen contained in the block; the second cell contains the starting address of the buffer; the last byte contains the status of the buffer (0 = not in use, 1 = in use, $81 = in use and updated).

QUERY —
Accepts input of up to 120 characters of text (or until a "return") from the operator's terminal. Text is placed in the terminal input buffer, whose starting address is indicated by TIB IN is set to 0.

QUIT —
Clears the return stack, stops compilation, and returns control to the operator's terminal. No message is given. This is the outermost word in FORTH; it contains a loop which continually waits for input and then invokes INTERPRET.

R — n
Copies the top of the return stack to the computation stack.

R# — addr
A user variable which may contain the location of an editing cursor, or other file related function.

R/W addr blk f —
The fig-FORTH standard disk read-write linkage. addr specifies the source or destination block buffer, blk is the sequential number of the referenced block, and f is a flag where f=0 write and f=1 read. R/W determines the location on mass storage, performs the read-write and performs any error checking. The type of disk drive and the drive # are also indicated by blk; see the user manual for proper values.

R0 — addr
A user variable containing the initial location of the return stack. This ideogram is not actually defined in Coin-Op FORTH. However, the proper value has been stored in the user area at UP + 8 (see UP in the assembler glossary) and the user can access it if necessary.

R> — n
Removes the top value from the return stack and leaves it on the computation stack. See >R and R.

REPEAT addr1 n1 addr2 n2 — (compiling)
Used within a colon-definition in the form:
BEGIN ... WHILE ... REPEAT
At run-time, REPEAT forces an unconditional branch back to just after the corresponding BEGIN.
At compile-time, REPEAT compiles BRANCH and the offset from HERE to addr1. It then compiles the forward branch offset from addr2 to HERE and stores it at addr2 (resolving WHILE). n1 and n2 are used for error testing.

ROT n1 n2 n3 — n2 n3 n1
Rotates the top three values on the stack, bringing the deepest to the top.

RP! —
Initializes the return stack pointer from user variable R0. (R0 is not directly accessible to the user).

S->D n — d
Extend a signed single number to form a signed double number.

S0 — addr
A user variable which contains the initial value of the stack pointer. This ideogram is not actually defined in Coin-Op FORTH and as such is not directly accessible to the user. However, the proper value has been stored in the user area at UP + 6 (see UP in the assembler glossary) and the user can access it if necessary. It is useful for judging the depth of the stack and for setting up non-destructive stack dumps.
Pronounced S-zero. See SP!.

SCR — addr
A user variable containing the screen number most recently listed. This is used by various screen editing functions.

SECIO n —
Sets up the device control block to indicate a disk I/O operation should be performed on disk sector n, and then executes SIO. If SIO returns an I/O error condition, an error message is sent to the terminal and ABORT is executed. Expects the command byte, buffer address and length, and device ID # to be already set in the device control block.

SIGN n d — d
Stores an ATASCII "-" sign just before a converted numeric output string in the text output buffer when n is negative. n is discarded, but double number d is maintained. Must be used between <# and #> (just before #).

SIO 0 — n
Executes the Atari SIO (serial input/output) routine. Assumes that the device control block is properly set up. SIO is used for disk access. It returns n, where n=0 indicates no error occurred during SIO, and n>0 indicates that an error # n+1 occurred during SIO.

SMUDGE —
Used during word definition to toggle the "smudge bit" in a definition's name field. Setting this bit prevents an uncompleted definition from being found during dictionary searches, until compiling is completed without error. Once the bit is set to zero, the definition can be found in searches. (NOTE: This allows the possibility of recursive definitions.)

SP! —
Initializes the stack pointer from S0. (S0 is not directly accessible to the user. See S0.)

SP@ — addr
Returns the address of the top of the stack as it was just before SP@ was executed.

SPACE —
Transmits an ATASCII blank to the output device.

SPACES n —
Transmits n ATASCII blanks to the output device. Takes no action if n=0.

STATE — addr
A user variable containing the compilation state. A non-zero value indicates compilation, while a value of 0 indicates execution.

SWAP n1 n2 — n2 n1
Exchanges the top two stack values.

TASK —
A no-operation word which marks the boundary between the Coin-Op FORTH boot and applications.

TCIOV — b f
n — n b f
Executes a call to CIO (the Atari central Input/Output routine) using IOCB # 0. Assumes that IOCB # 0 has been properly set up for whatever operation is desired. If the buffer length cell in IOCB is set to 0 on a "put" operaton, the top value on the stack will be sent to the device (through the accumulator) but it will not be removed from the stack. The routine returns the value $80 if a break was encountered during the CIO routine; otherwise it returns a 0. Do not use interactively.

THEN —
An alias for ENDIF.

TIB — addr
A user variable containing the address of the terminal input buffer.

TOGGLE addr b —
Complements the 8-bit contents of addr by the bit pattern of byte b.

TRAVERSE addr1 n — addr2
Moves across the name field of a fig-FORTH variable length name field. addr1 is the address of either the length byte or the last letter. If n=1, the motion is toward the end of the name; if n=-1, the motion is toward the length byte. The addr2 resulting is the address of the other end of the name.

TRIAD scr —
Displays on the selected output device the three screens which include that numbered scr, beginning with a screen evenly divisible by three. Output is suitable for source text records, and includes a reference line at the bottom taken from line 15 of screen 4.

TYPE addr n —
Transmits n characters starting at addr to the screen. No action occurs if n=0. Vectors to ABORT if the break key is pressed.

U* u1 u2 — ud
Performs an unsigned multiplication of u1 by u2, leaving the unsigned double number product ud.

U/ ud u1 — u2 u3
Leaves the unsigned remainder u2 and unsigned quotient u3 from the unsigned double dividend ud and unsigned divisor u1.

U< u1 u2 - f
Leaves a true flag if unsigned 16 bit value u1 is less than unsigned 16 bit value u2; otherwise leaves a false flag.

UNTIL f -- (run-time)
addr n -- (compile)

Occurs within a colon-definition in the form:

BEGIN ... UNTIL

UNTIL marks the end of a BEGIN-UNTIL loop. At run-time, UNTIL controls the conditional branch back to the corresponding BEGIN. If f is false, execution returns to just after BEGIN; if true, execution continues beyond UNTIL.

At compile-time, UNTIL compiles the run-time (0BRANCH) and an offset from HERE to addr. n is used for error tests.

UPDATE --

Marks the most recently referenced block (pointed to by PREV) as modified. The block will subsequently be transferred automatically to disk should its buffer be required for storage of a different block, or upon execution of FLUSH.

USER n --

A defining word used in the form:

n USER cccc

which creates a user variable cccc. The parameter field of cccc contains n which is the cell offset relative to the user pointer (register UP) for this user variable. When cccc is later executed, it places the absolute user area storage address for cccc on the stack; it is at this address that cccc's "value" is kept.

VARIABLE --

A defining word used in the form:

n VARIABLE cccc

When VARIABLE is executed, it creates a dictionary for cccc and allots 2 bytes in cccc's parameter field for storage; the contents of these 2 bytes is initialized to n. When cccc is later executed, the address of its parameter field (containing n) is left on the stack, so that a fetch or store may access this location.

VLIST --

Lists the names of the definitions in the CONTEXT vocabulary. "Break" will terminate the listing. The listing will continue into the CONTEXT vocabulary's parent vocabulary. (See the user manual section on the linking system.)

VOC-LINK -- addr

A user variable containing the address of the vocabulary link 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 used in the form:

VOCABULARY cccc

to create a vocabulary definition cccc. Subsequent execution of cccc will make it the CONTEXT vocabulary which is searched first by INTERPRET. The sequence "cccc DEFINITIONS" will also make cccc the CURRENT vocabulary into which new definitions are placed.

In fig-FORTH, cccc will be so chained as to include all definitions of the vocabulary in which cccc is itself defined (a "child/parent" relationship is thus established between cccc and the vocabulary in which it is defined). All vocabularies ultimately chain to FORTH . By convention, vocabulary names are to be declared IMMEDIATE. See VOC-LINK. See the user manual for a description of the structure of a vocabulary definition.

WARNING — addr

A user variable containing a value controlling messages. If = 1 then error messages are taken from disk, and screen 4 of drive 0 is the base location for messages. If = 0, messages will be presented by number. If = -1, executes (ABORT) for a user specified procedure. See MESSAGE, ERROR.

WHILE f — (run-time)

addr1 n1 — addr1 n1 addr2 n2 (compile-time)

Occurs in a colon-definition in the form:

BEGIN ... WHILE (tp) ... REPEAT

At run-time, WHILE selects conditional execution based on boolean flag f. If f is true (non-zero), WHILE continues execution of the true part through to REPEAT, which then branches back to BEGIN. If f is false (zero), execution skips to just after REPEAT, exiting the structure.

At compile time, WHILE compiles OBRANCH and leaves addr2 of the reserved offset. The stack values will be resolved by REPEAT. n2 is used for error checking.

WIDTH — addr

A user variable containing the maximum number of letters saved in the compilation of a definitions' name. It must be 1 thru 31, with a default value of 31. The name character count and its natural characters are saved, up to the value in WIDTH. The value may be changed at any time within the above limits.

WORD c —

Reads the next text character from the input stream until a delimiter c is found, and stores the packed character string beginning at the dictionary buffer HERE. WORD leaves the character count in the first byte at HERE, then leaves the characters, and ends with two or more blanks. Leading occurrences of c are ignored. If BLK is zero, text is taken from the terminal input buffer, otherwise from the disk block whose number is stored in BLK. See BLK, IN.

X —

This is a pseudonym for the "null", i.e. a dictonary entry for a name of one character of ATASCII 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

Leaves the bitwise logical exclusive-or (n3) of two values (n1,n2).

[—

Used in a colon-definition in form:

: xxx [words] more ;

Ends compilation and sets the execution mode. The words after [are executed, not compiled. (This allows the user to perform calculations or make compilation exceptions before resuming compilation with].) See LITERAL,].

[COMPILE] —

Used in a colon-definition in form:

: xxxx [COMPILE] FORTH ;

[COMPILE] will force the compilation of an immediate definition, which would otherwise execute during compilation. The above example will select the FORTH vocabulary when xxxx executes, rather than at compile time.

] —

Sets the compilation mode; text from the input stream is subsequently compiled. See [.

II.3 ATARI EXTENSIONS

This section includes about ninety words, which can be divided into five groups: Graphics and sound words, controller and player/missile graphics words, IOCB words, words from the DEC User's Group, and a few miscellaneous words. Source code for most of these words is included on the Coin-Op FORTH boot disk for the user's convenience; the words are already in the boot, however. (The source code is in highly compressed form and is best used with the formatted printer words, Section III.4.)

The following discussions are aimed at familiarizing the user with the main words in this section. Some of the words are not discussed, however, so browsing through the glossary, which is complete, could be quite worthwhile.

Graphics and Sound

Graphics

Read chapter 9 of the Basic Reference Manual (Graphics Modes and Commands) and chapter 3 of De Re Atari (Graphics Indirection and Character Sets); the latter will give you an idea of how color registers work, while the former lists the specific Atari graphics commands. Coin-Op FORTH uses the same graphic commands as Atari BASIC (using post-fix notation, of course!).

The main graphic commands are as follows:

GRAPHICS	GR.
COLOR	C.
DRAWTO	DR.
LOCATE	LOC.
PLOT	PL.
POSITION	POS.
PUT , GET	
SETCOLOR	SE.
XIO18	

See the glossary for an explanation of how these words work in Coin-Op FORTH. The glossary also contains graphics primitives and a number of constants which return the addresses of OS and hardware graphics registers.

Sound

Read chapter 10 (Sound) of the Basic Referene Manual and chapter 7 (Sound) of De Re Atari for an explanation of sound on the Atari.

The only Coin-Op FORTH sound word is SOUND, which operates exactly like the BASIC command except that it uses post-fix notation. See the glossary entry below.

Controllers and Player/Missiles

Controllers

Read chapter 10 (section on controllers) in the Basic Reference Manual for a discussion of controllers.

Coin-Op FORTH uses the controller words:

PADDLE
PTRIG
STICK
STRIG

These work like their BASIC counterparts. See the glossary entries below for an explanation. Note also that the constant CONSOL returns the address of the register which reads the START, SELECT, and OPTION buttons.

Player/Missiles

Read chapter 4 (Player - Missile Graphics) of De Re Atari for an explanation of how players and missiles work.

Coin-Op FORTH includes the following player/missile words:

COLPM! - sets the color of a player
H'POS! - sets the horizontal position of a player or missile
PBASE - returns the base address of player/missile DMA
SIZE! - sets up player/missile graphics.

In addition, the following constants return the addresses of registers which are useful in manipulating player/missiles (consult the Atari OS and Hardware manuals for an explanation): DMACTL, GRACTL, PRIOR.

Consult the glossary below for descriptions of how to use these words.

Creation of the actual player/missile (through storing values into player/missile memory) is left to the user, who will no doubt find C! and CMOVE (in the Coin-Op FORTH kernel) to be very useful commands.

IOCB WORDS AND CIO

These words are used to set up I/O control blocks; these blocks in turn are used to perform I/O with a variety of devices. No attempt will be made here to describe in detail the various devices, commands, and formats used in this process. The user should read De Re Atari chapter 8 (Operating System) and the OS manual section on I/O in order to gain such knowledge.

I/O is performed by setting up an IOCB (there are 10 of them, each with 16 bytes) and executing the OS CIO routine. Coin-Op FORTH facilitates this process by providing the user with a number of words with which to handle IOCBs.

First, the user selects an IOCB #n to operate on by executing

n IOCB

where n is the number of the IOCB which is to be used. The words ICDNO , ICCOM , ICSTA , ICBAL , ICBLL , I1CAX , I2CAX , AND ICPTL will then return to the user the addresses of various entries in IOCB # n. For example, 0 IOCB I1CAX will return the address of the first auxillary information byte in IOCB #0 (which handles the screen editor); 3 IOCB ICBLL will return the address of the buffer length cell in IOCB #3.

These addresses can then be used to set up the IOCB. When this is complete, execution of the

word CIO will perform the desired I/O operation. (A related word, ACIO, is used in certain circumstances to send a single character to a device through the 6502 accumulator; see the glossary.) OPEN and CLOSE may be used to open and close the currently selected IOCB; PUT and GET will put and get characters to and from the device specified by the current IOCB.

DECUS

These words were written by the DEC Users Group, and perform some useful operations.

OSET sets a memory location to 0.

1+! , 1- , and 2* do exactly what one would expect.

ALLOC functions like ALLOT, but it reserves cells rather than bytes. ARRAY and TBL are defining words used to create tables. See the glossary for details.

BDUMP and \ are used in conjunction with each other.

```
addr1  addr2  BDUMP
```

will cause 2 digit hexadecimal representations of the contents of each byte from addr1 to addr2 to be placed on the screen. The routine places the first 8 values on one screen line, then performs a carriage return and places the next 8 on the next line, and so on. At the start of each line, BDUMP also prints the address (in hexadecimal) of that line's first byte, and at the end of each line the routine points the FORTH word \ . \ , in turn, takes an address and 8 values on the stack - precisely the format in which BDUMP has printed out its data. \ then takes the low byte of each value and stores it into memory, starting at the address on the bottom of the stack. It ends with a QUIT, which performs a line-feed without printing out "oK".

Thus, it is possible to make use of the Atari screen editor to move the cursor over a line which BDUMP has printed out and type over the values which are printed on that line; when a [RETURN] is then performed, these values will be written into memory at the locations which were read by BDUMP. Furthermore, data printed by BDUMP on subsequent screen lines will not be damaged by "oK"s, since QUIT is used to exit. This is a clever and powerful interactive tool; try it for yourself to see how it works.

Miscellaneous

These include a number of OS and hardware constants (i.e., constants which return the addresses of OS and hardware registers), the historically obsolete constants LPWORDS and FORMY (used to indicate where the screen code of printing and code formatting words lay on disk), and the word SAVENFAs.

Read through the glossary to discover which OS and hardware constants are available - these are quite useful, and most of them are used by various graphics routines.

SAVENFAs takes the vocabulary entry pointers stored in the FORTH vocabulary and saves them in the boot parameter area (the 30 or so bytes after the "origin" , which is at $D00)] This will prevent a cold start (see COLD) from "forgetting" any non-booted words which the user has added. It is used by WTOBJ.

II.3 GLOSSARY OF THE ATARI EXTENSIONS

.IOC n --

A defining word used in the form:

n .IOC cccc

Creates a header for cccc and places the value n in cccc's parameter field. When cccc is later executed it returns the address of the memory location which is n bytes above the first byte of the "current" IOCB. See IOCB.

0SET addr --

Sets the 16-bit value at addr to 0.

1+! addr --

Increments the 16-bit contents of addr by 1.

1- n -- n-1

Subtracts 1 from n.

2* n1 - n2

Multiplies n1 by 2, leaving the product n2.

ACIO n --

Places n in the 6502 accumulator and calls CIO, which will use the IOCB previously selected by the user (see IOCB). Assumes that the IOCB is already set up. Can be used to send a single character to a device. Normally used with ICBLL set to zero.

ALLOC n --

Reserves n cells in the dictionary (i.e., advances the dictionary pointer by 2n bytes).

ARRAY n --

A defining word used in the form:

n ARRAY cccc.

Creates a dictionary entry for cccc and reserves n cells (n*2 bytes) as cccc's parameter field. Later execution of cccc returns the address of the first cell in this parameter field. This address may then be used to store and retrieve values in the parameter field.

ATACHR -- addr

A constant which contains the ATASCII value of the character most recently read from or written to the screen, or the value of the most recently plotted or read color pixel.

AUDCTL -- addr

A constant which contains the address of the audio mode control register ($D208)

BOTSC -- addr

A constant which contains the address of the split screen register ($2BF). If the contents equal 4, split screen text is enabled; if they equal 24, full screen text is enabled.

BDUMP addr1 addr2 ——
Displays the contents of memory addresses addr1 through addr2, byte by byte, on the output device. The contents are displayed in a 2 digit hexadecimal representation, with 8 values per screen line; the address of the first byte on a given line is printed at the beginning of that line, and each line ends with the word \, which allows memory locations in the "dumped" area to be easily altered. See \ and the user manual section on DECUS words.

C. n ——
Sets the color register (pixel data to be used in subsequent plotting, drawing, and printing) to n. n ranges from 0 to 4.

C1AUD —— addr
A constant which contains the address of the audio control register for audio channel 1 ($D201).

C2AUD —— addr
A constant which contains the address of the audio control register for audio channel 2 ($D203).

C3AUD —— addr
A constant which contains the address of the audio control register for audio channel 3 ($D205).

C4AUD —— addr
A constant which contains the address of the audio control register for audio channel 4. ($D207)

CH —— addr
Returns the address of the register which contains the ATASCII value of the most recently pressed key. ($2FC).

CH? addr ——
Converts the 8-bit contents of addr to a 2 digit hexadecimal value, and sends this value to the output device.

CHBAS —— addr
A constant which contains the address of the register which contains the most significant byte of the starting address of the current character set. ($2F4).

CHH n ——
Types the least significant byte of n as a 2 digit hexadecimal number. (E.g. $0014 CHH types 14; $FF23 CHH types 23).

CIO ——
Calls the Atari CIO routine, using the "current" IOCB. This assumes that an IOCB has been selected using n IOCB and that IOCB #n has been set up properly. (This can be done using the words defined by .IOC, after n IOCB has been executed).

CLEAR ——
Clears the screen.

CLOSE —

Closes the currently selected IOCB. n IOCB must be performed before calling CLOSE. Do not attempt to close IOCB # 0, as this controls the screen editor.

CN n —

An alias for CONSTANT.

COL0 — addr

A constant which contains the address of the playfield color register for playfield color 0. ($2C4).

COL1 — addr

A constant which contains the address of the playfield color register for playfield color 1. ($2C5)

COL2 — addr

A constant which contains the address of the playfield color register for playfield color 2. ($2C6).

COL3 — addr

A constant which contains the address of the playfield color register for playfield color 3. ($2C7).

COL4 — addr

A constant which contains the address of the playfield color register for playfield color 4. ($2C8).

COLOR n —

Alias for C.

COLPM — addr

A constant which contains the address of the first player/missile color register (i.e., the byte corresponding to player 0). ($2C0).

COLPM! c n —

Sets the color of player n to c, where the highest 4 bits of c indicate the color (0-15) and the lowest 4 bits indicate the luminance (0-15, although the lowest bit is not actually read so that luminances 0 and 1 are the same, as are 2 and 3, etc.). n=0 to 3. Missile #n will have the same color as player#0, and so forth, unless the "five player" option is enabled. See De Re Atari.

CONSOL — addr

A constant which contains the address of the register which reads the console switches START, SELECT, OPTION. ($D01F)

CRSINH — addr

A constant which contains the address of the cursor inhibit register ($2F0). When the contents equal 0, a cursor is displayed on the display screen.

Color — addr
A variable which contains the number of the color register to be used in drawing and printing.

DLST — addr
A constant which contains the address of the OS shadow of the register which contains the starting address of the display list. ($230).

DMACTL — addr
A constant which contains the address of the OS shadow of the DMA control register. ($22F). Alias for DMCT.

DMCT — addr
A constant which contains the address of the OS shadow of the DMA control register. ($22F).

DR. x y —
Draw a line from the current cursor position to x,y, in the color previously selected (using SETCOLOR and COLOR).

DRAWTO x y —
Alias for DR.

F1AUD — addr
A constant which contains the address of the audio frequency control register for audio channel 1. ($D200).

F2AUD — addr
A constant which contains the address of the audio frequency control register for audio channel 2. ($D202).

F3AUD — addr
A constant which contains the address of the audio frequency control register for audio channel 3. ($D204).

F4AUD — addr
A constant which contains the address of the audio frequency control register for audio channel 4. ($D206).

FILDAT — addr
A constant which contains the address of an OS register which holds data used in fill operations. ($2FD)

FORMY — 39
Returns decimal 39, which historically was used as the starting screen of a code formatting program.

GET — n
Gets one character from the device specified by the "current" IOCB and leaves it on the stack. Assumes that the IOCB is set up (except for the command byte) prior to the call to GET.

GR. n —
Clears the current screen, opens the graphics screen using graphics mode n. Add decimal 16 to n to allow full screen graphics (i.e., no text window; this can only be used during program execution when no screen editing functions are necessary). Add decimal 32 to n to prevent the screen from being cleared.

GRACTL — addr
A constant which contains the value of the graphics control register. ($D01D).

GRAPHICS n —
Alias for GR.

Get — n
A machine code routine which gets one piece of informaton from the device specified by the "current" IOCB and leaves it on the stack. The IOCB must be set up before Get is called.

H? addr —
Converts the 16-bit contents of addr to a 4 digit hexadecimal value, and transmits this value to the output device.

HD n -
Types the single character representation of n, as if BASE were set to n+1 (e.g. 5 HD types 5, 15 HD types F, 35 HD types Z).

HH n —
Converts the 16-bit value n to a 4 digit hexadecimal value, and transmits this value to the output device.

HPOS! x n —
Sets the horizontal position of player n to x. n = 0 to 7 (missile 0 = player 4, etc.). x = 0 to 255.

I1CAX --- addr
Defined by .IOC. Returns the address of the first auxilliary information byte in the "current" IOCB.

I2CAX --- addr
Defined by .IOC. Returns the address of the second auxilliary information byte in the "current" IOCB.

ICBAL --- addr
Defined by .IOC. Returns the address of the low byte of the buffer address entry in the "current" IOCB.

ICBLL --- addr
Returns the address of the cell which contains the buffer length entry in the "current" IOCB.

ICCOM --- addr
Defined by .IOC. Returns the address of the command byte of the "current" IOCB.

ICDNO ---- addr
Defined by .IOC. Returns the address of the device # byte of the "current" IOCB.

ICPTL --- addr
Defined by .IOC. Returns the address of the "put character entry pointer" in the "current" IOCB. This is only used by BASIC.

ICSTA --- addr
Defined By .IOC. Returns the address of the status byte of the "current" IOCB.

IOB --- addr
A variable which contains an offset from the first byte of IOCB # 0 to the first byte of the IOCB which is currently selected. See IOCB.

IOC --- addr
A variable which contains the starting address of the IOCB currently selected by the user. See IOCB.

IOCB n ---
Sets the "current" IOCB to IOCB # n. (This changes the values in IOC and IOB).

LMARGN --- addr
Constant which contains the address of the register which contains the left margin value. ($52).

LOC. x y --- n
Gets the value of the pixel on the screen at position x,y, and positions the graphics cursor at this point. n will indicate which color register the pixel's color is taken from.

LOCATE x y --- n
Alias for LOC.

LPWORDS --- 37
Historically, line printer words were placed on screen 37 (decimal); LPWORDS returned the starting screen number so one could write LPWORDS LOAD.

NOPLY ---
Disenables player/missile DMA and sets the player/missile size and position registers to 0. (This does not stop the player/missile DMA cycle steal, however, as DMACTL is left intact.)

OPEN n1 n2 n3 ---
Opens the "current" IOCB, using the parameters on the stack. n1 is the value to be stored in ICAX2, n2 is the value to be stored in ICAX1, and n3 is the buffer address (the buffer should contain an ATASCII string by which CIO will identify which device is being linked to the IOCB in question; see the Atari OS manual for details).

PADDLE n1 --- n2
Reads paddle n1 (n1 = 0 to 7), returns paddle position n2 (n2 = 0 to 256).

PBASE --- addr
Returns the address of the player/missile DMA area. This value is set by GRAPHICS and changes with different modes.

PL. x y ---
Plots a point in the color last selected (using SETCOLOR and COLOR commands) at screen position x,y.

PLAYER n ---
Sets up player/missile DMA (by setting PBASE, GRACTL, and DMACTL). n = 1 for single line resolution; n=2 for double line resolution. See De Re Atari.

PLOT x y ---
Alias for PL.

PMBASE — addr
A constant which contains the address of the hardware register which holds the most significant byte of the base address of the player/missile DMA area. ($D407).

POS. x y —
Sets the current graphics cursor position to x,y. (0,0 is the top left corner of the screen). Does not plot any points.

POSITION x y —
Alias for POS.

PRIOR — addr
Returns the address of the hardware priority register (GPRIOR; $D018).

PTRIG n1 — n2
Reads paddle trigger n1 (n1 = 0 to 7), returning n2 where n2 = 0 indicates trigger pressed, n2 = 1 indicates trigger not pressed.

PUT n —
Puts the character with ATASCII value n at the current cursor position on the screen.

Qbase — n
A variable which contains the address of the player/missile base. See De Re Atari, chapter 4.

RND — n
Returns a random number n on the stack, where n ranges from 0 to 255. (This value comes from hardware timer SKRES).

RTCLK — addr
Returns the address of the first byte of system timer 1 ($14). This is a three byte timer.

S: — addr
A variable which contains the ATASCII characters "S" and ":". The address of this variable is used as the buffer address in an "open" call to CIO and indicates that the device being opened is the graphics screen.

SAVENFAs —
Moves the NFAs in the vocabulary entry pointers of the FORTH vocabulary to the boot parameter area (starting at $D22); after this is done, a cold start (COLD) will not erase any words defined prior to the execution of SAVENFAs from the dictionary.

SE. n1 n2 n3 ——
Sets color register n1+1 to hold color n2 with luminance n3.

SETCOLOR n1 n2 n3 ——
Alias for SE.

SIZE! s n ——
Sets the width of player n to s, where s = 0 or 2 indicates normal width, s = 1 indicates double width, and s = 3 indicates quadruple width. n = 0 to 4; all missiles are set by n = 4.

SKCTL —— addr
A constant which contains the address of the serial port contol register ($D20F) which controls the configuration of the serial port as well as the Fast Pot Scan and the Keyboard Enable Functions.

SOUND n p d v ——
Sets audio channel n (n = 0 to 3) to have pitch p (p is actually a divisor; see the Basic Reference Manual for pitches corresponding to a given p), distortion d, and volume v.

SPB ——
Sets the base address of the player/missile DMA area and HIMEM (the highest available dictionary address, which is $17F bytes above the player/missile base address); the PM base address must fall on a 2K boundary.

STICK n1 —— n2
Reads joystick n1 (n1 = 0 to 3) and returns n2, where n2 indicates which direction the stick is being pushed. See the Basic Reference Manual for these values.

STRIG n1 —— n2
Reads joystick trigger n1 (n1 = 0 to 3) and returns n2, where n2 = 0 indicates trigger pressed and n2 = 1 indicates trigger not pressed.

TBL ——
A defining word used in the form:

```
TBL  cccc  n1 , n2 , n3 , ...
```

TBL creates a header for "table" cccc; values may then be stored into cccc's parameter field. When cccc is later executed, its PFA is returned on the stack, allowing the user to then index values in the parameter field. Used to create date tables at compile time.

VDELAY — addr

A constant which contains the address of a hardware register which is used to "delay" the appearance of a player or missile by 1 scan line. This is useful when two-line player resolution is used. ($D016).

XIO18 n —

Fills an area of the screen using the color in color register n. See description of use in the Basic Reference Manual.

\ addr n0 n1 n2 n3 n4 n5 n6 n7

Takes the low byte of each stack value n0, n1, ... , n7, and stores those bytes in memory beginning at addr. n0 is stored at addr+0, n1 is stored at addr+1, etc. This is used in conjuncton with BDUMP to facilitate rapid alteration of the contents of an area of memory. See BDUMP and the discussion of DECUS words before this glossary.

II.4 THE FIG EDITOR

The usual way of entering the fig Editor is by using n EDIT, which sets up editing on screen n. The command words in the fig Editor can be divided into three groups: screen editing words, line editing words, and words from the FORTH vocabulary that are useful for editing. The commands in these three groups are listed below:

Screen Editing Commands
UL LL DOIT

Line Editing Comands
HL DL IL RL CL SL BL TL $ %

Other Commands
L N COPY SHOW LIST WHERE

The cursor control (arrow) and editing (Clear, Insert, Delete) keys are also used in the fig Editor.

Screen Editing Words

There are three screen editing words in the fig Editor: UL and LL clear the display screen and list the 'upper' or 'lower' (first or last) 16 lines of the current screen. The third word, DOIT, causes the contents of the top 16 lines of the display screen to be copied to the appropriate half of the screen buffer and marks that buffer as updated. For convenience, UL and LL print the word DOIT two lines below the 16 lines that they list. To use the screen editor, do a UL or LL (depending on which half of the current screen you want to work on), then use the arrow and editing keys to change the information on the top 16 lines of the display. When you are finished editing that half-screen, register the changes by moving the cursor to the line with DOIT on it and hitting [RETURN]. An Important Note: don't hit [RETURN] while you are on the top 16 lines of the display. If you do, the interpreter will try to interpret the stuff you are editing. Use the arrow keys to move from line to line instead.

Line Editing Words

Most of the fig line-editing commands involve moving lines back and forth between the current screen and the pad. IL, DL, HL, RL, and CL all fall into this category (though CL copies a specified line from another screen to the pad). SL and BL put a blank line on the current screen, but in two different ways: SL moves the lines below the new blank line down to make room for it, while BL just overwrites the specified line with blanks. $ and % both put the text following them onto the screen: $ puts it on the specified line; % puts the text on the line after the specified line, pushing the lines below it down to make room. TL sets up editing of a specified line by typing the line with the line number and a $ preceding it (it also copies the line to the pad). This way, you can TL a line, edit it with the arrow keys, and put the edited line back by just hitting [RETURN] (since the line number and $ are already there).

Except in certain situations (inserting lines onto a screen that already has text on it, for example), the line editing commands are not as useful as the fig screen editor, since moving a cursor around and making changes on the screen are easier than typing lots of commands.

Other Editing Words

There are also some words from the FORTH vocabulary that are useful for editing. LIST and SHOW are good for looking at one or more screens (to find a certain word, for example). COPY, as its name implies, copies the contents of one screen to another. L lists the current screen. N lists the next screen. WHERE is useful for finding errors in a screen that won't LOAD. Typing WHERE after compilation halts will print the screen number and the text of the line where the compiler stopped, with a caret indicating exactly where the error was found.

The normal way of exiting the fig Editor is with FLUSH, which resets the screen margins, copies updated screens to disk, and puts the system back in the FORTH vocabulary.

FIG EDITOR GLOSSARY

#OFLINES — n — FORTH
Returns the number of lines per editing screen (32).

-MOVE — addr n — — EDITOR
Moves a line of characters that starts at addr to line n of the current edit screen. Marks the current screen as updated.

$ — n — — EDITOR
Puts the text following $ on line n of the current edit screen. Marks the screen as updated.

% — n — — EDITOR
Puts the text following % after line n of the current edit screen. The lines below line n are pushed down to make room (the last line of the screen is lost). Marks the screen as updated.

BL — n — — EDITOR
"Blank line." Replaces line n of the current edit screen with blanks. Marks the screen as updated.

COPY — n1 n2 — — FORTH
Copies the contents of screen n1 to screen n2 on the disk.

CL — n1 n2 — — EDITOR
"Copy line." Copies line n1 of screen n2 to the pad. The current edit screen number does not change.

DL — n — — EDITOR
"Delete line." Deletes line n from the current edit screen, moving the lines below the deleted line up. Saves the deleted line in the pad and marks the screen as updated.

DOIT — — — EDITOR
Causes the top 16 lines of the display screen to be copied to the screen buffer. If TOPFLAG contains $0, the display is copied to the 'upper' half (first 512 characters) of the buffer; if TOPFLAG contains $200, the display is copied to the 'lower' half (last 512 characters) of the buffer. The buffer is then marked as updated, the screen is cleared, and the top 16 lines are reprinted.

EDIT n — FORTH
Sets up editing of screen n by establishing EDITOR as the CONTEXT vocabulary and setting SCR (the current editing screen) to n.

EDITOR — FORTH
Makes EDITOR the CONTEXT vocabulary. It will then be searched first when the input stream is interpreted. The EDITOR vocabulary contains screen- and line-editing functions.

FLUSH — EDITOR
The EDITOR version of FLUSH resets the screen margins to (left) 2 and (right) 38, establishes FORTH as the CONTEXT vocabulary, and moves updated blocks from buffer to disk.

HL n — EDITOR
"Hold line." Copies line n of the current edit screen to the pad.

IL n — EDITOR
"Insert line." Inserts the line in the pad after line n of the current edit screen, pushing the lines below it down (and losing the last line of the screen). Marks the screen as updated.

L — FORTH
Lists the current editing screen (screen number is obtained from SCR).

LINE n — addr FORTH
Returns the starting address in disk buffer of line # n of the screen currently being edited. If the line is not on the screen (n>32), an error message is printed.

LIST n — FORTH
Lists the contents of screen n with the margins set to (left) 2 and (right) 38. The number of each line is printed at the beginning of that line.

LL — EDITOR
Clears the display screen and displays the lower half of the current edit screen on the top 16 lines. The word DOIT is printed two lines below the half-screen for the user's convenience.

N — FORTH
Lists the contents of the next screen (screen number is obtained by incrementing the contents of SCR by 1).

RL n — EDITOR
"Replace line." Replaces line n of the current edit screen with the line in the pad. Marks the screen as updated.

SHOW n1 n2 — FORTH
Lists the contents of screens n1 to n2, inclusive.

SL n — EDITOR
"Spread line." Creates a blank line n on the current edit screen by pushing all of the lines below it down. The last line of the screen is lost. Marks the screen as updated.

TEXT — FORTH
Moves the next line in the input stream to the pad; the line will end with a series of blanks.

TL n --- EDITOR

"Type line." Copies line n of the current edit screen to the pad, then types the line with the line number and the word $ preceding it. The line number and $ are there to facilitate changes to the line.

TOPFLAG --- addr EDITOR

A variable used by the editor to keep track of which half of a screen is being edited. Contains $0 if it's the 'upper' half (first 16 lines) or $200 if it's the 'lower' half (last 16 lines).

UL --- EDITOR

Clears the display screen and displays the upper half of the current edit screen on the top 16 lines. The word DOIT is printed two lines below the half-screen for the user's convenience.

ULL n --- EDITOR

Clears the display screen and displays one half of the current edit screen on the top 16 lines. The screen offset, n, determines which half is displayed ($0 for the 'upper' half, $200 for the 'lower' half). The screen margins are set to (left) 3 and (right) 34. The word DOIT is printed two lines below the half-screen for the user's convenience.

WHERE n1 n2 --- FORTH

After compilation of FORTH code stops due to an error, the block # n2 and an offset n1 from the start of the block to the point where the error was detected are left on the stack. Subsequent execution of WHERE will print the block # and the line on which the error was found; EDITOR is then established as the CONTEXT vocabulary and SCR is set to the block #.

THE FIG-FORTH ASSEMBLER

The Atari computer contains a 6502 microprocessor and FORTH provides easy access to its assembly language. This section does not explain assembly language programming in general or the 6502 instruction set in specific. There are good books available on the subject which should be consulted first. This is intended to explain assembly language programming in FORTH, with a general discussion of the instructions specific to the FORTH assembler and how they are used.

Syntactical differences: Like the rest of FORTH, the assembler uses reverse polish notation. In standard assembly language programming the syntax is:

[op-code mnemonic] [addressing mode] [operand]

e.g. LDA #0 or STA (addr), Y . With the FORTH assembler these appear in the order

[operand] [addressing mode] [op-code]

e.g. 0 #LDA, or addr)Y STA. Naturally, to accomodate this change there are certain particular ways of writing the address modes that are not found in standard 6502. These are:

,A	-accumulator mode	,X	-indexed with X
,Y	-indexed with Y	X)	-indexed indirect with X
)Y	-ind. indir. with Y		

The immediate mode is the same: # . Also, one need not write only one operation per line.

Op-code changes: There are differences in the appearance of 6502 op-code mnemonics used by the FORTH assembler; an alphabetic list of these differences appears below. Each op-code mnemonic has a comma affixed to its end. E.g. ADC becomes ADC, . This sets off what would normally be one line of assembly code and helps to differentiate assembler words that might be interpreted as hexadecimal numbers, such as ADC . Also, "," compiles into the dictionary, so the comma here indicates the point at which code is generated.

Conditionals: One thing missing from the FORTH assembler vocabulary is the set of standard branch statements such as BCS or BEQ. Instead, this ASSEMBLER vocabulary is equipped with conditional structures analogous to those in the FORTH vocabulary, including BEGIN, - UNTIL, loops and IF, - ELSE, - ENDIF, . The comma serves to differentiate these words from those in the FORTH vocabulary. With these conditionals comes a set of tests which specify branching by the conditionals which follow them. For example,

CS IF, ... ELSE,

will generate a branch to the code following the IF, if the carry status bit is set, but will generate a branch to beyond the ELSE, if not. Other such tests include

0= 0< VS >=

NOT reverses the result of any test appearing; for example, CS NOT IF, executes the code following IF, if the carry status bit is not set. This method of branching eliminates the need for labels within assembly code.

CODE and C; : It is possible to write assembly code when not in the ASSEMBLER Vocabulary by using the defining word CODE . CODE changes the CONTEXT vocabulary from CURRENT to ASSEMBLER making available the assembler op-code mnemonics. At the end of a CODE definition the word C; is used,

returning CONTEXT to CURRENT. CODE lets one name assembler words as if they were in FORTH.

XSAVE : The 6502 x register contains the FORTH parameter stack pointer. When the x register is being used in an assembly code definition for other purposes it must be saved elsewhere and restored before leaving ASSEMBLER. XSAVE leaves the address of a zero-page buffer for storing the x register. It appears in assembly definitions in the form XSAVE STX, and XSAVE LDX, .

NEXT and return of controls : NEXT is a constant which returns the address of the FORTH address interpreter, a subroutine that moves execution from one definition to the next. At the end of a CODE definition control must be returned to NEXT (i.e., NEXT JMP,) or to another routine which returns to NEXT such as POP , POPTWO , PUSH , and PUT (see the glossary for details on these words).

The glossary that follows contains the instructions used by the Atari FORTH assembler. The vocabulary in which a word is found appears at its right. Any effects on the stack, either at assembly or run-time are also given along with more detailed definitions and explanations.

6502 op-code mnemonics as employed by the FORTH assembler:

one mode op-codes:					multi-mode op-codes:				
BRK,	CLC,	CLD,	CLI,	CLV,	ADC,	AND,	ASL,	BIT,	CMP,
DEX,	DEY,	INX,	INY,	NOP,	CPX,	CPY,	DEC,	EOR,	INC,
PHA,	PHP,	PLA,	PLP,	RTI,	JMP,	JSR,	LDA,	LDX,	LDY,
RTS,	SEC,	SED,	SEI,	TAX,	LSR,	ORA,	ROL,	ROR,	SBC,
TAY,	TSX,	TXS,	TXA,	TYA,	STA,	STX,	STY,		

ASSEMBLER GLOSSARY

\# — ASSEMBLER
Specifies immediate addressing mode for the next op-code generated.

)Y — ASSEMBLER
Specifies indirect indexed with Y addressing mode for the next op-code generated.

,X — ASSEMBLER
Specifies indexed with X addressing mode for the next op-code generated.

,Y — ASSEMBLER
Specifies indexed with Y addressing mode for the next op-code generated.

,A — ASSEMBLER
Specifies accumulator addressing mode for the next op-code generated.

0< — cc (assembling) ASSEMBLER
Specifies that the following conditional generate a branch if the previous operation has produced a negative result, i.e., the sign bit of the status register is set (s=1). The flag cc is left at assembly time; there is no run time effect on the stack.

0= — zz (assembling) ASSEMBLER
Specifies that the following conditional generate a branch if the previous operation has produced a zero result, i.e., if the zero-bit of the status register is set (z=1). The flag zz is left at assembly time, there is no run-time effect on the stack.

>= — cc (assembling) ASSEMBLER
Specifies that the following conditional generate a branch if the zero bit of the status register is set (z=1), or if the sign bit is positive (s=0). It is only correct after SUB, or CMP,.

ASSEMBLER — FORTH
Makes ASSEMBLER the CONTEXT vocabulary. It will then be searched first when the input stream is interpreted.

BEGIN, — addr1 (assembling) ASSEMBLER
— (run-time)
A word in CODE definitions used to mark the start of a loop. Used in the form:
BEGIN, ... cc UNTIL,
At run-time, it marks the return point for the corresponding UNTIL,. When reaching UNTIL,, abranch ack to BEGIN, will occur if the status bit cc left by the conditional prior to the UNTIL,, is false; otherwise execution continues ahead. At assembly time, BEGIN, leaves the dictionary pointer address addr and the value 1 for later testing of conditional pairing by UNTIL,.

BOT — n ASSEMBLER
Used in code assembly in the form:
BOT LDA, or BOT 1+ X) STA,
It addresses the bottom of the parameter stack by selecting the indirect x mode and leaving n=0 at assembly time. n may be modified to another byte offset into the parameter stack. It must be followed by a multi-mode op-code mnemonic.

CODE — FORTH
Defining word used in the form:
CODE cccc ... C;
Creates a dictionary entry for cccc in the CURRENT vocabulary; the CFA of cccc contains the address of its parameter field. The CONTEXT vocabulary is set to ASSEMBLER, to make the assembler mnemonics available. When cccc is later executed the machine code in this parameter field will execute.

CPU n — (compiling assembler) ASSEMBLER
An assembler defining word used to create assembler mnemonics that have only one addressing mode. For example: 88 CPU DEY, creates the word DEY, with its op-code 88 as a parameter. When DEY, executes it assembles 88 as a one byte op-code.

CS --- cc (assembling) ASSEMBLER
Specifies that the following conditional generate a branch if the carry bit of the status register is set (c=0). The flag cc is left at assembly time; there is no run-time effect on the stack.

ELSE, --- (run-time) ASSEMBLER
addr1 2 --- addr2 2 (assembling)
Occurs within a code definition in the form:
cc IF, (true) ELSE, (false) ENDIF,
At run-time ELSE, sets off a section of code that will be executed if the condition specified by cc is false. At assembly-time ELSE, assembles a forward jump to just after ENDIF, and resolves a pending forward branch from IF,. The twos are used for error checking of paired conditionals.

END, --- ASSEMBLER
A compiling version of UNTIL,.

ENDIF, --- (run-time) ASSEMBLER
addr2 --- (assembly-time)
Occurs in a code definition in the form:
cc IF, (true) ELSE, (false) ENDIF,
At run-time ENDIF, marks the conclusion of a conditional structure. Execution of either the true or false part continues after ENDIF,. When assembling addr and 2 are used to resolve the pending forward branch to ENDIF,.

IF, cc --- addr2 (assembly-time) ASSEMBLER
--- addr2 (run-time)
Occurs within a code definition in the form:
cc IF, (true) ELSE, (false) ENDIF,
At run-time IF, branches based on the condition code cc. If the status bit specified by cc is true execution continues ahead. If false a branch is generated to the next ELSE, (if present) or to the following ENDIF,.
When assembling IF, creates a forward branch to an undetermined location based on the condition code cc, and leaves addr and 2 for resolution of the branch by the corresponding ELSE, or ENDIF,. Conditionals may be nested.

INDEX --- addr (assembling) ASSEMBLER
An array used within the assembler, which holds bit patterns of allowable addressing modes. (Do not confuse with INDEX in FORTH vocabulary.)

IP --- addr ASSEMBLER
A constant that returns the zero-page address of the Interpreter pointer, specifying the next FORTH address which will be interpreted by NEXT. It is used in code definitions in the form:
IP STA, or IP)X LDA,

M/CPU n1, n2 --- (compiling assembler) ASSEMBLER
An assembler defining word used to create assembler mnemonics that have multiple addressing modes. For example: 1C6E 60 M/CPU ADC, creates the word ADC, with two parameters. When ADC, executes it uses these parameters, the stack values, and the contents of MODE to calculate and assemble the correct op-code and operand.

MEM --- ASSEMBLER
Used within the assembler to set MODE to the default value for direct memory addressing, zero-page.

MODE --- addr ASSEMBLER
A variable used within the assembler, which holds a flag indicating the addressing mode of the op-code being generated.

N --- ASSEMBLER
A constant address of an 8 byte utility area in zero-page. See SETUP.

NEXT --- addr (assembling) ASSEMBLER
This is the inner interpreter that uses the interpretive pointer IP to execute compiled FORTH definitions. It is not directly executed but is the return point for all code procedures. It acts by fetching the address pointed by IP, storing this value in register W. It then jumps to the address pointed to by the address pointed to by W. W points to the code field of a definition, which contains the address of the code which executes for that definition. This usage of indirect threaded code is a major contributor to the power, portability, and extensibility of FORTH.

NOT cc1 --- cc2 ASSEMBLER
Used in ASSEMBLER conditionals, it reverses the condition code for the conditional that follows. For example: CS NOT UNTIL, will terminate the loop if the carry is <u>not</u> set.

POP --- addr (assembling) ASSEMBLER
n --- run-time
A constant which leaves (during assembly) the machine address of a routine which at run-time will pop a 16-bit value from the computation stack and return control to NEXT.

POPTWO --- addr (assembling) ASSEMBLER
n1, n2 --- (run-time)
A constant which leaves (during assembly) the machine address of a routine which at run-time will pop two 16-bit values from the computation stack and return control to NEXT.

PUSH — addr (assembling) ASSEMBLER
— n (run-time)

A constant which leaves (during assembly) the machine address of a routine which at run-time will add the accumulator (as high byte) and the bottom of the machine stack (as low byte) to the computation stack. It returns control to NEXT.

PUT — addr (assembling) ASSEMBLER
n1 — n2 (run-time)

A constant which leaves (during assembly) the machine address of a routine which at run-time will write the accumulator (as high byte) and the bottom machine stack byte (as low byte) over the existing top of the computation stack (16-bit value n1).

RP) — n (assembly-time) ASSEMBLER

Used in code definitions in the form:

```
RP) LDA,  or  RP) 3 +  STA,
```

Addresses the bottom, low order byte of the return stack by selecting the ,X mode and leaving n=$101 or maybe modified to another byte offset. Before operating on the return stack the x register must be saved in XSAVE and TSX, executed, and the x register must be restored before returning to NEXT.

SEC — n ASSEMBLER

Identical to BOT except that n=2. Addresses the low byte of the second 16-bit parameter stack value, the third byte on the parameter stack.

SETUP — ASSEMBLER

A subroutine which moves stack values to the N area. In this subroutine the accumulator specifies the quantity of 16-bit stack values to be moved (1-4). For example:

```
3 #LDA, SETUP JSR,
```

takes the first three pairs of bytes from the stack and moves them to N.

```
Stack before   N after    Stack after
    H                        H high byte
    G                    BOT G low byte
    F          F
    E          E
    D          D
    C          C
    B high byte B
BOT A low byte  A
```

THEN, — ASSEMBLER

A compiling version of ENDIF,.

UNTIL, — (run-time) ASSEMBLER
addr, 1, cc — (assembling)
A word in CODE definitions that is the terminator of loops, used in the form:
BEGIN, ... cc UNTIL,
At run-time UNTIL, controls the conditional branching back to BEGIN,. If the specific status bit specified by cc is false execution returns to BEGIN,, if true it 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) ASSEMBLER
A constant that returns the address of the pointer to the base of the user area. It is used in CODE definitions in the form:
UP LDA, or UP)Y STA, .

UPMODE addr f — addr f) ASSEMBLER
Used within the assembler to adjust the addressing mode based on operand size and op-code type.

VS — cc ASSEMBLER
Specifies that the following conditional generate a branch if the overflow bit of the status register is set.

W — addr (assembling) ASSEMBLER
A constant that returns the address of the pointer to the code field (execution address) of the FORTH dictionary word being executed. It is used in CODE definitions in the form:
W 1+ STA, or W 1- JMP, or W)Y ADC,
Indexing relative to W can yield any byte in the definition's parameter field. For example 2 # LDY, W)Y LDA, loads the first byte of the parameter into the accumulator.

X) — ASSEMBLER
Specifies indirect indexed with x addressing mode for the next op-code generated.

XSAVE — addr (assembling) ASSEMBLER
Used in code definitions in the form:
XSAVE STX, or XSAVE LDX,
A constant which leaves the address of a temporary buffer for saving the x register at assembly time. Since the x register indexes to the data stack in zero-page, it must be saved and restored when used for other purposes.

III.1 WTOBJ: WRITE BOOTABLE OBJECT

WTOBJ is a way to create bootable object disks. When the screens containing the WTOBJ words are loaded, the FENCE is reset to the top of the dictionary as it was just before the screens were loaded. Short instructions for using FORMAT and WTOBJ are printed out. To create an object code boot disk, you should FORMAT it (if it isn't formatted already), then put the disk in drive #1 and type WTOBJ. WTOBJ will write a short machine code routine (pointed to by BOOT) onto sector 1, followed (on succeeding sectors) by the contents of the dictionary up to FENCE. When this new disk is booted, the routine on the first sector will load in the rest of the boot. This version of WTOBJ will only create boot disks of 255 sectors or less.

CALLDK ---

Routine for moving a sector to or from disk. Assumes that the device control block has been set up.

DKIO addr n1 n2 ---

Instructs the disk drive specified by the lower byte of n2 to perform a disk operation on sector n1. The upper byte of n2 determines which operation (read, write, format, etc.) is performed. addr is the address of the 128 byte buffer used for this operation.

WTSEC addr n ---

Writes 128 bytes, starting at addr, to sector n of drive #1.

RDSEC addr n ---

Reads 128 bytes from sector n of drive #1 to memory, starting at addr.

FORMAT n ---

Formats the disk in drive n. First asks whether the user really wants to format, just in case.

BOOT --- addr

Points to the starting address in memory of the code routine that is put onto the first sector of a bootable object code disk (by WTOBJ). When this disk is booted later, this routine loads in the rest of the object code that was put onto the disk by WTOBJ.

WTOBJ ---

Creates a bootable object disk out of the FORMATted disk in drive #1. First puts the code routine that BOOT points to onto sector 1, then writes the contents of the dictionary (up to FENCE) to the disk. FENCE was reset to the top of the dictionary when the WTOBJ screens were loaded. When this new disk is booted, the code in the first sector will load in the rest of the boot.

III.2 DISK COPY ROUTINES

Coin-Op FORTH provides a set of disk copying routines in source code which can be loaded on top of the boot. The most important of these are CLONE , OBJ , and COPYDISK . For each of these, the source disk should be inserted in drive 1, and the destination disk in drive 2. None of them has any effect on the stack. Executing CLONE will copy the entire source disk. OBJ copies only the original Coin-Op boot screens (probably screens $ - B to -1, unless PHYSOFF or OFFSET have been changed by the user). If extra boot screens are added OBJ will not copy them. COPYDISK will copy the entire disk except for the Coin-Op FORTH boot screens.

III.3 Line Printer Words

Coin-Op FORTH provides a set of words for printing disk screens on the line printer. Although written for use with the Atari 822 printer, one small modification will make them suitable for use on an Atari 825 or an equivalent printer (such as the Centronics 739). This modification is as follows: on the second line of the first screen of source code for the printer words, the definition

```
3A50 VARIABLE P:
```

should be changed to

```
3250 VARIABLE P: 3A ,
```

This places the ATASCII characters P2: , rather than P: , into the parameter field of this variable. This change only needs to be made if you are not using an Atari 822 printer.

The printer should be turned on (as should any interface module used to connect the printer to the Atari) and switched to ON-LINE when the printing words are loaded, or else the printer will not be "opened". The command LPOPEN will open the printer after a SYSTEM RESET (or if the printer was not on line when the printing words were loaded).

There are three basic line printer routines available to the user. The first is LISTLP, which simply takes a disk screen number off the stack and prints that screen on the line printer. SHOWLP takes two arguments, a final screen m, and a starting screen n, and prints screens n to m on the line printer. If the number of characters per line on a screen is equal to 32, the screens will be printed in two columns; otherwise the screens will be printed in one column. Finally, LPINDEX takes the same arguments as SHOWLP, but prints only the first line of each screen.

The Atari words also allow the use of two different styles of print. After SHRINK is executed, files will be printed in condensed print. EXPAND returns the print size to normal.

.CLP n ---

Takes a number n from the stack and prints it at the line printer. If n is a single-digit number, it will print a preceding 0.

.LP n ---

Takes a number n from the stack and prints it on the line printer.

CRLP ---

CRLP checks LPCNT, and if the bottom of the page has been reached (if the number in LPCNT is greater than $3D) performs four carriage returns on the line printer to get to the next page.

EXPAND ---

After EXPAND is executed, files printed on the line printer will be printed with normal width characters.

FFLP ---

(Form feed.) FFLP performs successive carriage returns on the line printer until the top

of the next page has been reached.

LINELP n ---
Prints line n of the most recently referenced disk screen on the line printer.

LISTLP n ---
Prints screen n with line numbers on the line printer.

LPCNT --- addr
A variable which holds the number of carriage returns the printer has performed on a page.

LPCR ---
Performs a carriage return on the line printer and increments LPCNT.

LPEMIT n ---
Takes one ATASCII character from the stack and prints it on the line printer.

LPINDEX n, m ---
Prints line 0 of screens n through m on the line printer.

LPOPEN ---
OPENs a line printer device/file. This must be called before using any printer words.

LPSPC n ---
Prints n spaces on the line printer.

LYP1 addr, m ---
Sets the beginning of the line printer I/O buffer to addr with length m bytes, and prints the contents of the buffer.

LYPE addr, m ---
Prints m bytes of the buffer starting at addr on the line printer, followed by a space. If the buffer is longer than one line (80 characters) it increments LPCNT.

P: --- addr
A variable which contains the ATASCII equivalent of the characters P:, if an Atari 822 printer is being used, or P2: if an Atari 825 or equivalent printer is being used. P: is used as a buffer in opening the printer device/file.

PCIO --- n
Used after parameters are set for line printer I/O. PCIO sets up IOCB #7 and jumps to the CIO entry point ($E456). The I/O status is left on the stack.

PERR? n --

Takes the I/O status n from the stack and checks for error. If an error is found, PERR? executes ERROR.

SCR# -- addr

A variable which holds the ATASCII equivalent of the characters SCR #.

SHOWLP n, m --

Prints screens n through m on the line printer. If the number of characters per line on the screens is equal to 32, the screens will be printed in two columns on the page. Otherwise the screens are printed in one column.

SHRINK --

After SHRINK is executed, files printed on the line printer will be printed in condensed print.

III.4 FORMATTED PRINT WORDS
(Coin Op Screens)

These words are useful when you have to decipher compressed FORTH source code (you should rarely have to write such code yourself, since disk space is not expensive). The words allow you to list or print screens in a formatted fashion - they come out looking the way they should have in the first place.

The line printer words must be loaded before the formatted print words.

The main words are:

FLST	a formatted equivalent of	LIST
FLSTLP	- - - - - - - - - - - - - - - -	LISTLP
FSHW	- - - - - - - - - - - - - - - -	SHOW
FSHWLP	- - - - - - - - - - - - - - - -	SHOWLP

Each first lists or prints the screen in normal fashion, then in formatted form.

FSHWLP m n ---
For screens m through n, first prints then prints a formatted version.

FSHW m n ---
For screens m through n, first lists, then lists a formatted version on the TV screen.

FLSTLP n ---
Lists screen n on the printer, then prints the formatted version of the screen.

FLST n ---
Lists screen n on TV screen, then lists formatted version.

IV.1 MISCELLANEOUS WORDS

$ —

Causes the next word in the input stream to be interpreted as a hexadecimal number. At compile-time, the number is compiled into the next available dictionary space; at run-time, the number is left on the stack.

% —

Causes the next word in the input stream to be interpreted as a decimal number. At compile-time, the number is compiled into the next available dictionary space; at run-time, the number is left on the stack.

2DROP n m —

Drops the two top elements of the stack.

2DUP n m — n m n m

Duplicates the top two elements of the stack.

ATASCII>INTERNAL a — i

Code routine which takes the ATASCII value of a given character and returns the corresponding Internal value.

COLCRS — addr

A constant which returns the address that controls the full text screen text cursor's horizontal position (the split screen text cursor's horizontal position is controlled by TXTCOL).

INTERNAL>ATASCII i — a

Takes the Internal value of a character and converts it to the ATASCII value.

MAP — addr

Returns the address of the first entry in a table used in the conversion of characters from ATASCII to Internal representation, and vice versa.

MOVE addr1 addr2 n —

A "smart" form of CMOVE which moves bytes without danger of incorrect overwriting. Takes n bytes starting at addr1 and moves them to the n bytes starting at addr2. Written in machine code.

PICK xn ... x3 x2 x1 n — xn x1 x2 x3 ... xn

Takes the nth value from the stack, duplicates it, and puts it on top of the stack (a generalized form of OVER).

ROLL xn ... x3 x2 x1 n — xn x1 x2 x3 ...

Pulls out the nth value on the stack, xn, and puts it on top of the stack. The values which were above xn are pushed down one position to fill its place (a generalized ROT).

ROWCRS — addr

A constant which returns the address that controls the full text screen text cursor's vertical position (the split screen text cursor's vertical position is controlled by TXTROW).

SAVMSC — addr
A constant which contains the starting address of the graphics screen data.

SETBV — addr
A constant which returns the address of the vector into the OS routine SETVBV; this is used to set timers and all new routines to VBLANK. See the OS manual for details.

TXTCOL — addr
A constant which returns the address that controls the split screen text cursor's horizontal position.

TXTMSC — addr
A constant which contains the address of the beginning of the text screen data.

TXTROW — addr
A constant which returns the address that controls the split screen text cursor's vertical position.

VVBLKD — addr
A constant which returns the address of the deferred vertical blank vector.

VVBLKI — addr
A constant which returns the address of the immediate vertical blank vector.

IV.2 FAST INDEX

Use this exactly the same as regular INDEX. It is much faster since it looks at only the sector containing the top line of a screen, rather than reading in the whole screen. We have it compiled into our normal boot.

This word requires $ from the Miscellaneous Words.

Programmed by Charles A. Clinton

INDEX m n --
Writes the top lines of screens m to n on the TV screen.

Supporting words on the source screen:
DISKIO , INDEX_ONE_SECTOR , INDEX_ONE_SCREEN

IV.3 CASE

The CASE structure is a powerful branching system which offers an easy-to-use and easy-to maintain alternative to nested IF...THENs. The basic CASE structure is as follows

```
n CASE
    m1 OF xxx1 FO
    m2 OF xxx2 FO
         .
         .
         .
    my OF xxxy FO
    OTHERWISE OF xxxo FO
  ESAC
```

At execution, the entry value n is compared with m1, m2, ... my. If n is equal to some mi, the string of words xxxi is executed, and execution exits the CASE structure. If n is not equal to any of m1, m2, ... my, the OTHERWISE condition, xxx0, will be executed, and execution will leave the structure (if no OTHERWISE condition is included, ESAC will clean up the stacks, and execution continue to the next word in the definition). An important thing to note is that m1, m2, ... my can be either literals or set of words, provided they leave a number on the stack to be compared with n by OF.

EXAMPLE:

Suppose you need a program called M&MS that will keep track of the number of different colored m&ms in a bag as you eat them ,e.g.

```
3  BROWN  M&MS
```

will increment the variable BROWN by 3. This can be done using CASE as follows:

```
0  VARIABLE  BROWN
0  VARIABLE  YELLOW
0  VARIABLE  ORANGE
0  VARIABLE  GREEN

: M&MS            (n, addr ---)        CASE
     BROWN  OF  BROWN +! FO
     YELLOW  OF  YELLOW +! FO
     ORANGE  OF ORANGE +! FO
     GREEN  OF  GREEN +! FO
  ESAC ;
```

When, for example, the line:

```
2  GREEN  M&MS
```

is executed, the number 2 and the address of the variable GREEN are left on the stack. This address is first compared with the address of the variable BROWN, then that of YELLOW and ORANGE, and finally GREEN. Since the address of GREEN left on the stack outside the CASE structure will equal the address of GREEN left on the stack inside the CASE structure, the string GREEN +! will be executed, and the contents of GREEN will be incremented by two.

This can be a very useful program, since it is important not to eat too many green m&ms (they make you sterile).

GLOSSARY

$CASE ——

Run-time procedure compiled by CASE which places the following in-line parameter on the return stack. This parameter is the address later used by $FO to exit from the CASE structure.

$FO ——

The run-time procedure compiled by FO which causes execution to exit from the CASE structure.

$OF n m —— (if n=m), or
n m —— n (if n <>m)

The run-time procedure compiled by OF. This compares two values on the stack, and if they are equal, executes the words following $OF. Otherwise it leaves n on the stack and execution continues after the next $FO.

CASE

run-time: ——
compile-time: —— addr

Used in the form:

```
n  CASE
     m1 OF xxx1  FO
     m2 OF xxx2  FO
            .
            .
            .
     my OF xxxy  FO
    OTHERWISE OF xxxo FO
   ESAC
```

CASE opens the CASE branching structure which compares n to m1, m2, etc. If n is equal to mi, then the string of words xxxi will be executed and FO will cause execution to exit the CASE structure. Any mi may be a literal or a string of words which leave a value on the stack to be compared with n. If none of m1,... my are equal to n, the OTHERWISE string, xxxo, will be executed. Then execution continues after ESAC.
At compile-time, CASE compiles the run-time word $CASE, and reserves space at addr for use by ESAC.

ESAC

run-time: n ——
compile-time: addr ——

Used at the end of a CASE structure. At run-time, ESAC cleans up the return and parameter stacks if no match for the entry argument was made inside the CASE structure (if a match was made, FO would have caused execution to exit the structure, and ESAC would not be executed).
At compile-time, ESAC compiles R> and 2 DROPs to clean up the stacks, and stores the next available dictionary position into addr, which was left on the stack by CASE.

FO

run-time: ---
compile-time: addr ---

At run-time, FO causes execution to exit from the CASE structure if a match for the entry argument has been made. (See CASE.)
At compile-time, FO compiles the run-time word $FO and stores the address of the next available dictionary position into addr, which was left on the stack by OF.

OF

run-time: n m ---(if n=m)
or: n m --- n (if n<>m)
compile-time: --- addr

At run-time, OF compares two values on the stack (n was left on the stack before the CASE structure was entered; m was left on the stack inside the CASE structure). If these values are equal, the string of words, xxx will be executed, and the following FO will send execution out of the CASE structure. If the values are not equal, OF leaves n on the stack, and execution continues after the following FO.(See CASE.)
At compile-time, OF compiles the run-time word $OF, and reserves space at addr for use by FO.

OTHERWISE

n --- n n

Used in the CASE structure in the form:

OTHERWISE OF xxx FO

OTHERWISE will cause the string of words xxx to be executed. OF is expecting to compare two values on the stack and execute xxx if they are equal; OTHERWISE simply duplicates the value on the stack.

IV.4 DISPLAY LIST ASSEMBLER WORDS

These words allow the user to create ANTIC display lists in the FORTH dictionary. Read De Re ATARI chapter 2 (ANTIC and the Display List) before using these words.

These words have four advantages. First, because they employ mnemonics, creation and maintenance of display lists becomes much easier than would be the case if the user had to deal directly with the numerical ANTIC commands. This is roughly analagous to the difference between writing machine code via an assembler and writing it "by hand" - number by number. (Of course, in the case of both machine code and display lists, the user should have at least some understanding of the underlying numerical commands.)

The second advantage lies in the convenience of placing display lists in the FORTH dictionary, where they are created at compile time and can be easily accessed. This saves the user from the hassles of moving custom-built display lists in from disk or building them at run-time, and it avoids the problems of finding space for a display list between player/missile memory and display memory.

The words are quite easy to use. Bear in mind that, like FORTH assembler mnemonics, they operate in execution rather than compile mode. (This means that they can be placed inside other FORTH words to create "macros".)

A dictionary based display list starts with the command :DLIST , followed by the user's name for that diplay list (let us use the name DL.EX). :DLIST will create a FORTH header for DL.EX. Subsequent execution of DL.EX will leave its PFA on the stack; this in turn is the first byte of the new display list.

After the header, the diplay list is created. This is done through the use of the following four commands. (All of these commands have a , (comma) as the last character of their name; this indicates that they store a value onto the dictionary.)

JMP, - This command causes display list execution to "jump" to a new address (other than the next byte in memory), where the display list will continue. It has the effect of leaving one blank scan line on the display screen, and it is rarely used. JMP, is used in the form:

```
addr JMP,
```

where addr is the address to which the display list will jump.

BLANK, - This is used in the form:

```
n BLANK,
```

This command places between 1 and 7 blank scan lines on the screen. n indicates how many blank lines are desired. It is always used at the beginning of display lists (see De Re Atari), but it can be used anywhere the user desires.

MODE, - This is used to set up graphics mode lines. It is used in the form:

```
[address (of LMS)] [options] [mode #] MODE,
```

This creates an ANTIC instruction indicating one mode line of graphics mode [mode#]. It may also indicate one or more of these four options - LMS (load memory scan), VSCROLL (vertical scrolling enabled), HSCROLL (horizontal scrolling enabled), or DLI (display list interrupt). The option or options should be indicated before the mode line #. Finally, if one of the options is an LMS, the word LMS should be preceeded by the LMS address (i.e., the address of the start of display data).

For example,

```
$ B000 LMS VSCROLL  2  MODE,
```

will create an ANTIC mode 2 (graphics 0) mode line, with vertical scrolling enabled and an LMS which indicates display memory starts at $ B000. Another example:

```
6 MODE,
```

creates an ANTIC mode 6 (graphics 1) line.

The word OF-THEM is used to indicate that not one but several mode lines identical to the one most recently created should be placed in the display list. Thus, the command:

```
VSCROLL  7  MODE,  10  OF-THEM,
```

will compile 10 mode 7 (graphics mode 2) lines with vertical scrolling enabled; this avoids the necessity of repeating complicated (or even uncomplicated) instructions a number of times. OF-THEM, can also be used with BLANK,.

JVB, . This is used in the form:

```
[addr] JVB,
```

and indicates that the screen is completed, and as such display list execution should jump to [addr] and wait for vertical blank to occur. [addr] will almost always be the address of the first byte of the display list; this address can be obtained by executing the name of the display list:

```
DL.EX  JVB,
```

will thus cause display list execution to jump to the top of display list DL.EX.

A dictionary-based display list ends with the FORTH word DL; . This word checks to see if the display list crosses a 1K boundary (which is not allowed). If it does, a message to this effect is issued to the user, who must then either rearrange the source code or allot empty space in the dictionary so that the display list does not cross the boundary.

NOTE: There is one real disadvanage (other than the boundary problem just mentioned) with dictionary based display lists. Because they are created at compile time and used directly, it can be very difficult to "clean up" a display list after it has been altered in some way (for example, scrolling will change the LMS addresses). It may be necessary to store LMS addresses and the like in separate constants and to have a word which uses these constants to recreate the "original" display list.

After the display list is created, execution of its name will return its starting address. This address can then be stored into S-DLISTL ($230) in order to activate the display list.

Here are 2 examples of display lists (in decimal form):

```
:DLIST  DL.EX1
    7 BLANK,
    3 OF-THEM
    $ BC60 LMS 2 MODE,
    2 MODE, 23 OF-THEM,
    DL.EX.1 JVB,
DL;
```

(This creates a normal graphics mode 0 screen of 24 lines, the first of which has an LMS.)

```
:DLIST  DL.EX2
     7 BLANK,  3 OF-THEM,
     $ B000 LMS VSCROLL 7 MODE,
     VSCROLL 7 MODE, 9 OF-THEM,
     DLI 7 MODE,
     $ BF60 LMS 2 MODE,
     2 MODE, 3 OF-THEM,
     DL.EX2  JVB,
DL;
```

(This creates a 10-line scrolling graphics mode 2 screen with a four line graphics mode 0 text window; there is a DLI just efore the text window. See Se Re Atari chapter 6 for an explanation of why there are 11, rather than 10, ANTIC mode 7 instructions.)

The Display List Assembler words were programmed by Charles A. Clinton.

DISPLAY LIST ASSEMBLER GLOSSARY

:DLIST — addr

Used in the form:

```
:DLIST  xxxx ... DL;
```

:DLIST creates a dictionary entry for xxxx and sets MASK to 0, thus setting up the conditions for compiling a display list into the dictionary. ANTIC instructions can then be stored into xxxx's parameter field by using the display list assembler instructions; compilation of the display list is terminated by DL;. When xxxx is executed, it leaves its PFA on the stack; this is the address of the first byte of the display list, which can then be used to instruct ANTIC to use display list xxxx. At compile-time :DLIST leaves addr (xxxx's PFA) on the stack for subsequent error checking by DL;.

BLANK, n —

Compiles an ANTIC instruction in the dictionary. The instruction calls for n blank scan lines, where n = 1 to 7.

DL; addr —

Finishes compiling a display list by ensuring that the display list has not crossed a 1K boundary; if it has, a message to this effect is issued to the terminal, and interpretation of the input stream continues normally. addr is the address of the first ANTIC instruction in the display list, which is left on the stack by :DLIST.

DLI —

Adds $80 to the display list instruction currently being created (contained in MASK). This indicates that a display list interrupt will occur on the mode line corresponding to the instruction.

HSCROLL —

Adds $10 to the ANTIC instruction currently being created. This indicates that horizontal scrolling is enabled on the mode line corresponding to the instruction.

JMP, addr ---
Compiles an ANTIC JMP instruction, along with the address with which to reload the display list counter, into the dictionary. This instruction will leave one blank scan line on the screen.

JVB, addr ---
Compiles an ANTIC "Jump and wait for vertical blank to end" instruction, along with the address with which to to reload the display counter. This address is usually the starting address of the display list.

LMS ---
Adds $40 to the ANTIC instruction currently being created. This indicates that a "load memory scan" should occur on the mode line corresponding to the instruction.

MASK --- addr
A variable used to hold display list instructions temporarily as they are being created.

MODE, addr, n --- (LMS)
n --- (non-LMS)
Compiles an ANTIC instruction into the dictionary. The instruction indicates one mode line of ANTIC mode n, along with any other conditions (LMS, vertical and/or horizontal scrolling, DLI) preset in MASK. If the mode line contains an LMS, the new display data address is compiled immediately following the ANTIC instruction. MASK is reset to zero.

OF-THEM, n ---
Follows BLANK, or MODE,. Compiles n-1 ANTIC instructions, identical to the last one compiled before entry into OF-THEM, into the dictionary (e.g., 2 MODE, 23 OF-THEM will compile a total of 23 ANTIC mode 2 lines).

RESET ---
Sets MASK to 0, in preparation for the creation of the next ANTIC instruction.

VSCROLL ---
Adds $20 to the ANTIC instruction currently being created (contained in MASK). This indicates that vertical scrolling will be enabled on the mode line corresponding to the instruction.

IV.5 CHEAP CHARACTER GENERATOR

The word on the source screen is actually a template. To use just change the 8 x 8 matrix of 0's and 1's to form the character desired and rename the resulting word. Replace "beginning addr," by the beginning address of your character set (see the discussion below). To replace a character by your character, find its Internal number n in the Internal Character Set (Table 9.6 of the Basic Reference Manual), and do n <your name for word>. For example 0 ARROW-HEAD would replace "space" by the arrowhead.

This routine requires $ from the Miscellaneous Words.

Character set manipulation:

The highest two hexadecimal digits of the address of the beginning of the character set currently being used are stored in the variable CHBAS (normal value $E0), the other two digits of the address are 00.

To change the normal character set, select a memory address beg.addr, which lies on a 1K boundary (i.e. its last 2 digits must be 00), and for which you can afford to dedicate the following 1024 bytes to character set storage. Then CMOVE $400 bytes from $E000 to beg.addr. This gives you a changeable character set ($E000 to $E400 is in ROM). Change characters as described above. To make your new character set the current one, take the highest two digits of beg.char and do CHBAS C! After having modified a character set, you may wish to store it on disk.

IV.6 SEARCH

To find all appearances of a string xxxx on screens scr1 through scr2 inclusively, type:

```
scr1  scr2  SEARCH  xxxx
```

The computer will print out each line that contains the string, along with the line number and the screen number.

SEARCH will recognize leading, but not trailing blanks. Thus, letting "b" denote blank,

```
M n SEARCHbFF
```

would pick up both "BUFFER" and "$ FF" but

```
M n SEARCHbbFF
```

would only pick up "$ FF".

Supporting words on the source screens:
#LOCATE, #LEAD, #LAG, INFORM, TOP MATCH, 1LINE, FIND, COUNTER, BUMP.

Programmed by Jay Scott.

IV.7 PRINT COMMANDS FOR GRAPHICS MODES

The graphics printing words work exactly like the normal FORTH text printing words, except that they print characters in graphics modes. For each of three graphics modes (modes 1, 2 and 7), there are words that correspond to EMIT, TYPE, and ." , and words that position the cursor for printing. There are also counterparts to EMIT, TYPE, and ." for the text window (the parameters that position the cursor in the text window are given directly to T"). Since graphics characters are represented in memory in INTERNAL rather than ATASCII form, there are words that do what # , #S , HOLD , SIGN , and WORD do, except that they create a string in INTERNAL form instead of ATASCII. These words are used to format pictured numeric output. TO_INTERNAL will convert a string from ATASCII to INTERNAL representation. Other useful words in the graphics printing set include 7REVERSE, a variable which determines whether mode 7 characters should be printed in normal or inverse video form; and CLEAR-TEXT-WINDOW, which clears a portion of the text window.

Internally, the graphics printing words use SAVMSC and TXTMSC, which tell where the first bytes of graphics and text memory are, respectively. The mode 7 words use XPAND&COLOR, which takes an 8-bit slice of a character in the character set and widens it to 16 bits for printing in the higher resolution mode 7. (Mode 7 characters look exactly like mode 2 characters.)

These words require $ from the Miscellaneous Words.

The graphics printing words were programmed by Chris Helck.

GLOSSARY

#-I d1 — d2

From a double number d1, generates the next character (in Internal, not ATASCII, representation) which is placed in the pictured-numeric output stream. The double number d2 is the quotient after d1 is divided by BASE and is maintained for further processing. (Direct conterpart to #.) Use only between <# and #>.

#S-I d1 — d2

Converts all digits of the double number d1 to pictured numeric representation, adding each digit to the output text; the resulting text is in Internal, rather than ATASCII, representation. The double number d2 is the remaining quotient, and is always zero. (This is a direct counterpart to #S.) Use only between <# and #>.

(1") —

Run time code for 1" and 2". Prints the following in-line string (in Internal representation), with a count in the first byte, on a mode 1 or 2 screen. The string's color will be determined at run-time, on the basis of what color was most recently selected by the user.

(7") —

Run-time code for 7". Prints the following in-line text string (in Internal representation), with a count in the first byte, on a mode 7 screen; the string will be printed in the color most recently set.

(T") n1 n2 —

Execution procedure for T". Prints the following in-line text string, with the count in the first byte, in the text window starting at column n1, row n2.

1" —

Used in the form:

1" xxxx"

Compiles an in-line string xxxx, delimited by " (double quotes), with an execution procedure (1") to transmit the text to a mode 1 screen (in the color most recently set by a COLOR command). If executed outside of a colon definition, 1" immediately prints the string. The string may not exceed 255 characters.

1EMIT n —

Places the mode 1 character with internal representation n on the screen, at the current cursor position (see 1POSITION). The character will have the color most recently selected by the user. The cursor position is updated by 1 character.

1POSITION x y —

Sets the current cursor postion to x, y on a mode 1 screen; the next character printed will be placed at this spot.

1TYPE addr c —

Places the string beginning at addr, with character count c, on a mode 1 screen starting at the cursor position (see 1POSITION). The string will have the color most recently selected by the user. The string must already be in Internal representation.

2" —

Used in the form:

2" xxxx"

Compiles an in-line string xxxx, delimited by " (double quotes), along with the execution procedure (1") which transmits the text to a mode 2 screen (in the color most recently set). If executed outside of a colon definition, 2" immediately prints the string. The string may not exceed 255 characters.

2EMIT n —

Places the mode 2 character with Internal representation n on the screen, at the current cursor position (see 2POSITION). The character will be printed in the color most recently selected by the user. The cursor position is updated by 1 character.

2POSITION x y —

Sets the current cursor position to x, y on a mode 2 screen; the next character printed will be placed at this spot.

2TYPE addr n —

Puts the string beginning at addr, with character count n, on a mode 2 screen starting at the cursor position (see 2POSITION). The string must already be in Internal represenation, and it will have the color most recently selected by the user.

7" —

Used in the form:

7" xxxx"

Compiles an in-line text string, delimited by " (double quotes), along with the execution procedure (7") which, when executed, transmits the text to a mode 7 screen. The characters have the appearance of mode 2 characters; the full range of ATASCII characters is available to the user. The string will be in the color most recently set. If executed outside of a definition, 7" causes the string xxxx to be printed immediately.

7EMIT n —

Prints the character with Internal representation n on a mode 7 screen. The character will be in the color most recently selected by the user, and its upper-left corner will be at the graphics cursor. The cursor will be updated by 8 pixels. The character will have the appearance of a mode 2 character.

7POSITION x y —

Sets the graphics cursor to x, y on a mode 7 screen; the next mode 7 character printed will have its upper-left corner at this spot. Horizontally, characters must start on byte boundaries (every four pixels); vertically characters may start on any scan line.

7REL.SCRN! n1 n2 —

Stores the 16-bit value n1 into the address which is n2 bytes after the start of screen memory.

7REVERSE — addr

A variable which holds an indicator as to whether mode 7 characters should be printed normally or in inverse video. Contents=0 indicates normal, contents ◇ 0 indicates inverse.

7TYPE addr n —

Places the text string beginning at addr, with character count n, on a mode 7 screen, starting at the graphics cursor (see 7POSITION). The string will have the color most recently selected by the user; the characters will have the appearance of mode 2 characters. The string must already be in Internal representation.

CLEAR-TEXT-WINDOW n —

Clears the first n lines of the text window. Assumes that the starting address of the text window is stored in TXTMSC.

HOLD-I c —

Inserts the character whose Internal representation is c into the pictured numeric output string. For example, $E HOLD-I will place a decimal point into the next spot of the output text. (Direct counterpart of HOLD.) Use only between <# and #>.

REL-SCRN! c n —

Stores the 8-bit value c into the nth byte after the start of screen memory.

SIGN-I n d — d

Inserts the Internal value corresponding to "-" (minus sign) into the pictured numeric output string (if n is negative). (Direct counterpart to SIGN.) Use only between <# and #>.

T" — (compiling)
n1 n2 — (execution)
Used in the form:
n1 n2 T" xxxx".
Compiles the in-line text string xxxx, delimited by " (double quotes), along with the execution procedure (T"). When executed, the string will be placed in the text window at column n1, row n2. If executed outside of a definition, T" prints the string immediately.

TEMIT n —
Puts the character with ATASCII value n in the text window at the current cursor position, and updates the cursor position by 1 character.

TO.INTERNAL addr —
Converts the string beginning at addr, with a count stored in the first byte, from ATASCII to Internal representation.

TTYPE addr c —
Prints the ATASCII string starting at addr, with count c, in the text window beginning at the current cursor position.

WORD-I c —
Reads the input stream until a delimiter c is encountered, moving the text to HERE (with the character count in the first byte), and converts the text from ATASCII to Internal representation. (Direct counterpart to WORD.)

XPAND&COLOR n1 — n2
Expands an 8-bit value n1 (obtained from the character set) to a 16-bit value n2 for use on a mode 7 screen. When placed on a mode 7 screen, the bit pattern will reproduce 1 scan line of a character in the color currently selected by the user. This allows characters to be printed in mode 7 using the character set. (Each bit of n1 maps to two bits of n2; bit x of n1 maps to bits 2x and 2x+1 of n2; a 0 maps as 00, while a 1 maps as 00, 01, 10, or 11, depending on the current color.) If 7REVERSE is non-zero, the ones complement of n2 is returned, allowing the printing of "stencil" characters.

Non-boot words used: ATASCII>INTERNAL, INTERNAL>ATASCII, MAP.

IV.8 SCREEN COPY ROUTINES

The word CCOPY copies consecutive screens from some given range to screens starting at a new beginning number. It's a smart copy routine, i.e. the new and old screens can overlap.

CCOPY [1st] [last] [new 1st] ——
Copies screens [1st] to [last] to screens starting with [new 1st]

MCOPY makes multiple copies of a screen over some interval.

MCOPY [1st] [last] [scr. to be copied] ——
Makes copies of [scr. to be copied] over the screens in the interval from screen [1st] to screen [last].

IV.9 VERTICAL SCROLLING

Read chapter 6 (Scrolling) of De Re Atari for a discussion of scrolling on the Atari.

These words were created for use with Atari text modes. The following procedure explains how to use them.

1. Load the vertical scrolling words. $ must first be loaded from the Miscellaneous Words.

2. Create a display list, a portion of which has vertical scrolling enabled; this section should begin with an LMS command and it should not have more than one LMS. The section should include one more mode line than you wish to appear on the screen, and the very last display instruction in the scrolling section should not have vertical scrolling enabled. (Thus, if 10 mode lines of scrolling graphics mode 2 are desired, 11 mode 2 instructions should be placed in the display list, but only the first 10 should have vertical scrolling enabled. See De Re Atari.)

3. Set up the scrolling variables. These are:

VSCROLL.LMS
Store the address of the memory location in the display list which immediately follows the scrolling section's LMS (i.e., the address of the LMS address, which is the address of the start of the scrolling display data) into this variable.

SCAN.LINES/LINE
This variable should be set to the value which is one less than the number of scan lines per mode line in the graphics mode being scrolled. (Graphics mode 2 (ANTIC 7) has 16 scan lines/mode line, so the proper value for it is 15 ($F); graphics modes 0 and 1 (ANTIC modes 2 and 6) have 8 scan/lines per line; the value for them should be 7.)

BYTES/LINE
This variable should contain the number of diplay data bytes per mode line in the graphics section. This value is 20 ($14) for modes 1 and 2 (ANTIC modes 6 and 7), and 40 ($28) for mode 0 (ANTIC mode 2).

4. Clear the scrolling registers by executing the word CLEAR.SCROLL.REGS .

At this point, scrolling should be enabled and ready to go. Execution of the word UP will scroll the text one scan line up; execution of the word DOWN will scroll the text one scan line down.

Timing is crucial in scrolling. It must happen regularly and it must happen when the TV scan is beneath the graphics area of the screen (otherwise, the scrolling will jerk and the screen will flicker). For this reason, UP and DOWN make sure that changes to the scrolling registers occur only after the TV scan is beneath the screen by performing a delay loop; this prevents screen flicker. It serves also to make sure that scrolling occurs very regularly, as it can thus occur no faster than every 1/60th of a second (the time it takes to "draw" the TV screen), and as long as there are not too many routines between successive cells to UP and DOWN, it will occur no slower than every 1/60th of a second. (We have found that quite complicated tests can be perfomed between successive calls to UP and DOWN without slowing down the scrolling, so the user need not worry too much about this problem.)

The variables VSCROLL.COARSE.REG and VSCROLL.FINE.REG exist to allow checks as to how far in a given

direction scrolling has proceeded; the values in these registers can thus be used to stop scrolling at a given point. Bear in mind that both variables must be tested to see how far scrolling has progressed. (Note that, in addition to these two variables, the scrolling routines also change the LMS address in the display list and the hardware scroll register.)

Here is a typical scrolling routine:

```
: VERTICAL.SCROLL  ( -- )
  0 STICK DUP ( RETURNS STICK VALUE )
  $ E = ( UP )
  IF
    UP DROP
  ELSE
    $ D = ( DOWN )
    IF
      DOWN
    THEN
  THEN ;
```

This checks the joystick and scrolls up or down, depending on which direction the joystick is pressed. This word could then be used in a BEGIN ... UNTIL loop, or a similar loop, along with various tests of the scrolling variables, to allow the user to scroll over text, up and down, using the joystick.

The Vertical Scrolling words were programmed by Brock A. Miller.

GLOSSARY

BYTES/LINE -- addr

A variable which contains the number of data bytes per mode line in the current graphics mode. Must be set by the user.

CLEAR.SCROLL.REGS --

Sets the hardware and shadow vertical scrolling registers (VSCROL, VSCROLL.COARSE.REG, VSCROLL.FINE.REG) to 0. Used to initialize scrolling.

COARSE.DOWN --

Performs a coarse scroll by decrementing the LMS address in the vertical scroll section of the display list (pointed to by VSCROLL.LMS). It also decrements VSCROLL.COARSE.REG by 1 and sets the fine scrolling registers to their maximum values (contained in SCAN.LINES/LINE). The overall effect is to move text down one scan line on the screen.

COARSE.UP --

Performs a coarse scroll by incrementing the LMS address in the vertical scroll section of the display list (pointed to by VSCROLL.LMS). It also increments VSCROLL.COARSE.REG by 1 and sets the fine scrolling registers to zero. The overall effect is to move text one scan line up on the screen.

DOWN —

Scrolls text one pixel down in that portion of the screen which has vertical scrolling enabled. Performs a fine scroll or a coarse scroll, as needed.

FINE.DOWN —

Decrements the vertical scroll register (VSCROL) by one, causing the text on the screen to move one scan line down. Also decrements the shadow register (VSCROLL.FINE.REG). Waits until the TV scan is off-screen to change the registers, in order to avoid flicker.

FINE.UP —

Increments the vertical scroll register (VSCROL) by one, causing the text on the screen to move one scan line up. Also increments the shadow register (VSCROLL.FINE.REG). Waits until the TV scan is off-screen to change the registers, in order to avoid flicker.

OFF.SCREEN.WAIT —

A machine code routine which loops until VCOUNT, the TV scan line counter, indicates that the TV scan is below the graphics area; this is used before scrolling words to ensure that scroll registers are not changed while the scan is on the screen.

SCAN.LINES/LINE — addr

A variable which contains a value which is one less than the number of scan lines per mode line in the current graphics mode. Must be set by the user. (e.g., for graphics modes 0 and 1, the value is 7; for graphics mode 2, it is 15.)

UP —

Scrolls text one pixel up on that portion of the screen which has vertical scrolling enabled. Performs a fine scroll or a coarse scroll, as needed.

VSCROL — addr

A constant which contains the address of the hardware vertical fine scroll register ($D405). This is a one byte, write-only register.

VSCROLL.COARSE.REG —addr

A variable which contains the number of coarse scrolls performed relative to the starting point of the scrolling.

VSCROLL.FINE.REG —addr

A variable which is used to shadow the hardware vertical fine scroll register (VSCROL), which is a write-only register.

VSCROLL.LMS — addr

A variable containing the address in memory of the LMS address which is to be changed by the scrolling routines (i.e. the address of the first byte of scrolling memory).

IV.10 PLAYER-MISSILE GRAPHICS

The words presented here are easier to use and make code more readable and maintainable than simply writing to player-missile and graphics registers. Nonetheless, the chapter in De Re Atari on player-missile graphics should be read carefully before using these words.

The first steps in setting up player-missile graphics are to choose an address for the beginning of player-missile memory, choose either single- or double-line resolution, and set the appropriate bits in DMACTL and GRACTL. All of this is done by SET-P&M. SET-P&M takes two arguments off the stack; the resolution (1 for single-line, 2 for double-line), and the address of the start of player-missile memory, SET_P&M then enables the players in DMACTL and GRACTL, sets the variables MISSILE_0, PLAYER_0, etc., and clears player-missile memory.

With player-missile graphics set up, the next step is to get something on the screen. Since the Atari provides so many special features, it may take several steps to form any particular image on the screen. The first step is to store the image in player-missile memory (see De Re Atari page 4-3). Setting the horizontal position of players and missiles can be done using HPOS!. HPOS! takes two arguments off the stack. The first is the player or missile number - 0 through 3 for players 0 through 3; 4 through 7 for missiles 0 through 3. The second is the horizontal position, from about $ 50 for the left-hand side of the screen to $ 200 for the right-hand side. The widths of the players and missiles are controlled by SIZE!, which also takes two arguments off the stack. The first is again the player or missile number. This is 0 through 3 for players 0 through 3 respectively, but, unlike HPOS!, all the missiles are controlled by 4. This is because the sizes of all the missiles are controlled by one register, SIZEM, whereas each has its own horizontal position register. Bits 0 and 1 of SIZEM control the size of missile 0, bits 2 and 3 missile 1, etc. The second argument that SIZE! takes from the stack is the width; 0 or 2 for single-width, 1 for double-width, or 3 for quadruple-width. If SIZE! is used to set the width of the missiles, they must all be set at once. For example:

```
$ AB  4 SIZE!
```

will set missile 0 to quadruple width, and the others to single-width.

The priority of objects on the screen (which object will appear to be "in front" when two objects overlap) can be set by PRIORITY, which takes a number from 0 to 15 off the stack and stores it in the first four bits of GPRIOR. There are four basic priorities; 1 PRIORITY gives all four players priority over the four playfield colors; 2 PRIORITY gives players 0 and 1 priority over the playfield colors, which then have priority over players 2 and 3; 4 PRIORITY gives all four playfield colors priority over the players; 8 PRIORITY gives playfield colors 0 and 1 priority over the players, which have priority over playfield colors 2 and 3 (background color is always lowest priority). If priorities other than these four are used, objects with conflicting priorities will appear black in regions where they overlap.

The colors of players and missiles can be set by COLPM!. COLPM! takes two arguments; the first is the player number, 0 through 3 (missiles take the colors of their respective players); the second determines the color, 0 to $ FF. Fifth and sixth player-missile colors can be enabled using COLORED-P&M-ON (no stack effect). After COLORED-P&M-ON is executed, where players 0 and 1, or players 2 and 3 intersect, their colors will be ORed.

Use of the fifth player can be enabled by executing FIVE-PLAYERS (no stack effect). Note that FIVE-PLAYERS only assigns the color of playfield color 3 to all the missiles; they must still be positioned together by the programmer.

ERASE-P&M will fill all of player-missile memory will zeros, but an easier way to remove players and missiles from the screen is to set their horizontal positions to zero.

GLOSSARY

AND! n addr —
Performs the logical and of the contents of addr and n, leaving the result as the new contents of addr. Used with both shadowed and unshadowed players and missiles.

COLORED-P&M-OFF —
Disenables fifth and sixth player-missile colors.
Used with both shadowed and unshadowed players and missiles.

COLORED-P&M-ON —
Enables the use of fifth and sixth player missile colors. When this is called, where player 0 and player 1 intersect, and where player 2 and player 3 intersect, their colors will be ORed.
Used with both shadowed and unshadowed players and missiles.

DISENABLE-MISSILES—
Disenables the missile graphics. Used only with shadowed players and missiles.

DISENABLE-P&M —
Disenables player-missile graphics.
Used only with shadowed players and missiles.

DISENABLE-PLAYERS—
Disenables the player graphics.
Used only with shadowed players and missiles.

ENABLE-MISSILES —
Enables the missile graphics.
Used with both shadowed and unshadowed players and missiles.

ENABLE-P&M —
Enables player-missile graphics.
Used with both shadowed and unshadowed players and missiles.

ENABLE-PLAYERS —
Enables the player graphics.
Used with both shadowed and unshadowed players and missiles.

ERASE-P&M —
Stores zeros into all of player-missile memory.
Used with both shadowed and unshadowed players and missiles.

FIVE-PLAYERS —
Assigns the color of playfield color 3 to all the missiles allowing them to be moved together as a fifth player(otherwise, the missiles take the colors of their respective players).
Used with both shadowed and unshadowed players and missiles.

FOUR-PLAYERS —

Assigns each missile the color of its respective player.
Used with both shadowed and unshadowed players and missiles.

FROM-SHADOW-P — addr

A variable which holds the address of the code routine attached just before SHADOW-PLAYERS. The parameter field of FROM-SHADOW-P must not cross a 1K boundary.

GENTLE-GR. graphics # —

Switches graphics modes without bothering the players and missiles.
Used with both shadowed and unshadowed players and missiles.

GRAFP0 — addr

A constant that returns the address of player 0's graphics register. Used with both shadowed and unshadowed players and missiles.

HPOS! x n —

Moves player n to the horizontal position x on the screen.
This word is used both with shadowed and unshadowed players and missiles.

HPOSP0 — addr

A constant that returns the address of player 0's horizontal position register. Used only with shadowed players and missiles.

MISSILE_0 — addr

A variable which contains the address of the beginning of missile memory.
This word is used in both shadowed and unshadowed players and missiles.

NOT n — m

Returns the ones complement of the top of the stack.

OR! n addr —

Performs a logical or on the contents of addr and the value n. Leaves the result as the new contents of addr. Used with both shadowed and unshadowed players and missiles.

PLAYER_0 — addr

A variable which holds the address of the beginning of player 0's memory. Similarly for PLAYER_1, PLAYER_2, etc.
These words are used with both shadowed and unshadowed players and missiles.

PLAYER_4 — addr

A variable which contains the address of the beginning of missile memory. This is the same as MISSILE_0, but it will make code clearer if this is used when FIVE-PLAYERS is enabled.
This word is used with both shadowed and unshadowed players and missiles.

PM-RESOLUTION n —

Sets player-missile resolution; takes either n=1 for single-line resolution or n=2 for double line.
This word is used with both shadowed and unshadowed players and missiles.

PMBASE-SHADOW -- addr
A variable which holds the page number of the beginning of player-missile memory. This word is used with both shadowed and unshadowed players and missiles.

PRIORITY n --
Sets the bit in GPRIOR which controls the priority of objects on the screen. An object with higher priority will appear to be "in front" of an object with lower priority. Used with both shadowed and unshadowed players and missiles.

S-HPOSP0 -- addr
A constant which contains the "shadow" registers for the player-missile horizontal position registers HPOSP0, HPOSP1, etc. Used only with shadowed players and missiles.

S-SIZEP0 -- addr
A constant which contains the "shadow" registers for the player-missile size registers SIZEP0, SIZEP1, etc. Used only with shadowed players and missiles.

SET-P&M addr, n --
Enables player-missile graphics, setting the start of player-missile memory to addr, and the resolution to n. SET-P&M also sets variables MISSILE_0, PLAYER_0, PLAYER_1, etc., and clears player-missile memory. Used with both shadowed and unshadowed players and missiles.

SETUP-SHADOW-P&M --
Attaches SHADOW-PLAYERS to the immediate VBLANK routine. SETUP-SHADOW-P&M must be re-enabled after a reset. This word is only used with shadowed players and missiles.

SHADOW-PLAYERS --
A VBLANK routine that reads values from the "shadow" registers S-HPOSP0 and S-SIZEP0 to the player-missile horizontal position and size registers. SHADOW-PLAYERS also shadows the graphics control register GRACTL in S-GRACTL. This word is only used with shadowed players and missiles.

SIZE! size, n --
Changes the width of player n. Width will be either single width (size=0 or 2), double width (size=1), or quadruple width (size=3). This word is used both with shadowed and unshadowed players and missiles.

WAIT-VB --
Waits for the next vertical blank and returns control after interrupt ends. Used only with shadowed players and missiles.

IV.11 SHADOWED PLAYERS AND MISSILES

Chris Helck of the Swarthmore Trigonometry Project designed a set of "shadowed" player-missile words to correct two problems which occur when using unshadowed players and missiles. The first of these happens when display list interrupts are used to display a player in two (or more) parts on the screen. With unshadowed players, this requires two display list interrupts, one in the middle of the screen to change the players horizontal position, and one at the bottom or top of the screen to set it back to its original position. Shadowing the horizontal position registers makes only one display list interrupt necessary. The second problem is that HPOS! writes directly to the horizontal position register. As a result, if it is being called while a player is being drawn, the bottom part of the player will move before the top, producing, briefly, a ragged line, rather than a straight line. Shadowing the players removes this problem. In addition to shadowing the horizontal position registers, SHADOW-PLAYERS also shadows the size registers and GRACTL.

In order to use the shadowed players, the only extra step is to call SETUP-SHADOW-P&M (no stack effect) after each RESET. All of the un-shadowed player-missile words have the same function in the shadowed words.

The vertical blank routines must be loaded before loading the shadowed player-missile words.

IV.12 TEXT WORDS FOR DISK STORAGE OF TEXT

The text words provide a convenient method of storing blocks of text on disk and moving them easily into memory. The text is included in the definition of words defined by :T (for mode 0 text) or :T12 (for modes 1 or 2 text). The text must begin on the line following the :T (or :T12), and the end of the text is signaled by a semi-colon at the beginning of a line. For example;

```
:T GEORGE
This is the text which
is being stored in this
example called  GEORGE
;
```

The text in words defined by :T12 should not have more than 20 characters to a line; characters beyond the 20th are ignored.

The text can be any number of lines long. If the text will not fit entirely on one screen, simply write to the bottom of the first screen and continue at the top to the next screen. Do <u>not</u> use a "—>" .

When a text word is compiled (i.e. the screen containing it is loaded), the definition in the computer memory does not contain the actual text. Rather, it contains a pointer to the screen on which the text is stored. Thus, when the word is executed it is essential that the disk containing the text is in the disk drive.

When a text word is executed it expects an address on the stack, and it moves the text from disk to memory beginning at the given address. Since the primary use of the text words is to move text on to the video display, there are three words designed to return the appropriate address.

n WINDOW-LINE returns the address of the nth line of the text-window (assuming one is in use).

n SCREEN-LINE returns the address of the nth line of a mode 0 screen, while

n SCREEN-LINE12 does the same for mode 1 or 2 screens. In all cases the top line is number 0. Thus

```
3 SCREEN-LINE GEORGE
```

would put the text in the definition of GEORGE on a mode 0 screen, beginning at line 3 (i.e. 4th line from top).

To increase flexibility, the pointers that are compiled into text words are not the absolute screen numbers on which the text appears, but are instead offsets from the screen-number stored in the variable TEXT-BASE. It is therefore necessary to put the word INITIAL-TEXT-SCREEN at the top of the first screen which contains words defined by :T or :T12. This sets the variable TEXT-BASE to the correct value.

The advantage of this system comes from the fact that it is sometimes desirable to load a program from one disk and store the text on a seperate disk (this is especially true if the program is precompiled). To do this, simply reproduce the screens containing text words on the text disk. Make sure that they are in the same order and relative position as they are on the program disk. It is not necessary, however, to put the text on the same numbered screens. Instead, store the number of the first text screen (the one that contains the word INITIAL-TEXT-SCREEN) in the variable TEXT-BASE.

It is sometimes desirable to bring a screen containing text into the buffer at some point before the text is actually used (so that the disk drive does not have to operate at that point). This is done using the word TEXT->BUFFER. For example, if GEORGE is a text word, the expression

```
: BUFFER-EXAMPLE
    ' GEORGE TEXT->BUFFER
;
```

will bring the screen containing GEORGE's text into a buffer. Note the format carefully; first a ' (tic), then the text word, then TEXT->BUFFER.

One final note: all of the text words are designed to use 4 space margins in mode 0 (32 character lines). This is particularly easy to work with when the QE editor is employed.

The Text Words were programmed by David Fristrom.

GLOSSARY

TEXT->BUFFER pfa ——

Calls from disk to buffer the screen containing the text word with the given pfa. Usually used in the format:

```
' XXXXX TEXT->BUFFER
```

where XXXXX is a word defined by :T or :T12.

MODE0 ——addr

A variable used as a flag by the text words.

(:T) flg ——

The main component of :T and :T12. A defining word. Compiles flg (1=mode 0, 0=mode 1 or 2), screen number of text (counting from TEXT-BASE), and line number of first text line. During run time, words defined by (:T) move text from disk to memory, beginning at the address on the stack (usually some part of screen memory).

:T ——

A defining word, used in the format:

```
:T XXXXX
The text which you want to store on disk
;
```

Note that the semi-colon must be at the beginning of its line. :T compiles into the definition of XXXXX a flag to indicate the text is in graphics mode 0, the number of the disk screen the word is defined on (counting from TEXT-BASE), and the number of the screen line on which the text begins. Words defined by :T are used in the format:

```
addr XXXXX
```

This moves the text from the disk to memory, starting at addr.

:T12 ——

:T12 is the graphics modes 1 and 2 equivalent to :T. It is identical to :T except it compiles a flag indicating mode 1 or 2 rather than mode 0. Because mode 1 or 2 characters are wider than mode 0, the text should not exceed 20 characters to a line.

INITIAL-TEXT-SCREEN ——

Stores the value of BLK on TEXT-BASE. It should be used at the beginning of the first source code screen which contains definitions of text words defined by :T or :T12.

SCREEN-LINE n--addr
Returns the address of the nth line of a graphic mode 0 screen (with top line being zero).

SCREEN-LINE12 n--addr
Returns the address of the nth line of a graphic mode 1 or 2 screen (with top line being zero).

TEXT-BASE --addr
A variable used by the text words to contain a pointer to the first disk screen which has words defined by :T or :T12 on it.

WINDOW-LINE n--addr
Returns the address of the beginnng of the nth line of text window (with top line being zero).

IV.13 TEXT COMPRESSION TEXT WORDS

Effective use of text on the TV screen requires a great deal of waste space. Readability is enhanced by writing in short phrases rather than 40 character lines, by small paragraphs, generous indentations, and by writing only on every other line. The Text Compression Text words allow you to get by with this without exacting a price in disk storage - typically we have 1/3 to 1/4 as many compressed text screens as "source" text screens. These words also allow the same handy manipulation of text as do the regular Text Words.

In general, you would use the regular Text Words when developing an application (please see Section IV.12). Text Compression text Words are used in exactly the same fashion as regular Text Words, and no changes need be made with two exceptions: (1) The word END-TEXT must be added immediately following your last :T or :T12 word. (2) You can force the usage of a new text screen (in order, for example, to avoid disk operations at an infelicitous time) by inserting the word NEW-SCREEN before a :T or :T12 word.

To make up a disk using the Text Compression Text Words, just load those words (FORGETting the regular :T words, if necessary). Then choose a screen number for the first screen of compressed text and store it in TEXT-BASE. Subsequent screens will be filled downward from this - we fill disks with compiled code upward and compressed text downward. If you prefer your screens to be filled in increasing order, instead, there is an obvious change you can make in MOVE.TO.DISK and one in NEXT.SCREEN.

Now load your :T and :T12 words. Various messages will inform you of the progress of the compilation, terminating in "END TEXT COMPILATION". Proceed to load your non-text words, as in the regular Text Word context.

Behind the Scenes

The Text Compression Text Words work as follows. When compiled there is an Encode Buffer which requires $420 bytes starting at EBUFF.START (currently set to $A400 - change if you're already using that area of memory). Compressed text is collected in the Encode Buffer in contiguous strings of the form:

bbnnccc...c

where bb is the number of blanks to be written to screen, nn is the number of characters following, and cccc...c are those characters in Internal form. Thus, decoding consists of writing the specified number of blanks and then CMOVEing the character string to screen memory. Setting the number of blanks bb to -1 is used to signify that the next screen of compressed text is to be brought in. For example when the Encode Buffer has been filled everthing up to the overhanging string is sent to the FORTH buffer and a -1 is tacked on at the end.

The end of a :T or :T12 word is indicated by placing a 0 in the "number of characters" slot nn. A text word defined by :T or :T12 returns an address to a table containing the screen offset, character offset in the screen, and a Mode 0? flag.

These words require $ from the Miscellaneous Words.

GLOSSARY

#C ——addr
Variable which holds the number of characters which are neither leading nor trailing blanks in a line of text on a QE screen.

#L —— addr
Variable which holds the number of leading blanks in a line of text on a QE screen.

#LEADING addr —— count
Calculates the number of leading blanks in the text line starting at the given buffer address.

#T —— addr
Variable which holds the number of trailing blanks in a line of text on a QE screen.

(;T) compile: flag ——
run: addr——
At compile time this defining word encodes a ;T word for Mode 0 if flag is 1, and Mode 1 or 2 otherwise. At run time, the word is decoded starting at screen memory given by addr.

;T ——
Defining word which at compile time encodes text word with line length and margins set for Mode 0. At run time the text word decodes at the memory address on top of the stack.

;T12 ——
Defining word which at compile time encodes text word with line length and margins set for Mode 1 or 2. At run time the text word decodes starting at the memory address on the top of the stack.

B.LINE ——
Encodes a line of blanks (i.e. adds the number of blanks to the value at M.PTR and updates B.PTR).

B.PTR —— addr
A variable which during encoding holds the address in the Forth disk buffer (between $500 and $CFF) of the next text character to be encoded. During decoding it points in the Forth disk buffer to the next text character, or to numeric information between character strings.

DECODE addr flag s.off c.off —
Takes screen given by offset s.off and reads to Forth buffer, sets B.PTR from BLOCK and offset c.off. DECODE.LINE starting at addr until EOT? is set to 1. Compiled for Mode 0 if flag is 1.

DECODE.LINE —
Writes on the TV screen the number of blanks stored at B.PTR, then CMOVE the number of characters stored at B.PTR+2 and starting at B.PTR+4. If 0 characters, set EOT? to 2.

EBUFF.START — addr
Constant pointing to beginning of the 1K Encode buffer.

ENCODE —
Encodes text of a ;T word until a ";" is encountered as the first character in a line on a QE screen.

ENCODE.CHARS n —
Moves next n characters from Forth buffer to Encode buffer, and transforms from ATASCII to INTERNAL. Increases pointer in both buffers by n.

ENCODE.LINE —
Encodes a line of text from the Forth buffer and moves IN to the next line.

END.ENCODE —
Stores a 0 in the "number of characters" position in the Encode buffer and moves M.PTR in positon for new word.

END-TEXT —
Placed by user after last ;T word to send final contents of Encode buffer to drive. Prints "END TEXT ENCODING" on TV screen.

EOT? — addr
Variable containing flag which indicates the end of a ;T word.

INITIAL-TEXT-SCREEN —
Placed by user prior to ;T text words to initialize S.OFFSET and M.PTR. When compiled it prints message "START TXT ENCODING".

LINE.LENGTH — addr
Variable indicating the length of the line of QE screen to be decoded (32 for Mode 0, 20 for Modes 1 and 2).

L.MARGIN — addr
Variable containing the number of characters in the left margin of the TV screen for decoded text (0 for Modes 1 and 2, 4 for Mode 0).

MOVE.TO.DISK —
Moves encoded text from Encode buffer to Forth buffer and UPDATE. Prints message "TS n" on the video screen and subtracts 1 from the contents of S.OFFSET.

M.PTR — addr
Variable used during encoding to store the current offset position from EBUFF.START in the Encode buffer. During decoding it points to the next byte in screen memory.

NEXT.SCREEN —
Called by DECODE to read in the next encoded text screen from disk if the "number of blanks" position in a encoded line is negative.

NEW-SCREEN —
Placed by user before a :T word (or :T12 word), this will cause the word to be encoded on a new text screen. Inserted by user if previous compilation of :T words has resulted in inconvenient disk read (e.g. in the middle of the following word).

NXT.SCR? —
Checks whether B.PTR is beyond the Forth buffer in which it started and, if so, brings in the next text screen to reset B.PTR.

NONB.LINE —
Encodes a non-blank line of text and updates pointers.

OVERFLOW —
Takes the Encode buffer up to the current line and MOVE.TO.DISK. Moves the current encoded line to th top of the Encode buffer and sets M.PTR to the end of the line.

R.MARGIN — addr
Variable containing the number of characters in the right margin of the TV screen for decoded text (0 for Modes 1 and 2, 4 for Mode 3).

SCREEN-LINE n — addr
Returns memory address addr of line n of Mode 0 screen memory (n = 0,1,2,...). Thus,if TEXT is the name of a :T word, then:

```
2 SCREEN-LINE TEXT
```

will decode TEXT starting at line 3 of the TV screen.

SCREEN-LINE12 n -- addr
Returns memory address addr if line n of mode 1 or 2 screen memory (n = 0,1,2,...). Thus, if TEXT12 is a :T12 word then:
2 SCREEN-LINE12 TEXT12
Will decode TEXT12 starting at line 3 of the TV screen.

SET.MODE flag --
The flag is 1 for Mode 0, 0 for Modes 1 and 2. L.MARGIN, R.MARGIN, and LINE.LENGTH are set accordingly.

S.OFFSET -- addr
Variable giving offset from current screen to TEXT-BASE.

TEXT-BASE -- addr
A variable set by the user to the first (and highest numbered) screen to be used for text storage.

WINDOW-LINE n -- addr
Retruns memory address addr of line n in text window (n = 0,1,2,or 3) for the standard 4 line text window). Thus, if TEXT is the name of a :T word then:
2 WINDOW-LINE TEXT
will decode TEXT beginning at line 3 of the text window.

IV.14 VBLANK ROUTINES

See the Atari OS manual for more information on the VBLANK process.

The Swarthmore Trigonometry Project VBLANK attaching words were developed to solve a problem which may occur when attaching code routines to the immediate or deferred VBLANK routines - by storing addresses in VVBLKI or VVBLKD. IF VBLANK occurs while the routine is being attached, it is possible that only half of the address of the routine will be updated, and the machine will crash. These words are a safeguard against this problem.

To attach a routine to the immediate or deferred VBLANK routines, first be certain that the last thing your routine does is jump to the previous initial VBLANK routine (the address of this routine can be fetched from VVBLKI for immediate or VVBLKD for deferred routines). Then, put the address of your routine on the stack and execute either START-DEFERRED-VBV or START-IMMEDIATE-VBV. To detach routines, simply execute either NO-IMMEDIATE-VBV, or NO-DEFERRED-VBV. Note, however, that these will only detach the routines which were atached after IVBLKI and IVBLKD were compiled, i.e. after the VBLANK screens were loaded.

GLOSSARY

START-DEFERRED-VBV addr ——
Attaches the machine code routine starting at addr to the deferred VBLANK routines. The new routine will now be the first deferred VBLANK routine to execute. It is assumed that the new routine already ends with a jump to the previous initial deferred routine; otherwise, the system will crash.

START-IMMEDIATE-VBV addr ——
Attaches the code routine starting at addr to the immediate VBLANK routines. The new routine will now be the first VBLANK routine to execute. It is assumed that the new routine already contains a jump to the previous initial VBLANK routine.

IVBLKD —— addr
Returns the starting address of the code routine which was the first deferred VBLANK routine at the time that IVBLKD was compiled.

IVBLKI —— addr
Returns the starting address of the code routine which was the first immediate VBLANK routine at the time IVBLVKI was compiled.

NO-DEFERRED-VBV ——
Detaches all deferred VBLANK routines from the VBLANK process, except for OS routines and routines which were attached before IVBLKD was compiled. See IVBLKD.

NO-IMMEDIATE-VBV ——
Detaches all immediate VBLANK routines from the VBLANK process, except for OS routines and routines which were attached before IVBLKI was compiled. See IVBLKI.

IV.15 Big WTOBJ

This version of WTOBJ works exactly the same way as the version described in Section III.1, except that it can create boots of more than 255 sectors. The source code for this version is also in a much more readable form.

IV.16 INTERRUPT-DRIVEN SOUND

The word SOUND is adequate for many sound generating purposes, but in some instances it leaves much to be desired. Specifically, if the sound to be produced is to vary dynamically (change volume, pitch, or timbre over time) or if it has to persist for only a certain amount of time, then the processor will be tied up and will be unable to do anything else during the duration of the sound. This means that (for example) dynamic sound and graphics are impossible to produce simultaneously with SOUND.

If interrupt-driven sound routines are used, however, this limitation disappears. A basic interrupt-run sound system consists of several words. There is a word that takes parameters for a note from the user and loads these parameters into sound variables. These variables correspond to pitch, timbre, volume, and duration (for each of the four sound channels). Another word (written in assembler) is set up to run in VBLANK. This word moves the sound parameters from the variables to the hardware sound registers. If a duration parameter is being used, this interrupt routine would also decrement a duration counter every VBLANK (i.e., every 1/60 second) and shut off a channel when that counter reached zero. A more involved sound routine that read from a table of sound parameters would allow timbre or volume changes during the course of a note.

See De Re chapter 7 for more information on sound routines.

IV.17 QE USERS MANUAL

Introduction

QE is a full-screen text editor, designed and programmed by Charles A. Clinton. Written in FORTH, it was created primarily for the entering and manipulation of FORTH text, but it can easily be used to enter and manipulate text for other purposes. It is large (about $2200 bytes of memory), but we find it sufficiently convenient that we include it in our boot. We provide both source code and a glossary, so enterprising programmers can modify the editor to their own tastes.

Because of the size of the Atari screen, 24 rows of 40 columns, one disk "text screen" of 1024 characters cannot fit entirely on the editor screen. The editor will only display 22 lines of text, but will allow you to enter data on 32 lines. The two extra rows at the bottom of the screen are reserved for the prompt line, explained below.

The editor screen can be scrolled over the text screen. When you scroll the screen up, the text at the top will disappear, but it has not been lost as it would be with the fig EDITOR.

Loading QE

The disk "Swarthmore QE" contains QE in the boot.

To load QE if it is not in the boot, it is necessary to first load $, ATASCII>INTERNAL, and MOVE from the Miscellaneous Words, IV.1. Then, one must add on IV.3 CASE, IV.4 Display List Assembler, and IV.10 Unshadowed Player Missile Words. Then the source code for QE can be loaded.

Beginner's Walk Through

This section contains the basic instructions which you will need. Further commands, are explained in later sections of this manual. Let us suppose that you have booted with a disk which has QE in the boot and that you have inserted a formatted disk to use for practice. To enter QE, merely type in QE and press return. The screen will flash, and there will appear a line at the bottom of the screen which says, something like "QE 0.1.1 18-June-82". The number at the right end of this prompt line is the screen that you are on . In order to get to another screen, first type the letter S.

The prompt line will now say, "Re Wr Ne Lk Ulk ?" and will have the screen number at the end of the line. Type in the new number you want, 45 for instance. The number 45 will replace the earlier number. Now type in "R" to read that screen from the disk. After a moment while the screen is being read, the bottom line will change to "Command". Now, if for example you want to clear the screen, type "T", then "D", then "A". This will move your cursor to the top of the screen, put you into Delete mode, and then delete the rest of the screen by filling it with spaces.

After having cleared the screen let us enter new text in Insert mode. Type in "I". The word "Insert" will appear on the prompt line, and the borders of the screen will turn red. These will stay red as long as you are in either Insert or Overwrite modes. Now, if you enter text, it will appear on the screen as if you were using a typewriter.

Until you use up all 32 lines of the FORTH screen, a new blank line will appear when you reach the bottom of the page, and the top line will disappear. You will know you've completely filled a

screen when no more blank lines appear.

Overwrite mode is sometimes better than Insert mode for editing existing text, so try switching modes, either by holding the CONTROL button down while typing "O", or by hitting the ESC button, which will put you back in Command mode (notice the bottom line), and then typing "O". In either case, "Overwrite" will appear on the bottom line. You can move your cursor about with the arrows, used while holding the CONTROL button down, or by other commands explained in the Insert and Overwrite section of this manual. When the cursor is in the correct place, you can type over the existing text.

When you are in Insert mode, new characters are inserted between the cursor and the next character. In Overwrite mode, hitting the carriage return will take you to the next line; in Insert, it will create a new line below the present one.

When you are finished your editing, press the ESC button to return to Command mode. From here, go back again to Screen mode by typing "S". If you now type "W", the screen you are on will be written to the disk. One nice option that Screen mode has is that you can type "N", and the screen you have been working on will be written to the disk, and the one after it will be read in from the disk. Once you've typed an "N" or a "W", you will be put back into Command mode. To quit the editor type "Q".

Another convenient way to enter the editor is to type the number of the screen you want to start with, and then type "SE". More detailed information on the various modes follows.

Command Mode

In order to reach any other mode while in Command mode, type the first letter of the mode's name: "S" for Screen mode, "D" for Delete mode, "I" for Insert mode, and "O" for Overwrite mode. To escape back to Command mode hit the ESC key. Normally, one should return to Command mode when not actually entering text.

Command mode is useful for elementary text manipulation (e.g. typing "B" will break one line into two; typing "X" will join two lines into one) and also for fast cursor manipulations.

Tab keys move the cursor horizontally to tabs set eight spaces apart. Tab itself will move the cursor to the right, while "A" will move the cursor to the left. To clear a tab mark, hit Shift Tab; to set a tab, hit <CTRL> Tab. You can jump to the next word by hitting "W", or to the end of the line by hitting "E", to the end of the text by hitting "Z", or to the top of the text by hitting "T".

It is possible to scroll the screen up or down in two different ways. If the cursor is at the top of the screen and you hit either <CTRL> up-arrow or "U" the screen will move up, exposing the line of text "above" the screen . Symmetrical commands hold at the bottom with <CTRL> down-arrow or "M". Hit "Y" to scroll the screen up, or "N" to scroll the screen down. Note that when you reach the last line of text at the bottom or the first line of text at the top, the screen will no longer scroll, but rather will flash and beep at you.

In all modes, capitals and lower case commands have the same function, so that, for example, "S" and "s" will both put QE into Screen mode.

"T" - Moves cursor to first line of text screen.

"Z" - Moves cursor to last line of text screen.

"W" - Moves cursor to beginning of next word.

"E" - Moves cursor to end of current line.

Tab - Moves cursor to next tab mark.

"A" - Moves cursor to next tab mark to the left.

Shift Tab - Sets a tab mark in current cursor position.

<CTRL>Tab - Clears a tab mark from present cursor position.

<CTRL>Up-Arrow - Moves cursor up one line.

<CTRL>Down-Arrow - Moves cursor down one line.

<CTRL>Left-Arrow - Moves cursor left one character.

<CTRL>Right-Arrow - Moves cursor right one character.

"H" - Same as <CTRL>Left-Arrow.
For this and the three commands that follow, notice the convenient arrangement of these keys on the keyboard.
"J" - Same as <CTRL>Right-Arrow.

"U" - Same as <CTRL>Up-Arrow.

"M" - Same as <CTRL>Down-Arrow

<Return> - Moves cursor down one line to first non-space character.

<Break> - Same as <Return>.

"Y" -Scrolls screen up one line.

"N" - Scrolls screen down one line.

"I" - Shifts QE to Insert mode.

"D" - Shifts QE to Delete mode.

"B" - Breaks the current line at the cursor position.

"X" - Joins two lines together.

"Q" - Exits from QE.

Screen Mode

Screen mode is reached through command mode by typing "S". It is intended for the handling and manipulation of entire text screens, and for accessing the disk. This is accomplished by using a combination of nine letter commands, and the ten digits at the top of the keyboard. Entering anything besides these nine commands or the digits will cause you to return to Command mode. ESC is the dignified way of returning to Command mode.

Three of the commands in Screen mode are for the manipulation of the screen number. Typing a "+" will add 1 to the screen number; typing a "-" will subtract 1 from the screen number; "/" is used to change the screen number to 0. You can't type in a negative number, and if you want to access a negative screen, you must go to 0 and keep the "-" pressed down until you reach the desired screen number.

There are three commands in Screen mode which have to do with "locking" screens. This "lock" is marked on the disk, and it tells QE that the screen involved should not be written to, though it may be read out into the editor. The three commands are "L", which will lock the screen you are on (according to screen number), "U", which will unlock the screen you are on, and "?" which will tell you whether a screen is locked or not by putting the word "LOCKED" or "UNLOCKED" in the prompt line. This "locking" system is only effective with QE, not with the fig Editor.

(If you want to either "lock" or "unlock" every screen on the disk, you may do so from outside QE. Type the drive number, a space and UNLOCK_ALL or LOCK_ALL to do this.)

The last three commands in Screen mode are for the actual accessing of the disk. These are: "R" which reads from the disk the screen corresponding to the screen number on the prompt line, "W", which will set a flag for writing the revised screen back onto the disk (unless it is write-protected or "locked"), and "N", which writes the present screen to the disk, adds one to the screen number, and reads this next screen back out into the editor. If the screen is "locked," this will cause an error and put the editor back in Command mode.
Note: "W" and "N" do not cause immediate writing to disk. To make sure the last two screens you've worked on are actually written to disk, type FLUSH after you exit the editor.

Screen Mode Commands

"L" - Locks the current screen to prevent inadvertently writing over it.

"U" - Unlocks the current screen to allow writing it to disk.

"?" - Checks to see whether the current screen is locked or not.

"R" - Reads the current screen from the disk.

"W" - Writes the current screen to the disk (but see above note).

"N" - Writes the current screen to the disk, and then reads the subsequent screen from the disk.

"+" - Adds 1 to the current screen number.

"-" - Subtracts 1 from the current screen number.

"/" - Changes the current screen number to 0.

Screen numbers can also be entered directly, using the digits at top of keyboard. (Negative numbers can only be reached by subtracting from 0.)

ESC - Returns QE to Command mode.

Delete Mode

Delete mode is different from the other modes in QE in that one cannot stay in the mode but can only use it for one command at a time. This gives the user the feeling of having two letter commands instead of one. For example, to delete a line while in Command mode, you type "DL". The "D" puts you in Delete mode, the "L" tells QE to delete the line the cursor is on, then QE puts you back into Command mode.

Delete mode can be reached from Command, Insert, and Overwrite, and it returns to the mode from which it was called. Thus, if one is in Insert mode and types <CTRL>"D", which is the way to reach Delete mode from Insert mode, one would enter Delete mode, but upon leaving would return to Insert mode. (In any other mode either an error or ESC returns one to Command mode.)

Delete mode has six commands besides the ESC key. Four are for basic editing; "C" deletes the character the cursor is on and pulls back the other characters on the line to fill in; "S" deletes all the spaces between the cursor and the next non-space character; "W" deletes the rest of the word following the cursor.

The other two commands are a little different. "A" clears everything on the screen after the cursor, and "B" clears everything before. They are intended for major editing, so "B" deletes all before the cursor by filling in spaces, instead of pulling back the rest of the text to fill in the deleted lines and letters. To clear a screen, jump to the top or bottom of the screen and type "D" and either "A" or "B" depending on whether you are at the top or bottom.

Delete Mode Commands

"A" - Deletes all text after the cursor.

"B" - Deletes all text before the cursor.

"C" - Deletes the character the cursor is on; pulls back the rest of the line to fill in.

"L" - Deletes the line the cursor is on; pulls up the lines below to fill in the empty line.

"S" - Deletes all spaces from the cursor to the next non-space character; pulls rest of line back to fill in.

"W" - Deletes word or part of word, from cursor onward; pulls rest of line back to fill in.

ESC - Returns to previous mode.

Insert Mode and Overwrite Mode

There are two modes in QE for entering and editing text directly, Insert mode and Overwrite mode. They have many commands in common, and other commands which are very similar. The most important difference in the two modes is that Insert mode inserts text, while Overwrite mode writes on top of existing text. If you hit the return key in Insert mode, a new line will be inserted (if there is room on the screen),but hitting the return key will just move the cursor down in Overwrite mode. Also, if you are in Insert mode and start typing in the middle of a sentence, the rest of the line will move aside as you type, whereas in Overwrite mode the old text will be overwritten.

Aside from this difference, most commands in Insert and Overwrite modes are the same. The arrows can be used, in conjunction with the <CTRL> key, to move the cursor about the screen, and most of the commands which are found in Command mode (e.g. "T" to move to the Top of the text screen) can be used in these two modes by holding the <CTRL> key down while typing the command. Added to these commands are a few special ones, Shift INSERT and <CTRL> INSERT, the first of which inserts a space after the cursor, the second of which inserts the next character typed after the cursor. The advantage to the latter is that any character may be inserted, including control characters which can't be entered via normal text editing.

<CTRL> "D" will put you into Delete mode. (As explained in that section, Delete mode will return QE to Insert or Overwrite mode if it is called from there). <CTRL> "I" will transfer QE to Insert mode from Overwrite mode, <CTRL> "O" to Overwrite mode from Insert mode. Thus, you do not need to go through Command mode to switch between these two.

Insert Mode Commands

<CTRL>"T" - Moves cursor to first line of text screen.

<CTRL>"Z" - Moves cursor to last line of text screen.

<CTRL>"W" - Moves cursor to beginning of next word.

<CTRL>"E" - Moves cursor to end of current line.

Tab - Moves cursor to next tab mark.

Shift Tab - Sets a tab mark in current cursor position.

<CTRL>Tab - Clears a tab mark from present cursor position.

<CTRL>Up-Arrow - Moves cursor up one line.

<CTRL>Down-Arrow - Moves cursor down one line.

<CTRL>Left-Arrow - Moves cursor left one character.

<CTRL>Right-Arrow - Moves cursor right one character.

<Return> - Adds line beneath current line and places cursor at same indentation as previous line. (If there is not space for one more line on screen, the cursor just moves down one line.)

<Break> - Same as <Return>.

<CTRL>"O" - Shifts QE to Overwrite mode.

<CTRL>"D" - Shifts QE to Delete mode.

Shift <Insert> - Adds space after cursor position.

<CTRL><Insert> - Inserts any one character typed in after current cursor position. This allows insertion of <CTRL> sequences that leave graphics characters.

<CTRL>"B" - Breaks the current line at the cursor position.

The other keys work normally. As long as there are spaces to the right, all other text will be moved to the right as text is inserted.

ESC - Returns to Command mode

Overwrite Mode Commands

<CTRL>"T" - Moves cursor to first line of text screen.

<CTRL>"Z" - Moves cursor to last line of text screen.

<CTRL>"W" - Moves cursor to beginning of next word.

<CTRL>"E" - Moves cursor to end of current line.

Tab - Moves cursor to next tab mark.

Shift Tab - Sets a tab mark in current cursor position.

<CTRL>Tab - Clears a tab mark from present cursor position.

<CTRL>Up-Arrow - Moves cursor up one line.

<CTRL>Down-Arrow - Moves cursor down one line.

<CTRL>Left-Arrow - Moves cursor left one character.

<CTRL>Right-Arrow - Moves cursor right one character.

<Return> - Moves cursor down one line to first non-space letter.

<Break> - Same as <Return>.

<CTRL>"I" - Shifts QE to Insert mode.

<CTRL>"D" - Shifts QE to Delete mode.

Shift <Insert> - Adds space after cursor position.

<CTRL><Insert> - Inserts any one character typed in after current cursor position. This allows insertion of <CTRL> sequences that leave graphics characters.

<CTRL>"B" - Breaks the current line at the current line at the cursor position.

Any other keys behave normally. Text is written over existing text.

ESC - Returns to Command mode.

GLOSSARY

NOTICE: The glossary which follows is intended for hackers and the idly curious. Users of QE will only need the following half dozen words, all of which are invoked from FORTH:

QE and SE, described above, which get you into the QE Editor

n LOCK_ALL , which "locks" all the screens on drive n; its cousin UNLOCK_ALL
n LOCK_BOOT , which "locks" the boot screens on drive n
WE , which is used like WHERE to locate compilation errors.

" — QED VOCABULARY
This sets up a table to be used for the prompt line. This table is allotted 34 bytes. To use, follow " with a four letter header starting with a quotes, then the prompt followed by a quote. Thus:

```
" "COM COMMAND                    "
" "SCU Re Wr Ne Lk Ulk Unlocked "
```

There must be 30 characters used between quotes.

"K:" — addr QED
Variable signifying the keyboard address.

$?OF f —
The run-time procedure compiled by ?OF. Works like $OF, except that it takes a flag from the stack, rather than comparing two values on the stack. (See CASE glossary.)

%END — n QED
This constant holds the address of the end of the data for the current text screen.

%F — flag QED
Constant with value of 0, signifying False. Clarifies code.

%T — flag QED
Constant with value of 1, signifying True. Clarifies code.

/COL — addr QED
Variable holding the number of the column the cursor is on.

/DONE — addr QED
Variable holding flag which is true if you are ready to leave QE, false otherwise.

/EXTRACT — n QED
This constant gives the memory address directly after %END. Holds deleted characters for use with SAVE.CH and EVAS.CH.

/EXTRACTP — addr QED
Variable holding place in the stack of deleted characters. (Must be offset by /EXTRACT).

/IMAGE — n QED
This constant holds the location of the top of the QE display list.

/INSERT — addr QED
Variable holding flag which is true if QE is in Insert Mode, false if not.

/KEY — addr QED
Holds the value of the key pressed.

?OF f —
Used in the CASE structure in the form:
f ?OF xxx FO
?OF works like OF, except that it takes a flag from the stack rather than comparing two values on the stack. (See CASE glossary.)

/LASTNUMBER — addr QED
Variable holding the number of the screen last used before the present one.

/LINE — addr QED
Variable holding the number of the line in memory to which the current row corresponds.

/NUMBER — addr QED
Variable holding the absolute value of the screen number.

/OLDDLA — addr QED
This variable holds the address of the display list on the screen prior to QE's display list. This is replaced on exiting QE.

/PROMPT n1 —n2 QED
This is the prompt line, set up as a table, allotted 50 bytes. It appears at the bottom of the screen.

/ROW — addr QED
Variable holding the row the cursor is on.

/SIGN — addr QED
Variable which is 1 if the screen number is positive and -1 if the screen number is negative.

/SMUDGE — addr QED
Variable signifying that there is a prompt. (0 if there is not).

/STOP! — addr QED
Variable holding flag which says whether or not to stop the mode QE is in.

/TABS n1 — n2 QED
Table of tabs, set originally to every eight characters. Changed by SETTAB and CLRTAB.

/WEIRD-FLAG — addr QED
Variable which holds flag saying whether a digit typed in should be part of old digit, or whether it should begin again. (e.g. 35 is displayed. Should a 4 typed in become 354 or 4).

/WHERE — n QED
This constant points to memory address of the first text character to appear on the screen.

ALENGTH — n QED
Returns the length of the first row displayed on the editor's screen, not including trailing blanks.

ALLOCATE_IMAGE — QED
Puts the address of the beginning of the text in the display so that the top of the video screen holds the first line of the text.

BEEP! — QED
Flashes yellow to signify error, then returns to previous color. Sets /STOP! to true (i.e. 1).

BEEP? — flag QED
Is /STOP! true (i.e. 1)? If so, returns true flag.

BIT_TABLE n1 — n2 FORTH
Table containing bit logic for lock/unlock flags. There are eight elements.

BREAK_LINE — QED
This is only operational if there is room below the line the cursor for an inserted line. This shifts all of the line after the cursor down to the line below, leaving blanks after the cursor on the original line, and blanks before the cursor on the added line.

b — n QED
Constant that returns code field address of BEEP!

CASBUF — n FORTH
This constant marks the beginning of the cassette buffer (plus 2 bits). This is used for buffer since we don't use cassettes.

CH! ch — QED
Stores the character given at the locaton in memory corresponding to the cursor location.

CH? — addr QED
Finds the location of the character in memory which corresponds to the cursor location. Puts that address on the stack.

CH@ — ch QED
Fetches the character at the location in memory corresponding to the cursor location on the screen.

C!PSE6 — QED
Turns off the keyboard. (i.e. won't read from the keyboard after this).

CLRTAB — QED
Clears a tab mark from current position. (Puts 0 in /TABS).

COMMAND_LOOP — QED
This runs continuously while in Command Mode, picking up commands and executing them. Aborts when /DONE returns false.

CURDOWN — QED
Moves cursor down 1 row.

CURUP — QED
Moves cursor up 1 row.

DI — addr QED
This is the display list for QE. It has 22 lines of Mode 2, a divider of 4 Blank, and 1 line of Mode 2 for the prompt.

DELETE_ONE addr — QED
Replaces character at this address with a space.

DEL_AFTER — QED
Replaces all characters after cursor to end of text screen with spaces.

DEL_BEFORE — QED
Replaces everything on test screen before the cursor with spaces.

DEL_CHAR — QED
Deletes a character and uses SAVE-CH to save the deleted character in a delete buffer.

DEL_LINE — QED
Deletes the current line and moves lines below up one.

DEL_SPACES — QED
Deletes all spaces from the cursor to the next non-space character or the end of the row, which ever comes first.

DEL_WORD — QED
Deletes all characters from the cursor to the next space, or the end of the row.

DEX_INIT — QED
Sets up all of the DLI's , display list, colors, missiles, etc. Initializes everything that needs it.

DISKIO — FORTH
Executes the system disk handler. Assumes that the device control block has been set up already.

DISP_NUM — QED
This takes the number of the screen in use and puts it on the prompt line.

DOWN — QED
If the cursor is at the bottom of the display, this moves data up one line without changing cursor (unless there is no more data to scroll), otherwise this just moves cursor down one line.

ED_COMMAND — QED
Executes commands entered in Command Mode.

END_OF_LINE — QED
Moves cursor to last non-blank character of current line.

EVAS_CH — ch QED
Retrieves character at address of /EXTRACT + /EXTRACTP and subtracts one from /EXTRACTP.

FLAGs — n FORTH
This constant leaves the location of the lock and unlock flags.

GET_BYTE scrn — addr mask FORTH
Gets the appropriate address and mask for the lock/unlock bit of this screen.

GET_KEY — ch QED
Gets the value of the key pressed and puts its ATASCII value on the stack. Also stores this value in /KEY.

INDENT? — n QED
This checks to see if the line is indented. It returns the number of spaces indented.

INSERT_CHAR ch — QED
Inserts a character in row after cursor, if there is room.

INSERT_CHAR_R ch —
Takes a character and tries to insert it to the right of the cursor. If it is at the end of the row, it inserts a new line and inserts the character at the first column.

INSERT_DOWN — QED
Inserts a line beneath current one if there is room. (i.e. blank lines at bottom of data screen.) Used by INSERT-LINE.

INSERT_LINE — QED
Inserts a line beneath current one, if possible, and moves cursor to indentation of previous line.

KBOTTOM — QED
Puts cursor at first column of last line of text screen.

KDELETE — QED
Puts QE in Delete Mode, and puts Delete prompt on prompt line.

KINSERT — QED
Sets /INSERT to true and invokes TEXT.

KJOIN — QED
Joins together current line and line below.

KOVERWRITE — QED
Sets /INSERT to false and invokes TEXT.

KQUIT — QED
Sets /DONE to true, alowing QE to quit.

KRECOVER — QED
Recovers a character from the deleted /EXTRACT buffer using EVAS-CH.

KRETURN — QED
Moves down one line if possible and to the first non-blank character (using INDENT?).

KSCREEN — QED
Puts QE into Screen Mode, from Command Mode only, and sets /WEIRD-FLAG to false to allow

numbers to be entered properly.

KSCROLLDOWN — QED
Used for scrolling the screen in Command Mode. Uses ROLLUP.

KSCROLLUP — QED
Used for scrolling the screen in Command Mode. Uses ROLLDOWN.

KTOP — QED
Puts cursor at first line of text screen, in the first column.

LEFT — QED
Moves cursor one column to the left, unless it is at the first column. If it is, this moves the cursor to the last column of the row above, if there is a row above.

LENGTH — n QED
Returns the length of the first line of the text screen, not including trailing blanks.

LOCK scrn — FORTH
"Locks" one screen by setting a bit in the cassette buffer (unused), thus preventing writing that screen to the disk from QE.

LOCKED? scrn — FORTH
Returns "Locked" if the screen is locked, and "Unlocked" if it is unlocked.

LOCK_ALL drive — FORTH
Locks all screens on given drive, so that none can be written to in QE.

LOCK_BOOT drive — FORTH
Locks all screens from -11 to the end of the boot, (this must be set into word) so that the boot screens cannot be written to from QE.

LOCK_LOOK scrn — flag FORTH
Sets flags to 1 if the screen is locked. 0 if it is unlocked.

MAP_BIT scrn — mask offset FORTH
Maps the screen number to the lock/unlock bit to which it corresponds.

MAP_CH! ch — QED
Changes ATASCII character value to corresponding Internal value before storing it in memory.

MAP_COMMAND ch — ch QED
Identifies upper and lower case letters, then maps letters to the commands they should cause to occur.

MAP_SCREEN scr — scr drv FORTH
Figures out which drive a screen is on; gives error if it is not a possible screen.

NEXT_SCREEN — QED
Writes current screen to disk using WRITE-SCREEN, reads in subsequent screen using READ-SCREEN and adds 1 to /NUMBER.

NGET — n QED

Returns the screen number being used.

OPEN_K: — QED
This opens the keyboard.

PROMPT addr — QED
Takes address of a prompt message, finds its total length and moves it to address of /PROMPT. Sets /SMUDGE to true.

QE — FORTH
Puts into QED vocabulary, uses DEX_INIT to initialize everything, and uses XED to enter the editor, and later to exit from the editor.

QED — FORTH
Establishes QED as the CONTEXT vocabulary.

READ_FLAGS drv — FORTH
This word reads the lock and unlock flags from the disk in given disk drive.

READ_SCREEN — QED
Reads screen held in /NUMBER (sign held in /SIGN) from disk. Puts Reading... prompt on prompt line.

RIGHT — QED
Moves cursor to the right one column, unless it is at the last column. If it is, this moves the cursor to the first column in the row beneath, if there is a row beneath.

ROLLDOWN — QED
Scrolls screen of text upward unless it is at bottom. Does not move cursor.

ROLLUP — QED
Scrolls screen of text downward unless there is no more above current line. Does not move cursor.

SCREEN_LOCKED? — QED
Checks to see whether the current screen is locked. If it is, puts "Locked" in inverse video on prompt line, if not, puts "Unlocked" on prompt line in inverse video.

SDIGIT — QED
Takes input and translates it to decimal numbers that /NUMBER can hold.

SE scrn — FORTH
Given a screen number, this simuultaneously enters QE and causes the screen to be read in.

SETDOWN_DL — QED
Reconstructs display list on screen prior to QE's display list. Uses /OLDDLA.

SETTAB — QED
Sets tab mark in current position. (Puts 1 in /TABS).

SETUP_DL — QED
Puts address of old display list in /OLDDLA; sets up D1 as display list.

SET_COL col — QED

Given a column number, this puts it in /COL and moves player 0 (which forms the cursor) over to that column.

SET_LINE n — QED
Puts line from memory on screen; puts line number in /LINE.

SET_ROW row — QED
Sets up the cursor, using player 0, at the row, and puts value in /ROW.

SHEXDIGIT — QED
Takes hex numbers as input and translates them to numbers that /NUMBER can hold.

SINCREMENT — QED
Increments /NUMBER by 1 or -1, depending on the sign (held in /SIGN).

SMINUS — QED
Subtracts 1 from current screen number.

STOP? — flag QED
Checks to see if /STOP! is true, or whether ESC has been hit. Returns true if either is true.

SAVE_CH ch — QED
Saves a character in address found by adding /EXTRACT and /EXTRACTP. Adds one to /EXTRACTP.

STOTAL — QED
Puts zero in /NUMBER and 1 in /SIGN.

TAB — QED
Moves the cursor right to the next tab mark, as found in /TABS.

TAB! — QED
Puts a flag in /TABS at current position.

TAB_WORD — QED
Pops to beginning of next word. Will continue, line after line, till it finds a non-space character.

TEXT — QED
This word is used to go into Insert or Overwrite Mode from Command Mode. It oversees the various commands that can be used in those modes. Gathers together many little words for use in actual editing.

TEXT_BS — QED
Insert Mode: Deletes character to left of cursor, drawing back rest of line to fill in.
Overwrite Mode: Overwrites character to the left of cursor with a space.

TEXT_CH ch — QED
If QE is in Insert Mode, this inserts the character to the right of the cursor using INSERT-CHAR-R.

If in Overwrite mode, this writes over the character under the cursor, moves cursor right using RIGHT, and saves overwritten character with SAVE-CH.

TEXT_INSERT —— QED
Puts Insert prompt in prompt line, and sets /INSERT to 1 (true).

TEXT_OVER —— QED
Puts Overwrite prompt on prompt line, and sets /INSERT to 0 (false).

TEXT_RETURN —— QED
If QE is in Overwrite Mode, this moves cursor to first non-blank character on line below current line. If QE is in Insert Mode, this inserts a line below current one, space permitting, and moves cursor to indentation of line above.

UNTAB —— QED
Moves the cursor to the next tab mark to the left, as found in /TABS.

UNTAB_WORD —— QED

Moves left till it finds a non-space character. Moves up screen till it finds a non-space character or the beginning of the screen.

UNLOCK scrn —— FORTH

"Unlocks" a given screen by setting a bit in the cassette buffer, thus allowing one to write that screen to the disk in QE.

UNLOCK_ALL drive —— FORTH
Unlocks all screens on given drive, so that all can be written to from QE.

UP —— QED
If the cursor is at the top of the display, this moves data down one line without changing cursor (unless there is no more data), otherwise it just moves the cursor up one row.

VECTOR n1 —— n2 QED
This is the table (54 bytes long) containing the address of the actual keys for entering commands in Command Mode.

WE scrn row col —— FORTH
Recovers QE after disk errors. This will put QE up with scrn as the screen, and the cursor at row and col.

WEIRD_CHECK —— QED
If /WEIRD_FLAG is false, this zeros the current screen number and sets /WEIRD_FLAG to true.

WRITE_FLAGS —— FORTH
Writes the lock/unlock flags to the disk.

WRITE_SCREEN —— QED
Writes screen held in /NUMBER (sign held in /SIGN) to disk. Puts "Writing..." prompt on prompt line.

XED —— QED
Puts in original QE prompt giving Revision number, etc. Sets up everything, initializes everything, puts into COMMAND_LOOP,

HOW TO WRITE GOOD FORTH

FORTH suffers from a combination of bad press and bad programmers. It is actually easier to write good FORTH than good anything else, since it is flexible, imposes few restrictions, and extracts no penalty for short programs. (Short programs, incidentally, are where it's at - breaking things up into small, sensible, lucid pieces is a key to good programming).

Why write good FORTH?

Good code is faster to write, if you consider total time, since it comes from clear thinking and good organization, and it is much easier to debug. You can understand and modify good code in amazingly shorter time.

Finally, you're much more employable if you write good code. Should a prospective employee tell me that he or she prides him or herself in writing good, clear, maintainable, well-documented FORTH I would, after picking myself up off the floor, endeavor to hire the person on the spot.

How to write good FORTH:

You need to (1) know what good FORTH is, (2) discipline yourself to write it. Item (1) will be addresed next. For (2), I've observed that a few thoughtful rewritings of a reasonably sized piece of code will not only help a decent programmer understand what good code is, but will also establish the necessary habits so that writing passable code becomes almost automatic, and good code a distinct possibility.

What is good FORTH?

Good FORTH consists of short, well thought out pieces. A word should rarely take up more than half a screen. If you use more than that, see whether you can break up the word into smaller, more coherent pieces. A good FORTH word is one which is easy to understand, and which you might be tempted to use again elsewhere.

Good FORTH is vertical, not horizontal. That is, there are few words per line and the lines are left-indented in an intelligent fashion. DOs and LOOPs, BEGINs and UNTILs, IFs, THENs, and ELSEs, etc. are all left justified, with inner groupings to the right of outer ones. Take a look at the Swarthmore source code to see what this means.

Each line should contain no more than a single idea.

A comment accompanying each word should show its effect on the stack preferably with helpful mnemonic symbols for the stack elements.

Words should be liberally sprinkled with comments. More lengthy comments can be put after the "-->" or ";S". The best code is almost self-documented.

The first line of a screen should consist of a comment which describes the contents of the screen. You should rarely start with more than half a screen full of code.

Will you go broke buying the number of disks necessary to write good FORTH? Hardly. Even in

this expansive form, FORTH gets you a lot of mileage. Moreover, few items connected with computers are as cheap as disk space. Finally, truth to tell, revisions and reworking usually result in screens filling up more than the minimal recommended (starting minimally gives you room for such expansion).

FORTH is supposedly a "stack oriented" language, but the programmer should not be stack oriented. Juggling several elements on the stack can cause a surprising number of errors. In almost all situations, it's much clearer and leads to many fewer errors if you introduce sensibly named variables. It is hard to find a situation when such a practice will measurably slow down a program. It is possible to juggle stack elements by sending them to and from the return stack (it's possible to do lots of wild things in FORTH), but it's usually just not worth it.

If you simply must play with several items on the stack, you're probably best off making up special stack manipulation words. If a word deals with several items on a stack, I've also found it very helpful to put in full comment lines within the word which show the stack changes in clear mnemonics.

Well-thought-out names can contribute immensely to good code. It's worth spending time making up useful names (self-documenting code, again). One helpful approach is to try to name what the word does, not how it does it. Don't use an unidentified number in a word. Either identify it in a comment or define a constant (with a good name) and use it in the code.

Avoid abbreviations when you're writing code. The extra typing is trivial and the clarity introduced is considerable. It's fair and reasonable to introduce abbreviations in a testing/debugging session, however.

Is our code good?

It varies. It's getting better. At first we committed the negation of the above ideas. Look over our source code and see what you like and what is clear. There are probably 8 different programmers represented on the Swarthmore disks, so there are lots of different styles, ideas, and approaches.

Further thoughts on writing good FORTH are to be found in Leo Brodie's Starting FORTH.

G.K.