AIM 65/40

Forth Users Manual

microcomputer system

Rockwell International

...where science gets down to business

Document No.
29651NO7

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued

TABLE OF CONTENT (Continued)

TABLE OF CONTENT (Continued

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

AIM 65/40 FORTH V1.4 ERRATA

The AIM 65/40 FORTH release V1.4 includes the following limitations:

1. Compilation using SOURCE does not return control to the keyboard upon error detection.
2. The XOR word operates improperly.
3. The READ word does not display data input
4. The key used to terminate VLIST is also entered as an input character.
5. Compilation using SOURCE will accept lines greater than 79 characters in length.
6. "OK" is not displayed upon completion of compilation.

The following words should be entered into the text editor and compiled as shown (see Section 4.5). Run a VLIST (see Section 4) to verify proper compilation. These words will then be used in lieu of the XOR , FINISH , SOURCE and READ system words.

```
{T}
HEX
={L}/      OUT=
HEX
: VLIST
 VLIST KEY DROP ;

: 'KEY
[ UKEY @ ] LITERAL EXECUTE
 DUP EMIT ;

: READ
 UKEY @ ROT ROT
 [ ' 'KEY CFA ] LITERAL
 UKEY ! READ UKEY ! ;

: -ABORT
 CR HERE COUNT TYPE  "  ?" CLOSE QUI
 [ SMUDGE

: SOURCE
 TIB @ 80 ERASE
 WARNING @ UABORT @
-1 WARNING ! [ ' -ABORT CFA ]
 LITERAL UABORT ! SOURCE ;

: FINIS
 UABORT ! WARNING !
 CR  " OK" FINIS ;

CODE XOR
TOP LDA, SEC EOR, PHA,
TOP 1+ LDA, SEC 1+ EOR,
BINARY JMP, END-CODE

WARNING.@ UABORT @
FINIS
 *END*
={Q}

{5}
 AIM 65/40 FORTH V1.4
SOURCE  IN=M
VLIST NOT UNIQUE
READ NOT UNIQUE
SOURCE NOT UNIQUE
FINIS NOT UNIQUE
XOR NOT UNIQUE
OK
```

SECTION

INTRODUCTION

FORTH is a unique programming system that is well suited to a variety of applications. Because it was originally developed for real-time control applications, FORTH has features that make it ideal for machine and process control, data acquisition, energy and environmental management, automatic testing, and other similar applications. The speed performance of assembly language is required in many of these applications, however a high-level language is often desired to improve program development productivity and program reliability. FORTH is designed to satisfy both speed and programming efficiency requirements.

FORTH can be called a computer language, an operating system, an interactive compiler, a data structure, or an interpreter, depending upon your point of view. It was designed to combine the strengths of both compilers and interpreters. The result is a unique language based on pre-defined operations that minimizes software development time and costs, supports structured programming and program modularity, compiles interactively to ease debugging and to reduce programming errors, compacts into small object code, and executes extremely fast. Additional words may be defined to allow usage by non-programmers.

AIM 65/40 FORTH in ROM combines the benefits of FORTH and the features of the AIM 65/40 Microcomputer with its resident printer, display, keyboard, and interactive Monitor and Text Editor firmware, to produce a stand-alone development and run-time system.

AIM 65/40 FORTH can also be used in a Rockwell RM 65 Single Board Computer (RM65-1000), in either a run-time, or development, mode with user provided peripherals and input/output software drivers. AIM 65/40 peripherals may easily be connected to the RM 65 SBC module, either directly, or through an RM 65 Multi-Function Peripheral (RM65-5223) module.

1.1 AIM 65/40 FORTH USER'S MANUAL DESCRIPTION

This manual is designed to provide both introductory instruction and detail language reference information. If you are new to FORTH, be sure to read and follow the manual chapter-by-chapter using the AIM 65/40 Microcomputer as a teaching aid in order to learn the FORTH language and operation concepts. If you already know the FORTH language you can probably skip certain sections and still use the language, however it is recommended to review all sections to become familiar with the AIM 65/40 FORTH mechanization and unique features.

Section 1, Introduction, introduces the AIM 65/40 FORTH language and the AIM 65/40 FORTH User's Manual.

Section 2, Installation and Operation, explains how to install the AIM 65/40 FORTH ROMs and how to enter, exit and re-enter AIM 65/40 FORTH.

Section 3, FORTH Concepts, provides a general overview into FORTH concepts and advantages. This is a good chapter to read if you are new to FORTH.

Section 4, Elementary Operations, leads you through elementary and common FORTH operations. By following this section step-by-step you will learn how FORTH operates to a sufficient level to implement simple applications in FORTH.

Section 5, Advanced Operations, takes you into more complex FORTH operations once you have become familiar with the elementary FORTH operations described in Section 4.

Section 6, AIM 65/40 FORTH Assembler, describes concepts and operating procedures associated with the AIM 65/40 FORTH Assembler.

Section 7, Handling Interrupts in FORTH, explains how to use machine level and interpretive interrupts in FORTH.

Section 8, Programming the R6522 VIA, explains how to use FORTH to program the R6522 Versatile Interface Adapter (VIA). These techniques can easily be applied to other peripheral devices.

Section 9, Notes on Style and Program Development, discusses the general approach to programming in FORTH and provides an example program.

Section 10, Preparing an Application Program for PROM Installation, tells how to structure and locate a FORTH application program in a PROM which will operate in conjunction with the AIM 65/40 FORTH ROMs.

Section 11, Using an Audio Cassette Recorder, describes how to dump and load source and object code for programs written in FORTH

Section 12, Interfacing to Mass Storage, tells how to prepare programs to store and retrieve program and data from mass storage Blocks, screens, and buffers are described. The technique to handle program overlays is also explained.

Appendix A, AIM 65/40 FORTH Func al Summary summarizes FORTH word operation by general . of usage.

Appendix B, AIM 65/40 FORTH Glossary, defines each FORTH word in ASCII sort order.

Appendix C, AIM 65/40 FORTH Assembler Functional Summary, summarizes FORTH assembler word operation by area of usage.

Appendix D, AIM 65/40 FORTH Assembler Glossary defines each FORTH Assembler word in ASCII sort order.

Appendix E, Error Messages and Recovery, identifies each FORTH error number and/or message, defines the error meaning, and describes the recovery action.

Appendix F, Page Zero and One Memory Map, defines the address, variable name and general usage of page zero parameters.

Appendix G, User Variables RAM Map, defines the address, variable name and purpose of each user variable. The cold and warm start initialization values are also listed.

Appendix H, ASCII Character Set, provides a list of 7-bit ASCII codes in decimal and hexadecimal corresponding to 32 control functions and the 96 upper and lower case alphabetic, numeric and special characters.

Appendix I, FORTH String Handling Words, describes how to create string handling functions in FORTH.

Appendix J, User 24-Hour Clock Program in FORTH, illustrates a program written in FORTH colon- and CODE-definitions, i.e., FORTH high-level words and 6500 assembly language.

Appendix K, Utility Functions, explains how to determine the time it takes for a FORTH word to execute.

Appendix L, AIM 65/40 FORTH Versus FIG-FORTH, identifies words incorporated in each FORTH that are not included in the other FORTH.

Appendix M, FORTH and the RM 65 FDC Module, lists a program written in FORTH to compute and display a ROM check-sum.

Appendix N, Selected Bibliography, lists references to many popular and tutorial FORTH articles and books.

1.2 REFERENCE DOCUMENTS

<u>Rockwell</u>

29650N30 Order No. 202	R6500 Programming Manual
29650N31 Order No. 201	R6500 Hardware Manual
29650N86 Order No. 280	AIM 65/40 System User's Manual
29651N08 Order No. 2104	AIM 65/40 FORTH Reference Card

SECTION 2

INSTALLATION AND OPERATION

The AIM 65/4Ø FORTH object code is provided in two Rockwell R2332 4K-byte ROM devices. After installing the ROMs in the AIM 65/4Ø SBC Module, FORTH is ready for use. Figure 2-1 shows the overall FORTH memory map.

2.1 INSTALLING THE FORTH ROMS

Before removing the ROMs from the shipping package, be sure to observe the handling precautions listed in Section 2.1 of the AIM 65/4Ø System User's Manual. Since MOS devices may be damaged by the inadvertent application of high voltages, be sure to discharge any static electrical charge accumulated on your body by touching a ground connection (e.g., a grounded equipment chassis) before touching the ROMs or the SBC module. This precaution is especially important if you are working in a carpeted area or in an environment with low relative humidity.

Ensure that power is turned OFF to the AIM 65/4Ø microcomputer. Carefully remove any ROM or PROM devices that may be installed in sockets Z7Ø and Z71 of the AIM 65/4Ø SBC Module. Remove the FORTH ROMs from the shipping package. Inspect the ROMs to ensure the pins are straight and free of foreign material. While supporting the AIM 65/4Ø SBC module beneath the ROM socket, insert ROM number R32PØ in Socket Z7Ø, being careful to observe the device orientation. Now insert ROM number R32P1 into Socket Z71. Be certain that both ROMs are completely inserted into their sockets. Make sure addresses $CØØØ-$DFFF are selected on the AIM 65/4Ø SBC module (see Section 2.2.1 in the AIM 65/4Ø System User's Manual). Then turn ON power to the AIM 65/4Ø microcomputer.

FFFF – FØØØ	AIM 65/4Ø I/O ROM and On-Board I/O	
EFFF – EØØØ	User Available	
DFFF – CØØØ	AIM 65/4Ø FORTH ROMs	
BFFF – AØØØ	AIM 65/4Ø Debug Monitor/ Text Editor ROMS	
AFFF – 8ØB	FORTH User Dictionary (Continues Upward in Memory)	
8ØA – 8ØØ	Dummy Word TASK	Start of FORTH Dictionary in in RAM.
7FF – 78Ø	Terminal Input Buffer (TIB)	
77F – 76Ø	User Variables	
75F – 7ØØ	FORTH User Variables	See Appendix G
6FF – 6ØØ	RM 65 FDC Module Default Buffers	
5FF – 4AØ	RM 65 Module Variables	
49F – 2ØØ	System Variables and Constants	
1FF – 1ØØ	R65Ø2 CPU Stack and FORTH Return Stack	
FF – FØ	I/O ROM Variables	
EF – D7	RM 65 Module Variables	
D6 – A9	User Available	
A8 – 1Ø	FORTH Variables	See Appendix F
F – Ø	User Available	

Figure 2-1. AIM 65/4Ø FORTH Memory Map

2.2 ENTERING, EXITING AND RE-ENTERING FORTH

2.2.1 Entering FORTH

Press 5 to enter and initialize FORTH when the AIM 65/4Ø Monitor prompt is displayed. AIM 65/4Ø will respond with

```
{5}
 AIM 65/4Ø FORTH V1.4
```

The last line of data displayed may be printed upon FORTH entry along with the {5}.

To re-initialize FORTH while in FORTH, type COLD followed by pressing the <RETURN> key. AIM 65 will respond with

```
COLD
 AIM 65/4Ø FORTH V1.4
```

Initializing FORTH with either of the above methods will remove any user words previously defined and added to the FORTH vocabulary or to any other application vocabulary (see Section 5.5). User variables are initialized to the default values described in Appendix F. The FORTH number base is also initialized to DECIMAL for input/output operations.

2.2.2 Exiting FORTH

Two methods can be used to exit FORTH. The ESC key can be pressed any time FORTH is in a command input mode. Control will be immediately returned to the AIM 65/4Ø Monitor, however, any values currently in the stack will not be saved. The significance of this will be apparent as you become more familiar with FORTH.

Control can also be returned to the AIM 65/4Ø Monitor from the FORTH command input mode by typing

```
MON
```

followed by pressing the <RETURN> key. This causes an R6502 BRK machine instruction to be executed and AIM 65/40 microcomputer to display

```
        MON
        = B0 D9 92 00 FD D9D          BRK
```

More importantly, exiting FORTH in this manner preserves any values on the stack.

2.2.3 Re-Entering FORTH

Once FORTH has been entered and control returned to the AIM 65/40 Monitor, you can re-enter FORTH by either of two methods without re-initializing the user variables or deleting previously defined words.

You can re-enter FORTH by pressing 6 anytime the AIM 65/40 Monitor prompt is displayed. The system will respond with

```
        {6}
         AIM 65/40 FORTH V1.4
```

The last line of data displayed may be printed upon FORTH re-entry along with the {6}.

Note that re-entering FORTH with the 6 key will delete any values previously stored in the stack, however the I/O number base is retained (See Section 4.11.3).

If FORTH has been exited using the MON command, FORTH can be re-entered by pressing G, typing D9D2 then pressing the <RETURN> key.

Re-entering FORTH in this manner retains any numbers on the stack saved by the FORTH MON exit to the AIM 65/40 Monitor If FORTH is re-entered properly in this manner, FORTH displays

```
        OK
```


SECTION

FORTH CONCEPTS

FORTH is quite different from more conventional languages suc as BASIC, FORTRAN, or Pascal. It creates a computing environ ment with unique strengths, tools, and styles. Some of the structures of FORTH have little correspondence with those of other languages. This overview of the language and the AIM 65/40 FORTH implementation provides background for the how-to-do-it chapters which follow.

3.1 FEATURES OF FORTH

FORTH is EXTENSIBLE, meaning that you add your own operations to the language. New words (operations) are defined from old words or assembly language, until a single word is the entire desired program. The program word can then be executed by typing its name. Except that your words may be defined in RAM, or user provided PROM or ROM, while those of the FORTH system itself are provided in the FORTH system ROMs, there is no distinction between your new operations and those originally part of the language. Extensibility allows users to define libraries or even their own languages for particular applications, greatly facilitating maintenance as requirements change.

FORTH keeps all definitions in a DICTIONARY. The dictionary includes virtually all the object code of the system itself and of your applications. Only the AIM 65/40 Monitor, I/O buffers, any source code which may be in RAM, and the "user area" of system or application variable values are outside the dictionary. Your own data structures may be in the dictionary or outside it, at your option. The internal structure of the dictionary is uniform and much simpler than the internals of most other languages; therefore, application programmers typically learn much more of the inner workings of the FORTH system

FORTH object code is extremely COMPACT in memory, even compared to machine language. Short programs, however, that are assembled or compiled to machine language may take less space since the entire FORTH system in the 8K ROM normally stays in memory as a run-time package. The 8K AIM 65/40 FORTH ROMs include code for the FORTH compiler, an assembler, terminal handling, etc. which are unnecessary at run-time for most applications. It is possible to shrink FORTH's run-time package to much less than 8K by special compilation techniques, but the software for doing that is not included in this system. In any case, FORTH's hierarchical structure allows application code to build on itself, increasing the memory advantage for larger programs, and with little loss in speed.

FORTH code is recursive, suited to multi-tasking applications, and can be programmed in RAM, PROM or ROM.

FORTH is STRUCTURED. There is no GOTO statement in the language. IF and ELSE control structures, and DO, UNTIL, and WHILE loops are provided; all of these can be nested to any practical depth.

FORTH uses a STACK and its associated POSTFIX NOTATION, also called Reverse Polish Notation (RPN), in which the operation codes are written after the operands which they use. For example, <2+2> in BASIC would be written <2 2 +> in FORTH. Why does FORTH use a stack explicitly when most other languages hide their stacks from the user and avoid postfix in favor of more conventional notation?

Part of the answer is that the stack allows very low overhead for linking between subroutines. FORTH reduces the cost of subroutines to very little, and the whole language is built around subroutine calls. Routines can accept and return any number of arguments, without the complexity or other overhead of formal parameter or local variable declarations.

The stack encourages extremely MODULAR programming, which can be debugged with great reliability. Consider FORTH's programming environment. Each module (i.e., word or procedure) has only one entry and one exit point. Usually all communication with the outside world is through the stack, so there are no side effects on other modules, variables, etc., unless explicitly programmed. Usually each module is short; commonly three to five 40-column lines. The smaller a module is, the easier it is to test all paths through it.

FORTH is INTERACTIVE. Testing is immediate, because almost all FORTH words can be executed directly as commands from the keyboard, and will behave exactly the same in this mode as when compiled into later definitions. Any arguments required can simply be typed onto the stack before the test, or generated by other operations, and results can be printed immediately. Usually each component of the new definition can also be executed interactively from the keyboard, to aid in debugging.

FORTH debugging seldom requires examining any code except the single definition being tested. Documentation of the behavior of the defined words in glossary form is required, i.e., inputs, outputs, and actions, but there is no need for their code to be listed. Fewer listings are therefore required during FORTH program development than with other languages. Everything you need to work with is directly in front of you.

FORTH allows easy MACHINE ACCESS, unlike most other high-level languages. All of memory and I/O (data ports and control registers) can be addressed, although run-time protection can be implemented simply by redefining appropriate system or user words to include run-time bounds or other checks during testing. Except for direct access to machine-specific registers (A, X, Y, etc. in the 6502 CPU) which require assembly language subroutines, FORTH can do anything machine language can do. And FORTH runs fast enough that usually no assembly language subroutines are necessary.

But if full machine speed is needed, AIM 65/40 FORTH includes an assembler. It also allows machine language subroutines to be tested immediately as soon as the assembly source has been typed in or otherwise entered, with no waiting for separate assembly and linking passes. It encourages structured programming even in assembly language; IF...ELSE and BEGIN... UNTIL macros are provided. Users can define their own macros, and use the full power of FORTH for address arithmetic and other assembly-time utilities. All 6502 op codes and addressing modes are available. This one-pass assembler is implemented in about 1.5K bytes, illustrating the compactness of FORTH's object code. It is resident in the AIM 65/40 FORTH ROM set.

The routines created by this assembler have FORTH names and behave exactly like regular FORTH definitions. The user needn't know which words are programmed in assembly language. Therefore, an application can first be written entirely in high level using FORTH words, and, if more speed is necessary, parts can be converted to assembly language code with no changes required elsewhere.

FORTH code is extremely TRANSPORTABLE between machines. It is common for substantial programs to be moved between different computers such as 6502, 8080, and PDP-11 with very little change or none at all. The AIM 65/40 system follows the FORTH Interest Group (FIG) language model, probably the most common dialect of FORTH, and one closely aligned with the International Standard for the language. The FIG model is available on the common small computers and is rapidly being implemented on others. Therefore the AIM 65/40 microcomputer can be used to develop software for other computers, and it can use published FIG-model code regardless of the machine on which it was developed. Published programs are commonly written entirely in FORTH with no machine code or other dependencies, but designed so that short, time critical words can be rewritten in assembly language for optimization on any particular host machine. These programs can first be run unchanged, then optimized only if needed.

As in any programming, good style makes the application program easier to debug and verify, and easier to read and modify when requirements change. Many recommended FORTH practices are familiar from other language environments, but some are different. Practices such as top-down design and bottom-up coding and testing, short modules, indentation of control structures, and a glossary as the principal documentation during development, are discussed throughout this manual.

3.2 DEBUGGING

The FORTH environment's convenient and powerful debugging and error control features are an important advantage of the system. FORTH allows complete access to the machine, without the restrictions of many other languages such as BASIC and Pascal which try to guard the programmer against mistakes. Most users report that FORTH allows them to quickly produce and modify programs which are exceptionally reliable.

Although AIM 65/40 FORTH includes extensive compile-time checking which detects most of the detectable errors (see Appendix E), the most important error control is in the tools which the FORTH environment itself gives to the programmer.

Like most other modern languages, FORTH encourages "structured programming" design techniques, which helps to control errors. FORTH is extremely modular, even compared to other structured languages; each software module can be tested and debugged independently. Usually all communication between a module and the outside world is through an internal stack. Each module relies on earlier modules which have already been debugged, and in turn, the new testing helps catch any errors that may still be hidden in the earlier work.

Testing is immediate and interactive; simply type arguments onto the stack, execute the word, and print the results. If more elaborate test data is needed, a special word can generate it. This ease of testing means that a large number of tests can be run quickly.

Each module should be short, in the programming style preferred by most FORTH users, so that all possible paths of control can be tested easily.

If correct results are not obtained, it is possible to step through the definition by executing each component word individually, checking the stack whenever desired. AIM 65/40 FORTH has a special word, .S , which non-destructively prints the stack contents to help in this kind of debugging. Any unexpected results can be localized to a particular component word, which in turn can then be examined in detail. Because FORTH words work identically when compiled, or when executed as commands, the programmer can debug at either a batch or interactive operation mode.

Because FORTH is extensible, words can be re-defined to perform their original functions and, in addition, give special debug print-outs or do run-time error checks. These redefinitions can be inserted into programs for testing and removed later; nothing else in the program need be changed.

AIM 65/40 FORTH also includes a memory dump and other words for examining or changing memory. These commands can be compiled into programs or executed from the keyboard.

In contrast to most other operating systems, all of these tools are part of the normal FORTH environment. No special syntax or command language must be learned for debugging.

Each FORTH word is documented by a glossary (see Appendix B) which lists the arguments it takes from the stack and the results returned, and gives a short verbal description (usually one to three sentences) of its action. Such a glossary completely describes the word as it is seen by any other part of the program. When a new word is being tested, all earlier words should have these descriptions available. Therefore, the programmer seldom needs to look at the source code of any other word; the glossary fully describes its functions. During

testing debugging, on one word at a time needs to be
examined this greatly s down the need for program
listings ring developme

One important debugging procedure applies only to FORTH. After a word appears to work correctly it must be tested to make sure that it does not take any unexpected numbers off the stack, or return unexpected results. One way to check is to leave markers, easily-recognized numbers, such as 1, 2 and 3, on the stack and then execute the word being debugged. After an operation, use .S to make sure that the markers are still on the stack, below any arguments returned by the test word. This check is important because otherwise the word may look like it works, but causes later program crashes at unexpected and seemingly random places making the problem hard to debug.

SECTION

ELEMENTARY OPERATIONS

This section provides a step-by-step description of elementary AIM 65/40 FORTH operations, such as:

- Performing simple arithmetic and comparisons
- Entering and retrieving data from memory
- Using the stack
- Compiling interactively or in a batch mode from memory
- Defining new FORTH words
- Performing looping and conditional sequences

A major portion of FORTH is the FORTH dictionary itself. Each word in the FORTH dictionary causes specfic actions or operations to be performed. The use of FORTH is explained primarily by describing how each word operates and how to use it, either individually or with other words. Let's start by seeing what is in the FORTH dictionary.

Enter FORTH from the AIM 65/40 Monitor.

```
{5}
 AIM 65/40 FORTH V1.4
```

List the contents of the FORTH dictionary by running a VLIST . Type

```
VLIST
```

and then press the <RETURN> key. The entire FORTH dictionary will be displayed (and printed if the printer control is ON). There are 249 words in the AIM 65/40 FORTH dictionary (not counting the assembler, see Section 6.1) so the printout will take about one minute.

Terminate the listing at any time by pressing any key. The entire VLIST is shown in Figure 4-1. Note that the words do not appear to be in any general order; the words are listed by their address in the AIM 65/40 FORTH ROMs. (The FORTH dictionary structure is explained in detail in Section 5.5, but leave that for later.) These FORTH words are described in ASCII sort order for convenient lookup in the glossary in Appendix B and summarized by associated function in Appendix A.

AIM 65/40 FORTH may be readily learned by performing the following procedure. As each new FORTH word is encountered in this section, read the explanation and perform the accompanying examples. Then read the word definition in the Appendix B glossary. Repeat the examples, but vary one or more of the parameters until you thoroughly understand the operation of the described FORTH word.

As you are learning FORTH, you may make errors that either cause an error message to be displayed or cause the AIM 65/40 to hang up or to run away, i.e., the display may go blank or show random data. If an error occurs with a displayed error message or number, refer to Appendix E for the error definition and suggested recovery. If the program appears to hang up or run away, press the <RESET> key to reinitialize the AIM 65/40 microcomputer and to return control to the AIM 65/40 Monitor. You can then re-enter FORTH and try the example again. You may have to back up a few steps, however, to recover the example initialization.

In the following descriptions, a FORTH word comprising of letters and numbers is written in upper case. Since some FORTH words contain special characters that may be confused with sentence structure, e.g., periods, commas, or apostrophes, the FORTH words are set off by spaces, e.g., .S . These single spaces are not part of the FORTH word and should not be entered.


```
VLIST
  809 TASK            D9DC .S
 D9D1 MON             D9C1 HANG
 D97A VLIST           D943 ?
 D937 .               D927 .R
 D918 D.              D8F5 D.R
 D8DF #S              D8B7 #
 D8A2 SIGN            D88B #>
 D87C <#              D85F SPACES
 D84E WHILE           D82C ELSE
 D815 IF              D7FE REPEAT
 D7E7 AGAIN           D7D9 END
 D7C5 UNTIL           D7AF +LOOP
 D799 LOOP            D786 DO
 D77B THEN            D760 ENDIF
 D74E BEGIN           D6C5 FORGET
 D6AC '               D6A2 R/W
 D677 -->             D645 LOAD
 D5EC MESSAGE         D5D8 .LINE
 D5B4 (LINE)          D573 DUMP
 D548 FLUSH           D4EA BLOCK
 D4A2 BUFFER          D47F EMPTY-BUFFE
 D459 UPDATE          D42E +BUF
 D411 M/MOD           D3FF */
 D3EE */MOD           D3DE MOD
 D3CE /               D3BE /MOD
 D3AF *               D389 M/
 D36E M*              D359 MAX
 D343 MIN             D335 DABS
 D326 ABS             D314 D+-
 D302 +-              D2F3 S->D
 D27D COLD            D247 ABORT
 D216 QUIT            D206 (
 D1F6 DEFINITION      D1E0 ASSEMBLER
 D1CA FORTH           D19A VOCABULARY
 D182 IMMEDIATE       D134 INTERPRET
 D10B ?STACK          D0F0 DLITERAL
 D0BF LITERAL         D0A3 [COMPILE]
 D02D CREATE          D006 ID.
 CFD4 ERROR           CFC6 (ABORT)
 CF9A -FIND           CF44 NUMBER
 CEF9 (NUMBER)        CEA9 WORD
 CE97 PAD             CE7F HOLD
 CE70 BLANKS          CE5F ERASE
 CE31 FILL            CDF4
 CDDE QUERY           CD6F EXPECT
 CD3F ."              CD26 (.")
 CCF9 -TRAILING       CCD1 TYPE
 CCBE COUNT           CC8C DOES>
 CC7C <BUILDS         CC64 ;CODE
 CC4E (;CODE)         CC3B DECIMAL
 CC26 HEX             CC16 SMUDGE
 CC02 ]               CBF3 [
 CBDD COMPILE         CBC2 ?CSP
 CBB0 ?PAIRS          CB9A ?EXEC
 CB83 ?COMP           CB6D ?ERROR
 CB5A !CSP            CB46 PFA
 CB31 NFA             CB25 CFA
 CB17 LFA             CB07 LATEST
 CAD7 -DUP            CAC8 SPACE
 CAB2 PICK            CA9F ROT
 CA91 >               CA75 <
 CA5E U<              CA51 =
```

Figure 4-1. VLIST of AIM 65/40 FORTH Words

```
CA45 -              CA35 C,
CA24 ,              CA18 ALLOT
CA08 HERE           C9F9 2-
C9EC 1-             C9DF 2+
C9D2 1+             C9C5 B/SCR
C9B5 B/BUF          C9A5 LIMIT
C995 FIRST          C985 C/L
C97C MODE           C972 HLD
C969 CSP            C960 DPL
C957 BASE           C94D STATE
C942 CURRENT        C935 CONTEXT
C928 OFFSET         C91C SCR
C913 IN             C90B BLK
C902 ULIMIT         C8F5 UFIRST
C8E9 PREV           C8DF USE
C8D6 TIB            C8CD R0
C8C5 S0             C8BD UR/W
C8B3 UABORT         C8A1 UB/SCR
C895 UB/BUF         C889 UC/L
C87F VOC-LINK       C871 DP
C869 FENCE          C85E WARNING
C851 WIDTH          C846 UCR
C83D U?TERMINAL     C82D U-CR
C823 UEMIT          C818 UKEY
C80E UCLOSE         C802 U?OUT
C7F6 U?IN           C7EB BL
C7E2 4              C7DA 3
C7D2 2              C7CA 1
C7C2 0              C7A5 USER
C790 CODE           C777 VARIABLE
C757 CONSTANT       C73E ;
C711 :              C704 C!
C6EC !              C6DB C@
C6C6 @              C6B7 TOGGLE
C696 +!             C687 BOUNDS
C676 2DUP           C665 DUP
C64D SWAP           C644 2DROP
C63A DROP           C629 OVER
C612 DNEGATE        C5F8 NEGATE
C5D1 D+             C5BD +
C5B0 0<             C5A9 NOT
C593 0=             C57D R
C56E R>             C55E >R
C545 LEAVE          C534 ;S
C521 RP@            C50E RP!
C4FE SP!            C4EF SP@
C4E0 XOR            C4CC OR
C4B7 AND            C46E U/
C434 U*             C40F CMOVE
C3EC FINIS          C3A4 SOURCE
C389 WRITE          C36F READ
C31C -CR            C310 CLOSE
C302 ?OUT           C2F4 ?IN
C2E8 PUT            C2DC GET
C2D0 CR             C2C5 ?TERMINAL
C2B3 KEY            C2A7 EMIT
C28C CLRLINE        C241 ENCLOSE
C1DF (FIND)         C1B0 DIGIT
C1A6 I              C18D (DO)
C156 (+LOOP)        C125 (LOOP)
C104 0BRANCH        C0E3 BRANCH
C0CB EXECUTE        C0A8 CLIT
C057 LIT OK
```

Figure 4-1 VLIST of AIM 65/40 FORTH Words (Con't)

4.1 SIMPLE ARITHMETIC

FORTH arithmetic, like that of advanced pocket slide rule calculators, uses a stack to store operands and results. Operations such as + - * / (add, subtract, multiply, and divide) take their arguments from the stack, and return thei results to it.

To see how the stack works, give FORTH a cold restart by typing

COLD

and pressing the <RETURN> key. AIM 65/40 will respond with

AIM 65/40 FORTH V1.4

Now type the following five numbers

1 22 333 -44 5

and terminate the input by pressing the <RETURN> key. <RETURN> at the end of a line signals that your input is complete. (The <RETURN> is shown in the initial examples, but is not shown in later examples, except where needed to clarify data or command entry.) Be sure to insert one or more spaces between each number. Now the numbers 1 through 5 are separate numbers stored on the stack with 5 at the top. FORTH responds to your input by displaying (and printing if the printer control is ON) OK . OK means that the system has correctly acted on your command and is waiting for another command to be entered. (The OK is not shown in most of the examples, however, it is implied in all operations.) After <RETURN> is pressed, the following is printed (if the printer control is ON)

1 22 333 -44 5 OK

Note that the printer can be turned ON and OFF in FORTH using the <CTRL> and P keys as in the AIM 65/40 Monitor.

Notice that the "blinking" cursor indicates the input character position. A typing error during FORTH command or data entry can be corrected by pressing the <DEL> key as necessary.

4.1.1 Examine Stack Contents with .S

The word .S (pronounced dot-s) may be used at any time to examine the contents of the stack without altering the values or removing the numbers from the stack. Try it by typing

```
.S <RETURN>
```

The numbers entered in the prior section will be displayed (in some examples the displayed data is underlined to distinguish it from entered data)

```
5
-44
333
22
1 OK
```

The .S word is very useful when learning AIM 65/40 FORTH or debugging a FORTH program to determine the stack contents immediately prior to and/or after executing a FORTH word.

4.1.2 Print from the Stack using .

The print command removes a number from the stack and displays it (and prints it if the printer is ON) in the current I/O number base. In FORTH, the print command is represented by a period and is called "dot". Type

```
. <RETURN>
```

The 5 will be printed and removed from the stack.

```
. 5 OK
```


Verify this by typing .S and <RETURN> to show the new contents of the stack.

```
-44
333
22
1 OK
```

The next dot (and <RETURN>) will print the -44. Multiple commands separated by spaces, can be typed on one line like this

```
. . <RETURN>
```

to display two numbers from the stack, e.g.,

```
. . <RETURN> 333  22 OK
```

Now only 1 is left on the stack. Output it with

```
. <RETURN>
```

which displays

```
. 1 OK
```

Trying to examine or print the stack contents when there are no numbers on the stack will result in an error message. Try .S which will show

```
.S <RETURN>
EMPTY
```

Note that the word . will now cause a stack underflow and will display an indeterminate value along with a stack empty message. Try it now

```
. Ø  (typical number)
. ? STACK EMPTY
```

Similar FORTH operations trying to pull a number from an empty stack will result in this error message. This error message, as well as others, are described in Appendix E.

Notice that the data was displayed (and printed if the printer is installed and enabled) on the same line as the commands, i.e., the FORTH word . in this case. Many times it is desired to display and print data on a new line. The FORTH word CR issues a carriage return to the display and printer. Repeat the previous examples but insert CR before the . word and note that the command is displayed (and printed) on a separate line prior to the data. Also command CR after the . and observe the results.

Perform a cold restart before continuing.

```
COLD
 AIM 65/40 FORTH V1.4
```

4.1.3 <u>Clearing the Stack</u>

It is sometimes desirable to delete data from the stack without performing a COLD restart. The stack may be cleared by trying to execute a word that is not currently defined in the FORTH dictionary. This causes an error condition in which FORTH echos the missing word followed by a "?" (see Appendix E for error descriptions) and then clears the stack. Initially, the word Q is not defined in the FORTH dictionary and can be conveniently used to clear the stack.

Note also that commanding a word that is not in the dictionary will also delete data that you may want on the stack -- so be careful with your word entries or you may have to re-enter data or repeat prior steps.

Enter some numbers on the stack and display the stack contents.

```
678 356
.S
356
678 OK
```

Type Q now and verify that the stack is cleared.

```
Q
Q ?
.S
EMPTY OK
```


4.1.4 <u>Add + and Subtract -</u>

Let's now perform some simple arithmetic. Put two numbers on the stack, say

```
12809 135 <RETURN>
```

Now type the add command

```
+ <RETURN>
```

The + takes whatever two numbers are on top of the stack and adds them. It removes those numbers (by convention, most FORTH operations destroy their arguments on the stack), and replaces them with their sum. Type

```
. <RETURN>
```

to verify this. The sum will be displayed as

```
. 12944 OK
```

As before, multiple operations can be placed on one line, e.g.,

```
12809 135 + . <RETURN> 12944 OK
```

Subtract works in a similar manner. Try

```
12809 135 - . <RETURN> 12674 OK
```

Repeat these last two examples but, insert CR before and after the word . to display the result on a separate line.

4.1.5 <u>Multiply * and Divide /</u>

Multiply and divide also work in a similar manner. Try the following

```
38 78 * . <RETURN> 2964 OK
```

The word * multiplies the top two items on t[illegible]ack and leaves only the result on the stack. The word [illegible] divides the second item on the stack by the top item. Try

```
13036 50   . <RETURN>
```

which displays

```
13036 50   . 260 OK
```

Note also that the divide limited the result to an integer value (the full answer is 260 with a remainder of 36). Other operations allow the remainder to be saved (see Section 5.1). In all FORTH arithmetic and comparison words requiring two data items, the operator behaves as if it were between the top two values on the stack. Thus, 13036 50 / behaves as if it were 13036 / 50.

Each number on the stack is 16 bits wide, therefore these single numbers have the range -32768 to 32767 since the most significant bit (bit 15) is used for the arithmetic sign. This is enough for many applications, but AIM 65 FORTH also has double-precision (32-bit) numbers which are discussed in Section 5.1.

4.1.6 Postfix Notation and Stack Operation

Note that in the preceding examples, the operators (+ , * and /) were typed after their arguments, not between them. This style of arithmetic notation is called POSTFIX or Reverse Polish Notation (RPN). It can represent complex formulas without any use of parentheses. For instance

```
(42-50)*(128-1090/3)
```

would appear in postfix as

```
42 50 - 128 1090 3
```

Note that the operands (the numbers) are in the same order in the postfix and infix (ordinary arithmetic) expressions. Don't forget to type . and <RETURN> to display/print the result.

If you are new to postfix, you may want to follow this example by using stack diagrams, as shown in Figure 4-2. This illustration shows the successive states of the stack after each number or operation has been processed. Each column shows the stack at one time. The number on top is the most accessible number on the stack, ready to be used first by any operation which takes a number from the stack. We say that this number is at the TOP of the stack.

In the execution of the postfix formula shown above, 42 is placed on the stack (first column of Figure 4-2) -- then 50 is entered. The subtraction destroys those arguments and leaves the difference, -8. You can follow the rest of the process similarly.

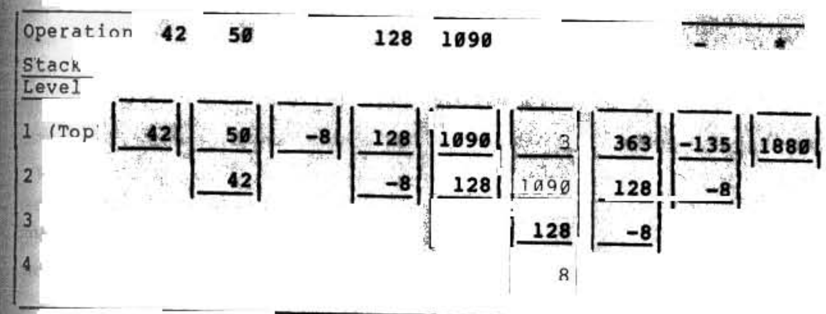


Figure 4-2. Stack Diagram of Postfix Example

Each column in Figure 4-2 shows the stack at the time after each successive number or operation of the formula has been processed. Note that any numbers which may have been below these numbers on the stack will be undisturbed. Repeat the above example but insert .S after each number and operator to examine the stack contents after each operation.

Only numbers go on the stack. Strings or other data structures do not reside there directly -- although some data such as pointers (addresses), length and offset information, ASCII values, are frequently on the stack.

How many numbers can reside on the stack at one time? AIM 65/40 FORTH limits the stack depth to 65 16-bit values, in order to keep the parameter stack in page zero to maximize 6502 CPU execution speed. Except for certain recursion problems, very few programs ever need a stack depth of more than about 20.

4.1.7 Decimal and Hexadecimal Number Base

Up to now we have been working in DECIMAL . FORTH allows input and output data to be represented in different number bases. We will consider only two pre-defined bases now -- DECIMAL and HEX . FORTH is initialized to DECIMAL (base 10) during initial entry or upon commanding COLD . DECIMAL is best used when working with numeric calculations. HEX operates in hexadecimal (base 16) and is most useful when working with addresses or logical operations on individual bits.

Type DECIMAL or HEX to change FORTH to the desired base before entering or displaying data in that base. FORTH will stay in the selected base until the base is changed or until FORTH is reinitialized (to DECIMAL). Note that DECIMAL and HEX affect the input and output data representation and not internal data handling.

Reinitialize FORTH and put the following numbers on the stack and print them using different combinations of DECIMAL and HEX .

```
COLD <RETURN> ( Initializes DECIMAL
 AIM 65/40 FORTH V1.4
```

Press <RETURN> after the word   in each of the following examples


```
16 .  16 OK
16 HEX .  10 OK
10 DECIMAL .  16 OK
255 .  255 OK
255 HEX .  FF OK
DECIMAL 32767 .  32767 OK
32767 HEX .  7FFF OK
DECIMAL -32768 .  -32768 OK
-32768 HEX .  -8000 OK
```

Note that DECIMAL numbers -1 to -32768 entered on the stack will be displayed in HEX in 2's complement form with a leading minus sign.

We will examine other number bases later (see Section 4.11.3).

4.2 STACK MANIPULATION

Since most FORTH words use the stack to hold input or output numbers, let's explore some FORTH words that are used to rearrange or copy numbers near the top of the stack. While these functions are sometimes necessary, you should avoid using them where possible. FORTH code is more readable when less stack manipulation is used. Common stack manipulation words are discussed here, however, to give you additional experience in working with the stack before proceeding into other FORTH word descriptions.

4.2.1 DUP , DROP , SWAP and OVER

The most common stack manipulation words are DUP , DROP , SWAP and OVER . Let's explore these, but first place some markers on the stack for reference

```
DECIMAL 333 222 111 <RETURN>
```

If we accidently pull too many numbers from the stack we will know where we are. Type .S to check

```
.S <RETURN>
111
222
333 OK
```

DUP pushes a copy of the top number onto the stack to create a new top number. In sequence

```
123 DUP . . <RETURN>
```

duplicates 123 on the stack then displays both numbers

```
123 DUP . . 123 123 OK
```

DROP deletes the top number from the stack. Try this with

```
456 789 DROP . <RETURN>
```

which deletes 789 and displays

```
456 789 DROP . 456 OK
```

SWAP exchanges the top two numbers on the stack. Put two numbers on the stack

```
456 789 <RETURN>
```

Use .S to look at the stack

```
.S <RETURN>
789
456
111
222
333 OK
```

Now swap the numbers on top the stack and examine the stack with

```
SWAP .S <RETURN>
```

which prints

```
456
789
111
222
333 OK
```


Notice that the top two number are reversed. Now try OVER which copies the second item to the top

```
OVER .S <RETURN>
789
456
789
111
222
333 OK
```

4.2.2 Test and Duplicate with -DUP

A related word -DUP duplicates the top number on the stack only if it is non-zero; otherwise -DUP does nothing. Continuing from the prior example, type

```
-DUP .S <RETURN>
```

to show that the top number was duplicated.

```
789
789
456
789
111
222
333 OK
```

Let's remove and display the top four numbers from the stack before continuing

```
CR . . . . <RETURN>
```

which displays

```
789 789 456 789 OK
```

Now, enter

```
Ø -DUP CR .S <RETURN>
```

which displays

```
Ø
111
222
333 OK
```

Notice that the top number was not duplicated. -DUP is usually used before an IF (see Section 4.8.1). In the non-zero case, some action is usually performed using the value; the extra copy made by -DUP is therefore removed by the IF processing. In the zero case, no additional action is performed, thus, the extra copy of the top number is not needed.

4.2.3 Delete the Top Stack Item with DROP

The word DROP deletes the top item on the stack. Drop the zero now and check the stack contents

```
DROP .S <RETURN>
111
222
333 OK
```

4.2.4 Rotate Stack Items with ROT

ROT rotates the top three items, moving the third item to the top, the previous top item to the second, and the previous second item to the third.

For example,

```
8ØØ 7ØØ 6ØØ .S <RETURN>
6ØØ
7ØØ
8ØØ
111
222
333 OK
```

Now rotate and print with

```
ROT .S <RETURN>
```


which outputs

```
8ØØ
6ØØ
7ØØ
111
222
333 OK
```

Now remove and display the top three numbers

```
CR . . . <RETURN>
8ØØ 6ØØ 7ØØ OK
```

4.2.5 Copy a Stack Item with PICK

PICK looks down any depth into the stack and copies the nth number from the top (not counting the n itself) and places it on top.

```
1 PICK
```

is the same as DUP , and

```
2 PICK
```

is the same as OVER . Put several numbers on the stack and check them

```
4Ø 5Ø 6Ø 7Ø 8Ø .S <RETURN>
8Ø
7Ø
6Ø
5Ø
4Ø
111
222
333 OK
```

Now pick the 4th item (i.e., 5Ø), and look at the results

```
4 PICK .S <RETURN>
5Ø
8Ø
7Ø
6Ø
5Ø
4Ø
111
222
333 OK
```

4.3 MEMORY OPERATIONS

Several FORTH words move data between the stack and memory, or from memory to memory.

4.3.1 16-Bit Store ! and Fetch @

The FORTH word

!

(pronounced "store") takes an address from the top of the stack and the 16-bit value beneath it and stores the value into the address (and address +1).

A corresponding word

@

(pronounced "fetch") takes an address from the top of the stack, fetches the 16-bit data from that address (and address +1) and replaces the address on top of the stack with the data from memory. Both the address and the data are specified in the current number base. Initialize FORTH and try

```
COLD
 AIM 65/40 FORTH V1.4
HEX OK
30FF 900 ! OK
900 @ CR .
30FF OK
```

which stores 30FF into addresses $900 and $901 with ! , fetches the contents of addresses $900 and $901 with @ and displays it with . . Return back to the AIM 65/40 Monitor and examine addresses $900 and $901 with the M command and note that data is stored in low-byte, high-byte order

```
ESC
{M}0900 FF 30 XX XX XX XX XX XX XX
```


Re-enter FORTH with

```
{6}
 AIM 65/40 FORTH V1.4
```

Try

```
DECIMAL
16000 HEX 900 ! OK
```

to store a decimal number in an address entered in hexadecimal. Now display the data in decimal by

```
900 @ DECIMAL CR .
16000 OK
```

which fetches the contents of addresses $900 and $901 and stores it on the stack, switches to the decimal mode, and outputs the data in decimal when . is commanded.

Now fetch and display the value in hexadecimal by

```
HEX 900 @ CR .
3E80 OK
```

4.3.2 8-Bit Store C! and Byte Fetch C@

Similar words allow byte length data to be stored and fetched. The word

C!

("c-store") stores the least significant 8-bits of the second item on the stack into the address determined by the number on top of the stack. The word

C@

("c-fetch") accesses the 8-bits stored at the address on top of the stack and stores it on top of the stack (replacing the address). Try

```
HEX OK
41 900 C! OK
F4 901 C! OK
```

which stores 41 and F4 into addresses $900 and $901, respectively. Display the contents of those address with

```
900 C@ 901 C@ CR . .
F4 41 OK
```

4.3.3 Initializing Memory with ERASE , BLANKS and FILL

Three words allow a block of memory to be initialized to various values.

ERASE fills memory with zeros ($00) starting at a specified address (second on the stack) and continuing through the number of bytes specified (top number on stack)

```
HEX 900 100 ERASE OK
```

Spot check with

```
900 @ 9FE @ CR . .
0 0 OK
```

Note that if the contents of $9FF were fetched, a non-zero number may be displayed since '@' fetches two bytes ($9FF and $A00) and address $A00 was not erased. The last byte could have been checked with

```
9FF C@ CR .
0 OK
```

BLANKS works like ERASE except that memory is initialized to ASCII blank ($20) instead of zeros. Try

```
HEX 900 100 BLANKS OK
900 C@ 9FF C@ CR . .
20 20 OK
```

4.3.5 Moving a Block of Memory with CMOVE

It is often useful to move a block of data from one area of memory to another. This can be done with the word CMOVE which takes three arguments on the stack: a from-address, a to-address, and a byte count. It moves the given number of bytes starting with the first address to the area of memory starting at the second address. Try

```
900 80 80 FILL OK
980 80 FF FILL OK
900 A00 8 CMOVE OK
980 A08 8 CMOVE OK
A00 10 DUMP
  A00  80 80 80 80 80 80 80 80
  A08  FF FF FF FF FF FF FF FF
OK
```

CMOVE works from the left to right, so be careful if the "from" and "to" memory areas overlap.

4.4 DEFINING YOUR OWN OPERATIONS

FORTH allows you to create your own operations. These new FORTH words become an integral part of the language, just like those which are pre-defined in AIM 65/40 FORTH. Your new words can take any number of arguments from the stack, and return any number of results.

The names of your operations can have up to 31 characters. They can use any ASCII characters except blank, delete and carriage return. For instance, an operation name could be a number, or even be non-displaying or non-printing control characters, although such names are discouraged. Even names already used by the system may be redefined as something else; therefore there is no reserved word list in FORTH. When a name is redefined, the old definition becomes inaccessible for later use in the program (although all earlier references to that name will remain as before). So, do not redefine a name if you want to use the old definition later.

Names which are descriptive of the function they perform make the code easier to read. Good choice of names is important for later use of the code, especially by other programmers.

As new words are defined, they are added to the FORTH vocabulary (described in more detail in Section 5.6). These definitions are normally stored in RAM starting at address $080E and build upward in memory. (They can also be stored in PROM/ROM as described in Section 10.) The FORTH word VLIST allows you to check what words have been added to the FORTH vocabulary

4.4.1 Colon-Definition

Suppose we want an operation to take the number on top of the stack, multiply it by 5, and print the result. Let's pick the name TEST-OP . We could define it simply as

```
: TEST-OP 5 * . ; <RETURN> OK
```

(Later we will rewrite this definition, using indentation and commenting conventions for more readable code). Enter the colon-definition as follows

a. Start the definition with a colon which tells FORTH to look ahead in the input stream for the word name. Follow the colon with a space.

b. Enter the word name (up to 31 characters). The FORTH word here is TEST-OP .

c. Enter the definition of the word. TEST-OP does the following

   1. Puts 5 on the stack
   2. Multiplies the top two numbers, i.e., the number on top of the stack when TEST-OP is executed by the 5 put on the stack by TEST-OP .
   3. Prints the result, i.e., the top number on the stack.

d. End the definition with a semi-colon (be sure to insert a space first). A FORTH definition may be continued on as many lines as needed.

This TEST-OP operation takes one number from the stack, as we have seen. It does not return any result (but if the . were omitted, the product would stay on the top of the stack). Note that no formal parameters are used to show the inputs and outputs of an operation. These are implicit -- TEST-OP takes one argument because it puts one number (5) on the stack then performs a multiply which uses two numbers (the 5 and one other). Check the operation of TEST-OP by placing a value on the stack and executing TEST-OP , e.g.

```
6 TEST-OP <RETURN> 3Ø OK
8 TEST-OP <RETURN> 4Ø OK
```

If the word being defined is already in the vocabulary dictionary, the message <name> NOT UNIQUE will be displayed. The NOT UNIQUE message is displayed only as a reminder that you have redefined a word which was previously defined and has no effect on the compilation process.

```
: TEST-OP 1Ø * . ;
TEST-OP NOT UNIQUE OK
```

and try

```
6 TEST-OP <RETURN> 6Ø
8 TEST-OP <RETURN> 8Ø
```

Note that only the new definition of TEST-OP is found and executed.

4.4.2 Find a Word in the Dictionary with '

Use the word ' (pronounced "tick") to find if a word is already contained in the dictionary and to return its parameter field address (PFA).

Type the word <name> after the word ' , i.e.

```
' <name>
```


FORTH will respond with OK for a found word and put the word's parameter field address on the stack (See Section 5.5 for description of the parameter field address). If not found the name is echoed with a "?" and the stack is cleared.

Check TEST-OP now (and print the address in the dictionary)

```
HEX OK
' TEST-OP <RETURN> OK
.S
82C OK
```

We can also run a VLIST to determine if TEST-OP is in the dictionary and to verify the address returned by ' . This is easy in this case since only two colon-definitions have been added to the dictionary and these two entries are printed immediately. Press any key to terminate VLIST.

```
VLIST
  82C TEST-OP          817 TEST-OP
  8Ø9 TASK            D9DC .S
 D9D1 MON             D9C1 HANG
OK                    ( a key was pressed here)
```

While both versions of TEST-OP are listed, only the version at address 82C is valid since it was defined last.

4.4.3 Print a Message with ."

You can print a message of up to 127 characters with the wor ." (dot-quote). Start the message one or more spaces after the ." word. Terminate the message with " (a double quote). Be sure to leave a space after the ." .

Now define a new word to use

```
: MULTIPLY
CR ." ANSWER="5 *    ; <RETURN> OK
```

and test it

```
DECIMAL
1Ø8 MULTIPLY
ANSWER=54Ø OK
1345 MULTIPLY
ANSWER=6725 OK
```

4.4.4 Commenting

Because the inputs and outputs are not explicit in FORTH code, it is very important to show them in the documentation. It is recommended that they be included as comments in the code and also in a separate glossary of operations. Each glossary entry should include the inputs, outputs and a short description of what the operation does -- usually two or three sentences are enough.

Comments in FORTH are enclosed in parentheses. A space must follow the left parenthesis because the left parenthesis is itself a FORTH operation. The closing right parenthesis need not be preceded by a space however, since it is a delimiter and not an operation. A <RETURN> also acts like a right parenthesis to terminate a comment. FORTH comments can be included on as many lines as needed; however, the comment must start with a left parenthesis followed by a space on each new line.

A conventional form of comment first lists the inputs, then three dashes, then the outputs. A period may be used to separate the last output word from the words of any description of the function of the operation. Therefore the TEST-OP definition could look like

```
: TEST-OP    N ---     MULT BY 5 AND PRINT)
   5 * . ;
```

A common style is to have only the colon, the word being defined, and the comment on the first line, then indent subsequent lines three columns. If the comment is too long, put it on the second line. There is no object code penalty for including comments and spaces so they can be used freely to improve readability.

When there is more than one input or output in a command, the right-most numbers are toward the top of the stack. A comment for a definition of a multiply operation might therefore be

```
: MPY    ( N1 N2 ---     MULTIPLY & PRINT
   * CR . ;
```


Note that an empty comment must consist of a left parenthesis, two spaces, and a right parenthesis. The reason is that the parsing word (WORD in FORTH) skips over leading occurrences of the delimiter. So if you leave only one space as in

the first character encountered by WORD is the right parenthesis, therefore the system skips it and continues looking for another right parenthesis.

4.5 EXECUTING AND COMPILING USING SOURCE

Up to now you have been operating in a manner where FORTH operations are compiled or executed immediately upon entry in an interpretive mode. If a new FORTH word is formed using a colon-definition (see Section 4.4) the word is immediately compiled and entered into the FORTH vocabulary upon completion of entry. Upon commanding the new FORTH word, the defined function is executed.

FORTH words can also be compiled and executed in a batch mode. In this mode, the FORTH words are compiled or executed upon entry from memory or mass storage. The source program for colon-definitions is not lost upon compilation with this technique, therefore, changes can easily be made without requiring re-entry of the whole program.

There are two methods of batch compiling in AIM 65/40 FORTH. The first method uses AIM 65/40 microcomputer Monitor/Editor capabilities to enter and edit source programs in FORTH and to load and save source and object programs. Entering and compiling source code using the AIM 65/40 Text Editor is explained in this section, while loading and saving FORTH source and object programs using an audio cassette recorder is described in Section 11.

The second method of batch compiling uses the standard FORTH technique of multiple RAM buffers and 1024 byte screens. This technique is commonly used for manipulating, saving, and retrieving data files on mass storage. This method is discussed in Section 12.

Perform the following steps to enter and compile FORTH source code using the AIM 65/40 Text Editor:

a. If you are in FORTH, return to the AIM 65/40 Monitor by pressing <ESC>.

(ESC)

b. Initialize the AIM 65/4 Editor above the maximum expected address for the compiled FORTH colon-definitions (remember that new words entered into the FORTH vocabulary start at address $80B and build upward).

```
{E}
 EDIT FROM=2000 TO=3FFF IN=<RETURN>
```

Type your FORTH source code in colon-definitions, for example,

```
: TEST-OP     N        MUL by 5 AND PRINT
   5 * . ;
```

d. Type any comment words to be executed during compilation. These words can serve as progress markers during compilation of large programs, e.g., ." 1" , ." 2" , etc. You may want to indicate completion of compilation with a different word or message. For example, enter

DONE"

Terminate your program with the FORTH word FINIS which indicates the end of the source program. Then type <RETURN> twice to end the text input

```
FINIS

 *END*
```


CAUTION

If FINIS is not included, your source program may be altered when compilation is attempted.

f. Quit the Text Editor and return to the AIM 65/40 Monitor.

={Q}

g. Enter or re-enter AIM 65/40 FORTH with the 5 or 6 key. If previous words have been compiled and the Text Buffer relocated below the previous source code using the Monitor C command, you may want to re-enter FORTH with the 6 key to save previous definitions. If you re-enter FORTH and compile the program, the latest word definitions will be used upon execution. In this case, the "ISN'T UNIQUE" message will be displayed as each word previously defined is compiled -- otherwise, enter FORTH with the 5 key to recompile the whole program.

```
{5}
 AIM 65/40 FORTH V1.4
```

NOTE

If words are repeatedly compiled without reinitializing FORTH or FORGETing previously defined words, the vocabulary may build up too high in memory and overwrite your source code. A common technique for preventing this is to FORGET <name> at the beginning of the source code in the Text Buffer. The dummy word TASK has been defined for just this purpose. FORGETing TASK then redefining it will remove all previously defined words from the vocabulary, e.g.,

```
FORGET TASK : TASK ;
```

Execute the SOURCE word to indicate that the FORTH program is to be input non-interactively, i.e., not from the keyboard. Be sure to press <RETURN> after SOURCE . When IN= is displayed, press M to tell AIM 65/40 FORTH that the input is from the Text Editor.

```
SOURCE <RETURN> IN=M DONE
OK
```

In this example the word DONE was displayed to indicate completion of input from the Text Buffer.

If the input is from the Editor, note that the source code is always compiled from the top of the Text Buffer. Should you desire to compile starting with a different line, you can use the AIM 65/40 Monitor C function (Recover Text Buffer) to move the top of Text Buffer.

If an error is detected during compilation, an error message is displayed and control returns to the FORTH command level. Consult Appendix E for the definition

of the error message anc required corrective action. Run a VLIST to the words properly defined and entered int dictionary to help locat the incorrect code.

If an error is detected during compilation from the Editor, the Monitor variable MEMRW ($02EF) will point to the last byte of source code read by FORTH. The bad source code can then be easily located in the Tex Buffer by checking memory around that address.

i. Check VLIST to verify the new word was entered in the FORTH vocabulary

```
VLIST
  817 TEST-OP            809 TASK
 D9DC .S                D9D1 MON
 D9C1 HANG              D97A VLIST
 D943 ? OK              ( <SPACE> was pressed here
```

j. Use the newly defined FORTH word to verify proper operation.

```
2 TEST-OP <RETURN> 10 OK
17 TEST-OP <RETURN> 85 OK
567 TEST-OP <RETURN> 2835 OK
```

k. You may want to define new FORTH words in terms of this word

```
: MUL TEST-OP ; <RETURN> OK
```

Verify the new word is in the FORTH vocabulary with VLIST

```
VLIST
  828 MUL               817 TEST-OP
  809 TASK             D9DC  .S
 D9D1 MON              D9C1 HANG
 D97A VLIST OK         ( <SPACE> was pressed here)
```

Verify proper operation of the new word

```
2 MUL <RETURN> 10 OK
87 MUL <RETURN> 435 OK
```

4.6 DO LOOPS

4.6.1 DO ... LOOP

The DO and LOOP statements allow repeated execution of a block of code. For example, the following definition creates a word SERIES , which prints a series of 25 numbers, zero through 24:

```
: SERIES   ( --- .  PRINT A SERIES
   CR 25 Ø DO I . LOOP ;
```

You may want t enter th source code Text Editor so you can easily cha e it and experiment w ferent values. Us the procedure scribed n Section 4. this. For example

```
 (ESC)
{E}
 EDIT FROM=2000 to=3FFF IN=<RETURN>
FORGET TASK : TASK ;
: SERIES ( ---. PRINT A SERIES)
   CR 25 Ø DO I . LOOP ;
." DONE"
FINIS


 *END*
={Q}

{5}
 AIM 65/40 FORTH V1.4
SOURCE  IN=M DONE
OK
```

Now execute

```
SERIES
Ø 1 2 3 4 5 6 7 8 9 1Ø 11 12 13 14 15 16
 17 18 19 2Ø 21 22 23 24 OK
```

Note that after the whole string is displayed the first line is redisplayed.

DO and LOOP must always be used as a pair. The code section which they enclose can be of any length. This code is executed repeatedly, and an index value I is available.

When DO sets up the loop (at run-time), it always takes two arguments from the stack. The top stack number (Ø) is the initial index value of the loop, and the second argument (25) is the final value plus one. If the initial value is zero, as is often the case, the second argument is the number of times around the loop. Also, ordering the loop limits this way makes the loop upper limit more accessible from outside a definition. We can see how this is done in the definition of NSERIES, below.

The loop index value is kept by the system and incremented automatically. The FORTH word I retrieves this index and copies it onto the stack. In the example above, the index value is zero the first time through the loop, then it is 1, 2, etc. through 24. In this example, the index is printed each time. SERIES takes no arguments from the stack and returns no results.

The recommended code-writing style for using DO and LOOP is to have the entire loop in a single line if possible; if not, LOOP should be indented to the same column as its corresponding DO. This style makes the program's structure easier to see.

The definition

```
NSERIES   ( N --- .  VARIABLE SERIES)
 CR Ø DO I . LOOP ;
```

creates NSERIES , which is almost like SERIES , except that it takes one argument from the stack, the number of times around the loop. You can use the Editor Text Buffer to enter the source code then compile both SERIES and NSERIES.

```
 (ESC)
{T}
FORGET TASK : TASK ;
={B}
FINIS
=U}/1
."  DONE"
={R} IN=<RETURN>
: NSERIES ( N ---. VARIABLE SERIES)
   CR Ø DO I . LOOP ;
."  DONE"
<RETURN>
={Q}
 AIM 65/4Ø FORTH V1.4
SOURCE  IN=M DONE
```

Now execute NSERIES

```
1Ø NSERIES
Ø 1 2 3 4 5 6 7 8 9 OK
```

Redefine SERIES now in terms of NSERIES , as

```
: SERIES    ( ---.   PRINT A SERIES)
   2Ø NSERIES ;
```

This redefinition will cause a "NOT UNIQUE" warning message to be printed. The warning can be ignored in this case; remember, it's purpose is to let you know that the word has also been defined previously. As mentioned before, FORTH allows any word to be redefined -- even the system words such as DO itself. Any further use of the word will refer to the latest definition, but all earlier uses still refer to the definition which was in effect when the earlier references were compiled.

Execute SERIES now.

```
SERIES
Ø 1 2 3 4 5 6 7 8 9 1Ø 11 12 13 14 15 16
 17 18 19 OK
```


In the examples above, notice that the only difference between SERIES and NSERIES is that the latter does not place a loop terminating value on the stack. Instead, it uses whatever was on the stack when NSERIES was executed. The NSERIES example also shows that the arguments to DO , the loop initial and terminating values, need not be literal numbers; instead they can be computed or obtained in any way. DO doesn't care how its arguments got onto the stack. This feature helps keep FORTH code modular and reduces side effects when changes are made.

DO ... LOOP , and the other control structures which will be introduced later, can only be used inside colon-definitions, i.e., they cannot be executed directly as commands at the terminal. DO and LOOP are in a special class of words called immediate words. These are not compiled like other words used in colon-definitions, but instead they execute at compile time to handle special compilation functions, e.g., to compile an internal branch back from the LOOP to its corresponding DO. Immediate words are discussed in Section 5.

An example of DO ... LOOP is a one millisecond time delay word:

```
: MS    ( N --- .  MILLISECOND DELAY)
   Ø DO 5 Ø DO LOOP LOOP
   CR ."  TIME-UP" CR ;
```

This word will cause delays of n milliseconds when used by putting n on the stack and then typing the word. To execute a 9 millisecond delay, simply enter

```
9 MS
```

At the end of the delay, the message

```
TIME-UP
```

is displayed. Try it with larger delays, e.g 1ØØØ, to visually notice the delay time.

4.6.2 +LOOP

The DO ... LOOP index always increments by 1. Another word, +LOOP , allows other increments. Each time around the loop, it takes a number off the stack for the increment, DO ... 2 +LOOP would increment by 2. The increment can be computed and it can change during loop execution. It can also be negative. The following word causes an odd number in the range 1 to N to be printed.

```
: ODD-SERIES    ( N --- .  PRINT ODD SERIES)
   1 CR DO I . 2 +LOOP ;
```

Execute ODD-SERIES with 25 as the input number (don't forget to put the input number on the stack or a STACK EMPTY error may occur.

```
25 ODD-SERIES
1 3 5 7 9 11 13 15 17 19 21 23 OK
```

4.6.3 LEAVE

LEAVE is another word used with DO loops. If LEAVE is executed within a loop, it will set the limit to the index value, causing the loop to exit when LOOP or +LOOP is next executed. LEAVE (and also the index I) can only be used inside a DO loop.

4.7 COMPARISON AND LOGIC OPERATIONS

The DO loop is one form of structured control in FORTH. Other structures described later (IF ... THEN , ELSE ... THEN , BEGIN ... UNTIL , BEGIN ... WHILE ... REPEAT , and BEGIN ... AGAIN) may test Boolean values (truth values) to control program execution. Comparison and logic words place Booleans on the stack and then the control words use these values.

4.7.1 > and

Simple FORTH comparison words are < (less than), > (greater than) and = (equal). Each of these operations takes two arguments from the stack (destroying those arguments) and returns one result (a Boolean) to the stack. The second item on the stack is compared to the top item in accordance with the FORTH word. If the comparison is true, a true ("1") is returned; if false, a false ("Ø") value.

4.7.2 U< , Ø< and Ø=

Other comparison operations are U< , Ø< and Ø= . U< (unsigned less than) compares the top two stack numbers as unsigned 16-bit integers (see Section 5.1). Ø< (zero less) and Ø= (zero equals) differ from the others in taking only one argument from the stack; it is tested for being less than zero (or equal to zero, respectively). Ø< leaves a true on the stack if the number is less than zero, otherwise a false is left. Ø= returns a true if the number equals zero, otherwise a false is returned. Ø= works the same as

```
Ø =
```

written as two words; similarly for Ø< The one-word forms are more efficient, however.

Ø= is equivale [illegible] to a logic [illegible] ıot", because [illegible] erses the truth value of [illegible] e top stack [illegible] ı (it changes [illegible] o 1, and or any other no [illegible] zero value [illegible] ro).

Experiment with the comparison operations

```
HEX
10 20 < . <RETURN> 1 OK
20 10 = . <RETURN> 0 OK
5 0= . <RETURN> 0 OK
5 5 - 0= . <RETURN> 1 OK
10 -10 < . <RETURN> 0 OK
10 -10 U< . <RETURN> 1 OK
1 0= 0= . <RETURN> 1 OK
8 0= 0= . <RETURN> 1 OK
```

Note that the Boolean false value is always zero and any non-zero value (not only '1') is taken as a Boolean true. However, the value returned by these comparisons is always zero or 1.

4.7.3 Logical Operations

Logical operations AND , OR , and XOR (exclusive OR) are provided. These are bit-wise operations. Each takes two arguments from the stack and returns one result. Each of the 16 bits of the result is obtained by applying the logical operation to the corresponding bits of the arguments. All bit positions are treated independently.

```
HEX
F7 01 AND . <RETURN> 1
08 01 OR . <RETURN> 9
F7 01 XOR . <RETURN> F6
```

The word NOT is provided as a synonym for 0= (see Section 4.7.2) to improve readability in logic expressions. Note that NOT is not a bit-wise operation; it is only a Boolean inversion and just returns the right-most bit of the word. To negate all the bits of a word (i.e., to take its one's complement), use

```
-1 XOR
```

For example

```
HEX
AAAA -1 XOR . <RETURN> 5555
AAAA FFFF XOR . <RETURN> 5555
```


These logical operations can also be applied to truth values returned by comparisons; in this case, only the right-most bit of each word is important. For example, suppose that a word ?HOT has already been defined to return a value of true if a sensor detects a temperature higher than a pre-set limit, false otherwise. Also suppose that a voltage value is previously stored on the stack. The test

```
8   ?HOT OR
```

will return true if the voltage (on the stack) is greater than 8, or the temperature is high, or both. In this example the voltage value on the stack is first compared to 8 by use of the relational operator. This results in a Boolean value left on the stack. Then ?HOT puts another Boolean on the stack and the two Boolean values are OR'ed together.

Note that

```
?HOT 8 > OR
```

would be erroneous in this case, because the Boolean left on the stack by ?HOT would be compared with the 8 and the result of that comparison (always false) would be OR'ed with the voltage that was on the stack before this phrase was extended.

4.8 CONDITIONAL CONTROL STRUCTURES

The following FORTH control structures test a Boolean result generated by the comparison or logical operations, and direct the flow of program execution accordingly.

4.8.1 IF ... ELSE ... THEN

As with other control structures, the IF and THEN must be used as a pair; if they are not, error message #19 or #20 will be generated (see Appendix E) at compile-time. Any correct block of FORTH programming may occur between the IF and the THEN

The IF takes one argument, a Boolean value , from the stack. If it is true (non-zero), the code between IF and THEN is executed; if false (zero), that code is skipped. In either case control resumes with the THEN . For instance,

```
GET-VOLTAGE 8 > ?HOT OR
IF SHUT-DOWN THEN
```

will execute the (predefined) operation SHUT-DOWN if the previously defined word GET-VOLTAGE returns a value greater than 8 or ?HOT returns true (or both).

An optional ELSE clause allows a block of code to be executed only if a test is false. For example, the simple control loop

```
10000 0                          ( Loop 10000 Times)
DO
   GET-VOLTAGE 8 > ?HOT OR       ( Danger?)
   IF GO-SLOWER ELSE GO-FASTER THEN
LOOP
```

repeatedly tests whether temperature or voltage exceed their limits, and executes predefined operations GO-SLOWER or GO-FASTER accordingly.

4.8.2 Nesting Control Structures

The previous example shows that control structures can be nested; an IF ... ELSE ... THEN is inside a DO ... LOOP . Any of FORTH's control structures can be nested within any other to any practical depth. The recommended coding technique is to keep each definition short and simple, breaking complex operations into two or more shorter ones. For this reason, great depth of nesting is not normally used. For instance, in the examples above, operating like GO-SLOWER and GO-FASTER may themselves contain complicated control, it is best to define them as separate words to avoid cluttering a single word with many levels of nesting. Also, this is an example of top down coding as GET-VOLTAGE , GO-SLOWER , GO-FASTER and ?HOT may not exist in final form yet as the programmer experiments with the overall design of the control loop.

Of course GET-VOLTAGE , GO-FASTER and ?HOT must exist in some form at least before the loop would compile in a definition. If not, the first unknown word name encountered would cause the error message

```
<name>?
```

to be output.

Another recommended coding style is to indent IF ... THEN or IF ... THEN ... ELSE like DO ... LOOP . Keep the whole structure on one line if it is short enough, otherwise, indent the IF , ELSE (if present) and THEN to line up vertically. Each new level of nesting structure should be indented at least one space.

4.8.3 Masking and Setting Bits

The operations used for masking -- selecting certain bits within a 16-bit word, and turning them OFF or ON, or complementing or testing them -- were largely covered in Section 4.7.3. This section further explores these operations in many of the control applications to which the AIM 65/40 microcomputer is well suited.

Mask values are best presented in hexadecimal. In hexadecimal, the values 0000 through FFFF can be input; a minus sign can also be used to input numbers 8000 to FFFF (-8000 to -1). The dot (period) works for output, but if the first bit is set, use the phrase

```
0 D.
```

(zero, double-precision print) instead, to avoid having the number interpreted as negative. This makes the top stack item a double integer, whose most significant 16 bits are zero, and then uses the double integer print word to output the resulting positive 32-bit integer.

In the following examples we will be changing or testing the last three bits of a word; i.e., the mask value will be 0007 (last three bits set, all others off). This value could be written simply as 7, but the leading zeros are conventionally used on address and mask values for program clarity. The mask of course need not be a literal value as shown in these illustrations; it could be computed, perhaps by previous logical operations, or input from the terminal, etc.

To turn ON the last three bits of the word on top of the stack (leaving all other bits unchanged), execute

```
0007 OR
```

The OR operation, as described earlier, does a logical OR of each bit independently. The sign bit is treated like any other. (In these examples, we will assume that HEX has been executed to set the number base to 16.)

Similarly to turn OFF the last three bits, use

```
FFF8 AND
```

To test if any of the last three are ON, use

```
0007 AND
```

The stack top will now be zero if none of the last three bits were ON, and non-zero otherwise. This value can be used as a Boolean by IF , UNTIL , or WHILE , but be careful if the value is used as input to another AND or OR ; input to these operations should be a Boolean zero vs. one, not zero vs. non-zero. If such further logic is to be done, use

```
0= 0=
```


which leaves the truth-value unchanged but converts the zero/non-zero result into a more correct zero/one Boolean. Use

```
0007 AND 0= 0=
```

or

```
0007 XOR
```

to complement (reverse values of) the last three bits of the top stack word.

To complement all bits. Use

```
FFFF XOR
```

(which could also be written

```
-1 XOR
```

because the numeric representations FFFF and -1 are the same in 16-bit 2's-complement arithmetic.)

With the operations AND , OR , and XOR , any truth-value functions of one, two or more arguments can be built.

4.8.4 <u>BEGIN ... Loops</u>

In a BEGIN ... UNTIL loop, the UNTIL takes a Boolean value from the stack. If false, it loops back to the BEGIN ; if true, it terminates the loop, i.e., the loop continues UNTIL a condition is true. The following loop executes until ?HOT is true (non-zero).

```
BEGIN
   PERFORM-AN-ACTION
   ?HOT ( STOP IF HOT)
UNTIL
```

The BEGIN ... WHILE ... REPEAT loop is almost opposite; it will continue to execute the statement(s) between WHILE and REPEAT while the condition between BEGIN and WHILE is true. WHILE tests the Boolean; if true, it does nothing, allowing control to remain in the BEGIN ... REPEAT loop; if false, it branches out of the loop (to beyond the REPEAT). The REPEAT always branches back to the BEGIN . The following loop is almost the same as the UNTIL loop above

```
BEGIN
   ?HOT Ø=
WHILE
   PERFORM-AN-ACTION
REPEAT
```

The difference is that the words contained between WHILE and REPEAT loop can execute zero times, but the words in the BEGIN ... UNTIL loop will always execute at least once since the test is made at the end of the loop, not the beginning. Note the use of Ø= (equivalent to a logical NOT) to reverse the truth-value returned by ?HOT.

A BEGIN ... AGAIN structure creates an infinite loop. AGAIN takes no arguments from the stack -- it always causes control to return to its corresponding BEGIN . This structure could be used in a real-time control program to execute a final procedure until interrupted. It is also possible to exit this loop with a ;S word.

All of these structures can be nested within any others. Again, avoid long or complicated definitions. Short definitions make programs easier to read, debug, and modify.

4.9 DATA STORAGE

How can you get an address of available memory to use for data storage?

Let's review the memory map (see Figure 2-1). The AIM 65/40 FORTH system occupies 8K of ROM (CØØØ-CFFF). This area contains definitions of the words already defined by the system. Your own word definitions start in RAM memory at $8ØB and continue upward. AIM 65/4Ø FORTH uses parts of RAM memory $Ø-$AB (see Appendix F) and $7ØØ-$7FF (see Appendix G) for its variables and buffers.

One way to find available memory for data structures is to use the top of your RAM memory and work down, since your word definitions start at $8ØB and work up. For instance, if you have 16K of RAM, addresses such as $3FFØ-$3FFF might be used for data, depending on the size of your program (i.e., the number and size of your word definitions). As described below the system uses the first 68 (decimal) bytes of available dictionary memory (after your latest definition) as its own scratchpad area; do not put data too close to the end of your definitions.

Another approach is to allocate memory for data within the dictionary -- the words CONSTANT , VARIABLE and ALLOT , described in the next chapter, do this.

4.9.1 Find Next Dictionary Location with HERE

The word HERE returns the address of the next available dictionary location. HERE can be used to determine the size, i.e., the memory required of a colon-definition.

The procedure is to type:

```
HERE   puts current dictionary address on stack)
```

Enter the colon-definition

```
: &lt;name&gt; --
```

Enter the current dic ary addre s on sta , swap to subtract the smaller address f the large , and th print the size of the defined word.

```
HERE SWAP - .
```

Enter the following example of a square function. Note the first available dictionary location before and after entry of the SQUARE colon-definition. The length of the colon-definition in the dictionary is $13.

```
HEX OK
HERE DUP . <RETURN> 80B OK
: SQUARE DUP * . ; OK
HERE DUP . <RETURN> 81E OK
SWAP - . <RETURN> 13 OK
```

Check the operation of the SQUARE word.

```
DECIMAL
4 SQUARE <RETURN> 16
```

4.9.2 Use PAD for Temporary Storage

A common location for temporary storage is the address returned by the word PAD , and the memory above. PAD returns a starting address 68 bytes beyond the next available dictionary location (which is returned by the word HERE). The space between HERE and PAD is used by AIM 65/40 FORTH itself for temporary memory; the byte at PAD and the locations above it are free for your temporary use. Let's restart and check the starting address of the FORTH dictionary using HERE and the starting address of the temporary storage area using PAD .

```
COLD
 AIM 65/40 FORTH V1.4
PAD HERE HEX .S
80B
84F OK
```

Verify that the PAD starts 68 bytes above the start of the FORTH dictionary

```
DECIMAL - CR .
68
```

Since PAD is located relative to the current top of the RAM dictionary it will change when any new words are defined, or when words already in the dictionary are forgotten. Usually this is not a problem because any particular test or run would move data into its temporary storage at PAD , and not rely on data stored there previously.

As an example, add a word to the FORTH dictionary that can be used to check where HERE and PAD are located, as other words are either added or deleted from the dictionary. Then use it first to check itself. Let's also define it in the Text Editor in case we want to modify it later.

```
{E}
 EDIT FROM=3000  TO=3FFF IN=<RETURN>
FORGET TASK : TASK ;
: CK-PAD ( ---. CHECK PAD & HERE)
  PAD HERE HEX
  CR ." HERE=" .
  CR ." PAD=" . ;
FINIS

 *END*
={Q}

{5}
 AIM 65/40 FORTH V1.4
SOURCE IN=M
OK

CK-PAD
HERE=837
PAD=87B  OK
```

Now the memory fetch and store words can be tested, using PAD as available memory. Try the sequence

```
DECIMAL OK
PAD 20 BLANKS OK
15 PAD ! OK
15 PAD 10 + C! OK
PAD 20 HEX DUMP
  879   F  0 20 20 20 20 20 20
  881  20 20  F 20 20 20 20 20
  889  20 20 20 20 45 D3 5A  8
OK
```

The output shows the blanks (ASCII $20), the 15 ($000F) stored as a word (with the bytes reversed by the 6502 CPU so it looks like 0F00' ,and the 15 ($F) stored as a byte 10 bytes later. The

```
10 +
```

shows use of an offset to an address; this technique can be used to create data structures such as arrays, records and fields, etc.

4.9.3 Increment Memory with +!

Two miscellaneous memory words are +! (pronounced "plus store") and TOGGLE . +! takes a stack value and a memory address and adds the value to the contents of the address; for example, it is used for incrementing counters in memory.

Define the word BUMP to increment the contents of address $9ØØ by one, eight times, and prints the contents of $9ØØ after each increment.

```
HEX
: BUMP
  CR 8 Ø DO 1 9ØØ
  +! 9ØØ C@ . LOOP ;
```

Initialize $9ØØ to zero and execute BUMP

```
Ø 9ØØ C!

BUMP
1 2 3 4 5 6 7 8 OK
```

Try it again but first initialize $9ØØ to $1Ø

```
1Ø 9ØØ C!
BUMP
11 12 13 14 15 16 17 18 OK
```

Define another function UPBY6 to increment the memory contents by six and display the results

```
: UPBY6
  CR 8 Ø DO 6 9ØØ
  +! 9ØØ C@ . LOOP ;
```

Clear $9ØØ contents and try it.

```
Ø 9ØØ C! OK
UPBY6
6 C 12 18 1E 24 28 3Ø OK
```

4.9.4 Exclusive-OR Memory Using TOGGLE

TOGGLE takes an address and a one-byte mask as arguments; it does an exclusive-OR between the byte and the address contents, updating the latter.

Experiment with TOGGLE by first initializing $9ØØ to $FØ

```
HEX OK

FØ 9ØØ C! OK
```

TOGGLE the value

```
9ØØ 55 TOGGLE OK
```

Print the result

```
9ØØ C@ . <RETURN> A5 OK
```

Note that both +! and TOGGLE could be performed otherwise using multiple FORTH words, however, these words are convenient.

4.1Ø CONSTANTS AND VARIABLES

4.1Ø.1 CONSTANT

The word CONSTANT creates a new FORTH word which returns a value to the stack whenever it is executed. For example,

```
5Ø CONSTANT X
```

creates a constant named X . When this new word is executed, it will return 5Ø to the stack. Print the value of X with

```
X . <RETURN> 5Ø
```

Constants are commonly used to give names to values which are fixed parameters in programs.

The same result could also have been accomplished by using a colon-definition,

```
: X 5Ø ;
```

But the former is more efficient in both memory use and run-time speed.

If it is ever necessary to change the value of a CONSTANT after entry in the dictionary it can be done using the following technique

```
<new value> ' <name> !
```

For example, to change the 50 in the prior example to 78, use

```
78 ' X
```

check it now with

```
X . <RETURN> 78
```

Note that trying to change the value of a constant, by putting a new definition of the constant in the dictionary after compiling a word using it, will not work since existing linkage to the prior value will not change. However, when compiling from the Text Editor, the value can be changed in the source code to allow the constant and the using words to be recompiled for proper linkage.

4.10.2 VARIABLE

VARIABLE is like CONSTANT , but the word it creates returns the address of a value instead of the value itself. Therefore new values can be stored into the variable. Try

```
50 VARIABLE Y       ( Defi e variable Y, initialize
                      to 5  )
Y @ . <RETURN> 50   ( Fetc  and print Y
60 Y !              ( Stor  60 into Y )
Y @ . <RETURN> 60   ( Fetc  and print Y
```

Although this example illustrates the use of the word VARIABLE to initialize the value (to 50), the better practice is to always create the variable as zero or some dummy value, and initialize if necessary in an initialization section of the code. If the program is later moved to ROM, the variable location will have to be in RAM, where it cannot be initialized at compile time (see Section 4.10.4).

4.10.3 Defining Words

CONSTANT and VARIABLE are both in a special class of words called "defining words". Defining words add new words to the dictionary. The only other defining word we have seen so far is the colon used to begin colon definitions. As with the colon, the names created by CONSTANT and VARIABLE can be up to 31 characters long and can redefine other names.

The AIM 65/40 FORTH system includes eight defining words which are commonly used: the colon, CONSTANT , VARIABLE , USER , VOCABULARY , CODE , <BUILDS ... DOES> , and ;CODE . Each defining word is equivalent to a data type or class of operations. Later we will learn how the user can create entirely new data types (new defining words) by using the special operations <BUILDS ... DOES> or ;CODE .

4.10.4 USER

USER is a defining word which creates a different kind of variable. A user variable, like an ordinary variable, returns an address of where a value is stored. But user variables store their values in a special "user area" which is always in RAM from address $700 through $77F; not in the dictionary (which may be in ROM). (The name "user area" originated on large, multi-user FORTH systems. Each user has a unique memory area for system variables, e.g., the number base currently in effect for that user, and the programmer's own variables.) The user variables are defined in Appendix G.

USER , like CONSTANT and VARIABLE , takes one argument from the stack, but the argument is not an initial value; instead it is an offset from $700 into the user area. For example,

```
96 USER A
98 USER B
```

creates two variables, A and B, with offsets of 96 and 98 bytes, respectively, from the user variables base address at 1792 ($700). USER is configured to allow offsets of 0-255 ($FF). Offsets between $56 and $7E should be used however, to place the USER variables at $756 through $77E. Note that offset values below 86 ($56) and above 126 ($7E) may cause conflict with other system user variables or the Terminal Input Buffer (see Appendix G). Be sure that your assignment allows one word (two bytes) for each user variable.

4.1Ø.5 ALLOT

FORTH programs can use arrays, records, virtual arrays (if mass storage is available), and other data structures. The most elegant way to create such structures is described in the chapter on user-defined data types. But a simple method which is sometimes good enough uses VARIABLE and another word, ALLOT .

ALLOT takes one argument from the stack and leaves space for that many bytes in the dictionary. For example,

```
Ø VARIABLE RECORD
```

creates a variable called RECORD ; two bytes are available for the value. Suppose 1ØØ bytes are needed. Then

```
Ø VARIABLE RECORD 98 ALLOT
```

would create the variable RECORD and leave the 98 extra bytes for it.

Suppose RECORD were to be used for a customer name and address; the programmer could create such operations as

```
: LAST-NAME Ø + ;
: FIRST-NAME 2Ø + ;
: MIDDLE-INITIAL 3Ø + ;
: ADDRESS1 31 + ;
: ADDRESS2 51 + ;
```

Then

```
RECORD FIRST-NAME
```


would return the address of the start of th FIRST-NAME field

In a similar manner, arrays can be generated and manipulated. To define an array of 3ØØ bytes, use

```
Ø VARIABLE ARRAY 298 ALLOT
```

To fetch the nth value of this array, one can use

```
: GETN ARRAY SWAP 2 * + @ ;
```

Type

```
41 GETN
```

to place the value of the 41st element onto the stack.

4.11 CHANGING THE NUMBER BASE

We have already seen the words DECIMAL and HEX , which set the number base to 1Ø and 16, respectively. FORTH can work in any number base (even above 16) but in practice only 1Ø, 16, 2, and perhaps 8 are commonly used.

The number base can be changed by storing the desired base value into the user variable BASE , which is available as part of the system. For example,

```
2 BASE !
```

sets FORTH terminal input and output t binary. The user could define a word to do this,

```
: BINARY 2 BASE
```

and then later just execute

```
BINARY
```

The words DECIMAL and HEX similarly change BASE ; for convenience, these words are already defined in the system as supplied.

Note that BASE only affects input and output. Internal computation is always in binary so there is no computation-speed penalty for using different bases. Also note that the base will remain as set until changed again.

You can easily determine the current I/O number base with

```
BASE @ DUP DECIMAL .
```

The word @ puts the value of BASE on the stack. DUP duplicates the base value for the later restore. DECIMAL converts the I/O number conversion base to decimal and . prints the base and removes it from the stack.

If you need to check the base often, you can define a colon-definition word to do it, such as

```
: BASE? BASE @ DUP DECIMAL . BASE ! ;
```

When a colon-definition is compiled, the base in effect at compile time is the one that counts. Notice that the following code is erroneous and fails to compile:

```
DECIMAL
: MASK HEX ØØFF OR ;
ØØFF?
```

The ØØFF is unrecognized because the base is decimal at compile time; the word HEX does not change the base immediately (as was intended), but compiles as part of the definition of MASK ; it would change the base when MASK was executed. The correct code is

```
HEX
: MASK ØØFF OR ;
DECIMAL
```


A possible source of confusion is the fact that in binary, the numbers 2, 3 and 4 (as well as Ø and 1) are correctly recognized on input. This happens because the numbers Ø-4 are so commonly used that they were made into constants to save memory space. Since these common numbers are FORTH words in the dictionary, they are recognized regardless of the number base in effect.

4.12 OUTPUT WORDS

4.12.1 Print Right-Justified with .R

We have already seen the word . (dot) used for printing numbers. Other operators are available to output single-precision and double-precision numbers left-justified and right-justified.

The word .R prints a 16-bit number right-justified in a field of a given width. It takes two arguments, the number and the desired field width; the latter is on top of the stack. For example,

```
4734 CR 1Ø .R  CR
                         4734
OK
```

prints 4734 right-justified 26 columns. Note the use of CR to cause OK to print on the following line.

Later (in Section 5.2.2) you will see that the corresponding double-precision (32-bit) output word D. prints a double-precision signed number left-justified, while D.R prints a double-precision signed number right-justified.

4.12.2 Output Spaces with SPACE and SPACES

The word SPACE outputs one space, and SPACES takes one argument from the stack and outputs that number of spaces; such as

```
CR . TEXT1" 4 SPACES    TEXT2" CR
TEXT    TEXT2
OK
```

4.12.3 Output a Number to the Display/Printer with EMIT

Use the word EMIT to take the top stack number as an ASCII value and output it to the display/printer. For example

```
DECIMAL 65 EMIT
```

outputs A to the display/printer.

Use EMIT in conjunction with the input word KEY (see Section 4.13.1) to display/print an entered character. Try it with

```
KEY <RETURN> <input character>
EMIT
```

Note that the input character (from the keyboard) is not displayed/printed by the word KEY -- only by EMIT. Now, define one word to do both

```
: ?KEY KEY CR EMIT CR ;
```

Check it with

```
?KEY <RETURN> A
A
?KEY <RETURN> #
#
OK
```

Now try a few other characters of your own choice -- try lower case letters also.

4.12.4 Output a String to the Display/Printer with TYPE

To print an ASCII string given its address and length (length on top of the stack), use TYPE . Try

```
HEX 900 10 TYPE
```


which displays 16 byte st 'ting from HEX address 900

This will convert whatever is in these locations to ASCII and output it -- which will display random characters and spaces until known data is placed in these locations.

Try it after first entering in string of data from the keyboard in RAM (let's use the PAD area for temporary storage) using the word EXPECT (see Section 4.13.2).

```
DECIMAL PAD 40 CR EXPECT
<character string> <RETURN> ( if less than 20 characters)
PAD 40 CR TYPE
```

Try it with a message of up to 40 characters. Note that if the string is less than 40 characters, whatever is in memory between the last entered character through the 40th character will be converted and displayed/printed.

4.12.5 Prepare to Output a String with COUNT

Sometimes a string is stc   s a length byt   llowed by the string itself, and only t   ldress of the s   g (of the length byte) is on the st   this is an alt   te form for storing a string.

To convert from this form, the word COUNT takes the address and return the arguments required by TYPE. Therefore,

```
COUNT TYPE
```

prints a string given the address of its length byte. Try the following

```
HERE COUNT CR TYPE
TYPE
```

More advanced output operations are discussed in Section 5.3, "Output Formatting". These allow you to create your own output formats which may include decimal points, dollar signs, commas, etc. More on string handling is discussed in Section 5.4.

4.12.6 Set the Active Output Device with ?OUT

The word ?OUT allows you to set the active output device to a device other than the display/printer. ?OUT calls the AIM 65/4Ø Monitor Subroutine WHEREO (see Section 7.7 in the AIM 65/4Ø System User's Guide). After ?OUT , enter the input code for the desired device as follows

```
?OUT <RETURN> <output device code>
```

where the output device code can be

```
<RETURN> or <SPACE> = display/printer
 P = printer
 F = floppy disk (user defined)
 S = Serial (user defined)
 T = audio cassette recorder (AIM 65/4Ø format)
 U = user defined
 V = user defined
 Other = display/printer
```

See Section 11 for audio cassette recorder I/O procedures. See Section 6.1 of the AIM 65/4Ø System User's Manual for user defined I/O guidelines. Refer to Sections 9 and 1Ø of the AIM 65/4Ø System User's Manual for audio cassette recorder and teletype interface information.

Once the active output device is selected, ?OUT does not have to be used again until the output device is to be changed.

4.12.7 Output a Character to the Active Output Device with PUT

The word PUT operates like the EMIT word but outputs the character on top of the stack to the active output device rather than the display/printer. Input a character and output it to the printer only with

```
?OUT <RETURN> OUT=P
KEY <RETURN> <input character>
PUT
```

Notice that the character is printed and not displayed.

4.12.8 Output a String to the Active Output Device with WRITE

The word WRITE outputs a string of characters to the active output device like TYPE outputs a string to the display printer. Put the starting address of the string and the character length (top of the stack) on the stack followed by WRITE to use it.

Try it by putting message in as you did with EXPECT and outputting it the printer

```
DECIMAL
PAD 4Ø CR EXPECT <RETURN> <input string>
?OUT <RETURN> OUT=P
PAD 4Ø CR WRITE CR
```

Notice the commands will not be echoed to the display until

```
?OUT <RETURN> OUT=<RETURN>
```

is entered

4.13 INPUT WORDS

FORTH handles input by taking all characters (tokens) separated by spaces and first trying to look them up in the dictionary. If the token is not in the dictionary, the system tries to make a number of it, using the number base currently in effect. Then if the token contains a non-digit character, the system reports an error condition by typing the token followed by a question mark, indicating an unrecognized word (see Appendix E).

Most programs can use the FORTH system itself for terminal input. You type the numbers onto the stack and execute operations to use them. Many programs run without a terminal so no special input is needed. You seldom need to write operations to accept input from the keyboard, except for turnkey programs which do not run under the FORTH interpreter (i.e., which do not give the 'OK' to the user). When special input is required, several primitive operations are available.

4.13.1 Input a Character from the Keyboard with KEY

The word KEY accepts a single character from the keyboard, returning its ASCII value to the top of the stack. It is the opposite of EMIT (see Section 4.12.3). It is often used to accept a single-letter menu choice from the user. The entry procedure is

```
KEY <RETURN> <character>
```

Note that the entered character is not displayed/printed. Upper or lower case letters may be entered, however, FORTH words must be in upper case.

Clear the stack with an undefined word, enter a character, and check the entered value on the stack.

```
Q
Q ?
HEX KEY <RETURN> A     Type A)
.S
41
```

Notice the hexadecimal representation of the ASCII code for the entered number. Change the I/O base to DECIMAL and check the value again

```
DECIMAL .S
65
```

Use EMIT now to output the numbers to the display/printer.

```
EMIT <RETURN> A
```

You can use the words KEY and . along with the I/O base to easily convert the ASCII code for an entered character into the number base of your choice. This is especially useful if you do not have an ASCII/HEX/DECIMAL conversion table handy.

To enter a number and display it in hexadecimal, use

```
KEY <RETURN> <input character> HEX .
```


To display an entered number in decimal, use

```
KEY <RETURN> <input character> DECIMAL
```

A word can easily be defined to display the entered number i both bases

```
: ASC
KEY DUP DUP CR EMIT HEX . DECIMAL . ;
```

The input procedure is

```
ASC <RETURN> <character>
```

Try it with a couple of numbers.

```
ASC <RETURN> A     (A will not be displayed/printed
A 41 65
ASC <RETURN> 1
1 31 49
ASC <RETURN> ?
? 3F 63
```

Experiment with a few other numbers a mpare your results with Appendix H.

4.13.2 Input a String from the Keyboard with EXPECT

The word EXPECT accepts a one-line string from the terminal. EXPECT takes two arguments from the stack, a starting address in RAM and a maximum length of the input string; it returns no result to the stack. When executed, EXPECT waits for the terminal input; it keeps accepting characters until you press <RETURN>, or until the maximum length is reached. Note that EXPECT terminates the input string with a null byte ($ØØ); be sure there is room for it in the input area.

For example, use EXPECT to prepare to input 15 characters, enter the data, then dump the input data in hexadecimal which represents the ASCII code for the input data (see Appendix H). After you type EXPECT , FORTH will wait for your input -- 15 characters maximum. Press <RETURN> to end the input early. Notice that the last byte dumped is the null byte.

```
DECIMAL OK
PAD 15 CR EXPECT
123456789012345OK
PAD 16 HEX DUMP
  86B  31 32 33 34 35 36 37 38
  873  39 30 31 32 33 34 35  0
OK
```

In this example, the temporary storage area specified by PAD (see Section 4.9.2) was used to store the input data.

Use TYPE to display the input data as it was entered:

```
DECIMAL OK
PAD 15 CR TYPE
123456789012345OK
```

Using the two preceding examples as a guide, set up an input of 40 characters and display it in HEX and in ASCII. Then establish a permanent input buffer area in RAM where you want it instead of PAD and try it again.

4.13.3 Set the Active Input Device with ?IN

In addition to the keyboard, AIM 65/40 FORTH allows you to specify a different active input device. The word ?IN does this by calling the AIM 65/40 Monitor subroutine WHEREI (see Section 7.7 in the AIM 65/40 System User's Guide). Use ?IN as follows

```
?IN <RETURN> IN= <input device code>
```

The acceptable codes are

<RETURN> or <SPACE> = keyboard
F = floppy disk (user defined)
S = serial (user defined)
T = audio cassette recorder (AIM 65/40 format)
U = user defined
V = user defined
Other = keyboard

See Section 11 for audio cassette recorder I/O handling. See Section 6.1 of the AIM 65/40 System User's Manual for user defined I/O considerations. See Section 9 of the AIM 65/40 System User's Manual for audio cassette recorder and teletype interface information.

Once the active input device is selected, ?IN does not have to be used again until the input device is to be changed.

4.13.4 Input a Character from the Active Input Device with GET

Like KEY , the word GET inputs one character. Unlike KEY, however, GET inputs the character from the active input device rather than just the keyboard. For example, GET can be used like KEY as follows

```
?IN <RETURN> IN= <RETURN>
GET <RETURN> <input character>
```

to input a character from the keyboard.

4.13.5 Input a String from the Active Input Device with READ

The word READ allows a string of characters to be input similar to EXPECT except that the string can be input from the active input device instead of just the keyboard. Use the word ?IN to first select the active input device. Try

```
DECIMAL OK
?IN IN= <RETURN>
PAD 16 CR READ
( Type 0123456789012345 6)
OK
PAD 16 HEX DUMP
  86B  31 32 33 34 35 36 37 38
  873  39 30 31 32 33 34 35 36
OK
```

Note that the input t displayed duri ntry and that a terminating null is aced in the inpu ea as with EXPECT

TYPE can also be used here to display the input data the form it was entered

```
PAD 15 CR TYPE
123456789Ø12345OK
```

4.13.6 Test for Character Input with ?TERMINAL

The word ?TERMINAL tests the terminal keyboard and [illegible]aves a true flag (1) on the stack if any key is depressed. An example of a word that waits for a key depression is

```
: ANY-KEY?  BEGIN  ?TERMINAL UNTIL ;
```


SECTION

ADVANCED OPERATIONS

5.1 OTHER SINGLE-PRECISION ARITHMETIC OPERATIONS

There are other FORTH arithmetic words that perform simple operations. While these words are not required for many elementary arithmetic operations, they simplify implementat[illegible] of more complex functions.

5.1.1 Modulus Operators MOD and /MOD

The word MOD takes a dividend (second on the stack) and a divisor (top of the stack), and leaves only the remainder o[illegible] division on the stack; for example,

```
22 7 MOD . <RETURN> 1
```

The word "/MOD" ("divide-mod") leaves both the quotient (top of the stack) and the remainder (second on the stack), for example

```
22 7 /MOD CR . .
3 1
```

5.1.2 Absolute ABS and Negate NEGATE

To get the absolut[illegible] of a number [illegible] the word ABS. For example, take the [illegible]te values of b[illegible]h a positive and a negative number

```
 22 ABS . <RETURN> 22
-22 ABS . <RETURN> 22
```

To reverse the sign of a nu[illegible] the word NEGATE. Negate both a positive and a negat[illegible] for example,

```
-33 NEGATE . <RETURN> 33
 33 NEGATE . <RETURN> -33
```

5.1.3 Simple Increment and Decrement 1+ , 2+ , 1- , 2-

Four words are included for convenience of incrementing or decrementing a value on the stack by one or by two. They are

1+	("one-plu	Increment by 1
2+	("two-plu	Increment by 2
1-	("one-min	) Decrement by 1
2-	("two-min	) Decrement by 2

Try the following examples,

```
1 1+ . <RETURN> 2
2 2+ . <RETURN> 4
3 1- . <RETURN> 2
5 2- . <RETURN> 3
```

5.1.4 Minimum MIN and Maximum MAX

When you wish to limit the range of number between a lower and upper value, the words MAX and MIN will compare the values of the top two numbers on the stack and leave only the greater, or smaller number, respectively.

```
  1  2 MIN . <RETURN> 1
-1Ø  5 MIN . <RETURN> -1Ø

  4  7 MAX . <RETURN> 7
-1Ø  5 MAX . <RETURN> 5
```

A word that will limit numbers to a range between 1 and 9 uses the following colon-definition:

```
: RANGE 1 MAX 9 MIN ;
```

For example:

```
 6 RANGE   <RETURN> 6
 1 RANGE   <RETURN> 1
 Ø RANGE   <RETURN> 1      (Ø is smaller than 1)
 9 RANGE   <RETURN> 9
1Ø RANGE   <RETURN> 9      (1Ø is larger than 9)
```


5.2 UNSIGNED MIXED AND DOUBLE-PRECISION ARITHMETIC

The FORTH stack is 16 bits wide, and the numbers we have seen so far are signed values internally formatted in 2's complement binary arithmetic. In this number representation, bit 15 (the most significant bit) contains the arithmetic sign, and bits Ø to 14 contain the numeric magnitude value. A 'Ø' in the sign bit indicates a positive number while a '1' indicates a negative number. A positive signed 16-bit number may range from Ø ($ØØØØ) to 32,767 ($7FFF) while a signed negative number may vary from -1 ($FFFF) to -32,768 ($8ØØØ). Signed values are used most often for arithmetic calculations.

16 bits can also hold an unsigned number, where bit 15 is interpreted as an additional order of magnitude rather than the arithmetic sign. In this case, bit 15 represents a value of 32,768 (2^{15}) with the sign implicitly positive. The value of a 16-bit unsigned number may, therefore, range from Ø ($ØØØØ) to 65,535 ($FFFF). Unsigned values are used most often for addresses.

5.2.1 Entering Double-Precision Numbers

AIM 65/40 FORTH also supports 32-bit (double-precision) 2's complement numbers. These are represented as two 16-bit numbers on the stack, with the high-order number on top. Double-precision allows positive or negative decimal integers in the range -2147483648 to 2147483647 to be used.

FORTH interprets an input number as double-precision <u>if there is a decimal point anywhere in it</u>. The location of the decimal point does not affect the input number (although the number of decimal places is saved in the system variable DPL in case you need to know it, see Appendix G). For example, '555555555.' and '.555555555' are input as the same number -- only DPL is different. Input the following numbers in double-precision format and display the contents of DPL to check the number of decimal places in the input number:

```
1ØØ. DPL @ . <RETURN> Ø
156.7 DPL @ . <RETURN> 1
365.12 DPL @ . <RETURN> 2
496.436752 DPL @ <RETURN> 6
```

Double-precision numbers are integers, with the decimal point used only as a flag to indicate double-precision; the programmer must keep track of any implicit decimal point information.

Input the following small numbers in double-precision format and print out the two 16-bit numbers that make up the number. Notice that the most significant 16-bits is zero for positive numbers and is -1 ($FFFF) for negative numbers (consistent with 2's complement notation).

```
456. . . <RETURN> Ø 456
23145. . . <RETURN> Ø 23145
-879. . . <RETURN> -1 -879
-1289.4 . . <RETURN> -1 -12894
```

Change to hexadecimal and repeat the examples. Notice the difference since each hexadecimal digit represents four binary bits.

```
HEX
456. . . <RETURN> Ø 456
23145. . . <RETURN> 2 3145
-879. . . <RETURN> -1 -879
-1289.4 . . <RETURN> -2 -2894
```

5.2.2 Printing Double-Precision Numbers

Now that you understand how double-precision numbers are stored on the stack, let's look at two FORTH words that print the data in double-precision format. The word D. (pronounced "d-dot") prints the top two numbers on the stack as a 32-bit number, left-justified. Repeat the previous examples in decimal


```
DECIMAL
456. CR D.
456
23145. CR D.
23145
-879. CR D.
-879
-1289.4 CR D
-12894
```

It is often desirable to print the data right-justified. The word D.R ("d-dot-r") prints a double-precision number, right-justified in a variable width field. The top number on the stack is the column in which the least significant digit of the data is to be printed, while the second number is the double-precision number (the data) to be printed. Try the example data one more time, but right-justify it in the 3Ø-column field as follows (if the number prints in the wrong column, you forgot to switch back to decimal)

```
456. 3Ø CR D.R
                              456
23145. 3Ø CR D.R
                            23145
-879. 3Ø CR D.R
                             -879
-1289.4 3Ø CR D.R
                           -12894
```

Define a word to print multiple d ecision numbers right-justified 15 columns.

```
: PRINT-RIGHT ( N---.)
  Ø DO CR 3Ø D.R LOOP CR ;
```

Enter the data on the stack and print it with PRINT-RIGHT . Place the numbers and the number of items on the stack before calling PRINT-RIGHT .

```
456. 23145. -879. -12894.
4 PRINT-RIGHT
                           -12894
                             -879
                            23145
                              456
```

5.2.3 Other 32-Bit FORTH Operators

There are several other double-precision FORTH words which are analogous to the single-precision operations.

Double-precision ad D+ 'd-plus" operates in the same manner as + , usi g the top two d ble-precision numbers on the stack as inputs and lea ving one double-precision number, e.g.,

```
3456. 6576. D+ D. <RETURN> 10032
```

DABS ("d-abs") returns the absolute value of a double-precision number similar to the single-precision word ABS.

```
-76543. DABS D <RETURN> 76543
```

DNEGATE ("d-negate") changes the sign of the double-precision number on the stack, allowing subtraction.

```
-768945. DNEGATE D. <RETURN> 768945
```

The word S->D ("s-to-d") converts a single-precision number on the top of the stack to double-precision number.

```
6758 DUP CR .
6758
S->D CR D.
6758
```

The operation D+- ("d-plus-minus") applies the sign of the single-precision number on top of the stack to the double-precision number beneath it. Note that a minus number on top always changes the sign of the double-precision number below. Note also that the single-precision number is removed by the D+- operation.

```
56789 -78 D+- D. <RETURN> -56789
```


5.2.4 Unsigned Compare U<

Addition and subtraction are the same for signed or unsigned numbers so there are no special operations for these. Comparison is different, however, so an unsigned compare word U< ("u-less-than") should be used instead of the signed compare word < . Using < in a comparison where one number exceeds 32,767 will result in an incorrect answer. The comparison

```
20000 40000 < . <RETURN> 0
```

gives 0 (Boolean false), because 40,000 as a signed 16-bit number is negative and is therefore less than 20,000. The comparison

```
20000 40000 U< . <RETURN> 1
```

yields 1 (Boolean true) which is the correct result. Use U< to compare addresses, unless you are sure both of them will be below 32,768, or both above it.

5.2.5 Unsigned Multiply U* and Divide U/

Two other unsigned operations are provided. The unsigned multiply word U* ("u-times") multiplies two unsigned single-precision numbers to give an unsigned double-precision number. For example,

```
40000 40000 U* CR D
1600000000
```

The unsigned divide word U/ ("u-divide") divides a unsigned double-precision number (second on stack), by an unsigned single-precision number (top of stack), to give an unsigned single-precision quotient (top of stack) and unsigned single-precision remainder (second on stack).

The foll ing example gives a positive quotient and unsigned remainde

```
        120031  4 U/        <RETURN> 30007
```

Note that another example,

```
        140035. 4 U/        <RETURN> -30528 3
```

appears to give a negative quotient and unsigned remainder. In the single-precision format a number between 32,768 and 65,535 is displayed as negative unless printed as a double-precision number. The following example forces the quotient to a double-precision number and prints it along with the remainder.

```
        140035. 4 U/ 0 D. . <RETURN> 35008 3
```

5.2.6 Mixed-Mode Operations M* , M/ , and M/MOD

Some mixed-mode operations are also available. The operator M* ("m-times") multiples two signed numbers and returns a signed double-precision product. Two examples illustrate the operation.

```
        4532 8765 M* D. <RETURN> 39722980
        4876 -5467 M* D. <RETURN> -26657092
```

The operator M/ ("m-divide") divides a double-precision number (second on stack), by the single-precision number (on top of the stack), and returns a signed single-precision remainder (second on stack) and signed single-precision quotient (top of stack). Try this example:

```
        564755. 500 M/ . . <RETURN> 1129 255
```


The word M/MOD ("m-divide-mod") divides a positive double-precision number (second on stack) by a positive single-precision number (top of stack), returning an unsigned single-precision remainder (second on stack) and an unsigned double-precision quotient (top of stack). Examine with

```
        54000. 5000 M/MOD D  . <RETURN> 10 4000
```

5.2.7 Scaling

Suppose you are working with 16-bit integers and want to multiply one by a scaling factor such as the sine of 45 degrees. Since we are using only integers, this sine value (0.7071) could be represented as multiplied by 10000, i.e., 7071. We want to multiply our number by 7071 and divide it by 10000 -- the problem is that the intermediate product is too large to represent as 16 bits --- so FORTH provides an operation */ ("times-divide") which multiplies the third term on the stack by the second item and then divides the result by the top of stack item, while keeping a 32-bit intermediate product This is illustrated by

```
        12345 7071 10000 */ . <RETURN> 8729
```

Another operation */MOD ("times-divide-mod") performs the same operation but also returns the remainder as the second number on the stack. Repeat the last example but also print the remainder

```
        12345 7071 10000  */MOD .   <RETURN> 8729 1495
```

5.3 OUTPUT FORMATTING

The numeric output commands described in Section 4.11.1 are enough for most programs. However, some applications need special formats such as decimal points and dollar signs with printed numbers, or colons within numbers to indicate degrees, minutes and seconds. FORTH includes special output operations which let you define your own numeric formats.

5.3.1 S->D , <# , #S , SIGN and #>

To use these operations, first get a double-precision number on the stack. Then a special operation <# ("less-sharp") must be used to start numeric conversion. Digits are converted from the right, i.e., least significant digit first. ASCII characters such as decimal points and dollar signs can be added where needed. Then another special operation #> ("sharp-greater") must close the conversion.

For example, the following definition creates and tests a word .PRINT , which works like the print command . . This example illustrates a fairly simple case with no added character.

```
: .PRINT
   S->D SWAP OVER DABS
   <# #S SIGN #>
   TYPE SPACE ;
```

Enter a number to test .PRINT

```
12345 .PRINT <RETURN> 12345
```

First, S->D converts the top stack number to double-precision. The SWAP OVER , in effect, makes an extra copy of the high-order 16-bit part below the double-precision number of the stack; this is required to preserve the sign information since the numeric conversion itself requires a positive number -- hence the DABS.

The <# sets up the output conversion followed by the #S ("sharp-S") which converts all digits of the number to ASCII. The SIGN word then places an ASCII minus sign if necessary; it uses the extra copy of the high-order part of the double-precision number to detect if that number was originally negative.

The #> closes the conversion, and leaves stack arguments set up for TYPE -- i.e., the number of characters to type on top of the stack, and the address of the first one below it. The SPACE word leaves one space after the number to separate it from the next one.

5.3.2 # and HOLD

Here is an example showing creation of a word D$. which prints a double-precision number with decimal point and dollar sign. Besides the above operations, it also uses # ("sharp") which places a single digit into a string being created. It also uses HOLD which takes an ASCII value from the stack and places that character into the number being formed.

The following colon-definition shows how to convert digits, individually, placing additional characters such as decimal points and dollar signs where desired within a number.

```
DECIMAL
: D$. ( D ---)
   SWAP OVER DABS
   <# # # 46 HOLD      ( 46 is the decimal point)
   #S 36 HOLD SIGN #> ( 36 is the dollar sign)
   TYPE SPACE ;
```

The following examples show that the leading zeros are handled properly

```
555. D$. <RETURN> $5.55
5. D$. <RETURN> $Ø.Ø5
```

If three places after the d mal point were desired, one additional # would be nec ary before the '46'.

Let's define another word that uses D$. to print multiple numbers

```
: PRINT-D$.
   CR Ø DO D$. CR LOOP ;
```

Now put four numbers on the stack and print them

```
123. 45678.
3456. 23456.
4 PRINT-D$.              (Print four numbers)
$234.56
$34.56
$456.78
$1.23
```

The following word prints a mixed number when the integer double-precision number is on top of the stack and the position of the decimal point is held in the user variable DPL .

```
HEX
: XN.
SWAP OVER DABS          (Set form for sign and
                         conversion)
<# DPL @ -DUP           (Convert digits to right of
IF Ø DO # LOOP THEN      decimal point)
2E HOLD #S SIGN #>      (Convert decimal point and
                         and remainder of digits)
TYPE SPACE ;            (Print results)
DECIMAL
```

Verify proper conversion with an example such as:

```
34.786  XN.  <RETURN> 34.786
```

5.4 STRINGS

FORTH does not have a standardized package of string-handling operators, but it does have primitive operations from which string routines can be built. For many applications the primitives themselves are enough. A series of string handling functions that can easily be constructed in FORTH is described in Appendix I.

Because there is no ready-made standard, you can decide how to represent strings internally. Two formats are already in use within the system. In one, a length byte is followed by the string itself; string length cannot exceed 255 characters. The address of the string is the address of the length byte (this is used to store names of words in the dictionary). In the other format, only the string itself is stored in memory; its address is the address of its first character. The length is stored separately, and kept above the string address on the stack.

5.4.1 Address String Data with COUNT

The COUNT word returns the address (second on stack) of a character string and the number of characters, e.g., bytes, in the string (top of the stack). The character string can be up to 255 bytes in length. COUNT operates on the address preceding the first byte of the character data which must contain the number of bytes of the character data.

5.4.2 Output String Data with TYPE

The word TYPE takes the address of the first data byte (second on stack) and the data byte count (top of stack) and outputs it to the active output device. TYPE is usually preceded by COUNT which sets up the data address and byte count in a compatable format.

5.4.3 Input String Data with EXPECT

The word EXPECT (see Section 4.11.2) can be used to read a string into memory. Unfortunately it does not return the actual length of the input string; however, you can find this length if it is needed by searching for the trailing nulls (binary zero bytes).

5.4.4 Suppress Trailing Blanks with -TRAILING

To eliminate trailing blanks of a message, the word -TRAILING is used. If -TRAILING is given an address of a string (second on stack) and a count (top of stack) such as that output by COUNT , then -TRAILING will adjust the count to commands if necessary to eliminate any trailing blanks in the string. For example

```
HEX 9
9ØØ 9 EXPECT <RETURN>
```

allows nine characters to be entered into memory starting at $9ØØ. Enter

```
ONLY5 (followed by four spaces)
```

immediately after the <RETURN> following EXPECT (note that OK will not be displayed until after nine characters are entered). A five character message with four trailing blanks is now in RAM. Check it with

```
900 9 DUMP
  900  4F 4E 4  59 35 20 20 20
  908  20  0 )  XX XX XX XX XX
OK
```

Notice the terminating null character ($0) placed after the entered data. Now enter

```
900 9 -TRAILING  S
5                        (character count less trailing
                          blanks)
900                      (starting address)
```

To see the full message less trailing blanks, enter

```
CR TYPE CR
ONLY5
OK
```

5.4.5 Interpret a Number with (NUMBER)

Most of the words needed for terminal input are described in Section 4.11. This section covers the special situation of accepting a numeric string as input and interpreting it as a number. Such special input is seldom necessary, because most programs can accept input from the FORTH system itself (i.e., numbers typed onto the stack), if they use a terminal at all. This special terminal input is most often for turnkey programs not run under the direct control of FORTH (in which the user should not see the OK).

First use EXPECT to accept a string from the user (see Section 4.11.2). Then use (NUMBER) to interpret part or all of that string as a number (the parentheses are part of the name). This operation is a bit complicated. It needs a double-precision zero on the stack, as well as the address of

the first ASCII character of the number minus one, i.e., the address of one byte before the number begins. This address must be on top of the stack. (NUMBER) then returns the value of the number; it is accumulated into the double-precision zero. The address on top of the stack is incremented to point to the first non-numeric character, i.e., to the terminator of the number; the program may test this terminator, which would normally be a blank, and if it is an unexpected quantity, e.g. a letter erroneously typed by the terminal operator, error handling can be performed.

For example

```
: INPUT
   PAD 10 EXPECT 0 0 PAD 1 - (NUMBER) ;
```

defines a word

```
INPUT
```

which when executed, accepts a number, returning the address just beyond the number, and the number itself in double-precision form (as two numbers on the stack). (NUMBER) will not skip leading blanks or handle minus signs; you must do so if necessary. By defining INPUT , you have handled the difficult part of (NUMBER) just once. Subsequent inputs can be processed easily by using the INPUT word.

5.4.6 Input a Number with NUMBER

The word NUMBER (written without the parentheses) will handle leading blanks and the minus sign. But if the string being converted is in error (e.g., contains alphabetic letters), FORTH will handle the error itself by echoing the unrecognized string with a question mark; the user cannot get control to process the error differently. Therefore the more primitive (NUMBER) is usually preferred for turnkey applications.

5.5 DICTIONARY STRUCTURE

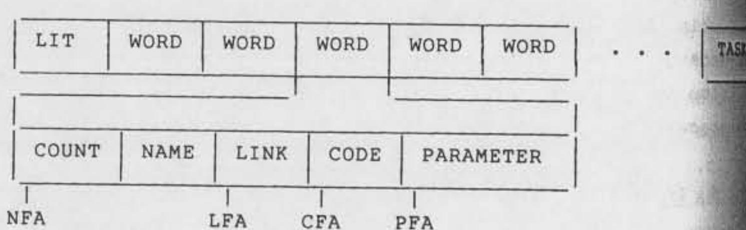
As you are well aware by now, FORTH consists primarily of a dictionary of words. The FORTH words were listed using VLIST in Section 4 and are shown in Table 4-1. This section describes the structure of the words in the dictionary.

5.5.1 FORTH Word Structure

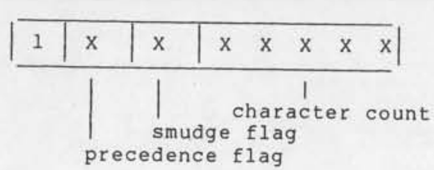
The FORTH words are arranged one after the other, starting with LIT to TASK , followed by all user-created words. Each word is composed of five sections:

- . flag bits and name character count
- . name
- . link address
- . code address
- . parameter field

Here is a picture of the dictionary with a word expanded with its sections:



The first byte of a word begins the name field and contains the number of characters in the word's name along with two flag bits:



The MSB is set to indicate the start of a name. The precedence flag indicates if the word is for compile or immediate execution. The smudge flag prevents the word from being found in the dictionary during compilation. If the compilation finishes successfully, the smudge bit is reset to zero allowing the name to be recognized. "SMUDGED" words show up in a VLIST.

The name field continues with the ASCII characters of the name with the MSB of the last character set to indicate the end of the name.

The link address is the address of the count byte of the previous word (i.e., the beginning of the previous name field). This allows the dictionary to be scanned, word-by-word, beginning with the most recent word and moving back. The last word in the dictionary has a link address of zero.

The code address indicates the code to be executed depending on the type of word, i.e.,

```
code  =  $C723          for  "colon-definition" words
      =  $C7AD          for  USER words
      =  $C77B          for  VARIABLE words
      =  $C75F          for  CONSTANT words
      =  next address   for  "CODE-definition" words
```

The parameter field changes meaning depending on type of word. If the word is a "colon-definition" word, the parameter field contains the addresses of the FORTH words that make up the definition. If the word is a "CODE-definition" word then the parameter field contains the actual assembly code for the logic to be performed.

Examine the TASK word; as an example,

```
HEX OK
800  C DUMP
  800  84 54 41 53 CB D5 D9 23
  808  C7 32 C5 XX XX XX XX XX
OK
```

For both CONSTANT and VARIABLE words, the parameter field is two bytes long and contains the value of the constant or variable. For USER words, the parameter field is one byte long and contains the offset into the user area for the USER variable.

Now look at it's component parts:

```
800  84           ( 8 =   MSB = 1 = start of a word)
                  ( 4 =   Number of characters in TASK)

801  54 41 53 CB  ( ASCII characters for  TASK  with)
                  ( MSB of last character set to 1)

805  D5 D9        ( Link address of $D9D5 links to .S)
                  ( word in AIM 65/40 FORTH ROMs)

807  23 C7        ( Code address of $C723 indicates)
                  ( colon-definition)

809  32 C5        ( Parameter address of $C532)
                  ( indicates the end of a)
                  ( colon-definition, i.e., ';')
```

5.5.2 Handling FORTH Word Addresses

There are five FORTH words concerned with finding the address of the various word fields. They are:

'	(tick)
PFA	(Parameter Field Address)
CFA	(Code Field Address)
LFA	(Link Field Address)
NFA	(Name Field Address)

a. ' leaves the parameter field address (PFA) of the following word on the stack.

b. NFA converts the parameter field address on the stack into the name field address (NFA).

LFA converts the PFA into the link field address (LFA).

d. CFA converts the PFA into the code field address (CFA).

PFA converts the name field address (NFA) to the parameter field address.

5.5.3 FORTH Word Handling Examples

To print the contents of LFA of CLIT , perform

```
HEX
' CLIT  LFA @ 0 CR D
C04F
```

To print the name of LIT , perform

```
' LIT NFA COUNT 1F AND CR TYPE
LIT
```

To print the topmost word name in the dictionary, perform

```
LATEST CR ID.
TASK
```

A simple list of all words in the FORTH dictionary can be obtained with

```
: DIR CR LATEST
  BEGIN
  DUP ID. CR
  PFA LFA @ DUP
  0= UNTIL ; OK
  DIR <RETURN>
  DIR
  TASK
  .S                  (Press <RESET> to terminate list)
```

5.6 VOCABULARIES

Vocabularies are groupings of FORTH words. They are used to allow the same names to be used for different operations in different application areas. If a name is redefined in the same vocabulary, only the latest definition will be accessible. But, if the same name is used in two or more different vocabularies, all the definitions can be selected. Every word defined in FORTH should be in only one vocabulary to minimize confusion between word usage.

The AIM 65/40 FORTH system as supplied includes two vocabularies: FORTH , which is the default vocabulary, where the example definitions illustrated earlier in this manual were all placed, and ASSEMBLER , which contains definitions of R6502 instruction mnemonics, mode symbols, and other operations only used for the assembler (See Section 6). For example, AIM 65/40 FORTH has two words, 0= and 0< , which are defined in both vocabularies and used differently (see Section 4.7.2 and 6.6) depending on which vocabulary is selected (see Section 6.1).

5.6.1 More on VLIST

As mentioned at the beginning of Section 4, you can list the FORTH vocabulary by executing the word

    VLIST

Press any key to terminate the VLIST . VLIST can also be used to list the words contained in the assembler vocabulary (see Section 6). Enter

    ASSEMBLER VLIST

which will print the ASSEMBLER vocabulary (and then link to the FORTH vocabulary and print that also). The FORTH link word (no name) is shown at address $726 in the VLIST. Then it is wise to execute

    FORTH

to set the vocabulary back to FORTH .

Vocabularies are effective only at compile time; they have no meaning after object code has been compiled. They only affect the search for names of words in the dictionary.

5.6.2 CONTEXT and CURRENT Specify Vocabularies

At any given time, two vocabularies are in effect: CONTEXT and CURRENT . CONTEXT specifies the vocabulary in which dictionary searches begin, while CURRENT gives the vocabulary into which new definitions are placed. Often CONTEXT and CURRENT are the same; e.g., when AIM 65/40 FORTH is initialized (initial entry or COLD word), both of them point to the FORTH vocabulary. But when a CODE-definition is being assembled, the CONTEXT vocabulary is ASSEMBLER , while CURRENT is usually FORTH or something else (CURRENT would be ASSEMBLER only if you were adding new capabilities, e.g., macros, to the assembler).

To set the CONTEXT , just execute the name of a vocabulary; e.g.,

    ASSEMBLER

switches to the ASSEMBLER vocabulary. To set the CURRENT , the word

    DEFINITIONS

changes the CURRENT to the CONTEXT . So to change both of them to ASSEMBLER , execute

    ASSEMBLER DEFINITIONS

Now any new colon, CODE- , or other definition will go into the ASSEMBLER vocabulary. Remember to get back by executing

    FORTH DEFINITIONS

after you are done extending the assembler

Incidently, any colon or other new definition will set CONTEXT back to CURRENT . This is done to help the programmer avoid errors. So if you are in FORTH and then execute just

    ASSEMBLER

without DEFINITIONS , and then define any new words, they will go into FORTH , and also the CONTEXT will be set back to FORTH ; i.e., executing ASSEMBLER alone will have had little effect.

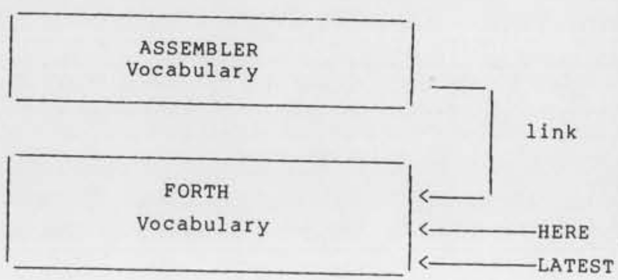
5.6.3 Use LATEST and HERE to Check Directory Addresses

The word LATEST leaves on the stack the name field address of the last word pointed to by CURRENT . Do a COLD start and check the FORTH dictionary

    HEX LATEST <RETURN> . 800

The word HERE leaves on the stack the next available dictionary address where new words can be added.

    HERE . <RETURN> 80B



5.6.4 Application Libraries

You can create your own vocabularies, in order to keep different application libraries separate from each other. Just execute

    VOCABULARY <name>

where <name> is the name (up to 31 characters) you want the new vocabulary to have. Then you would usually say

    <name> DEFINITIONS

and begin putting your application library words into the <name> vocabulary.

In the AIM 65/40 FORTH system, the new vocabulary will be linked to whatever vocabulary it was created in (usually FORTH). All vocabularies form a tree, allowing subvocabularies nested to any depth. All vocabularies from CONTEXT along the branching path back to the root of the tree (which is always FORTH) will be searched whenever a name is entered into the FORTH system for execution or compilation.

To create a new vocabulary, use the word (2) VOCABULARY (2) along with the vocabulary name to change (2) CONTEXT (2) to point to its last word, e.g.,

    VOCABULARY NEW

To add words to NEW, now type

    NEW DEFINITIONS

because DEFINITIONS sets CURRENT equal to CONTEXT allowing new words to be added to the NEW vocabulary.

Now add a new word

    : MYWORD ." NEW VOC" ;

and type VLIST and get

```
VLIST
  826 MYWORD           726
  813 NEW              809 TASK     (word in NEW)
D9DC .S                D9D1 MON     (link NEW to FORTH)
D9C1 HANG              D97A VLIST   (new vocabulary)
D943 ?                 D937.
D927 .R OK             ( <SPACE> bar pressed here)
```

Now type FORTH , this will set CONTEXT back to the FORTH vocabulary and MYWORD will not show up on a VLIST but it will execute.

Now type FORTH DEFINITIONS , changing both CURRENT and CONTEXT to the FORTH dictionary. Now MYWORD will not show up in VLIST and will not execute. To use MYWORD one needs only to link the NEW vocabulary to FORTH by typing NEW .

It is generally recommended that use of subvocabularies be avoided and all user-defined vocabularies be created in FORTH. This is for compatibility with many other FORTH systems which only allow one level of vocabulary nesting.

Vocabularies are optional, needed for advanced users only. Most programs only use the default FORTH vocabulary, and the programmers do not even need to know that vocabularies exist.

5.7 IMMEDIATE WORDS

Most FORTH words will be compiled, not executed, when they are used inside a colon-definition. Immediate words are the exception. They are executed even at compile time.

The words used for conditional branching and looping (e.g., IF , THEN , DO , LOOP , BEGIN , etc.) are all immediate words. They execute at compile time in order to handle forward or backward branch references, various error checks, and other functions. Some of these words such as DO and LOOP place special run-time words, not used directly by the programmer, into the object code. But some, (e.g., BEGIN) place nothing at all in the object code.

To define a new immediate word, use IMMEDIATE after its definition, i.e., after the semicolon. This causes the last word defined to be immediate.

On rare occasions the programmer must force compilation of an immediate word. To do this, use [COMPILE] (the brackets are part of the name.

For example, suppose you want to run source code written for an older version of FORTH which used the name ENDIF for THEN (AIM 65/40 FORTH supports both of these words). You don't want to go through the code and make all the changes. It would be wrong to define ENDIF by

```
ENDIF THEN ; IMMEDIATE
```

because the THEN would try to compute a conditional branch and cause an error message because there is no corresponding IF . The correct form would be

```
: ENDIF [COMPILE] THEN ; IMMEDIATE
```

This defines ENDIF to work the same as THEN .

5.8 CREATING YOUR OWN DATA/OPERATION TYPES

The AIM 65/40 FORTH system includes several 'defining words'; that is, words which create new words. The most important of those are: (the colon), CODE , CONSTANT , VARIABLE , USER , and VOCABULARY .

You may want to create new defining words. In general, each new defining word creates a new type of data structure of operation. Examples might be ARRAY , MATRIX , CUSTOMER-RECORD , and VIRTUAL-ARRAY . FORTH assemblers use similar structures for classes of instructions, such as one-address and two-address.

New data or operation types are usually created by the pair words <BUILDS and DOES> ; these words are always used together. The word ;CODE is an alternative way to create new data structures; they run faster but require use of the assembler (see Section 6.9).

For example, suppose we want a word to create arrays of 2-byte (16-bit) memory locations numbered from zero. We want to say, e.g.,

```
50 ARRAY X
10 ARRAY Y
```

to create arrays 'X' and 'Y' with 50 and 10 elements, respectively. Then we want to use these arrays as

```
0 X                 (0th element of ARRAY X)
49 X                (49th element of ARRAY X)
0 Y                 (0th element of ARRAY Y)
9 Y                 (9th element of ARRAY Y)
```

to return the addresses of the first (0th) and last elements of X and Y. We can then use the arrays to store and fetch data using ! and @ . Note that there are 50 elements in ARRAY X (numbered from 0 to 49) and, similiarly, there are 10 elements in ARRAY Y (numbered from 0 to 9).

How do we define ARRAY to do this? We could use

```
: ARRAY
<BUILDS 2 * ALLOT
DOES> SWAP 2 * + ;
```

How does this definition work?

The <BUILDS part tells what happens at compile time. The argument (on top of the stack) to ARRAY (50 or 10 in the above example) is multiplied by two, and ALLOT leaves that many bytes of space in the dictionary. Note that when X or Y or any other array is being defined, the appropriate number of bytes must be alloted for it.

The DOES part tells what happens when X or Y is executed At execution of 0 X , 49 X , etc., DOES> automatically causes the system to place the address of where the array begins on top of the stack; any arguments (0 , 49 or 9 in these examples) are below that address. The SWAP brings the array index to the top of the stack, where it is multiplied by two to get its byte offset from the beginning of the array. This offset is then added to the address of the array to get the desired address of the particular element.

To see how the allocation works, after entering the definition of ARRAY , type:

```
DECIMAL HERE .
```

to see where the next dictionary entry will occur . Then enter

```
50 ARRAY X HERE .
```

to how much dictionary space has been used by the array. Note that there are 8 bytes of overhead plus the 100 bytes for the array.

If you now enter

```
10 ARRAY Y HERE .
```

you will see that 20 bytes of array plus 8 bytes of overhead has been allocated. Entering

```
1234 5 X !
```

will now store 1234 in the fifth element in the X array. And entering

```
5 X @
```

will now place 1234 on the top of the stack

The data to go in an array may be loaded at compile time by the following technique:

```
: VECTOR <BUILDS Ø DO ,
  LOOP DOES>
  SWAP 2 * + ;
```

The data on the stack is in inverse order and the top value on the stack is the number of elements in the vector. Thus,

```
data n-1 data n-2 --- dataØ
n VECTOR ALPHA
```

creates a vector with n elements called ALPHA . For example:

```
55 4444 -33 2222 1111 Ø
6 VECTOR ALPHA
```

Now check the element data

```
3 ALPHA @ . <RETURN> -33
Ø ALPHA @ .  <RETURN> Ø
2 ALPHA @ .  <RETURN> 2222
```

These elements may be changed if so desired, e.g.,

```
1Ø1Ø Ø ALPHA !
```

Check with

```
Ø ALPHA @ . <RETURN> 1Ø1Ø
```

In the definition of VECTOR , a loop is executed the number of times indicated by the top value on the stack. The only function performed by the loop is to use the , command to store the current top value of the stack into the dictionary entry. This is repeated until all of the vector elements are stored in the dictionary definition. The remainder of the operation is the same as the definition. The remainder of the operation is the same as the prior example for ARRAY .

<BUILDS and DOES> can be used to create much more elaborate data types such as special array definitions which do bounds or other error checks at run-time. These definitions could be used during debugging and replaced with the regular (faster) definitions for production use, once you are assured that no out-of-bounds error will occur.

SECTION 6

AIM 65/40 FORTH ASSEMBLE

The AIM 65/40 FORTH structured assembler creates machine language execution procedures that would be time-inefficient if executed in high-level FORTH colon-definitions. A separate ASSEMBLER vocabulary provides the op-codes, addressing modes, conditionals, and other support words necessary to program functions in R6500 assembly language. A function written in assembler language is entered into a vocabulary in a similiar manner as a FORTH colon-definition. It is also executed in the same manner by referring to the word name. It is recommended that assembly language, or "code", as it is often referred to in FORTH terminology, be structured and written similiar to high-level FORTH for clarity of expression. A function can first be rapidly written and debugged in FORTH, tested for proper operation, and then recoded in assembly language for faster execution with a minimum of restructuring.

6.1 THE ASSEMBLY PROCESS

The AIM 65/40 FORTH assembler vocabulary is selected by the word ASSEMBLER or by the word CODE (explained in the following paragraphs). A separate ASSEMBLER vocabulary is linked ahead of the FORTH vocabulary. The words in the ASSEMBLER vocabulary are defined in Appendix D, AIM 65/40 FORTH Assembler Glossary, in ASCII sort order.

To examine the assembler words, perform a cold start, command ASSEMBLER , and run a VLIST . The Assembler VLIST is shown in Figure 6-1. Note that the ASSEMBLER VLIST continues into the FORTH vocabulary upon completion of the ASSEMBLER word list. Press any key to terminate the VLIST before completion.

```
ASSEMBLER OK

VLIST
 DFD0 END-CODE                DFC1 0<
 DFB8 0=                      DFAF VS
 DFA6 CS                      DF99 NOT
 DF73 ELSE,                   DF65 THEN,
 DF3B ENDIF,                  DF24 IF,
 DF04 REPEAT,                 DEF0 AGAIN,
 DED9 WHILE,                  DEBA UNTIL,
 DEA9 BEGIN,                  DE99 BIT,
 DE8B JMP,                    DE7D JSR,
 DE6F STY,                    DE61 LDY,
 DE53 LDX,                    DE45 CPY,
 DE37 CPX,                    DE29 STX,
 DE1B ROR,                    DE0D ROL,
 DDFF LSR,                    DDF1 INC,
 DDE3 DEC,                    DDD5 ASL,
 DDC7 STA,                    DDB9 SBC,
 DDAB ORA,                    DD9D LDA,
 DD8F EOR,                    DD81 CMP,
 DD73 AND,                    DD65 ADC,
 DD59 TXS,                    DD4D TYA,
 DD41 TXA,                    DD35 TSX,
 DD29 TAY,                    DD1D TAX,
 DD11 SEI,                    DD05 SED,
 DCF9 SEC,                    DCED RTS,
 DCE1 RTI,                    DCD5 PLP,
 DCC9 PLA,                    DCBD PHP,
 DCB1 PHA,                    DCA5 NOP,
 DC99 INY,                    DC8D INX,
 DC81 DEY,                    DC75 DEX,
 DC69 CLV,                    DC5D CLI,
 DC51 CLD,                    DC45 CLC,
 DC39 BRK,                    DB6C RP)
 DB5E SEC                     DB50 TOP
 DB45 )                       DB3C )Y
 DB32 X)                      DB28 ,Y
 DB1E ,X                      DB14 MEM
 DB09 #                       DB00 .A
 DACF SETUP                   DAC3 BINARY
 DAB6 PUT0A                   DAAA PUSH0A
 DA9D POPTWO                  DA90 POP
 DA86 PUT                     DA7C PUSH
 DA71 NEXT                    DA66 INTVECT
 DA58 INTFLAG                 DA4A XSAVE
 DA3E UP                      DA35 W
 DA2D IP                      DA24 N
```

Figure 6-1. VLIST of AIM 65/40 FORTH Assembler Words


```
 COLD
 AIM 65/40 FORTH V1.4
ASSEMBLER OK
VLIST
 DFD0 END-CODE DE        DFC1 0<
 DFB8 0=                 DFAF VS
 DFA6 CS                 DF99 NOT
 DF73 ELSE,              DF65 THEN,
    .
    .
    .
 DA3E UP                 DA35 W
 DA2D IP                 DA24 N
  726                     809 TASK
 D9DC .S                 D9D1 MON
 D9C1 HANG OK            ( <SPACE> bar pressed)
```

Code assembly consists of interpreting entered words with the ASSEMBLER vocabulary as CONTEXT (see Section 5.6.2). Thus, each word in the input stream is matched according to the FORTH practice of searching CONTEXT first, then CURRENT .

The vocabulary search order is

Order	Vocabulary	
1	ASSEMBLER	(Now CONTEXT)
2	FORTH	(Chained to ASSEMBLER)
3	User's Vocabulary	(CURRENT if one exits)
4	FORTH	(Chained to user's vocabulary)
5	Literal Number	

The above sequence is the usual action of FORTH's text interpreter, which remains in control during assembly.

6.1.1 CODE Definitions

The CODE word defines a word written in assembly code (called a CODE-definition) in a similiar manner as the : word defines a word written in FORTH (a colon-definition). The assembler vocabulary is automatically selected as CONTEXT when CODE is encountered. The name following CODE is entered into the dictionary as the FORTH word for the CODE-definition. Assembly language routines or program segments in CODE-definition form are often referred to as "CODE" or "code" in general FORTH literature. Assembly language instructions in RPN format (see Section 6.2) are then entered along with any instructions to save and restore return stack values (see Section 6.4) and conditionals (see Section 6.6) The END-CODE word terminates a CODE-definition in a similiar manner as the ; terminates a FORTH colon-definition.

During assembly of CODE-definitions, FORTH continues interpretation of each word encountered in the input stream (not in the compile mode). These assembler words specify operands, address modes, and op-codes. At the conclusion of the CODE-definition an error check verifies correct completion and then "unsmudges" the definition's name, therefore making it available for dictionary searches.

6.1.2 Assembly-time Versus Run-time

It is important to understand at what time a particular word definition executes. During assembly, each assembler word interpreted executes. Its function at that instant is called 'assembling' or 'assembly-time'. This function includes op-code generation from mnemonics, address calculation, address mode selection, and relative branch calculation.

The later execution of the generated code is called 'run-time'. This distinction is particularly important with the conditionals. At 'assembly-time', each word (i.e., IF, UNTIL, BEGIN, etc.) 'runs' to produce machine code (conditional branch and/or jump instructions) which will later execute at 'run-time' when its CODE-definition name is used.

6.1.3 CODE-Definition Example

As a practical example, here's a simple call to the AIM 65/40 Monitor, via the IRQ address vector (using the BRK op-code). Enter the following words.

```
CODE MONX          Exit to AIM 65/40 Monitor)
BRK,
NEXT JMP,
END-CODE
```


a. The word CODE is first encountered and executed by FORTH. CODE builds the name MONX into a dictionary header and calls ASSEMBLER as the CONTEXT vocabulary. Note that the <name> after CODE must be on the same line.

b. BRK , is next found in the assembler vocabulary as the op-code. When BRK , executes, it assembles the byte value 00 into the dictionary as the BRK instruction machine code. This causes the R6502 CPU to perform an IRQ interrupt, which in turn returns control to the AIM 65/40 Monitor (see Section 6.4 in the AIM 65/40 System User's Manual).

Note that the FORTH assembler word names end with a ",". The significance of this is:

(1) The comma distinguishes assembler control words from FORTH control words, e.g., IF, versus IF , etc.

(2) The comma shows the conclusion of a logical grouping that would be one line of classical assembly source code.

(3) "," compiles into the dictionary; thus, a comma implies the point at which code is generated.

(4) The "," distinguishes op-codes from possible hexadecimal numbers ADC, ADD, and BCC.

FORTH executes your word definitions under control of the address interpreter, named NEXT . This short code routine moves execution from one definition to the next. At the end of your CODE-definition, you must return control to NEXT or else to other code which returns to NEXT.

The BRK instruction executed by the word MONX returns control to the AIM 65/40 Monitor, e.g.,

```
        MONX
       = B0 08 92 00 FD 0814    BRK
```

Note that the address counter and processor status were saved by the IRQ processing. If G, followed by the address displayed plus one, and <RETURN> is now typed, execution will resume at the next instruction past BRK, which is the JMP to NEXT , e.g.,

```
        {G} 0815 <RETURN>
```

NEXT is a constant that specifies the machine address of FORTH's address interpreter (at $C06F). Here NEXT is the operand for JMP, . As JMP, executes, it assembles a machine code jump to the address of NEXT from the assembly time stack value. If control is not returned to this FORTH address as the last instruction in the CODE-definition, improper operation of the AIM 65/40 microcomputer and possible alteration of your program may result.

d. The END-CODE word terminates the CODE-definition with a SMUDGE of the name. It also exits the ASSEMBLER making CONTEXT the same as CURRENT .

The object code of our example is:

```
080B  84                          ( Name letter count with MSB set)
080C  45 58 49 D4  MONX           ( Name with MSB of last digit set)
0810  00 08        link field
0812  14 08        code field
0814  00           BRK
0815  4C 6F C0     JMP NEXT
```

6.2 ASSEMBLER OP-CODES

The bulk of the assembler consists of dictionary entries for the R6500 mnemonic op-codes. Refer to Appendix B in the R6500 Programming Manual to see the machine code that is generated by each mnemonic op-code.

6.2.1 Single Mode Op-Codes

The R6502 single mode op-codes are:

BRK,	CLC,	CLD,	CLI,	CLV,	DEX,	DEY,	INX,
INY,	NOP,	PHA,	PHP,	PLA,	PLP,	RTI,	RTS,
SEC,	SED,	SEI,	TAX,	TAY,	TSX,	TXS,	TXA,
TYA,							

When any of these op-codes are executed, the corresponding machine code byte is assembled into the dictionary.

6.2.2 Multi-Mode Op-Codes

The multi-mode op-codes are:

ADC,	AND,	CMP,	EOR,	LDA,	ORA,	SBC,	STA,
ASL,	DEC,	INC,	LSR,	ROL,	ROR,	STX,	CPX,
CPY,	LDX,	LDY,	STY,	JSR,	JMP,	BIT,	

These op codes take an operand which must already be on the stack. An address mode may also be specified. If none is given, the op-code uses z-page (when appropriate) or absolute addressing.

6.3 ADDRESSING MODES

The addressing modes are specified by:

FORTH Word	Addressing Mode	
.A	accumulator	none
#	immediate	8 bits only
,X	indexed X	z-page or absolute
,Y	indexed Y	z-page or absolute
X)	indexed indirect X	z-page only
)Y	indirect indexed Y	z-page only
)	indirect	absolute only
none	memory	z-page or absolute

Here are examples of FORTH vs. conventional assembler. Note that the operand comes first, usually followed by any addressing mode modifier, and then the op-code mnemonic. This makes best use of the stack at assembly-time. Also, each assembler word is set off by blanks, as is required for all FORTH source text.

FORTH	Conventional Assembler	
.A ROL,	ROL A	.A distinguishes A
1 # LDY,	LDY #1	from hex number ØA)
DATA ,X STA,	STA DATA,X	
DATA ,Y CMP,	CMP DATA,Y	
6 X) ADC,	ADC (Ø6,X)	
POINT)Y STA,	STA (POINT),Y	
VECTOR) JMP,	JMP (VECTOR)	

The words DATA , POINT , and VECTOR specify machine addresses defined by prior VARIABLE or CONSTANT words. In the case of "6 X) ADC," the operand memory address $ØØØ6 was given directly. This is occassionally done if the usage of a value does not justify devoting the dictionary space to a symbolic value.

6.4 R65Ø2 CONVENTIONS

6.4.1 Stack Addressing

The parameter stack is located in z-page, and is usually addressed by "Z-PAGE,X". This stack starts at $ØØ91 and grows physically downward. The X index register is the data stack pointer. Thus, incrementing X by two removes a data stack value; decrementing X twice makes room for one new data stack value.

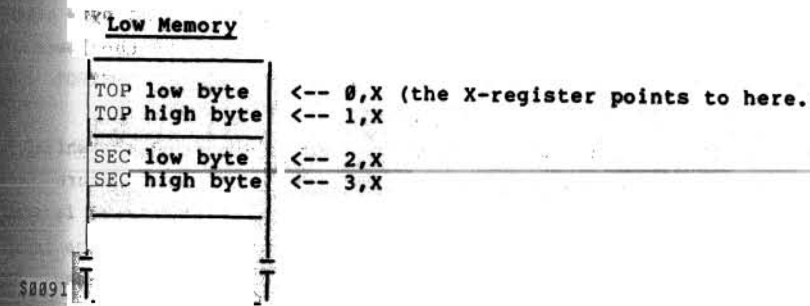
16-bit values are placed on the stack according to the R65Ø2 convention; the low byte is at low memory, with the high byte following. This allows "indexed, indirect X" instructions to be executed directly off of a stack value.

The top and second stack values are referenced often enough that the support words TOP and SEC are included. Using

TOP LDA, assembles LDA (Ø,X) and
SEC ADC, assembles ADC (2,X)

TOP leaves Ø on the stack and sets the address mode to ,X
SEC leaves 2 on the stack and also sets the address mode to ,X .

Here is a pictorial representation of the parameter stack in z-page (see Appendix F).



The Ø or 2 left by TOP or SEC is the base address above which X register indexes. You may further modify this at assembly-time to address at any byte in the parameter stack.

Here is an example of assembly code to "or" together the top four bytes on the stack:

FORTH	Conventional Assembler
TOP LDA,	LDA (Ø,X)
TOP 1+ ORA,	ORA (1,X)
SEC ORA,	ORA (2,X)
SEC 1+ ORA,	ORA (3,X)

To obtain the 14-th byte on the stack, use

TOP 13 + LDA,

6.4.2 Return Stack

The FORTH Return Stack (and the machine stack) is located in the R6502 machine stack area in Page One. It starts at $01FE and builds physically downward. No lower bound is set or checked as Page One has sufficient capacity for all (non-recursive) applications.

By R6502 convention the CPU's S register points to the next free byte below the bottom of the Return Stack. The byte order follows the convention of low significance byte at the lower address.

Return stack values may be obtained by: PLA, PLA, which will pull the low byte, then the high byte from the Return Stack. To operate on arbitrary bytes, the method is:

a. Save X in XSAVE .

b. Execute TSX, to move the S register contents to the X register.

c. Use RP) to address the lowest byte of the return stack. Offset the value to address higher bytes. (The address mode is automatically set to ,X .)

d. Restore X from XSAVE .

As an example, this CODE-definition non-destructively tests the second item on the Return Stack (also the machine stack), to see if it is zero.

```
CODE IS-IT          ( Is second item on Return Stack zero?)
     XSAVE STX,     ( Setup for Return Stack)
     TSX,
     RP) 2+ LDA,    ( Or second item's bytes together)
     RP) 3 + ORA,
     Ø= IF,         ( If zero, increment Y by one)
       INY,
     THEN,
     TYA,           ( Save low byte)
     XSAVE LDX,     ( Restore data stack)
     PUSHØA JMP,    ( Push Boolean and zero onto data stack)
     END-CODE
```



```
                 Low Memory
         ______________________________
        |                              |
        |          Free Byte           |   <--- S
        |______________________________|
        |     Top Item Low Byte        |   <--- RP) = $1Ø1,X
        |     Top Item High Byte       |
        |______________________________|
        |    Second Item Low Byte      |
        |    Second Item High Byte     |
        |______________________________|
        |         Return Stack         |
       -+-                            -+-
$Ø1FE   |______________________________|
```

6.5 FORTH REGISTERS

6.5.1 Assembly Registers

Several FORTH registers are available only at the assembly level and have been given names that return their memory addresses. These are:

IP Address of the Interpretive Pointer, specifying the next FORTH address which will be interpreted by NEXT .

W Address of the pointer to the code field of the dictionary definition just interpreted by NEXT . W-1 contains $6C, the op-code for the indirect jump instruction. Therefore, jumping to W-1 will indirectly jump via W to the machine code for the definition.

UP User Pointer containing the address of the base the user area.

N A utility area in z-page from N-1 thru N+7

6.5.2 CPU Registers

When FORTH execution leaves NEXT to execute a CODE-definition, the following conventions apply:

The Y register is zero It may be freely used

b The X register defines the l byte of the bottom data stack item relative to machi address $0000 X must point to the correct item up returning to FORTH.

The CPU sta er S points one by below the low byte of the item in the Return tack. Executing PLA, ull this byte to t accumulator.

d The accumulator may be freely used

The CPU is in the binary (i.e., not decimal) mode and must be returned in the binary mode (with a CLD prior to return, as needed).

6.5.3 XSAVE

XSAVE is a byte buffer in z-page, for temporary storage of the X register. Typical usage, with a call to a previously defined code word USER , which will change X, is:

```
CODE DEMO
  XSAVE STX,
  ' USER JSR,
  XSAVE LDX,
  NEXT JMP,
  END-CODE
```

6.5.4 N Area

When absolute memory registers are required, use the 'N Area' in Page Zero. These registers may be used to store pointers for indexed/indirect addressing or to store temporary values.

The assembler word N returns the base address ($0097). The N area spans 9 bytes, from N-1 thru N+7. Conventionally, N-1 holds one byte and N, N+2, N+4, N+6 are pairs which may hold 16-bit values. See SETUP for help on moving values to the N Area.

It is very important to note that many FORTH procedures use N. Thus, N may only be used within a single CODE-definition. Never expect that a value will remain within outside a single definition!

```
HEX
CODE DEMO
  6 # LDA,
  N 1- STA,           ( Setup a counter )
  0 # LDA,            ( Make Port A input)
  FFA3 STA,
   BEGIN,
   FFA1 BIT,          ( Test Port A )
   N 1- DEC,          ( Decrement the counter)
   0< UNTIL,          ( Loop until negative)
  NEXT JMP,
  END-CODE
```

Check the VLIST and HERE to det he starting and next address after the end of dictionary

```
VLIST
 814 DEMO                  809 TASK
D9DC .S                   D9D1 MON
 D9C1 HANG OK             (<SPARE> bar pressed)
HERE . <RETURN> 827 OK
```

Escape to AIM 65/40 Monitor and examine generated machine code.

```
 (ESC)
{K}=0814/8                OUT=<RETURN>

 0814 A9 06               LDA #$06
 0816 85 96               STA $96
 0818 A9 00               LDA #$00
 081A 8D A3 FF            STA $FFA3
 081D 2C A1 FF            BIT $FFA1
 0820 C6 96               DEC $96
 0822 10 F9               BPL $081D
 0824 4C 6F C0            JMP $C06F
```

6.5.5 SETUP

Often we wish to move stack data values to the N area. The word SETUP has been provided for this purpose. Upon entering SETUP the accumulator specifies the quantity of 16-bit stack values to be moved to the N area. That is, A may be 1, 2, 3, or 4 only:

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

Stack before		N after		Stack after	
TOP --->	A high	N --->	A		
	B low		B		
SEC --->	C		C		
	D		D		
	E		E		
	F		F		
	G low			TOP	G
	H high				H

6.6 CONTROL FLOW

FORTH discards the usual convention of assembler labels. Instead, two replacements are used. First, each FORTH definition name is permanently included in the dictionary. This allows procedures to be located and executed by name at any time as well as be compiled within other definitions.

Secondly, within a CODE-definition, execution flow is controlled by label-less branching according to "structured programming". This method is identical to the form used in colon-definitions. Branch calculations are done at assembly-time by temporary stack values placed by the control words:

BEGIN,	THEN,
UNTIL,	AGAIN,
IF,	WHILE,
ELSE,	REPEAT,

Here again, the assembler words end with a comma, to indicate that code is being produced and to clearly differentiate from the high-level form.

One major difference occurs! High-level flow is controlled by run-time Boolean values on the data stack. Assembly flow is controlled instead by processor status bits. You must indicate which status bit to test with one or two FORTH condition code (cc) words, just before a conditional branching word i.e., IF, UNTIL, or WHILE, .

The conditional specifiers for the R65Ø2 are

FORTH Condition Code (cc) Words	Test Function	Processor Status Bit
CS	carry set	C=1
Ø<	less than zero	N=1
Ø=	equal to zero	Z=1
VS	overflow set	V=1
CS NOT	carry clear	C=Ø
Ø< NOT	positive	N=Ø
Ø= NOT	not equal zero	Z=Ø
VS NOT	overflow clear	V=Ø

6.6.1 Conditional Looping

A conditional loop is formed at assembler level by placing the instructions to be repeated between BEGIN, and UNTIL, . Precede UNTIL, by a conditional specifier, e.g., Ø< . The assembler generates the proper conditional branch machine instruction, e.g., BEQ, to test the processor status and to conditionally branch back to the machine instruction immediately after the BEGIN, .

The general format is:

```
BEGIN
<assembly code>
<cc> UNTIL,
<continuing assembly code>
```

For example, enter the CODE-definition for LOOP-TEST

```
HEX
Ø VARIABLE TICK
CODE LOOP-TEST
  6 # LDA,
  N STA,
   BEGIN,
   TICK DEC,
   N DEC,
   Ø=  UNTIL,
  NEXT JMP,
END-CODE
```

Note where the variable TICK and LOOP-TEST are located in the FORTH dictionary:

```
VLIST
  824 LOOP-TEST            814 TICK
  809 TASK                D9DC .S
D9D1 MON                  D9C1 HANG
D97A VLIST OK             ( <SPACE> pressed)
```

Also, find the start ($832) of the next dictionary entry:

```
HERE . <RETURN> 832
```

Return to the AIM 65/40 Monitor and disassemble the machine code.

```
 (ESC)
{K}*=0824/6              OUT=<SPACE>

 0824 A9 06              LDA #$06
 0826 85 97              STA $97
 0828 CE 14 08           DEC $0814
 082B C6 97              DEC $97
 082D D0 F9              BNE $0828
 082F 4C 6F C0           JMP $C06F
```

This shows you how the assembly code is generated for a typical conditional loop.

First, the temporary storage byte at address N is loaded with the value 6. The beginning of the loop is marked (at assembly-time) by BEGIN, . Memory at TICK is decremented, then the loop counter in N is decremented. Of course, the CPU updates its status register as N is decremented. Finally, a test for Z=1 is made; if N hasn't reached zero, execution returns to BEGIN, . When N reaches zero (after executing TICK DEC, 6 times) execution continues ahead after UNTIL, . Note that BEGIN, generates no machine code, but is only an assembly-time locator. In this example, 0 = UNTIL, generated a BNE instruction to address $0828, the address located by BEGIN, .

6.6.2 Conditiona Execution

Paths of execution may be chosen at assembly in a similar fashion as done in colon-definitions. In this case, the branch is chosen based on a processor status condition code. The general format is (using 0= as a typical condition code word):

```
PORT LDA,
0= IF,
  <code for zero set>
THEN,
<continuing code>
```

In this example, the accumulator is loaded from PORT . The zero status is tested and, if set (Z=1), the code for zero set is executed. Whether the zero status is set or not, execution will resume at THEN, .

The conditional branching also allows a specific action for the false cas

```
PORT LDA,
0= IF,
  <assembly code for zero set>
ELSE,
  <assembly code for zero clear>
THEN,
<continuing assembly code>
```

The test of PORT will select one of two execution paths, before resuming execution after THEN, . The next example increments N based on bit D7 of a port:

```
PORT LDA,                      ( Fetch one byte )
0< IF,
  N DEC,                       ( If D7=1, decrement N )
ELSE,
  N INC,                       ( If D7=0, increment N )
THEN,                          ( Continue on )
```

6.6.3 Conditional Nesting

Conditionals may be nested, according to the conventions of structured programming. That is, each conditional sequence begun (IF, BEGIN,) must be terminated (THEN, UNTIL,) before the next earlier conditional is terminated. An ELSE, must pair with the immediately preceding IF, .

```
BEGIN,
<code always executed>
  CS IF,
  <code if carry flag set>
  ELSE,
  <code if carry flag clear>
THEN,
<loop until zero flag is non-zero>
Ø= NOT UNTIL,
<code that continues onward>
```

Next is an error that the assembler security will reveal.

```
CODE <name>
  BEGIN,
  PORT LDA,
    Ø= IF,
    TOP INC,
    Ø= UNTIL,
    ENDIF,
```

The UNTIL, will not complete the pending BEGIN, since the immediately preceeding IF, is not completed. An error trap will occur at UNTIL, and error number 19 "conditionals not paired" will be generated. To delete the erroneous code from the dictionary, first SMUDGE the word to allow finding it, then FORGET it, and correct the source code and recompile.

6.6.4 Some Nesting Examples

a. An 8-Bit Counter

An 8-bit counter illustrates simple conditional looping.

```
Ø VARIABLE COUNTS
-1 ALLOT
CODE COUNT-DOWN
COUNTS STA,
Ø # LDA,
COUNTS DEC,
  BEGIN,
  Ø= UNTIL,
NEXT JMP,
END-CODE
```

Execute the counter:

```
COUNT-DOWN <RETURN> OK
```

Dump the machine code for examination:

```
HEX ' COUNT-DOWN NFA 1C DUMP
  817 8A 43 4F 55 4E 54 2D 44
  81F 4F 57 CE  B  8 26  8 A5
  827  Ø 8D 16  8 CE 16  8 DØ
  82F FB 4C 6F CØ  4 44 55 4D
OK
```

The breakdown of the machine code is:

```
816 ØØ                                   ( COUNTS Variable)
817 8A                                   ( Name Field = Start)
818 43 4F 55 4E 54 2D 44 4F 57 CE       ( COUNT-DOWN Name)
822 ØB Ø8                                ( Link Field = Ø8ØB)
824 26 Ø8                                ( Code Field = Ø826)
826 A9 ØØ          LDA #ØØ               ( Parameter Field)
828 8D 16 Ø8       STA Ø816
82B CE 16 Ø8       DEC Ø816
82B DØ FB          BNE Ø82B              ( Next)
83Ø 4C 6F CØ       JMP CØ6F
```

The machine instructions can be disassembled with the AIM 65/4Ø Monitor K command to check the assembly code sequence.

In this example we use part of the RAM dictionary for the counter (COUNTS) . This counter is only 8 bits, however, so after we create the 16-bit named dictionary location COUNTS , we use ALLOT to back up over the extra byte and recover it for use.

The definition of the word COUNT-DOWN is a simple loop, decrementing COUNTS until it hits zero then jump to NEXT . First, of course, we clear COUNTS to its initial value by the LDA, and STA, instructions. The initializing to zero is no problem because right after we clear counts to zero we decrement it and it becomes FF. This way we loop 256 times before finally exiting when we decrement to zero.

b. A 16-Bit Counter

This counter is similar to the 8-bit one except that:

- COUNTS is the right size to begin with therefore ALLOT is unnecessary.
- We initialize two bytes to zero to start with.
- We use two nested loops to do the decrementing.

The assembly code is:

```
Ø VARIABLE COUNTS
CODE COUNT-DOWN
Ø # LDA,
COUNTS STA,
COUNTS 1+ STA,
  BEGIN,
    BEGIN,
    COUNTS DEC,
    Ø= UNTIL,
  COUNTS 1+ DEC,
  Ø= UNTIL,
NEXT JMP,
END-CODE
```

Execute the counter

```
COUNT-DOWN <RETURN> OK
```

The machine code is:

```
Ø816 ØØ ØØ                                    ( COUNTS Variable)
Ø818 8A                                       ( Name Field Start)
Ø819 43 4F 55 4E 54 2D 44 4F 57 CE            ( COUNT-DOWN Name)
Ø823 ØB Ø8                                    ( Link Field = $Ø8ØB)
Ø825 27 Ø8                                    ( Code Field = $Ø827)
Ø827 A9 ØØ             LDA #$ØØ               ( Parameter Field)
Ø829 8D 16 Ø8          STA $Ø816
Ø82C 8D 17 Ø8          STA $Ø817
Ø82F CE 16 Ø8          DEC $Ø816
```



```
Ø832 DØ FB             BNE $Ø82F
Ø834 CE 17 Ø8          DEC $Ø817
Ø837 DØ F6             BNE $Ø82F
Ø839 4C 6F CØ          JMP $CØ6F
```

c. A 24-Bit Counter

The value of indenting the loops for visual clarity is more obvious here than in the previous example. This example uses a three byte counter and so one more byte of dictionary space is alloted and three nested loops do the work.

```
Ø VARIABLE COUNTS
1 ALLOT
CODE COUNT-DOWN
Ø # LDA,
COUNTS STA,
COUNTS 1+ STA,
COUNTS 2+ STA,
  BEGIN,
    BEGIN,
      BEGIN,
      COUNTS DEC,
      Ø= UNTIL,
    COUNTS 1+ DEC,
    Ø= UNTIL,
  COUNTS 2+ DEC,
  Ø= UNTIL,
NEXT JMP,
END-CODE
```

Execute the counter:

```
COUNT-DOWN <RETURN> OK                    ( About 2 1/2 min.)
```

The breakdown of the machine code is:

```
Ø816 ØØ ØØ ØØ                                 ( COUNTS Variable)
Ø819 8A                                       ( Name Field Start)
Ø81A 43 4F 55 4E 54 2D 44 4F 57 CE            ( COUNT-DOWN Name)
Ø824 ØB Ø8                                    ( Link Field = $Ø8ØB)
Ø826 28 Ø8                                    ( Code field = $Ø828)
Ø828 A9 ØØ             LDA #$ØØ               ( Parameter Field)
Ø82A 8D 16 Ø8          STA $Ø816
Ø82D 8D 17 Ø8          STA $Ø817
Ø83Ø 8D 17 Ø8          STA $Ø818
Ø833 CE 16 Ø8          DEC $Ø816
Ø836 DØ FB             BNE $Ø833
Ø838 CE 17 Ø8          DEC $Ø817
Ø83B DØ F6             BNE $Ø833
Ø83D CE 18 Ø8          DEC $Ø818
```

6.7 RETURN OF CONTROL

When concluding a CODE-definition, several common stack manipulations are often needed. These functions are already in the nucleus, so we may share their use just by knowing their return points. Each of these words ultimately returns control to NEXT .

POP	Remove one 16-bit stack value.
POPTWO	Remove two 16-bit stack values.
PUSH	Push two bytes to the data stack.
PUT	Write two bytes to the data stack, replacing the present top of the stack.
PUSHØA	Push a zero and the accumulator to the data stack.
PUTØA	Replace the top of the stack with a zero and the accumulator.
BINARY	Combines the action of POPTWO and PUSH .

Our next example complements a byte in memory. The bytes' address is on the stack when INVERT is executed.

```
CODE INVERT        ( Code to invert a memory byte )
  HEX              ( Change I/O base to HEX)
  TOP X) LDA,      ( Fetch byte addressed by stack )
  FF  # EOR,       ( Complement accumulator )
  TOP X) STA,      ( Replace in memory )
  POP JMP,         ( Discard pointer from stack
  END-CODE         ( Return to NEXT )
```

A new stack value may result from a CODE-definition. We could place it on the stack by:

```
CODE ONE           Code to put 1 on the stack
  DEX,
  DEX,             Make room on the data stack
  1 # LDA,
  TOP STA,        ( Store low byte )
  TOP 1+ STY,     ( High byte stored from Y since = zero )
  NEXT JMP,
  END-CODE
```


A simpler version could use PUSH :

```
CODE ONE            ( Code to put 1 on the stack )
  1 # LDA,
  PHA,              ( Push low byte to machine stack
  TYA,              ( High byte to accumulator )
  PUSH JMP,         ( Push to data stack )
  END-CODE
```

The convention for PUSH , BINARY and PUT is:

- . Push the low byte on to the machine stack.
- . Leave the high byte in the accumulator.
- . Jump to PUSH , BINARY or PUT .

PUSH will place the two bytes at the new bottom of the data stack. PUT will over-write the present bottom of the stack with the two bytes. BINARY first pops two stack values (four bytes) then does a push. Failure to push exactly one byte on the machine stack will disrupt execution upon usage!

The simplest version would use PUSHØA :

```
CODE ONE
  1 # LDA,
  PUSHØA JMP,
  END-CODE
```

If the high byte of a result to be placed on the stack is zero, and the low byte is in the accumulator, the words PUSHØA and PUTØA are convenient. They work the same as PUSH and PUT but add to, or replace, the data on the stack with a zero in the high byte position and the contents of the accumulator in the low byte position.

6.8 ASSEMBLER SECURITY

6.8.1 Assembler Tests

Numerous tests are made by the assembler to detect errors in structure and syntax. These tests verify that

a. All parameters used in CODE-definitions are removed.
b. Conditionals are properly nested and paired.
c. Op-codes are valid.
d. Address modes and operands are allowable for the op-codes.

Note that a possible error not detectable by the assembler, is referencing a word in the wrong vocabulary, e.g., referring to Ø= in the FORTH vocabulary rather than the Assembler vocabulary.

6.8.2 Bypassing Security

Occasionally we may want to generate unstructured code. We can then control the assembly-time security checks, as follows: First, we must note the parameters utilized by the control structures at assembly-time. The notation below is taken from the assembler glossary in Appendix D. The "---" indicates assembly-time execution and separates input stack values from the output stack values.

```
BEGIN,  -->                   --- addrB 1
UNTIL,  -->          addrB 1 <cc> ---
AGAIN,  -->          addrB 1 ---
WHILE,  -->          addrB 1 --- addrB 1 addrW 3
REPEAT, -->          addrB 1 addrW 3 ---

IF,     -->          <cc>    --- addrI 2
ELSE,   -->          addrI 2 --- addrE 2
THEN,   -->          addrI 2 ---
                     or addrE 2 ---
```

Where the address values indicate the machine location of the corresponding "B"EGIN, , "I"F, , or "E"LSE, and <cc> represents the condition code to select the processor status bit referenced. The digit 1, 2 or 3 is tested for conditional pairing.

The general method of security control is to drop off the check digit and manipulate the addresses at assembly-time. The security against errors is less, but the programmer is usually paying intense attention to detail during this effort.

6.9 ADDING ASSEMBLY CODE TO A DEFINING WORD

The word ;CODE is used in a colon-definition to stop compiling and to add assembly code to the definition. The format is as follows:

```
: <name> [FORTH words] ;CODE [assembly code] END-CODE
```

where the [FORTH words] are run at compile time and the [assembly code] is executed at run-time.

When <name> is used later to define new words, this assembly code address will be put into the code sequence of the new words. Thus, the new words will cause this assembly code to be executed. For example,

```
: VALUE CREATE SMUDGE C,
  ;CODE
  Ø X) LDA,
  PUSHØA JMP,
  END-CODE
```

When used by typing 8Ø VALUE EIGHTY , the word EIGHTY is created which, when executed with a "dot" to print the stack top,

```
EIGHTY .
```

will yield

```
8Ø OK
```

SECTION 7

HANDLING INTERRUPTS IN FORTH

7.1 TYPES OF INTERRUPT HANDLERS

Interrupts can easily be handled in FORTH using one of two methods: machine level or interpretive interrupt processing. A machine level, or conventional, interrupt handler is written in assembly language and performs the entire interrupt processing before returning to the interrupted routine. NMI interrupts must be serviced with a machine level interrupt handler, as shown in the flowchart in Figure 7-1. The IRQ interrupts can also be serviced with a machine level interrupt handler, which is the method used for all AIM 65/40 peripheral interrupt processing. The general flowchart for using this method on the AIM 65/40 microcomputer is shown in Figure 7-2. This approach provides the fastest response to an interrupt, however, since it is written in assembly language it may take longer to develop and check out.

An interpretive IRQ interrupt has a minimum length assembly language subroutine to service the interrupt and to initiate interrupt processing, which is written in high level FORTH and is executed under control of the FORTH inner-interpreter, NEXT . The general flowchart for this approach is shown in Figure 7-3. Although the response to an interrupt may be longer with this approach, the development and checkout may be done quicker and easier since the main interrupt processing is done in FORTH.

When developing interrupt dependent software (regardless of the type of interrupt) try to take small steps between checkout. Carefully determine when system interrupts should be disabled or enabled. Avoid using any interrupt service routine that has not been first tested for logical integrity.

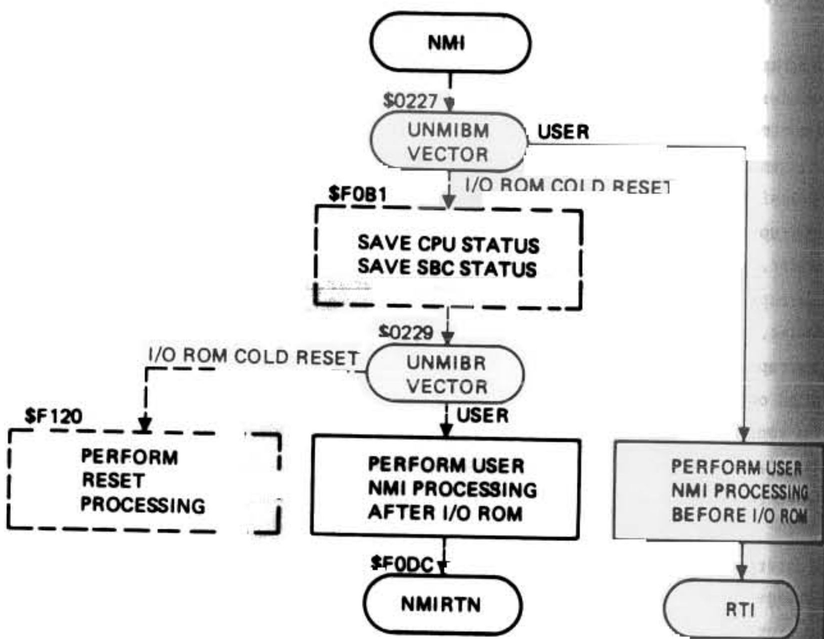


Figure 7-1. Machine Level NMI Interrupt Handling

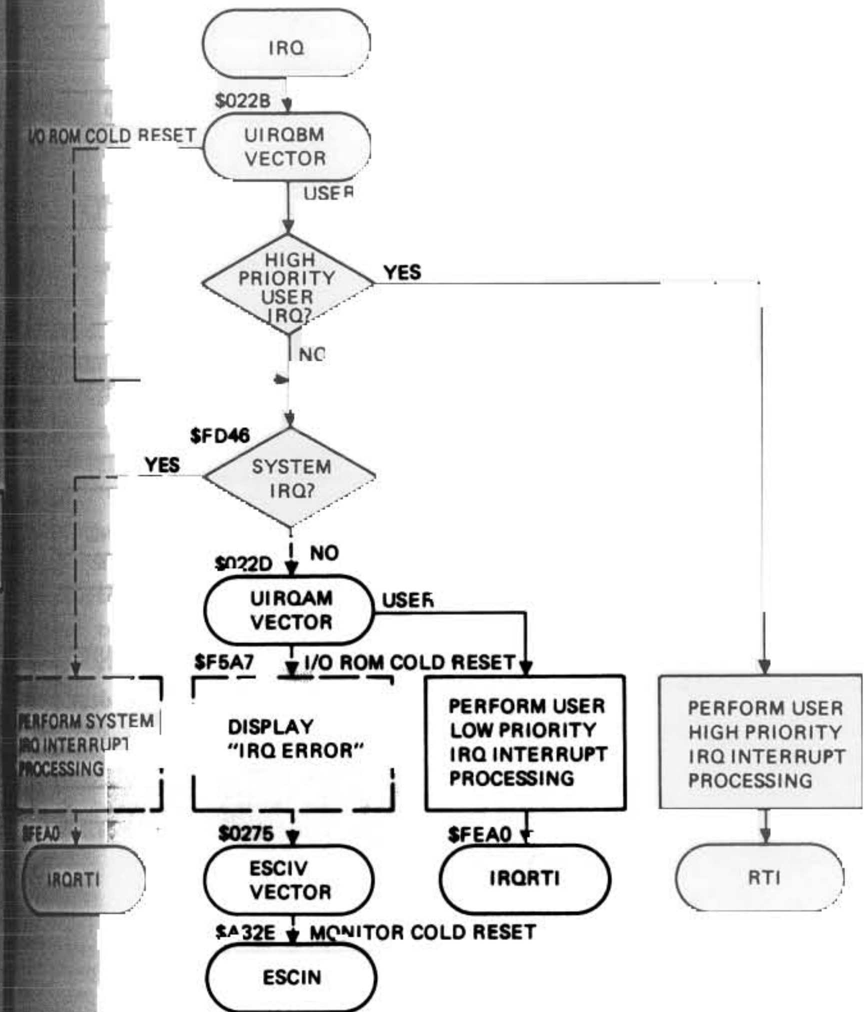


Figure 7-2. Machine Level IRQ Interrupt Handling

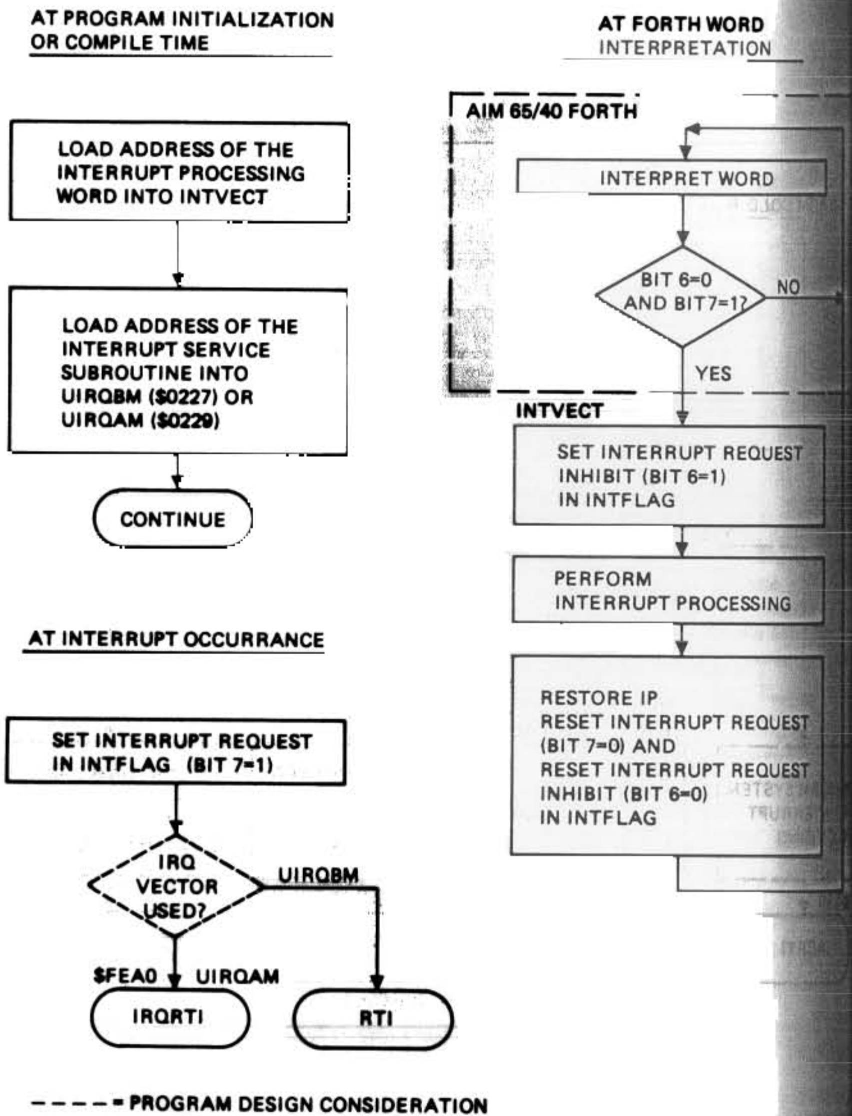


Figure 7-3. Interpretive IRQ Interrupt Handling

NOTE

Since the AIM 65/4Ø system is interrupt driven, care must always be taken with untested user interrupt routines to avoid hanging up the system. This hang up condition is always recoverable with a cold reset.

7.2 MACHINE LEVEL INTERRUPT HANDLING

Write a machine level interrupt handler in assembly language either as a CODE-definition (see Section 6.1) or as a code fragment. If written as a CODE-definition, assign a name to the interrupt handler and later address it by that name to load the interrupt vector. If written as a code fragment, include the assembly code directly into the dictionary, but first save the starting address for later loading into the interrupt vector. The code fragment (also called an orphan) eliminates the slight overhead of the dictionary header. In either case, terminate the interrupt handler with an RTI, to return to the interrupted program rather than NEXT JMP, which returns control to the inner-interpreter. Before continuing you may want to review the AIM 65/4Ø interrupt linkage and handling described in Section 6.3 and 6.4 of the AIM 65/4Ø System User's Manual and the R65Ø2 interrupt processing features discussed in Chapter 9 of the R65ØØ Programming Manual.

The AIM 65/4Ø interrupt vectors normally available to the user are:

```
NMI (Before I/O ROM Processing) - $Ø227 (UNMIBM)
NMI (Before Return to Monitor)  - $Ø229 (UNMIBR)
IRQ (Before I/O ROM Processing) - $Ø22B (UIRQBM)
IRQ (After I/O ROM Processing)  - $Ø22D (UIRQAM)
```

7.2.1 CODE-Definition Form

The form for an interrupt handler written as a CODE-definition

```
HEX
CODE <name>
<assembly code>
<for interrupt>
<handler>
RTI, or CONTINUE JMP,    (CONTINUE must end with an RTI,)
END-CODE
' <name>  022X !         ( Set interrupt vector)
```

where X = 7, 9, B or D

The use of either the RTI, or the CONTINUE JMP, will depend on the interupt vector used. For the NMI or IRQ interrupt vectors after I/O ROM processing (UNMIBR or UIRQAM), either an RTI, or CONTINUE JMP, (where CONTINUE is the address left in the interrupt vector by a cold reset) may be used. For the NMI or or IRQ interrupt vectors before I/O ROM Processing (UNMIBM or UIRQBM), the CONTINUE JMP, should be used (where CONTINUE is the address left in the interrupt vector by a cold reset). Don't forget the END-CODE as it completes the CODE-definition and makes <name> available for use to load the interrupt handler address in the interrupt vector.

The word ' ("tick") fetches the parameter field address (PFA) of the word <name> to the stack and 022X ! stores it in the appropriate vector. The PFA obtained is the start address of the executable machine code.

7.2.2 Code Fragment Form

The form for a machine level interrupt handler written as a code fragment is:

```
HEX
ASSEMBLER                 ( Include assembler vocabulary)
HERE                      ( Locate dictionary address)
<assembler code>
<for interrupt>
<handling>
RTI, or CONTINUE JMP,     (CONTINUE must end with an RTI,)
022X !                    ( Store dictionary address in
                            interrupt vector)
```


Since the interrupt handler is not named, the starting address of the machine code is saved on the stack by the word HERE until the coding is complete, then it is stored in the appropriate interrupt vector. Notice that in both the above cases the interrupt vector was loaded after the interrupt handler was assembled. This method allows an IRQ or NMI interrupt occurring immediately after the interrupt vector is loaded (and the IRQ interrupt is enabled) to be processed correctly.

7.2.3 Interrupt Disable/Enable Words

To further help with control of IRQ interrupt execution you will probably want to define two short CODE-definition words to enable and disable only the user IRQ interrupts. Define the user IRQ interrupt disable word as

```
CODE DISABLE
<perform action to>
<mask user interrupt>
NEXT JMP
END-CODE
```

and then disable the IRQ interrupt as required with DISABLE .

Define the user IRQ ENABLE word as

```
CODE ENABLE
<perform action to>
<restore user interrupt>
NEXT JMP
END-CODE
```

and also enable the IRQ interrupt as required with ENABLE .

For time-critical applications (such as writing data to a floppy disk), it is sometimes necessary to disable all IRQ interrupts except for the user interrupt. This type of masking is easily performed using the IRQ Interrupt Priority Mask PRIRTY. The PRIRTY mask is a write-only register that masks all IRQ interrupts below the set level (see Section 2.7.2 in the AIM 65/40 System User's Manual). Since the priority level is reconfigurable with a DIP header, these time-critical IRQ sources can be given the highest priority. For each IRQ source that is equal or higher in priority of the user IRQ source, this PRIRTY mask is not effective and separate enable and disable words should be created for each. The following FORTH code masks out all IRQ sources except for the RM65 Bus.

```
( DISABLE ALL IRQ SOURCES EXCEPT USERS)
HEX 60 USER UPRIRTY ( IMAGE OF PRIRTY)
00 UPRIRTY C! ( COLD RESET VALUE)
FF80 CONSTANT PRIRTY
FF CONSTANT USERMASK ( MASK BELOW USER)

( DISABLE ALL IRQ EXCEPT USER)
: DISABLE ( MASK OUT ALL IRQ ABOVE USER)
USERMASK PRIRTY C! (MASK BELOW USER) ;

( ENABLE ALL IRQ INCLUDING USER)
: ENABLE ( RESTORE ALL IRQ ABOVE USER)
UPRIRTY C@ PRIRTY C! ( RESTORE MASK) ;
```

The user variable UPRIRTY is an image of the PRIRTY register. Any application code that modifies PRIRTY from the cold reset value ($00) must also change UPRIRTY to reflect this value. The constant USERMASK is the bit pattern that masks off all IRQ sources below the highest priority. The DISABLE word disables all IRQ sources except for the user interrupt. First, any interrupts above the IRQ priority level are disabled (none in this example), then the mask for interrupts below the level is stored into the priority latch. The ENABLE word restores all IRQ sources to the previous state, with the value in UPRIRTY taken as the priority mask level.

CAUTION

Since the AIM 65/40 peripherals are IRQ interrupt driven, DISABLE <code> ENABLE should always be paired as closely as possible. Interactive debugging of code between DISABLE and ENABLE <u>cannot</u> be performed.

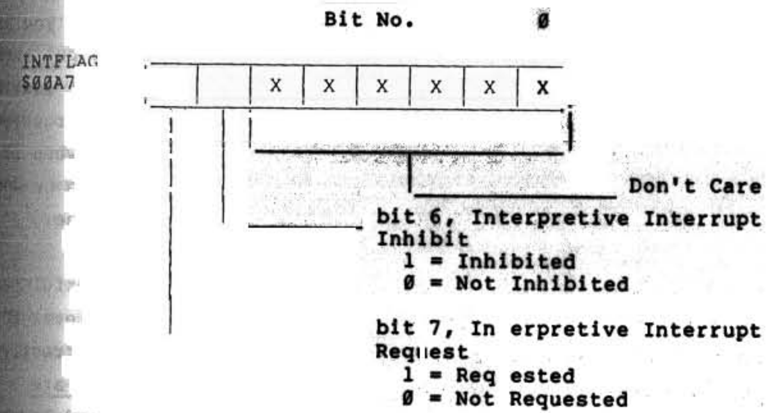
7.2.4 Example

An example of a conventional interrupt handler, written as a code fragment, is shown in Figure J-1 for the 24-Hour Clock example program described in Appendix J.

7.3 INTERPRETIVE INTERRUPT HANDLING PROCEDURE

7.3.1 Interrupt Service Subroutine

Write a minimum length interrupt service subroutine using the procedure described in Section 7.2 to load the AIM 65/40 interrupt vectors (and to disable/enable the IRQ interrupt). This routine needs only to set bit 7 in the FORTH interrupt flag INTFLAG (at $00A7) to one and return to the interrupted routine. The format of the INTFLAG variable word is



7.3.2 Interrupt Processing Word

The desired IRQ interrupt processing procedure uses a high level FORTH colon-definition word. Load the code field address (CFA) of this interrupt handler word into the FORTH interrupt vector INTVECT , a two-byte user variable located at $00A8 and $00A9. Note that upon FORTH initial entry, or upon executing FORTH word COLD , this vector is initialized to $D245, which points to ABORT processing.

When FORTH is executing its inner-interpreter, i.e., NEXT , it examines the interrupt request and inhibits bits of INTFLAG . When the interrupt inhibit (bit 6) of INTFLAG is ON, the interrupt request (bit 7) is ignored and NEXT executes the FORTH word. When the interrupt inhibit is OFF, the interrupt request (bit 7) of INTFLAG is tested. If the interrupt request is OFF, then NEXT executes the next FORTH word. If the interrupt request is ON, then NEXT passes execution to the word whose CFA is in INTVECT , i.e., the interpretive interrupt service word, and sets the inhibit bit.

The interpretive interrupt word now processes the service required without interruption (inhibit bit is set). When the interpretive interrupt word has finished, it must reset the inhibit bit to zero, restore the interrupted word to the interpretive pointer, and jump to NEXT to continue the interrupted execution. Remember to keep the assembly interrupt code as short and simple as possible. For example, if you are reading data values at specific times, read them and put them away in, say, a FORTH variable using a small interrupt routine for just that purpose. Meanwhile, a high level FORTH routine examines that variable for new data and processes it when it appears. FORTH is fast enough for much of the work to be done in high level which will speed program development time.

If FORTH is not fast enough for some purposes, a powerful technique is to first develop the program logic in high level FORTH and test the logic at reduced speed. When it works correctly, code in assembly language only those FORTH words that are required to bring the speed performance to the desired level. By this technique, program development time is reduced to the minimum.

7.3.3 Example

An example of an interpretive interrupt handler is shown in Figure J-3. Only two short CODE-definition words are defined; one to set the request bit in INTFLAG when an IRQ interrupt occurs due to VIA Timer 1 timeout, and one to clear the inhibit bit in INTFLAG when the interpretive interrupt word completes execution.

The changes to the 24-Hour Clock to use interpretive interrupts involve replacing the code interrupt handling routine with a colon-definition or code fragment service word, writing the interpretive interrupt arm and trigger words and then the FORTH interrupt processing word. The conventional interrupt handler from PHA, to RTI, is replaced with two smaller code routines, one a code fragment and the other the ARM word that goes at the end of the FORTH interrupt processing word.

The code fragment is the interrupt service subroutine that is executed with each IRQ interrupt. It sets the interrupt request bit in INTFLAG and clears the VIA's IRQ request bit which caused the interrupt. This code fragment serves as a typical example of all that is necessary to do at the code level for a wide variety of high level FORTH interrupt words.

The CODE word ARM turns OFF the interpretive interrupt inhibit bit (bit 6) of INTFLAG , restores the FORTH interpretive pointer into the interrupted FORTH word and then jumps to FORTH's inner-interpreter NEXT to continue execution. ARM could be written in high level FORTH using C@ , C! and ; words but it must not be interfered with by a high level interrupt. This interference cannot occur if the functions are done within a CODE-definition.

The FORTH interpretive interrupt word for the 24-Hour Clock is T+ . Another FORTH word, +!L is used often by T+ . These two words comprise the entire interpretive interrupt service word. T+ does just what the CODE-definition interrupt routine did, i.e., increment the hundredth's of a second byte by 5 and when it reaches 100, increment the seconds, minutes, etc. The utility word +!L increments a certain byte by a given amount and checks it against the given limit. If the limit is exceeded, it zeros the byte and returns a true value so that the next byte can be incremented. The arguments on the stack for +!L are:

```
limit 1-  byte-address  increment --- T/F
```

The CODE-definition word ARM stops execution of T+ . The word [switches FORTH from the compiling state to the interpreting state so that the word SMUDGE will be executed, which makes the name of T+ available in the FORTH dictionary.

In order for the interpretive interrupts to work, the code field address (CFA) of the interpretive interrupt word must be loaded into INTVECT . This is accomplished in this example by the following:

```
' T+          ( Obtain the PFA of T+)
CFA           ( Change it to the CFA)
ASSEMBLER     ( Switch to the FORTH assembler vocabulary)
INVECT        ( Obtain the address of  INTVECT )
!             ( Store the CFA of 'T+' in  INTVECT )
FORTH         ( Return to the FORTH only vocabulary)
```

which follows the definition of T+ .

Note that if the AIM 65 is executing machine code for an appreciable amount of time and not frequently executing NEXT , the FORTH interrupt routine will not be executed and interrupt requests may pile up or be lost (depending on the interrupt service subroutine). This can happen when using the printer or waiting for a key.

The proper choice of machine level or interpretive (or both) interrupt service routines can make a very flexible approach to control situations or understanding computer interrupts.

7.3.4 Points to Remember

a. Define and code all required words before loading INTVECT , or requesting an interpretive interrupt. The required CODE words are (see text for more detail)

   1 - the IRQ or NMI code fragment
   2 - the ARM word to rearm the interrupt, etc.
   3 - the ENABLE and DISABLE words for IRQ (if using IRQ).

   A colon-definition level FORTH word is also required to run at interrupt request. The last word executed by this word is ARM , above.

b. See that NMI and IRQ do not contend for the interpretive interrupt -- there is no stacking and they can get lost.

c. Do not alter any of FORTH's floating buffers (at HERE and PAD) or any of the USER variables (BASE , DPL , IN , etc.) or leave anything on the stack between interrupts.

d. Use caution when using interpretive interrupts -- think the sequence through before acting. If it does not operate correctly, perhaps you are overwriting something that FORTH needs. Try using a do-nothing word like

   : DUMMY ARM   SMUDGE

   for the interpretive :d and see if that works.

e. The X register contents must be saved if X is used during the interrupt processing, but not in XSAVE or any of the other regular FORTH "registers". For example, use the Return Stack instead, such as TXA, PHA

SECTION 8

PROGRAMMING THE R6522 IN FORTH

Chapter 7 in the AIM 65/4Ø System User's Manual explains how t program the R6522 Versatile Interface Adapter (VIA) in R65ØØ assembly language. FORTH can be used to program the VIA in a similar manner using its built-in assembler, however, in most cases, high-level FORTH can be used. This section repeats mos of the examples described in the AIM 65/4Ø System User's Manua but programs them in FORTH rather than assembler, except for a few instances where assembler is preferred.

The techniques shown in this section can easily be applied to other R65ØØ peripheral devices such as

R652Ø Peripheral Interface Adapter (PIA)
R6551 Asynchronous Communications Interface Adapter (ACIA
R6545 CRT Controller (CRTC)

or other similarly structured I/O or peripheral control devices.

8.1 VIA ORGANIZATION AND REGISTERS

In review, the R6522 is organized as shown in Figure 8-1 with its registers occupying 16 addresses as listed in Table 8-1. The addresses shown correspond to the user R6522 on the AIM 65/4Ø SBC Module. To summarize, the VIA operates as determined by the contents of four control registers.

a. Data Direction Register A (DDRA) determines whether the pins on Port A are inputs or outputs.

b. Data Direction Register B (DDRB) determines whether the pins on Port B are inputs or outputs.

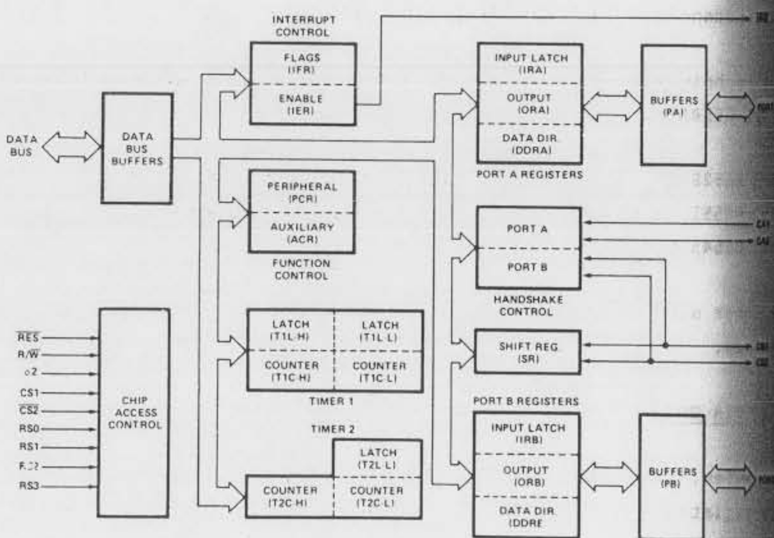


Figure 8-1. R6522 VIA Block Diagram

Table 8-1. R6522 Memory Assignments

Location	Function		
FFX0	Port B Output Data Register (ORB)		
FFX1	Port A Output Data Register (ORA) **Controls handshake**		
FFX2	Port B Data Direction Register (DDRB)	0 = Input	
FFX3	Port A Data Direction Register (DDRA)	1 = Output	
	Timer	R/$\overline{W}$ = L	R/$\overline{W}$ = H
FFX4	T1	Write T1L-L	Read T1C-L Clear T1 Interrupt Flag
FFX5	T1	Write T1L-H & T1C-H T1L-L → T1C-L Clear T1 Interrupt Flag	Read T1C-H
FFX6	T1	Write T1L-L	Read T1L-L
FFX7	T1	Write T1L-H Clear T1 Interrupt Flag	Read T1L-H
FFX8	T2	Write T2L-L	Read T2C-L Clear T2 Interrupt Flag
FFX9	T2	Write T2C-H T2L-L → T2C-L Clear T2 Interrupt Flag	Read T2C-H
FFXA	Shift Register (SR)		
FFXB	Auxiliary Control Register (ACR)		
FFXC	Peripheral Control Register (PCR)		
FFXD	Interrupt Flag Register (IFR)		
FFXE	Interrupt Enable Register (IER)		
FFXF	Port A Output Data Register (ORA) *No effect on handshake*		

X = A(User VIA), B(System VIA), C(Keyboard VIA)

The Peripheral Control Register (PCR) determines which polarity of transition (i.e., rising edge or falling edge) is recognized on the input status lines (CA1 and CB1) and how the other status lines (CA2 and CB2) operate.

The Auxiliary Control Register (ACR) determines whether the data ports are latched and how the timers and shift register operates.

Note that there is a data direction register for each side but only one pair of control registers. Ports A and B are almost identical. One important difference is that Port B can handle Darlington transistors which are used to drive solenoids and relays. We will generally use Port A for input and Port B for output in our examples.

8.2 SIMPLE I/O WITH THE VIA

8.2.1 Considerations

Since RESET clears all the VIA registers, disabling all interrupts and clearing all control lines, we can discuss simple I/O referring only to the data registers and the data direction registers. So simple I/O can be performed with the R6522 VIA as follows:

a. Establish the directions of the pins by storing the proper values in the data direction registers.

b. Transfer data by moving it to or from the data registers.

Note that most programs only have to execute step a once since the directionality of most input and output devices is fixed (i.e., you never want to read data from a display or printer or write data to a switch or paper tape reader).

You can establish directions as follows:

a. A '0' in a bit in the data direction register makes the corresponding pin an input.

For example, a '0' in bit 4 of data direction register A makes pin PA4 into an input.

b. A '1' in a bit in the data direction register makes the corresponding pin an output.

For example, a '1' in bit 6 of data direction register B makes pin PB6 into an output.

As for transferring data, remember that the R6502 microprocessor has no specific I/O instructions. Storing data in a VIA port that has been designated for output is equivalent to sending the data to the attached output device. Loading data from a VIA port that has been designated for input is equivalent to reading the data from the attached input device; but any instruction that acts on memory can serve as an I/O instruction if the specified address is actually an I/O device. You must be careful of the exact significance of such instructions in writing, reading, and documenting R6502 programs.

8.2.2 Examples

The first four examples can be done in almost exactly the same way in FORTH as they are done in assembler -- they are presented here as a bridge between assembler techniques and FORTH. First, the assembler label equivalences are emulated with some FORTH constants:

```
( R6522 ADDRESSES)
HEX
FFA0 CONSTANT UDRB
FFA1 CONSTANT UDRAH
FFA2 CONSTANT UDDRB
FFA3 CONSTANT UDDRA
FFA4 CONSTANT UT1L
FFA5 CONSTANT UT1CH
FFA6 CONSTANT UT1LL
FFA7 CONSTANT UT1LH
```

```
FFA8 CONSTANT UT2L
FFA9 CONSTANT UT2H
FFAA CONSTANT USR
FFAB CONSTANT UACR
FFAC CONSTANT UPCR
FFAD CONSTANT UIFR
FFAE CONSTANT UIER
FFAF CONSTANT UDRA
```

Then the examples are done as FORTH colon-definitions. These definitions will do exactly what the assembler code does for the same example. Note that we are in HEX the whole time and also, since FORTH uses page zero for its parameter stack it is not a very good idea to put things in there indiscriminently. After presenting these examples in exactly the same form as assembler they are done again in a way that completely avoids any conflict with the FORTH stack area. Note also that comments have been included beside the FORTH words for ease of understanding. In actual coding, you should include the comments along with the code. Remember that comments do not take up any space in the compiled FORTH object code.

a. Fetch da[illegible]om a simple [illegible]ut port (e.g., from a set of switc[illegible]r a keypad) [illegible] store it in memory location

```
HEX
: INDATA
Ø UDDRA C!              Set DDR A to Inputs)
UDRAH C@ 4Ø C! ;        Fetch input data and store
```

b. Send data to a simple output port (e.g., to set displays or relays) from memory location 4Ø.

```
: OUTDATA
FF UDDRB C!           ( Set DDR B to outputs)
4Ø C@ UDRB C! ;       ( Fetch data and output it)
```

You can mix inputs and outputs on a single port by establishing the directions of individual pins appropriately. Note that you can read the states of data pins even if they have been designated as outputs. Port B side is buffered so that it can always be read correctly; however, Port A side is not buffered so that it can only be read correctly if it is lightly loaded (or designated as inputs).

The above examples are all perfectly fine FORTH words and will work. They do not take advantage of FORTH's unique abilities though, and so to illustrate the point, they are done over now in better FORTH style. Note that with the proper choice of word names and order, the examples are more readable and nearly say in English what they actually do. Also, since proper FORTH coding almost always uses the parameter stack for temporary values, we have used it here as the comments indicate.

```
HEX
Ø CONSTANT BPORT
1 CONSTANT APORT

: INPUT                 --- b. Get Data from Port)
FFAØ + C@ ;

: OUTPUT                b ---. Output Data To Port)
FFAØ + C! ;

: DIRECTION           ( b---. Set Data Direction)
FFA2 + C! ;

: EX1                 ( ---b. Set Output and Input Data)
Ø APORT DIRECTION APORT INPUT ;

: EX2                 ( b---. Set Output and Output
                      Data)
FF BPORT DIRECTION BPORT OUTPUT ;
```

First, the constants APORT and BPORT are defined in such a way that their numeric value can be added to a fixed value to compute the A or B I/O port data direction addresses. Next, the words INPUT , OUTPUT and DIRECTION are defined to compute the correct port or direction register and then fetch or store data or store direction information. In this manner, then, we have defined convenient words that talk to or control portions of the AIM 65/4Ø SBC module User VIA. These words are now a part of FORTH just as VLIST or DUMP are and extend AIM 65/4Ø FORTH in the direction of peripheral control. These words are now used in a natural way in the new version of EX1 and EX2 and in the examples to follow.

If the I/O device is more complex, we may not be able to transfer data to or from it at will. In the input case, the processor must know when new data is available (e.g., a key has been pressed on a keyboard or a tape reader has read another character). In the output case, the processor must know whether the device is ready to receive data (e.g., a printer has finished printing the last character or a modem has completed the previous transmission).

8.3.1 Considerations

Normally, the input or output device provides a status signal. A transition on that line indicates the availability of data or the readiness of the device. The microcomputer I/O section must recognize the transition and allow the processor to determine that it has occurred.

You can handle this kind of I/O with the R6522 Versatile Interface Adapter as follows:

a. Attach the peripheral status input CA1 or CB1.

b. Determine which edge on the status line will be recognized by assigning a value to control register bit Ø (CA1) or 4 (CB1). A value of zero in that bit position means that the interrupt flag will be set by a high-to-low transition (or falling edge). A value of one means that the interrupt flag will be set by a low-to-high transition (or rising edge).

c. Determine whether a transition has occurred by examining bit 1 (CA1) or 4 (CB1) of the interrupt flag register. The bit will be one, if a transition has occurred.

d. Reset the [illegible]t flag by reading or writing the correspond[illegible] register. The flag is then ready to be used [illegible] next operation.

8.3.2 Examples

Let us now look at some examples:

a. Fetch data from an input port with an active high-to-low DATA READY strobe and place the data in memory location $4Ø.

```
: EX3
Ø UDDRA C!           ( Set DDR A to inputs)
Ø UPCR C!            ( Set CA1 Flag on falling edge)
BEGIN
UIFR C@ 2 AND
UNTIL                ( Wait for strobe occurance)
UDRAH C@ 4Ø C! ;     ( Fetch data and store in memory)
```

Clearing the Peripheral Control Register is unnecessary if the routine is starting from a reset. Note that reading the input Data Register clears the interrupt flag so that it is available for the next DATA READY signal.

b. Send data to an output port with an active high-to-low PERIPHERAL READY strobe. Get the data from memory location $4Ø and send it when the peripheral is ready.

```
: EX4
FF UDDRB C!          ( Set DDR B to outputs)
Ø UPCR C!            ( Set CB1 Flag on falling edge)
BEGIN
UIFR C@ 1Ø AND
UNTIL                ( Wait for strobe occurance)
4Ø C@ UDRB C! ;      ( Fetch data and output it)
```

Note that sending the data to the Output Data Register clears the interrupt flag so that it is available for the next PERIPHERAL READY signal.

For the second version of EX3 and EX4 the word ?STROBE has been defined to wait in a BEGIN ... UNTIL loop until a given bit in the IFR register turns ON. To help provide readable code the special word @IFR was defined to fetch the IFR register from the stack to be ANDed with a copy of the given mask byte. The word loops until the result of the AND operation is true (non-zero) and then exits and drops the extra copy of the mask byte. With similar motivation for clear coding, the word !PCR is defined to store a given byte in the PCR for setting up the desired event.

```
: !PCR   FFAC C! ;          ( Set PCR Register)

: @IFR   FFAD C@ ;          ( Fetch IFR)

: ?STROBE ( M ---.)         ( Wait for Strobe)
BEGIN                       ( Waiting)
DUP                         ( The Mask)
@IFR AND                    ( Pick Bit)
UNTIL                       ( It is on)
DROP ;                      ( Extra Mask)

: EX3
0 APORT DIRECTION
0 !PCR                      ( CA1 Falling)
2 ?STROBE                   ( CA1 Interrupt?)
APORT INPUT ;

: EX4
FF BPORT DIRECTION
0 !PCR                      ( CB1 Falling)
10 ?STROBE                  ( CB1 Interrupt?)
BPORT OUTPUT ;
```

Examples 5 through 8 are all modifications of EX3 and EX4 and use !PCR to setup for various I/O protocols.

Fetch data from an input port with an active low-to-high DATA READY strobe and place the data on the stack.

```
: EX5
0 APORT DIRECTION
1 !PCR               ( CA1 Rising)
2 ?STROBE            ( CA1 Interrupt?)
APORT INPUT ;
```

Note t[illegible] the VIA has [illegible] input and output latches. The ou[illegible] latches are [illegible]s enabled; output data is latche[illegible] [illegible]en it is sto[illegible] [illegible] an output data register. The in[illegible] latches, if [illegible] [illegible]re needed, can be enabled

by setting Bit 0 (Port A) or Bit 1 (Port B) of the Auxiliary Control Register. The input data will the be latched by the active transition on CA1 or CB1.

8.4 PRODUCING OUTPUT STROBES

The peripheral may also require information about when a transfer has occurred or whether the port is ready to receive data. For example, devices such as digital-to-analog converters commonly require a LOAD pulse to enter data into the converter. A multiplexed display requires an output signal that directs the next output properly. A communications device may need a signal to indicate that an input buffer is available or that an output buffer is full. Output signals may also be needed to turn devices ON or OFF, activate operator displays, or control operating modes.

8.4.1 Considerations

You can handle this kind of I/O [illegible]th the R6522 Versatile Interface Adapter as follows:

a. Attach the control output to CA2 or CB2.

b. Make CA2 (CB2) into an output by setting control register bit 3 (7).

c. Make CA2 (CB2) into a pulse by clearing control register bit 2 (6) or into a level by setting that bit.

d. If CA2 (CB2) is a pulse, make it into a handshake signal (low from the time the Output Register is read or written until the next active transition on CA1 (CB1) by clearing control register bit 1 (5) or into a single-cycle strobe by setting that bit.

e. If CA2 (CB2 [illegible]s a level, determine its value by clearing or [illegible]tting bit 1 (5).

8.4.2 Options

The options are:

a. CA2 goes low when the processor transfers data to or from Output Register A, and goes high when the next active transition occurs on CA1. The signal can indicate that the port is ready for more data or that output data is available. The peripheral's response then indicates that it has sent more data or has processed the previous data.

b. CA2 goes low when the processor transfers data to or from Output Register A and goes high after one clock cycle. This signal indicates that an input or output operation has occurred and can be used for multiplexing.

c. CA2 is a level controlled by the value of control register bit 1. This signal can provide an active-high or low pulse of arbitrary length. It can be used to load registers, turn devices ON or OFF, or control operating modes.

8.4.3 Examples

Let us now look at some examples:

a. Fetch data from an input device that requires a handshake signal and that produces an active high-to-low DATA READY strobe. Place the data on the stack.

```
: EX7
Ø APORT DIRECTION
8 !PCR                 ( CA1 Falling)
2 ?STROBE              ( CA1 Interrupt?)
APORT INPUT ;
```


The Peripheral Control Register bits are

bits 4-7 = Ø since CB1 and CB2 are not use

bit 3 = 1 to make CA2 an output

bit 2 = Ø to make CA2 a pulse

bit 1 = Ø to make CA2 a handshake acknowledgement that remains low until the next active transition on CA1

bit Ø = 1 to make the active transition on CA1 a falling edge (high-to-low transition)

b. Fetch data from an input device that requires a brief DATA ACCEPTED strobe for multiplexing or control purposes. Place the data on the stack.

```
: EX8
Ø APORT DIRECTION
A !PCR                 ( CA1 Falling)
2 ?STROBE              ( CA1 Interrupt?)
APORT INPUT ;
```

Here bit 1 of the Peripheral Control Register is set to 1 to make CA2 a brief strobe lasting one cycle after the reading of Port A Input Data Register.

c. Send data to an output device that requires a handshake signal and that produces an active low-to-high PERIPHERAL READY strobe. The data is assumed to be on the stack and is sent when the peripheral is ready.

```
: EX9
Ø APORT DIRECTION
9Ø !PCR                ( CB1 Rising)
1Ø ?STROBE
APORT OUTPUT ;
```

The Peripheral Control Register bits are:

bit 7 = 1 to make CB2 an output

bit 6 = Ø to make CB2 a pulse

bit 5 = Ø to make CB2 a handshake acknowledgement that remains low until the next active transition on CB1

bit 4 = 1 to make the active transition on CB1 a rising edge (low-to-high transition)

bits Ø-3 = Ø since CA1 and CA2 are not used.

d Send data to an output device that requires a brief OUTPUT or DATA READY strobe for multiplexing or control purposes. The data is assumed to be on the stack.

```
: EX1Ø
FF BPORT DIRECTION
AØ !PCR              ( CB2 Pulse)
BPORT OUTPUT ;
```

Here bit 5 of the Peripheral Control Register is set to 1 to make CB2 a brief strobe lasting one cycle after the writing of Port B Output Data Register.

e Fetch data from an input device that requires an active-high START pulse. The device produces an active high-to-low DATA READY strobe. Place the data on the stack.

```
: EX11
Ø APORT DIRECTION
C !PCR               ( Reset)
E !PCR               ( Set Start)
C ! PCR              ( Reset)
2 ?STROBE
APORT INPUT ;
```

Here bit 2 of the Peripheral Control Register is set to 1 to make CA2 a level with the value given by bit 1 of the Peripheral Control Register. This mode can be used to produce pulses of any length and polarity; it is called the manual output mode because there is no automatic pulse information.

In a typical application, an analog-to-digital converter or data acquisition system usually needs a START CONVERSION pulse to begin operations.

f. Send data to an output device that must be turned ON before the data is sent and turned OFF after the data is sent (a logic 1 on a control line turns the device ON). The peripheral produces an active low-to-high PERIPHERAL READY strobe. The data is assumed to be on the stack and is sent when the peripheral is ready.

```
: EX12
FF BPORT DIRECTION    ( Set CB2 High)
FØ !PCR               ( Set CB1 Flag on Rising
                        Edge)
1Ø ?STROBE            ( Ready?)
BPORT OUTPUT
DØ !PCR ;             ( Turn off)
```

In many applications, such as portable equipment, the output peripheral is only turned ON when data is to be sent to it. In other applications, the processor must issue an OUTPUT REQUEST and receive an acknowledgement before sending the data.

In the FORTH versions of EX1Ø , EX11 and EX12 , we use the previously defined DIRECTION , !PCR , INPUT , OUTPUT and ?STROBE words to our advantage. These extensions to AIM 65 FORTH make coding most types of VIA I/O words very convenient. For these examples a further refinement of naming a constant Ø ALL-IN and FF ALL-OUT would result in the very readable phases:

```
ALL-OUT BPORT DIRECTION
```

and

```
ALL-IN APORT DIRECTION
```

8.5 VIA INTERRUPTS

8.5.1 Considerations

You can easily use the R6522 Versatile Interface Adapter in an interrupt-driven mode. Figure 8-2 shows the Interrupt Enable Register (IER). Any of the various interrupt sources can be enabled by setting the corresponding enable bit. Note that the most significant bit controls how the other enable bits are affected:

> If IER7 = 0, each '1' in a bit position clears an enable bit and thus, disables that interrupt.
>
> If IER7 = 1, each '1' in a bit position sets an interrupt bit and thus, enables that interrupt.
>
> Zeroes in bit positions always leave the enable bits as they were.

8.5.2 Examples

For examples of how to set up the VIA's Interrupt Enable Register, the word !IER is defined to make things as simple as possible. The word takes an argument from the stack which, like the previous examples, is an 8-bit pattern to store in the Interrupt Enable Register.

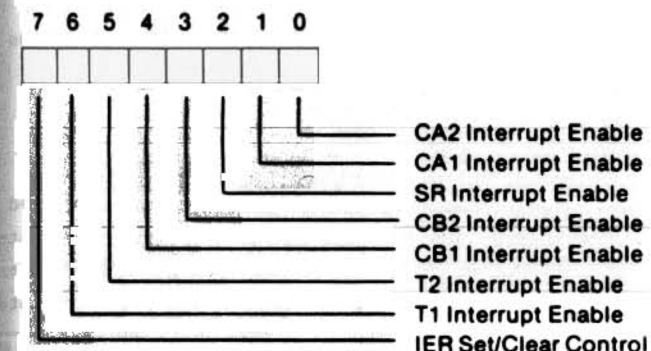
```
: !IER            ( Put Byte)
FFAE C! ;         ( In IER)
```

a. Enable CA1 interrupt; disable all others.

```
: EX13
7D !IER 82 !IER ;
```

The first operation clears all the interrupt enables except CA1. The second operation sets the CA1 interrupt enable.

R6522 INTERRUPT ENABLE REGISTER (IER), LOC. $FFXE



INTERRUPT ENABLE BITS (IER0-6)

IERn = 0 Disable interrupt
= 1 Enable interrupt

IER SET/CLEAR CONTROL (IER7)

IER7 = 0 For each data bus bit set to logic 1, clear corresponding IER bit
= 1 For each data bus bit set to logic 1, set corresponding IER bit.

Note: IER7 is active only when R/$\overline{W}$ = L; when R/$\overline{W}$ = H, IER7 will read logic 1.

Figure 8-2. R6522 Interrupt Enable Register (IER) b Enable CB1 and CB2 interrupts; disable all others.

```
: EX14
67 !IER 98 !IER ;
```

Note that we could disable all interrupts in the first step.

c. Disable CA1 interrupt; leave others as they were.

```
: EX15
1 !IER ;
```

d. Disable CB1 and CB2 interrupts; leave others as they were.

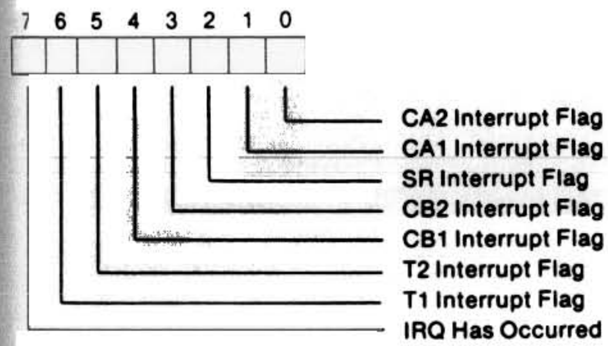
```
: EX16
18 !IER ;
```

The processor can determine which interrupt has occurred by examining the Interrupt Flag Register (Figure 8-3). Note that examining bit 7 determines if any interrupts have occurred on the VIA. Note also, the conditions for clearing the interrupt flags.

A typical polling sequence in regular R6502 assembly language would be:

```
LDA   UIFR    ;ANY INTERRUPTS ON THIS VIA?
BPL   NXT    ;NO, LOOK AT NEXT POSSIBLE SOURCE
ASL   A       ;IS INTERRUPT FROM T1
BMI   TIM1    ;YES, GO SERVICE T1 INTERRUPT
ASL   A       ;IS INTERRUPT FROM T2?
BMI   TIM2    ;YES, GO SERVICE T2 INTERRUPT
ASL   A       ;IS INTERRUPT FROM CB1?
BMI   CB1     ;YES, GO SERVICE CB1 INTERRUPT
.
.
.
```


R6522 INTERRUPT FLAG REGISTER (IFR), LOC. $FFXD



IFR Bit	Set By	Cleared By
0	Active transition on CA2	Reading or writing the ORA
1	Active transition on CA1	Reading or writing the ORA
2	Completion of eight shifts	Reading or writing the SR
3	Active transition on CB2	Reading or writing the ORB
4	Active transition on CB1	Reading or writing the ORB
5	Time-out of Timer 2	Reading T2C-L or writing T2C-H
6	Time-out of Timer 1	Reading T1C-L or writing T1L-H
7	Any IFR bit set with its corresponding IER bit also set	Clearing IFR0-IFR6 or IER0-IER6

Figure 8-3. R6522 Interrupt Flag Register (IFR)

The same fragment of assembler code, shown above, is coded below using the FORTH assembler. Note that this code is structured and therefore does not require labels.

```
HEX
FFAD LDA,                  ( IFR)
Ø< NOT IF,                 ( No VIA Interrupts)
 <assembly code>
 <to look for next>
 <source of possible>
 <interrupt>
ELSE,
  .A ASL,
 Ø< IF,                    ( T1 Interrupt on?)
  <T1 interrupt>           ( Yes)
  <service assembly>
  <code>
 ELSE,
   .A ASL,
  Ø< IF,                   T2 Interrupt on?)
   <T2 interrupt)
   <service assembly>
   <code>
  ELSE,
    .A ASL,                ( CB1 Interrupt on?)
   Ø< IF,                  ( Yes)
    <CB1 interrupt>
    <service assembly>
    <code>
   ELSE,
        (Etc.)
   THEN,
  THEN,
 THEN,
THEN,
```


The above FORTH assembler code will do exactly what the regular code does but even polling of interrupts can be done on a high level in FORTH. Using the word @IFR defined for EX3 to fetch the contents of the Interrupt Flag Register we then define:

```
: ?IRQ
@IFR 8Ø AND ;              ( From VIA)
```

which yields a true or false value depending on the state of bit 7 of the IFR. Then a direct form of the polling sequence would be:

```
: ?ON
OVER AND ;                 ( Mask bit on?)

  POLL-VIA                 ( Poll for VIA Interrupts)
?IRQ IF
 @IFR 4Ø ?ON               ( Any Interrupts on?)
  IF                       ( T1 Interrupt on?)
   <T1 interrupt>
   <service FORTH>
   <code>
  ELSE 2Ø ?ON
   IF                        T2 Interrupt on?)
    <T2 interrupt>
    <service FORTH>
    <code>
   ELSE 1Ø ?ON
    IF                       CB1 Interrupt on?)
     <CB1 interrupt)
     <service FORTH>
     <code>
        .
        .
      ( etc.)
        .
        .
    THEN
   THEN
  THEN
 DROP                      ( Copy of IFR)
THEN                       ( Done)
```

SECTION 9

NOTES ON STYLE AND PROGRAM DEVELOPMENT

9.1 GENERAL

Like most other programming languages FORTH is not particularly readable to someone who is not familiar with it. Because FORTH is unique among programming languages even experienced programmers have difficulty at first -- FORTH is unlike their past experience. After reading this manual FORTH should be understandable and some practice in coding will sharpen the eye. The key to easily readable FORTH code lies in logical factoring of key words and in choosing appropriate names for words.

Correct FORTH code should read in an almost natural English way for the higher level key words if these practices are followed. Factoring means writing a word as a collection of lower level words that actually name the functions performed by this word. These lower level words in turn are made up of other still lower level words which more precisely define the task accomplished by the word. Finally, you will arrive at a level which uses mostly regular FORTH words and is the most fundamental level, not particularly obvious to anyone but the system programmer, and then only when coding the word and for a short time afterwards.

Comments are necessary to understanding the operation of words. What may be perfectly clear to you today will not be next month or to another equally qualified programmer at any time. Comment in simple statements using the appropriate FORTH word names adjacently where you can. Be a bit more detailed than you think necessary when you write them, since they will be a bit too obscure later if you don't. However, comments are not tutorials to the novice or non-programmer. Too much verbage obscures the program flow in a sea of text. Commentary documentation that must speak to the non-programmer should be written elsewhere in a companion document.

Comments given at the start of a FORTH word should include a simple statement about what the word takes from and leaves on the stack.

9.2 EXAMPLE PROGRAM

To illustrate the style and comment forms described above, look at the many examples elsewhere in this manual and you will see the top-down approach in use, and reasonable comments that can be included on a 40 column printer. This section takes as an example simple problem of figuring the miles a car traveled per gallon of fuel from data kept in a common automobile record book.

a. Problem Definition

Start with the definition

$$mpg = \frac{\text{miles traveled}}{\text{gallons used}}$$

which is fine if you record mileage and gallons and you aren't particular about accuracy. For the newer automobiles, an error of a tenth of a gallon can cause an error of 0.5 mpg in the result.

For best accuracy you should accumulate mileage and gasoline used over several fill-ups. This averages your error in filling the tank non-uniformly and then by recording the data carefully you can have an accurate picture of your gas mileage.

$$mpg = \frac{\text{miles}}{\text{gallons}}, \text{ but gallons} = \frac{\text{amount}}{\text{price}},$$

so

$$mpg = \frac{\text{miles}}{\frac{\text{amount}}{\text{price}}} = \frac{\text{miles(price)}}{\text{amount}}$$

However, miles is not the odometer reading, it is th miles traveled, and the odometer will require a correction factor, so

$$\text{miles} = (m_1 - m_0)k$$

m_0 = last odometer readin
m_1 = current odometer reading
K = correction factor

therefore,

$$mpg = \frac{(m_1 - m_0)kp}{a}$$

p = price
a = amount

But price is either in cents per gallon or cents per liter, and since you are probably interested in miles per gallon you will have to multiply any cents per liter prices by 3.785 to correct to cents per gallon.

Finally

mpg = [illegible] for cents/gallon

or

$$mpg = \frac{(m_1 - m_0)k}{a}\left(\frac{3785p}{1000}\right) \text{ for cents/gallon}$$

To be efficient when doing more than one gas mileage check, the current odometer reading should be saved as m_0 for the next calculation.

b. Scaling

To correctly scale the calculation for integer computation, you must first decide the precision is desired in the answer. If you want mpg to a tenth of a mile per gallon, distance in miles, price to a tenth of a cent and amounts in cents, then

$$mpg(10) \quad \frac{(m_1-m_0)p(10)}{a}$$

If we enter p without the decimal, we get p times 10 automatically and the result will come out in 10's of mile per gallon as desired.

Program Design, Coding and Checkout

If the data are recorded in a data book as miles, price, and amount, it is convenient to enter them as written, therefore the stack would look like this:

m_1 p a

and they would all be 16-bit numbers.

Given the data on the stack as described above and the odometer correction and price adjustment necessary, the principle word looks (without any comments) like this:

```
: ?MPT    ROT   TRUE-MILES    ROT    CENTS/GAL
          ROT   */    .MPG
```

where the ROT words bring the stack values up to be operated on by the fairly obvious correction and adjustment words. The */ computes the final operation

$$\frac{mp}{a}$$

and the word .MPG prints the answer out in a nice format with a decimal point where we expect it.

Now that we have the 'top' level structure, let us define the lower level words TRUE-MILES , CENTS/GAL and .MPG .

It is not necessary but a convenience to use two memory storage locations in this calculation, one constant for k and a variable where we can store the current odometer reading (m_1) to use as the last odometer reading (m_0) for the next calculation.

Start with the word .MPG and use the normal FORTH output formatting words except include a decimal point in the output text string with the phrase 46 HOLD and embellish the result with the ending "MPG" .

Enter the program source code into the AIM 65/40 Text Buffer and compile from there. If any error occurs during text entry, it will be easy to correct the source code and recompile. Notice that blank lines (enter <SPACE> followed by <RETURN>) aid in source code readability.

```
{E}
EDIT FROM=2000 TO=3FFF IN=<RETURN>

( MPG PROGRAM )

( CONST & VARIABLES )
103 CONSTANT K
  0 VARIABLE OLD

: .MPG ( MPG * 10 ---. DISPLAY MPG )
S->D <# # ( 1 DIG. )
46 HOLD ( DEC. PT. )
#S #> ( FINISH IT)
CR TYPE ."  MPG  " ;
CR ." DONE"
FINIS

*END*
={Q}

{5}
 AIM 65/   FORTH V1.4
SOURCE <  TURN> IN=M
DONE
OK
```

The test of .MPG puts a few numbers on the stack before the test number 456 and then execute .MPG along with .S to see that the stack contents have not been altered.

```
1 2 3 456 .MPG .S
45.6 MPG
3
2
1 OK
```

With .MPG working correctly, define TRUE-MILES which uses */ as a scaling operator. The constant K (derived for each odometer and set of tires separately) is multiplied by the miles traveled (M - OLD) and then divided by 100. The decision point for CENTS/GAL tells if the price for a gallon or liter is $1.00 and should be correct for a while yet. The adjustment for liter prices is

$$P\frac{3785}{1000}$$

which results in cents per gallon equivalent.

Escape and re-enter the Text Editor. Go to the bottom and read in the rest of the program. After validating the source code entry, exit the Text Editor, enter AIM 65/40 FORTH and compile. You can quickly enter and compile the program if you do not code the comments, but don't forget the terminating ; for each word.


```
<ESC>
(ESC)
{T}
( MPG PROGRAM )
={B}
FINIS
={U}/1
CR ." DONE"
={R} IN=<RETURN>

: TRUE-MILES ( ODOMETE      ADJUST MILEAGE
OLD @ ( OLD #)
OVER OLD ! ( NEW #)
- ( MILES)
K ( CORRECTION )
100 */ ( ADJUST) ;

: CENTS/GAL ( PRICE --   CONVERT PRICE )
DUP ( FOR COMPARE)
1000 < ( ? < $1.00)
IF ( CENTS/LITER)
  3785 100 */
THEN ;

: ?MPG ( ODO PRICE AMT    . DISPLAY MPG
ROT ( GET ODOMETER)
TRUE-MILES
ROT ( GET PRICE)
CENTS/GAL
ROT ( GET AMOUNT)
*/ ( COMPUTE MPG)
.MPG ( & PRINT) ;
<RETURN>
CR ." DONE"

={Q}

{5}
 AIM 65/40 FORTH V1.4
SOURCE <RETURN> IN=M
DONE
OK
```

If any errors occur during compilation, check the source code for entry errors. Compile each word separately, if needed, to verify proper coding by bracketing each word before the word (before the : using the Monitor C command to move the top of the Text Buffer, and after the word (after the ;) with FINIS d **Program Final Testing**

The testing of ?MPG involves entering some known values into it and observing the results. Don't forget to set the OLD value for the odometer reading first, as shown below. Then enter test values on the stack for the current odometer reading, the price, and the miles traveled.

```
3198 OLD ! OK
3457 1199 841 ?MPG
37.9 MPG OK
3665 1169 1038 ?MPG
24.1 MPG OK
3839 1199 1063 ?MPG
20.1 MPG OK
4017 1339 1150 ?MPG
21.3 MPG OK
4200 1329 997 ?MPG
25.0 MPG OK
OLD ? 4200 OK
```

Finally, if you want to put the decimal points in the input numbers, remember that AIM 65/40 FORTH interprets this as a 32-bit number. So, for every number with a decimal point in it, you will have two 16-bit numbers on the stack.

e. **Program Enhancement**

You can redefine word ?MPG which will take such numbers and rearrange them to be acceptable to the original ?MPG . This time the input is in whole miles, cents and a tenth and dollars.

The 32-bit numbers go on the stack with the most significant part on top. Since none of the numbers are even close to using the most significant 16-bit part, simply drop them off the stack at the appropriate place and use the old version of ?MPG .


```
<ESC>
(ESC)
{T}
( MPG PROGRAM )
={B}
FINIS
={U}/1
CR ." DONE"
={R} IN= <RETURN>
: ?MPG ( 0DO CENTS $ --   COMPUTE MPG)
DROP ( UNUSED WD.)
SWAP ( OTHER ONE)
DROP ( IT TOO)
?MPG ( USE IT OVER)
;
<RETURN>
CR ." DONE"
={Q}
{5}
 AIM 65/40 FORTH V1.4
SOURCE <RETURN> IN=M
?MPG NOT UNIQUE
DONE
OK
```

Now test it with miles, cents and a t h, and dollars.

```
3198 OLD ! OK
3457 119.9 8.41 ?MPG
37.9 MPG OK
```

Check to be sure OLD was updated.

```
OLD ? 3457 OK
```

SECTION 10

PREPARING AN APPLICATION PROGRAM FOR PROM INSTALLATION

It is often desired to install an application program written in FORTH into one or more PROM/ROM devices for immediate operation upon microcomputer power turn-on, i.e., without requiring entry into FORTH under operator control from the Monitor, or compilation of the application FORTH source code. This section describes a method to develop a 4K-byte application program written in FORTH high level (code-definitions) and/or low level (assembly language CODE-definitions) using batch compilation, how to locate it for execution from either RAM or PROM/ROM on the AIM 65/40 microcomputer, and how to install a start-up driver program for FORTH initialization.

Three example startup drivers are illustrated. Each driver requires that the AIM 65/40 FORTH ROMs be installed. One driver operates with both the I/O and Monitor/Editor ROMs installed and takes advantage of the user key decoding provided by the Monitor linkage. The second example driver operates with only the I/O ROM installed. Both of these drivers provide unique cold and warm start initialization as well as common initialization paths. A third driver operates with neither the I/O ROM nor Monitor/Editor ROMs installed. This last configur-ation would be appropriate only for an application program providing the total initialization, reset and I/O processing.

The general procedure is illustrated for one of the drivers. This procedure can be easily modified, to accommodate larger programs, by changing address boundaries of the Text Buffer in Editor (source code), and/or the object code and by compiling from mass storage, if necessary. Since compiled FORTH object code is very compact, a fairly large application program can b provided in 4K bytes of object code.

CAUTION

This procedure changes the dictionary linkage variables from default AIM 65/40 FORTH values to application dependent values. Correct dictionary linkage is vital to the proper operation of FORTH, thus the procedure to change it must be carefully followed. Should the dictionary linkage be improperly altered, a cold reset may be necessary to recover proper operation.

10.1 GENERAL PROCEDURE

The general procedure is illustrated for the example driver described in Section 10.2.1. The memory map for this example is:

0 - $FF	Page 0
$100 - $1FF	Page 1
$200 - $7FF	System and FORTH Variables
$800 - $80A	TASK dummy word
$1000 - $7FFF	Applica on Sour e Code (Text Buffer)
$8000 - $803F	Applica on Obje t Code (Startup Driver)
$8040 - $8FFF	Applica on Obje t Code (FORTH words)

1. Verify that RAM is installed and selected from 0 - $8FFF.

2. If the source code is to be compiled from mass storage, go to step 18. Otherwise, enter the Editor. Select source code entry from the keyboard (illustrated) or from mass storage.

```
{E}
EDIT FROM=1000 TO=3FFF IN=<RETURN>
```

3. Now create the text file that will be compiled. First, forget TASK to allow subsequent linkage from TASK to the last application word:

```
( TEXT FILE FOR PROM-FORTH PROCEDURE)
FORGET TASK CR
```


4. Change the address of the dictionary pointer (in user variable DP) to point to the first address of the PROM/ROM area (now RAM):

```
HEX 8000 DP !
```

5. Change the first and last addresses of the data buffer (in user variables UFIRST and ULIMIT) to values equal to, or greater than, $A0 above the last address of the PROM/ROM area, to allow compilation:

```
90A0 DUP UFIRST ! ULIMIT !
```

Note that these last two steps may, alternatively, be done in an interactive manner in FORTH prior to compilation, however, including them with the source code saves re-entering the steps during development.

6. Include the follow u want to verif changing of DP , UFIRST , and during compila on:

```
HERE 0 D.
UFIRST @ 0 D. ULIMIT @ 0 D. CR
```

At this point, the start up driver for the application code to reside in the PROM/ROM is ready to be entered -- starting at $8000. This can be entered using the FORTH assembler as described below in steps 7 and 8, or using the AIM 65/40 Assembler as described in Section 10.2. If the driver is to be programmed using the AIM 65/40 Assembler, or the Enter Mnemonic Instruction function in the AIM 65/40 Monitor (see Section 4.5.1 in the AIM 65/40 System User's Manual), skip to step 9.

7. If the I/O ROM is installed, include constants to identify the PROM/ROM (at $8000) and to cause auto-start processing (at $8001 and $8002) -- see Section 6.2.2 AIM 65/40 System User's Manual.

```
( AUTO-START BYTES)
  40 C,                  40 = typical value)
  77 C, A5 C,            $5A, $A5 = do autostart)
```

Note that a value other than $5A, e.g., 77 in this case, is used until the program is fully debugged, otherwise a RESET may cause improper AIM 65/4Ø and/or FORTH operation.

8. Include the startup driver assembly code. The following code, in FORTH assembly format, implements the driver described in Section 1Ø.2.2. This driver enters the FORTH command mode immediately upon completion of auto-start processing (i.e., without returning to the I/O ROM). The value to load into the TASK NFA field is left zero until after the application words are compiled (see Step 15).

```
( START-UP DRIVER FOR AIM 65/4Ø FORTH)
ASSEMBLER                   ( ASSEMBLY VOCABULARY)
D293 JSR,                   ( INIT FORTH)
D2D1 JSR,                   ( INIT FORTH)
ØØ # LDA,                   ( INIT TASK LFA)
DP @ 1- Ø D. ." =LSB "
8Ø5 STA,                    ( STORE TASK LSB)
9F STA,                     ( STORE IP LSB)
ØØ # LDA,
DP @ 1- Ø D. ." =MSB "
8Ø6 STA,                    ( STORE TASK MSB)
AØ STA,                     ( STORE IP MSB)
92 # LDX,                   ( INIT PARAMETER STACK)
CØ6F JMP,                   ( TO FORTH COLD START ENTRY)
FORTH                       ( RETURN TO FORTH VOCABULARY)
```

Note that the CØ6F JMP, can be replaced with a NEXT JMP, to immediately begin application execution.

9. If the start-up driver is coded in FORTH, the compiled application words will be located immediately after the end of the driver machine code. In this case, skip to step 1Ø.

If the driver is to be merged later, room must be left for the driver code ahead of the application code. The dictionary pointer (in the user variable DP) is therefore changed to where the application code will start ($8Ø4Ø in this example). Words to verify the value during compilation may also be included.

```
8Ø4Ø DP ! ( START OF APPLICATION WORDS
CR HERE  Ø D.
```


1Ø. Enter the application source code colon and/or CODE definitions:

```
EXAMPLE APPL   ATION FORTH PROGRAM
 WD1X CR ."    T WORD 1" CR ;
 WD2X CR ."    T WORD 2" CR ;
```

11. Change the address of user variable DP back to the initial value to allow TASK to be redefined in low memory, i.e., at $8ØØ.

```
8ØØ DP !
```

12. Change the first and last addresses of the data buffer back to the initial values.

```
4ØØØ DUP UFIRST ! ULIMIT !
```

13. Include the following if you   rify restoration of DP , UFIRST , and ULIMIT ,   values during compilation:

```
CR HERE Ø D.
UFIRST @ Ø D  ULIMIT @ Ø D.
```

14. Redefine TASK at $8ØØ:

```
TASK ;
```

15. Put the name field address (NFA) of the last application word (referred to as LSB and MSB) into the TASK LFA field. Note that the LSB and MSB (i.e., $8ØØA and 8ØØF in this example) depend upon the driver assembly language code (see Step 8).

```
' WD2X NFA DUP Ø D. Ø 1ØØ U/
8ØØF C! 8ØØA C!
```

16 Add a message to indicate completion of compilation and include FINIS to indicate end of words to compile and to terminate text input:

```
CR ." DONE"
FINIS
<RETURN>

*END*
```

17. Return to the Monitor and enter FORTH:

```
={Q}

{5}
AIM 65/40 FORTH V1.4
```

18. Compile from the Text Editor or mass storage (illustrated from the Text Editor with user variable changes verified):

```
SOURCE <RETURN> IN=M
8000 90A0 90A0
800A =LSB 800F =MSB
8040
800 4000 4000 805B DONE
```

19. Run a VLIST to verify compilation (shown after example words):

```
VLIST
  809 TASK              8064 WD2X
 8049 WD1X              D9DC .S
 D9D1 MON OK            ( <SPACE> bar pressed)
```

20. Try the application words:

```
WD1X <RETURN>
TEST WORD 1
OK
WD2X <RETURN>
TEST WORD 2
OK
```

21 Escape to the Monitor

```
<ESC>
( ESC)
```


22. Change the value of $8001 to $5A to enable driver aut start processing (see Step 7):

```
{M}8000 40 77 A5 20 93 D2 20 D1 @[
{/}8000 40 5A <RETURN>
```

23. Verify operation after a cold reset, either at the FORT command or run-time level

```
<RESET>
AIM 65/40 FORTH V1.4
```

24. Test the application words.

```
WD1X <RETURN>
TEST WORD 1
OK
WD2X <RETURN>
TEST WORD 2
OK
```

25. Save the object code at $8000-$8FFF on mass storage, using the Monitor D command, for preparation of a PROM/ROM.

26. Save the source code in the Text Buffer on mass storage, using the Editor L command, for future updates.

Figure 10-1 shows a compilation and test of the example program using this procedure. The listing includes the application program source code, the compilation, command of the test words, and a cold reset, followed by automatic execution of the last application word. This listing assumes proper operation has been previously verified so that auto-start constant (i.e., $5A) is compiled rather than being changed later.

10.2 EXAMPLE START-UP DRIVERS

10.2.1 With I/O and Monitor ROMs Installed

The start-up driver listed in Figure 10-2 illustrates a start-up driver for use with both the I/O and Monitor firmware installed. During auto-start, this driver checks for the

```
FORGET TASK CR
HEX 8000 DP !
90A0 DUP UFIRST ! ULIMIT !
HERE 0 D.
UFIRST @ 0 D. ULIMIT @ 0 D. CR
40 C,
5A C, A5 C,
( START-UP DRIVER FOR AIM 65/40 FORTH)
ASSEMBLER
D293 JSR,
D2D1 JSR,
00 # LDA,
DP @ 1- 0 D. ." =TASK LSB "
805 STA,
9F STA,
00 # LDA,
DP @ 1- 0 D. ." =TASK MSB"
806 STA,
A0 STA,
92 # LDX,
C06F JMP,
FORTH
( EXAMPLE APPLICATION FORTX PROGRAM)
: WD1X CR ." TEST WORD 1" CR ;
: WD2X CR ." TEST WORD 2" CR ;
800 DP !
4000 DUP UFIRST ! ULIMIT !
CR HERE 0 D.
UFIRST @ 0 D. ULIMIT @ 0 D.
: TASK ;
' WD2X NFA DUP 0 D. 0 100 U/
8011 C! 800A C!
CR ." DONE"
FINIS
 *END*
=(Q)

(5)
 AIM 65/40 FORTH V1.4
SOURCE  IN=M
8000 90A0 90A0
800A =TASK LSB 8011 =TASK MSB
800 4000 4000 8039
DONE


WD1X
TEST WORD 1
OK
WD2X
TEST WORD 2
OK
```

Figure 10-1. Example Driver Compilation and Test


```
PAGE 0001    FORTH STARTUP DRIVER FOR AIM 65/40 WITH MONITOR

ADDR OBJECT  SOURCE

              ; PROGRAM EQUATES
              TSKLFA=$0805            ;LINK FIELD ADDRESS OF TASK
              INT4TH=$D293            ;FORTH VARIABLE INITIALIZATION
              FNDTOP=$D2D1            ;FIND FORTH VARIABLE DEFAULT VALUES
              LAST=$0000              ;POINTER TO LAST APPLICATION WORD (NFA)
              START=$8000             ;APPLICATION PROGRAM START
              SETLNK=$A432            ;LINK TO USER COMMAND DECODER
              R80LNK=$03FC            ;FOR APPLICATION AT $8000
              FORENT=$C545            ;ENTRY TO FORTH
              EX2MON=$03F4            ;INDIRECT EXIT FOR START DECODE
                     *=START
0000 81              .BYT $81         ;AUTO-START PROGRAM ID
0001 5A              .BYT $5A,$A5     ;AUTO-START KEY PATTERN
0002 A5
0003 10 15           BPL WARM         ;WARM RESET ==>
0005 C9 FF           CMP #$FF         ;CHECK FOR MONITOR PRESENT
0007 F0 11           BEQ WARM         ;YES ==>
              ;
              ; ON COLD RESET, PERFORM THE REQUIRED APPLICATION CODE
0009 EA       COLD   NOP              ;APPLICATION DEPENDENT
              ; ALSO SET UP THE FORTH LINKAGE TO THE APPLICATION PROGRAM
000A A2 1D           LDX #<DECODE     ;INITILIALIZE MONITOR DECODE LINKAGE
000C A0 80           LDY #>DECODE
000E 20 32 A4        JSR SETLNK
0011 8E FC 03        STX R80LNK
0014 8C FD 03        STY R80LNK+1
0017 4C 1B 80        JMP COMMON
              ;
              ; ON WARM RESET, PERFORM THE APPLICATION REQUIRED CODE
001A EA       WARM   NOP              ;APPLICATION DEPENDENT
              ;
              ; ON ANY RESET, SOME CODE IS COMMON TO WARM AND COLD
001B EA       COMMON NOP              ;APPLICATION DEPENDENT
001C 60              RTS              ;RETURN TO I/O ROM
              ;
              ; A MONITOR COMMAND CAN START FORTH WITH APPLICATION VOCABULARY
001D C9 39    DECODE CMP #'9          ;KEY TO DECODE
001F F0 03           BEQ FORINT       ;YES ==>
0021 6C F4 03        JMP (EX2MON)
              ;
              ; FORTH INITIALIZATION DRIVER
0024 20 93 D2 FORINT JSR INT4TH       ;DOWNLOAD RAM VARIABLES
0027 20 D1 D2        JSR FNDTOP       ;SET UP DEFAULT VALUES
002A A9 00           LDA #<LAST       ;INITIALIZE TASK LFA
002C 8D 05 08        STA TSKLFA
002F A9 00           LDA #>LAST
0031 8D 06 08        STA TSKLFA+1
0034 A2 92           LDX #$92         ;SET UP PARAMETER STACK
0036 4C 45 C5        JMP FORENT
              ;
              ; THIS IS WHERE THE FORTH APPLICATION CODE WILL BEGIN ...
0039 EA       FREE   NOP
                     .END

ERRORS=0000
```

Figure 10-2. Startup Driver with I/O and Monitor ROMs Installed presence of the Monitor ROMs in the cold reset path. If they are present, a pointer to a key-down check in the driver is added to the Monitor key decoding linkage (see Section 6.2.2 in the AIM 65/40 System User's Manual). Linkage is also shown for cold and warm reset processing in auto-start. The NOP instructions shown in the listing should be replaced with actual application dependent instructions.

Any time a key is pressed (when in the Monitor command level) the driver will check to see if it is the 9 key. If it is the 9 key, FORTH will be initialized at the command level with the application words linked to the dictionary.

10.2.2 With I/O ROM Installed

This driver, shown in Figure 10-3, is similar to the one listed in Figure 10-2 except that neither Monitor linkage nor key decoding is provided.

10.2.3. With I/O ROM Not Installed

This driver, illustrated in Figure 10-4, initializes FORTH, including linkage of the application words to the dictionary, then jumps to NEXT to start execution of the application program, i.e., of the last FORTH word compiled. No cold or warm reset interface with the I/O ROM is provided, therefore, the driver and/or the application program must provide all reset, interrupt and I/O handler functions.


```
PAGE 0001   FORTH STARTUP DRIVER FOR AIM 65/40 WITH I/O ROM ONLY
ADDR OBJECT   SOURCE
              ; PROGRAM EQUATES
              TSKLFA=$0805              ;LINK FIELD ADDRESS OF TASK
              INT4TH=$D293              ;FORTH VARIABLE INITIALIZATION
              FNDTOP=$D2D1              ;FIND FORTH VARIABLE DEFAULT VALUES
              LAST=$0000                ;POINTER TO LAST APPLICATION WORD (NFA
              START=$8000               ;APPLICATION PROGRAM START
              FORENT=$C545              ;ENTRY TO FORTH
              ;
                     *=START
0000 42              .BYT $42           ;AUTO-START PROGRAM ID
0001 5A              .BYT $5A,$A5       ;AUTO-START KEY PATTERN
0002 A5
0003 10 04           BPL WARM           ;WARM RESET ==>
              ;
              ; ON COLD RESET, PERFORM THE APPLICATION REQUIRED CODE
              ;
0005 EA       COLD   NOP                ;APPLICATION DEPENDENT
0006 4C 0A 80        JMP COMMON
              ;
              ; ON WARM RESET, PERFORM THE APPLICATION REQUIRED CODE
              ;
0009 EA       WARM   NOP                ;APPLICATION DEPENDENT
              ;
              ; ON ANY RESET, SOME CODE IS COMMON TO WARM AND COLD
              ; FORTH INITIALIZATION DRIVER
000A 20 93 D2 COMMON JSR INT4TH         ;DOWNLOAD RAM VARIABLES
000D 20 D1 D2        JSR FNDTOP         ;SET UP DEFAULT VALUES
0010 A9 00           LDA #<LAST         ;INITIALIZE TASK LFA
0012 8D 05 08        STA TSKLFA
0015 A9 00           LDA #>LAST
0017 8D 06 08        STA TSKLFA+1
001A A2 92           LDX #$92           ;SET UP PARAMETER STACK
001C 4C 45 C5        JMP FORENT         ;COME UP IN FORTH COMMAND MODE
              ;
              ; THIS IS WHERE THE FORTH APPLICATION CODE WILL BEGIN ..
001F EA       FREE   NOP
                     .END

ERRORS=0000
```

Figure 10-3. Startup Driver with I/O ROM Installed

```
PAGE 0001     FORTH STARTUP DRIVER FOR AIM 65/40 WITHOUT I/O ROM

ADDR OBJECT   SOURCE

              ; PROGRAM EQUATES
              TSKLFA=$0805              ;LINK FIELD ADDRESS OF TASK
              INT4TH=$D293              ;FORTH VARIABLE INITIALIZATION
              FNDTOP=$D2D1              ;FIND FORTH VARIABLE DEFAULT VALUES
              LAST=$0000                ;POINTER TO NFA OF LAST APPLICATION WORD
              LASTPF=$0000              ;POINTER TO PFA OF LAST APPLICATION WORD
              START=$8000               ;APPLICATION PROGRAM START
              NEXT=$C06F                ;ENTRY POINT TO EXECUTE NEXT FORTH WORD
              IP=$009F                  ;POINTER TO NEXT FORTH WORD
              ;
                     *=START
8000 20 93 D2 RESET  JSR INT4TH         ;DOWNLOAD RAM VARIABLES
8003 20 D1 D2        JSR FNDTOP         ;SET UP DEFAULT VALUES
8006 A9 00           LDA #<LAST         ;INITIALIZE TASK LFA
8008 8D 05 08        STA TSKLFA
800B A9 00           LDA #>LAST
800D 8D 06 08        STA TSKLFA+1
              ; SET UP THE WORD TO EXECUTE
8010 A9 00           LDA #<LASTPF
8012 85 9F           STA IP
8014 A9 00           LDA #>LASTPF
8016 85 A0           STA IP+1
8018 A2 92           LDX #$92           ;SET UP PARAMETER STACK
801A 4C 6F C0        JMP NEXT           ;COME UP RUNNING APPLICATION WORD ( LAST
                     .END

ERRORS=0000
```

Figure 10-4   Startup Driver Without I/O ROM Installed

SECTION 11

USING AN AUDIO CASSETTE RECORDER

An audio cassette recorder provides a low cost method of permanently saving programs written in FORTH as well as data for use during program execution. AIM 65/40 FORTH, in conjunction with the AIM 65/40 hardware and AIM 65/40 Monitor firmware, allows both source and object code to be saved and loaded using an audio cassette recorder.

If you want to save and load programs or data at separate times, connect only one recorder. Connect two recorders if you want to read from one recorder and write to another one. Refer to Section 9 in the AIM 65/40 System User's Manual for audio cassette recorder connection information and general operating procedure.

This section describes the procedures to use audio cassette recorders. AIM 65/40 FORTH also includes functions to interface with generalized mass storage. These functions are described in Section 12.

11.1 HANDLING PROGRAM SOURCE CODE FILES

11.1.1 Listing Program Source Code

There are several reasons to record the source code of a program written in FORTH:

- To have a permanent file of the source code for reading into the Text Editor for editing in case of accidental AIM 65/40 power turn-off or inadvertent overwrite of the source code in RAM.

- To compil from audio cas insufficient memory is availa le to have both d object code co-resident RAM.

To allow program updating and editing at a later date.

To transfer programs between the AIM 65/40 FORTH and AIM 65 FORTH.

The procedure to record source code from the Text Editor is:

NOTE

Recorder remote control must be used if extended FORTH execution is performed between blocks.

a. Position the tape and set up the recorder remote control commands as described in Section 9.3 of the AIM 65/40 System User's Manual.

b. Enter the Text Editor and read the source code into the Text Editor as described in Section 4.5.

If the file to be recorded is to be compiled from audio tape and contains the complete program, include as the last word in the Text Buffer:

```
FINIS
```

d. Position the Line Pointer to the first line of source code to be recorded.

e. List the source code to the recorder using the L (List) command.

```
={L}/. OUT=T UNIT=<recorder no.>
FILE=<filename> <RETURN>  XX W

*END* XX W
```


NOTE

Be sure to manually start the recorder before pressing <RETURN>, if manual recorder control is used.

11.1.2 Reading Program Source Code

The source code of a program written in FORTH can be read int the Text Editor using the AIM 65/40 Monitor E command (Enter Text Editor) or AIM 65/40 Text Editor R command (Read Lines Into Text Editor). Follow the procedures described in Sectio 5.2.1 and 5.4.1 in the AIM 65/40 System User's Manual, respectively. Also refer to Section 9.3 of that manual for instructions on how to use the audio tape recorder.

11.1.3 Compiling Program Source Code

The source code can be compiled from an audio cassette recorde using the word SOURCE similar to the procedure described in Section 4.5 The procedure is:

a. Previously record the source code on an audio cassette file using the AIM 65/40 Text Editor L (List) command.

b. If recorder remote control is used, turn the recorder off using the AIM 65/40 Monitor 1 or 2 command.

c. Enter or re-enter FORTH.

d. Compile using the word SOURCE as follows:

```
SOURCE <RETURN> IN=T
UNIT=<recorder no.> FILE=<filename><RETURN>
```

After the compilation is finished, control returns to the FORTH command level.

NOTES

(1) If a tape read error occurs, an error message is displayed and control is returned to the AIM 65/40 Monitor.

(2) If the word FINIS is not read at the end of a file, FORTH remains in the read mode and will not return control to the keyboard. In this case, press the RESET button to gain AIM 65/40 Monitor control then re-enter FORTH. Run a VLIST to see which words were compiled. If only the word FINIS was missing, all the words should have compiled.

11.2 HANDLING PROGRAM OBJECT CODE FILES

Loading a program written in FORTH in object code form is often desirable since the files are shorter and compilation is not required. While AIM 65/40 FORTH does not have words to save and load object code, the AIM 65/40 Monitor does. Using the AIM 65/40 Monitor dump and load functions, the AIM 65/40 FORTH variables and dictionary object code can be saved and retrieved to allow a program entry capability.

Note that FORTH <u>object</u> code is <u>not</u> compatible between the AIM 65 and AIM 65/40 microcomputers. The source program FORTH words are compatible, however, except as described in Appendix L.

11.2.1 Dumping Program Object Code

Perform the following steps to dump the dictionary object code, linkage and re-entry variables:

a. Validate operation of the FORTH program in FORTH.

b. Use HERE to find the last memory location used by the dictionary:

```
HEX HERE . <RETURN> LAST
```

where LAST is the value to be used in step d.

c. Escape to the AIM 65/40 Monitor.

d. Dump the FORTH variables and program object code to the audio cassette recorder using the Monitor D (Dump) command.

```
{D}
FROM=0700 TO=077F OFFSET=0000 MORE?Y
FROM=0800 TO 0878 OFFSET=0000 MORE?N
TYPE=<A,B> OUT=T
UNIT=<recorder no.> FILE=<filename><RETURN>
```

Where

```
700 = start of the FORTH variables.
77F = end of the FORTH variables.
800 = start of the program dictionary.
878 = end of the program dictionary
      (from step b, typical value shown).
```

Also, be sure to save any user variables that have been altered.

11.2.2 Loading Program Object Code

Perform the following steps to loa[illegible]t code which has been dumped according to the procedure i[illegible]ion 11.2.1.

a. If in FORTH, perform a cold start using COLD to initialize FORTH, then escape to the AIM 65/40 Monitor

If in the Monitor, first type 5 to enter FORTH and to initialize FORTH variables, then escape back to the Monitor.

Read the recorded FORTH variables and object code from the audio cassette recording using the Monitor L (Load) command.

```
{L} OFFSET=0000 IN=T
UNIT=<recorder no.> FILE=<filename><RETURN>
```

c Type 6 to re-enter FORTH.

d Run a VLIST to verify that the application words are in the dictionary.

11.3 HANDLING DATA FILES

Data files can be written to and read from an audio cassette recorder in several different ways. These include:

Dumping and loading in AIM 65/40 FORTH format using FORTH READ and WRITE words (see Appendix B).

Dumping and loading in AIM 65/40 Monitor Format (see Sections 4.8 and Appendix H of the AIM 65/40 System User's Manual).

Dumping and loading in FORTH screen format (see Section 12).

The program listed in Figure 11-1 contains several words that can be used to toggle or turn on/off the recorder remote control lines, dump and load data files in AIM 65/40 FORTH format under keyboard or program control, and to dump and load data files in AIM 65/40 Monitor format under keyboard control.


```
(                                       )
( AIM 65/40-FORTH AUDIO TAPE DRIVERS )
(                                       )

HEX FFB0 CONSTANT SYSORB

0 VARIABLE DRIVENO
0 VARIABLE NAME 4 ALLOT

: SYSBC@ SYSORB DUP C@ ;
: ON ." ON" ;
: OFF ." OFF" ;

(   TAPE RECORDER CONTROL ROUTINES
(
(  ---    TURN RECORDER 1 ON )
: T1-ON SYSBC@ EF AND SWAP C! ;

(  ---    TURN RECORDER 1 OFF )
: T1-OFF SYSBC@ 10 OR  SWAP C! ;

(  ---    TURN RECORDER 2 ON )
: T2-ON SYSBC@ DF AND SWAP C! ;

(  ---    TURN RECORDER 2 OFF )
: T2-OFF SYSBC@ 20 OR  SWAP C! ;

(  ---    TURN RECORDERS 1 & 2 OFF )
: T-OFF SYSBC@ 30 OR  SWAP C! ;

(  ---    TOGGLE RECORDER 1 CONTROL
: T1 SYSBC@ 10 XOR 2DUP SWAP C!
 10 AND IF OFF ELSE ON  THEN DROP ;

(  ---    TOGGLE RECORDER 2 CONTROL
: T2 SYSBC@ 20 XOR 2DUP SWAP C!
 20 AND IF OFF ELSE ON  THEN DROP ;
```

Figure 11-1. AIM 65/40 Audio Tape Handling Words

```
(    LOAD AND DUMP DATA ONTO TAPES
(
(  ---     SET OUTPUT DEVICE = T )
CODE SETOUT XSAVE STX, AEB9 JSR,
 ( WHEREO) XSAVE LDX, NEXT JMP, END-CODE

(  ---     SET INPUT DEVICE = T )
CODE SETIN XSAVE STX, AE9B JSR,
 ( WHEREI) XSAVE LDX, NEXT JMP, END-CODE

( addr no ---    DUMP IN AIM 65/40 FORTH)
: FDUMP CR SETOUT WRITE CLOSE ;

( addr no ---    LOAD IN AIM 65/40 FORTH)
: FLOAD CR SETIN READ CLOSE ;

(  ---    DUMP DIRECTLY FROM THE MONITOR)
CODE TDUMP XSAVE STX, A976 JSR,
 ( DUMP ) XSAVE LDX, NEXT JMP, END-CODE

(  ---    LOAD DIRECTLY FROM THE MONITOR)
CODE TLOAD XSAVE STX, ABB7 JSR,
 ( LOAD ) XSAVE LDX, NEXT JMP, END-CODE

FINIS
 *END*
```

```
VLIST
  99C TLOAD             988 TDUMP
  974 FLOAD             960 FDUMP
  94C SETIN             938 SETOUT
  90B T2                8E2 T1
  8CE T-OFF             8B7 T2-OFF
  89F T2-ON             888 T1-OFF
  870 T1-ON             85E OFF
  84F ON                840 SYSBC@
  82F NAME              824 DRIVENO
  816 SYSORB            809 TASK
```

Figure 11- AIM 65/40 Audio Tape Handling Words (Cont'd)

These words

T1	Toggle Recorder No. 1 On/Off
T2	Toggle Recorder No. 2 On/Off
T1-ON	Turn Recorder No. 1 On
T1-OFF	Turn Recorder No. 1 Off
T2-ON	Turn Recorder No. 2 On
T2-OFF	Turn Recorder No. 2 off
T-OFF	Turn Both Recorders Off
FDUMP	Dump in AIM 65/40 FORTH Format
FLOAD	Load in AIM 65/40 FORTH Format
TDUMP	Dump in AIM 65/40 Monitor Format
TLOAD	Load in AIM 65/40 Monitor Format

Other words are also included that are used by the above words

Load the program into the AIM 65/40 Text Editor and compile it as shown. Run a VLIST to verify compilation was completed and that the word locations are the same as listed. Note that this is an example program that can be used as is, or as a base for your own words. The flexibility of FORTH allows these words to be altered or other words to be defined easily to meet your specific application requirements.

Note that the word CLOSE is used in the program to return control to the keyboard upon completion of tape file read.

11.3.1 Using Recorder Remote Control

The recorder remote control words can be used er keyboard or program control.

(1) To turn a recorder on, use T1-ON or T2-ON .

(2) To turn a recorder off, use T1-OFF or T2-OFF .

(3) To turn both recorders off, use T-OFF .

(4) To toggle a recorder control line, use T1 or T2 .

11.3.2 Using AIM 65/40 FORTH Format

AIM 65/40 FORTH provides a data file format consisting of all data bytes as opposed to the AIM 65/40 Monitor ASCII format which includes other information in multiple records. Each AIM 65/40 Monitor record also includes the number of bytes, the starting address, and a checksum. Since the FORTH data file format contains only data, more data can be stored in a smaller space resulting in faster recording and reading. Since the data file does not include addresses, the recorded data can easily be loaded where needed in memory without skipping or processing the address information.

a. **Dumping a Data File Using FDUMP**

(1) Establish an output data buffer in RAM.

(2) Store the output data in the output data buffer.

(3) Set up the recorder for recording using the desired recorder remote control word.

(4) Enter or load the starting address and the number of bytes to record and initiate the dump.

<starting address> <no. of bytes> FDUMP <RETURN>

(5) Enter the output device code (in this example, the audio tape device code = T).

OUT=T

(6) Enter the recorder no.

UNIT= <1 or 2>

(7) Enter the file name (all 5 characters):

FILE= <filename> <RETURN>

The recorder will be started automatically.

(8) OK will be displayed upon dump completion.

For example, dump 20 bytes from locations $1000 through $101F to recorder no. 1 as a file named QWERT . Use the FILL and DUMP words to initialize and check the RAM contents for test purposes.

```
1000  20  FDUMP  <RETURN>
 OUT=T  UNIT=1  FILE=QWERTOK
OK
```

b. **Loading a Data File Using FLOAD**

(1) Set up the recorder for reading. Use the recorder remote control words as desired.

(2) Enter or load the starting address and the number of bytes to record and initiate the load.

<starting address> <no. of bytes> FLOAD <RETURN>

(3) Enter the input device code (in this example, the audio tape device code = T).

IN=T

(4) Enter the recorder no.

UNIT= <1 or 2>

(5) Enter the name of the file as recorded.

FILE=<filename> <RETURN>

The recorder will be started automatically.

(6) OK will be displayed upon load completion.

Try the following example. Again, use the FILL and DUMP commands to initialize and check the RAM contents before and after loading.

```
1000  20  FLOAD <RETURN>
 IN=T UNIT=1 FILE=QUERT <RETURN>
QUERT  00  R
OK
```

11.3. Using AIM 65/40 Monitor Format

It is sometimes desirable to dump and load a data file in FORTH that is compatible with the AIM 65/40 Monitor ASCII or binary format (described in Appendix H of the AIM 65/40 System User's Manual). Data files recorded in FORTH in this manner can then be read by the AIM 65/40 Monitor L (Load) command and data files recorded by the AIM 65/40 Monitor D (Dump) command can be read under FORTH control.

a. Dumping a Data File Using TDUMP

(1) Type TDUMP:

```
TDUMP <RETURN>
```

(2) Respond to prompts:

```
FROM= <address> TO= <address> OFFSET=<address>
  MORE?<Y,N>
TYPE=<A,B> OUT=<T> UNIT=<1 or 2> FILE=<filename>
```

(3) A block count will be displayed during recording and OK will be displayed upon dump completion.

For example:

```
TDUMP <RETURN>
FROM=1000 TO=1200 OFFSET=0000 MORE?N
TYPE=A OUT=T UNIT=1 FILE=DAT22<RETURN> 08 W
DONEOK
```

b. Loading a Data File Using TLOAD

(1) Type TLOAD:

```
TLOAD <RETURN>
```

(2) Respond to prompts:

```
OFFSET=<address> IN=<T> UNIT=<1 or 2>
  FILE=<filename> <RETURN>
```


(3) OK will be displayed upon load completion

For example:

```
TLOAD <RETURN>
OFFSET=0000 IN=T UNIT=1 FILE=DAT22 <RETURN>
DAT22 08 R
DONEOK
```

SECTION 1

INTERFACING TO MASS STORAGE

12.1 OVERVIEW

AIM 65/40 FORTH includes all of the fundamental words needed to interface with, and effectively use, mass storage devices. This chapter provides directions and guidelines on how to interface to a floppy disk, however, the procedure may be easily modified to include other peripherals.

Before you begin, you must have a mass storage device in correct functioning order. You must know how to get data to and from the device's controller, and what data the controller needs to function correctly. Finally, you must have enough memory in the AIM 65/40 microcomputer to hold a FORTH screen. The minimum RAM requirement is 2K bytes, but a practical minimum is 16K bytes. If you have more than 16K bytes RAM available, then so much the better.

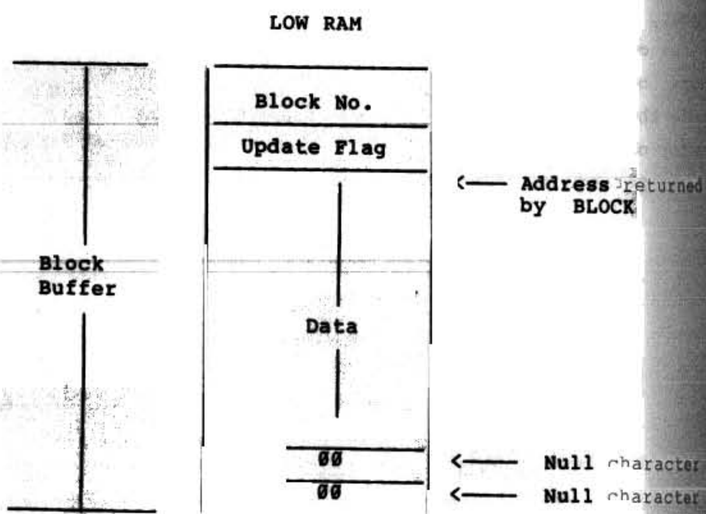
12.1.1 Mass Storage Terminology

FORTH accesses mass storage in uniformly-sized pieces called blocks, and keeps data, or source code, in RAM in 1024-byte pieces called screens. If the block is 1024 bytes, then the terms 'block' or 'screen' are often used interchangeably. Since these block sizes are commonly the size of a floppy disk sector of 128 or 256 bytes, there are normally eight or four blocks per screen, respectively.

Block Buffer

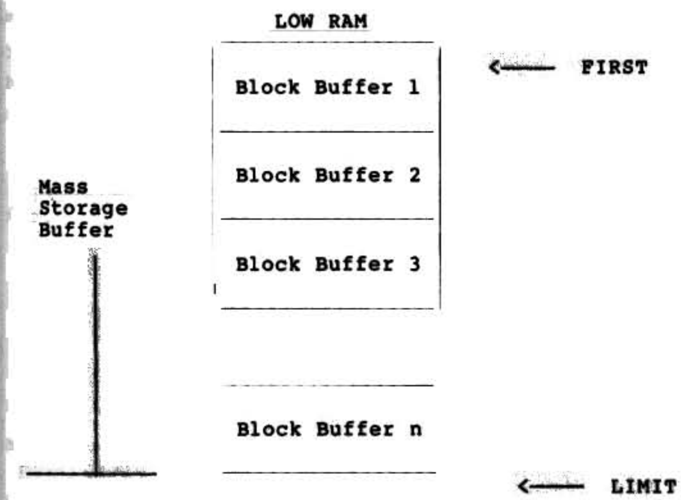
A particular block is referenced by the FORTH word BLOCK which takes the block number as the argument. If the block of data is in RAM, BLOCK returns immediately with the address of the buffer where the data is to be found. If the block is not in RAM, BLOCK uses R/W (described below) to fetch it from mass storage and put it in a buffer in RAM, then returns the address of that buffer. BLOCK also checks to see if the data in a particular buffer needs to be written out to mass storage before it uses the buffer for new data.

Each block buffer in RAM is four bytes larger than the mass storage block size. Two of these extra bytes are at the end of the buffer and both contain ASCII null characters ($ØØ) to mark the end of data. The other two bytes, located at the start of the buffer, contain the block number (byte 1) and a one-bit flag (byte 2) that indicates whether or not the buffer contains data that must be written to mass storage before the buffer can be used for new data. The layout of a block buffer is:



b. **Data Buffer**

The RAM area reserved for use by mass storage, commonly called the data buffer, or the mass storage buffer, must contain <u>more than two</u> of the block buffers described above. The first byte of the entire mass storage buffer area is referenced by the word FIRST and is stored in the variable UFIRST . The last byte of the entire buffer area is located at LIMIT -1 and the value returned by the word LIMIT is kept in ULIMIT . The layout of the buffer area is:



c. **Screen Size**

Conventionally, when a screen of source code is listed on a CRT display, it appears as 16 lines of 64 characters each. The lines are numbered Ø to 15 on the left of the text. If the display will not permit 67 or 68 characters on a line, other formats can be adopted.

12.1.2 Buffer Variables

The size, number and location of the block and data buffers in AIM 65/40 FORTH is controlled by four user variables (UB/BUF , UB/SCR , UFIRST , and ULIMIT). The logic of which one to use at any given time is controlled by three other variables (PREV , USE , and OFFSET). The names, description and access words for these variables are given in Table 12-1.

12.2 SETTING UP BLOCK AND DATA BUFFERS

R/W is the primary word that interfaces FORTH to mass storage. All of the FORTH logic which automatically handles the locating, reading and writing of mass storage data ultimately winds up using R/W . However, before R/W can work properly, it must have a set of data buffers to use. As explained earlier, AIM 65/40 FORTH needs more than two buffers in order for the buffer rotation logic to work correctly.

The general steps in the process of setting up the block and data buffers is a simple procedure as summarized below; the details are given in the following section.

1. Set the top (high RAM) of the data buffer area into ULIMIT .
2. Set the size of the block buffers into UB/BUF .
3. Compute and set the number of block buffers per screen into UB/SCR .
4. Compute and set the start of the data buffer area into UFIRST .
5. Set USE and PREV to FIRST .
6. Clear the data buffer.
7. Initialize the block offset value.

Table 12-1 Buffer Variables and Access Words

Variable	Access Word	Default Value	Description
UB/BUF	B/BUF	128	Holds the number of bytes of data in each block buffer. This is often a power of two in the range of 128 to 1024. The actual buffer size is four bytes larger than this value.
UB/SCR	B/SCR	8	Holds the number of block buffers per FORTH screen. Typical values are 1,2,4 or 8.
UFIRST	FIRST	*	Holds the location of the start of the data buffer.
ULIMIT	LIMIT	*	Holds the location of the end of the data buffer plus one.
PREV	none**	*	Holds the address of the block buffer most recently referenced.
USE	none**	*	Holds the address of the block buffer to use next.
OFFSET	none**	none	Holds a value to be added to the block number given to BLOCK .

NOTES

*On a cold start, AIM 65/40 FORTH V1.4 sets these variables to the top of 16K of RAM plus one byte ($4000 = 16384).

**Variables without a special access word to fetch the value are treated like any other FORTH variable. Use @ to fetch the data from its address and ! to put data into its address. Any variable can be accessed in this manner; the special access words are only for convenience.

In many cases, steps 1, 2 and 3 can be omitted. The default value of the top of RAM for ULIMIT is a good choice, unless special circumstances dictate that another value should be used. The default values for UB/BUF and UB/SCR are popular values for floppy disk systems, but may have to be changed, especially for magnetic tape, or fixed disk, mass storage.

In steps 3 and 4, it is convenient to use FORTH to compute the actual value to store. Step 5 provides the starting values for the buffer pointers, and step 6 sets up the buffers to use and clears them of prior data.

Step 7 is necessary as FORTH adds this offset value to each block number requested via R/W . The utility of OFFSET is in setting it to the first block number in an extra mass storage device. Then the block numbers of media inserted in that device will be the same to the user as when OFFSET is zero and the media is in the primary device.

12.3 CREATING SCREENS

This section illustrates the creation and testing of two different buffer arrangements. Assume you have an AIM 65/40 microcomputer with 16K bytes of RAM connected to a disk with a typical sector size.

12.3.1 Creating and Testing a One Screen Buffer

Source Code Entry

Enter the following source code into the AIM 65/40 Text Editor:

```
( SETTING UP DISK BUFFERS)
HEX
FORGET TASK ( FDC RAM $4A0-$55F)
1 CONSTANT S# ( ONLY 1 SCREEN)
80 UB/BUF ! ( 128 FOR SINGLE DENSITY)
8 UB/SCR ! ( 8 BLOCKS ARE A SCREEN)
LIMIT B/BUF 4 + B/SCR *
S# * - UFIRST ! ( BUFFERS AT TOP OF RAM)
0 OFFSET ! ( NOT NEEDED RIGHT NOW)
FIRST USE ! ( SET UP FIRST POINTER)
```



```
FIRST PREV ! ( AND ALSO PREV POINTER)
EMPTY-BUFFERS ( CLEAN OUT THE BUFFERS)
: ... CR . . . ; ( R/W WORD FOR DEMO)
' ... CFA UR/W ! ( SET UP R/W WORD POINTER)
CR ." DONE " ( FINISHED)
FINIS
```

Line 1 sets the input/output base to hexadecimal, as it is much easier to visualize memory layouts in this format. Line 2 shows use of the dummy word TASK to avoid piling up definitions. The constant S# in line 3 is the number of FORTH screen's worth of buffers desired.

Lines 4 and 5 depend on the size (5 1/4-inch mini-floppy or 8-inch standard floppy) and recording density (single- or double-density) of the disk drive. The values shown (80,8) are for a single-density mini-floppy, which has 128 bytes per sector. These values are the same for a single-density standard floppy. For a double-density mini-floppy or a double-density standard floppy, there are 256 bytes per sector, so there are $100 bytes per buffer and four buffers per screen.

Line 6 begins the actual calculation by putting LIMIT and B/BUF on the stack, adding four to allow for the extra four bytes in each block buffer, then multiplies this true block buffer size by the number of buffers in a screen. Line 7 then multiplies this by the number of screens desired. With two numbers on the stack, LIMIT and the computed size of the entire disk buffer area, a subtraction leaves the bottom of the RAM buffer, which is placed in the variable UFIRST .

Lines 8, 9 and 10 set the remaining buffer variables, and in line 11 EMPTY-BUFFERS completes the buffer generation by clearing all the RAM buffers to $00.

The code-definition word ... (dot-dot-dot) on line 12 is designed to test the FORTH mass storage processing and buffer use. Note that aside from the constant S# , this is the only new word defined and that all the other words are interpreted and executed as encountered. The word ... simply does a carriage return then prints the top three items on the stack. The function of ... is to print the three parameters that are supplied to R/W , to allow viewing of the overall operation of buffer selection and use. Line 13 shows how to install ... into the mass storage vector of UR/W . The FORTH word ' (tic) fetches the parameter field of the word following it (i.e., ...), CFA changes that address to the code field address, and the phrase UR/W ! stores it in UR/W . Line 12 and 13 are only shown to test the buffer operation without a mass storage device. For proper operation, these lines would be replaced with the mass storage interface words (see Section 12.(

The mass storage system is now ready to visually test for correct operation using parameters appropriate for a disk. Line 14 is only to show that the buffer creation step is done.

b. Interpretation and Operation

The interpretation of the buffer creation code is done using SOURCE with M specified for the location of source code. 'DONE' is displayed when the buffers are ready to use.

```
{5}
 AIM 65/40 FORTH V1.4
SOURCE <RETURN> IN=M
DONE
OK
```

The following example shows the operation of the test word ... with the words LOAD and UPDATE . First, load two screens.

```
10 LOAD <RETURN> ( Read screen 10 into the buffer)
1 80 3BE2
1 81 3C66
1 82 3CEA
1 83 3D6E
1 84 3DF2
1 85 3E76
1 86 3EFA
1 87 3F7E OK
20 LOAD <RETURN> ( Read screen 20, overwritting this buffer)
1 100 3BE2
1 101 3C66
1 102 3CEA
1 103 3D6E
1 104 3DF2
1 105 3E76
1 106 3EFA
1 107 3F7E OK
```


Using LOAD to test the buffer operation is convenient as it transfers an entire screen, allowing the overall operation of the buffer selection logic to be observed. Each line printed by ... shows the read/write flag (1), the block number (80, 81, etc.) and the buffer address. Remember that the screen number gets multiplied by B/SCR and that buffers are four bytes larger than B/BUF .

The word UPDATE sets the update bit in the buffer that PREV is pointing to and indicates that the buffer must be written out to mass storage before being over-written. This operation is seen in the last two lines. It is the responsibility of the user to use UPDATE in any word that modifies the contents of a disk buffer. If a disk buffer is not marked as updated when it has new data, it will be over-written and the data will be lost.

```
UPDATE 30 LOAD <RETURN>
1 180 3BE2
1 181 3C66
1 182 3CEA
1 183 3D6E
1 184 3DF2
1 185 3E76
1 186 3EFA
0 107 3F7E    ( Write out updated block)
1 187 3F7E OK ( Read in new block)
```

12.3.2 Creation and Testing a Two Screen Buffer

a. Source Code Entry

Change the value of S# to 2 in line 3 of the source code in the Text Buffer and recreate the disk buffers by compiling the code again.

```
{T]
={F}1 CON
1 CONSTANT S#
={C} OLD=1  <RETURN> NEW=2 <RETURN> /<RETURN>1
1 CONSTANT S#
2 CONSTANT S#

80 UB/BUF
={O
```

b Interpretation and Operation

After modifying the source code in the Text Editor, the buffer operation can again be simulated by first entering FORTH and compiling as follows:

```
{5}
 AIM 65/ ] FORTH V1.4
SOURCE < ETURN> IN=M
DONE
```

Now, testing using LOAD shows that two screens may be loaded before a buffer is over-written.

```
10 LOAD <RETURN> ( Read screen 10 into the buffer)
1 80 37C2
1 81 3846
1 82 38CA
1 83 394E
1 84 39D2
1 85 3A56
1 86 3ADA
1 87 3B5E OK
20 LOAD <RETURN> ( Read screen 20 into the buffer)
1 100 3BE2
1 101 3C66
1 102 3CEA
1 103 3D6E
1 104 3DF2
1 105 3E76
1 106 3EFA
1 107 3F7E OK
30 LOAD <RETURN> ( Read screen 30, overwritting the buffer)
1 180 37C2
1 181 3846
1 182 38CA
1 183 394E
1 184 39D2
1 185 3A56
1 186 3ADA
1 187 3B5E OK
```


The chain of events that results in a particular block of data being transferred to, or from, mass storage begins with a high level word, such as LOAD for programs, or BLOCK for data. When the user, or a word, executes LOAD or BLOCK , the internal logic decides if the data is in RAM, and if not, which buffer to use and whether or not a write is necessary. This information, along with the block number, is passed along in one or more calls to R/W .

The next step is handled by the user written interface words which translate the FORTH parameters supplied to R/W into parameters acceptable to the mass storage device. This interface then executes programs that do the actual work of reading or writing data. These programs may be part of the interface itself, or may be located in firmware supplied with the device.

Finally the status of the operation is returned, and control returns to the word following the original high level FORTH word that began the transfer. The status resulting from the transfer may, or may not, be acted upon, at the option of interface program.

The exact interface methods depend on the hardware, but a few points are applicable to most devices. Try to do most of the work in FORTH until you get to the point where you must call a subroutine in the driver firmware, or where you need the speed of machine code. Use the FORTH assembler for these final tasks, then take advantage of FORTH's parameter stack to pass control and sense information between the interface words.

For magnetic disk mass storage devices, you usually have to calculate the track and sector, then place these values where the device driver expects to find them. The act of reading or writing is often a matter of calling the appropriate sub-routines. For magnetic tape you may have to also keep track of the current location on the tape to know which way the tape must be positioned in order to access the desired block.

For example, consider the floppy disk system example in Figure 12-1. This system is compatible with the RM 65 Floppy Disk Controller (FDC) module (RM65-5101) with program PROM R32N5 dated 1/4/82 installed. This system uses the locations $D7 to $DC and $4A0 to $55F to keep the buffer pointers and variable data. Seeking, reading and writing are done by calling subroutines in the disk driver firmware located in memory space starting at $8000, with parameters passed in the A, X, and Y registers. Data is passed in the RAM buffers pointed to by RDBUF and WRTBUF , each which return a status byte in the A register. The interface driver's task is to take the data given to R/W , compute the track and sector, load the buffer address into the buffer pointers, and call the subroutine with the proper parameters in the registers. This system assumes a standard double-density mini-floppy, with 35 tracks and 16 sectors per track.

The entire disk interface is included in the program listed in Figure 12-1. As typical of FORTH, each word performs a simple function, with each new word building on previous ones. The first nine lines set up the mass storage buffers for use as previously described in Section 12.3. There is one screen buffer created with four blocks of 256 bytes each.

INIT initializes the module by setting up the interrupt vector and then calling the code INIT1 , which in turn calls the RM 65 FDC module initialization routine, then turns on the desired drive. SIZEOK checks that the block number is valid, i.e., less than the number of sectors times the number of tracks. BBUF stores the RAM block buffer address into the Read and Write buffer pointers for the FDC module. T&S takes the block number from the stack, and leaves a track and a sector number.

The next three words are the basic primitives that allow any disk sector to be read or written. The code word SEEK moves the disk head to the track number on the stack and leaves a non-zero status byte for an error condition. The DREAD and DWRITE words read or write between the RAM block buffer and the disk sector left on the stack, returning a non-zero status bit for error conditions.


```
( AIM 65/40 -- FORTH DOUBLE-DENSITY DISK ROUTINES)
HEX FORGET TASK ( FDC RAM $4A0-$55F)
1 CONSTANT S# ( ONLY 1 SCREEN NEEDED )
100 UB/BUF ! ( 256 FOR DOUBLE DENSITY )
4 UB/SCR ! ( 4 FOR DOUBLE DENSITY )
LIMIT B/BUF 4 + B/SCR * S# * - UFIRST ! ( TOP OF RAM )
0 OFFSET ! ( NOT NEEDED WITH ONLY 1 DRIVE )
FIRST DUP USE ! PREV ! ( SET UP FIRST BUFFER )
EMPTY-BUFFERS ( CLEAR OUT THE BUFFER AREA )
( INITIALIZE FDC & TURN ON DRIVE NO. 1)
CODE INIT1 XSAVE STX, 8000 JSR, ( CALL INIT )
 ( SET DRIVE PARAMETERS IN SRCDRV, SRCSID, & SRCDEN )
 0 # LDA, 4A5 STA, ( DRIVE ONE INTO SRCDRV )
 0 # LDY, 4AF STY, ( SIDE ONE INTO SRCSID )
 0 # LDX,  4B1 STX, ( DOUBLE DENSITY INTO SRCDEN )
 83E0 JSR, ( MOTON ) XSAVE LDX, NEXT JMP, END-CODE
( --- INITIALIZE FDC & AIM 65/40, TURN ON DRIVE NO. 1)
: INIT FD46 4FB ! ( UIRQBM > IRQOUT)
  83C9 22B ! ( SET UP IRQHAN) INIT1;
: SIZEOK OVER 230 < ; ( 16 SECTOR * 35 TRACK )
: BBUF DUP 4F1 ( RDBUF ) ! 4F3 ( WRTBUF ) ! ;
: T&S SWAP 10 /MOD ; ( LEAVE TRACK & SECTOR )
( SOME OF THE FDC PRIMITIVES)
CODE SEEK XSAVE STX, TOP LDA, 8104 JSR, ( CALL SEEK )
 XSAVE LDX, 99 # AND, PUSH0A JMP, END-CODE
CODE DREAD XSAVE STX, TOP LDA, 84BF JSR, ( CALL RDSEC )
 XSAVE LDX, BD # AND, PUSH0A JMP, END-CODE
CODE DWRITE XSAVE STX, TOP LDA, 8500 JSR, ( CALL WRTSEC )
 XSAVE LDX, FD # AND, PUSH0A JMP, END-CODE
( MUST DISABLE ALL OTHER INTERRUPTS FOR PRIMITIVES)
: INTDIS FF FF80 C! ; ( MASK OUT ALL BUT RM 65 IRQ)
: INTENB 00 FF80 C! ; ( RESTORE THE IRQ MASK)
: DERROR ( RESTORE IRQ MASK & PRINT ERROR)
  INTENB CR . " DISK ERROR - " ;
: DATA ( FETCH A BYTE ) ROT BBUF T&S SEEK DUP
  IF DERROR    ." SEEK A="    ( SEEK ERROR )  INTDIS ELSE DROP
   THEN DROP 1+ (START WITH SECTOR 1 ) SWAP
  IF DREAD DUP   IF DERROR ." READ A=" .  ( READ ERROR )
     ELSE DROP  THEN ( DO NOTHING )
   ELSE DWRITE DUP  IF DERROR ." WRITE A=" .  ( ERROR )
     ELSE DROP   THEN ( DO NOTHING )   THEN DROP ;
: DISK SIZEOK   IF INTDIS DATA INTENB
   ELSE CR ." BLOCK TOO LARGE ERROR " ABORT    THEN ;
' DISK CFA UR/W ! ( STORE INTERFACE WORD )
CODE FORMAT XSAVE STX, 8095 JSR, ( CALL FORMAT )
 XSAVE LDX, NEXT JMP, END-CODE
: FORMAT INTDIS FORMAT INTENB ;
: TASK ; ( THROUGH WITH CODE ) FINIS
```

Figure 12-1 RM 65 FDC Module Disk System

The FDC primitives (specifically DREAD & DWRITE) are very time critical because they must be synchronized with the diskette movement and must not be interrupted, or else an error may occur. Since the AIM 65/40 microcomputer uses interrupts also, all other interrupt sources except for the FDC module must be disabled while the primitives are being performed. The word INTDIS masks out all IRQ sources except for the RM 65 bus. INTENB restores the IRQ Priority Latch to the cold reset value ($00) with no interrupts masked. The word DERROR restores the interrupt mask and prints out an error message.

The word DATA , when given a block number, read/write status, and a RAM block buffer address on the stack, performs the disk transfer, or returns with an error indication if an error is detected. First, the disk buffers are setup and the block number is converted to a track and a sector. A seek to the track is performed, with a disk seek error shown if the seek is not successful. This is followed by a read or a write, depending on the status from the stack. For a read, the requested disk sector is transferred into the RAM block buffer, with a disk read error shown for errors. For a write, the RAM block buffer is transferred into the requested disk sector, with a disk write error shown for errors.

The word DISK integrates all of the disk management structure to be used by the FORTH mass storage device. DISK first does a size check; if the block is out of range, a disk error occurs and the operation terminates. Otherwise, interrupts are disabled, the disk access is performed, then the interrupts are restored (under an error condition they will already be restored). If more than one drive, or disk side, is available, DISK can be modified to select the appropriate disk and side. This word can also turn on and off the drive motor, but for this example, the motor continues to run. Finally, the disk interface word DISK is stored into the mass storage read/ write vector UR/W

The FORMAT word prepares new disks for use with the FORTH disk. This word formats the selected disk (selected by INIT) by writing track and sector identifiers for every sector of the disk, and filling each sector with byte $E5 pattern. Note that interrupts should be disabled during FORMAT.

12.5 USING MASS STORAGE

The simplicity of the disk interface and FORTH's ability to customize to a particular application allows mass storage devices to be easily used in powerful ways. Two such ways are described in this section. Remember all mass storage operations must use diskettes that are first formatted with either FORMAT or a similar word.

12.5.1 Data Storage and Retrieval - the Virtual RAM

Data storage and retrieval using a mass storage device is quite simple. Just think of the data as an array of numbers, and, given the element number of a data item in the array, compute the required block number and offset into that block. Knowing the block number, all that is left to do is to access the block and add the offset to the address returned by BLOCK .

Suppose you want to process an array of 250 16-bit numbers and you wish the data to start at block 25. If the disk uses 256-byte blocks, a word that would supply the RAM address of a given array element number (0-249) would look like:

```
: DATA 128 /MOD 25 + BLOCK SWAP 2 * + ;
```

The address produced by DATA can then be used like any other variable address. The normal FORTH words @ and ! would then fetch and store data as if it were always in RAM. One extra modification would be appropriate here -- the word ! should automatically indicate that data was put into a disk buffer so that the buffer will be written out automatically. This is easily done by redefining ! and @ as U! and U@ :

```
( value index ---
: U! DATA ! UPDATE ;
( index --- value )
: U@ DATA @ ;
```

The actual use of DA is shown by a couple of examples. To print the 153rd numbe simply type:

```
153 U@ .
```

To clear out the enti array use:

```
CLEAR 250 0 DO 0 I U! LOOP FLUSH ;
```

(The word FLUSH at the end writes all updated buffers out to the mass storage device.)

12.5.2 Program Loading and Overlays

Once a screen has been written with a FORTH program, it is necessary to compile the program into the dictionary. This is done with LOAD , which takes the screen number from the stack and begins compiling that screen, starting at line 0 and continues until a ;S is encountered. The ;S terminator may be placed at any position, and any number of ;S words may appear on a screen, but FORTH will always stop compiling at the first ;S encountered.

Programs of more than one screen may be compiled but only if the screens are contiguous. Each screen, except for the last, must end with the word --> , and the last screen must be terminated with ;S . Compilation starts with a LOAD of the first screen in the sequence.

With a disk connected to an AIM 65/40 microcomputer with 16K-bytes of RAM, you can run quite large programs in FORTH by dividing the program into convenient-sized pieces and using program overlays. The techniques for using program overlays are -- like the disk data storage -- quite straightforward. Use the FORTH words FORGET and LOAD to overlay programs.

Suppose you have a program that consists of three parts: input, processing and output. If these three parts do not need to be resident in RAM all at the same time, they can be loaded and run sequentially.

First, construct what is called a load screen, which contains the directions for loading and executing the entire program. Suppose the source code of the input part of the program is in screens 12, 13 and 14, the source code for the processing part is in screens 30, 31 and 32, and the output source code is in screens 33 to 35. Further suppose that some data manipulating words are in screen 102 and 103, and that these words are commonly used by the three overlays of the program. The resulting load screen might look like this:

```
FORGET TASK   : TASK ;  ( CLEANS DICTIONARY)
102 LOAD  103 LOAD   ( DATA WORDS)
: INPUT    12 LOAD  13 LOAD  14 LOAD ;
: PROCESS    30 LOAD  31 LOAD  32 LOAD ;
: OUTPUT    33 LOAD  34 LOAD  35 ;
: LEVEL ;    INPUT
```

Each of the three overlay programs, INPUT , PROCESS and OUTPUT should have the phrase

```
FORGET LEVEL : LEVEL ;
```

in the first screen to be loaded. This phrase discards the previous overlay and makes room in the dictionary for the next overlay. The process of overlays is started by interpreting the word INPUT in the load screen. Note that the three overlay words are defined <u>before</u> the dummy word LEVEL . This ensures that the overlay words will not be forgotten by the overlays themselves.

The process of overlaying can be a manually directed one, or if desired, the next overlay can be called as the last action of the current overlay. The process of overlaying can then continue indefinitely and unattended.

The methods outlined above for enhanced use of mass storage are very useful in actual practice even though the methods are quite simple. FORTH is capable of much more. By using the defining words <BUILDS and DOES>, different classes of new FORTH words can be created to take advantage of other mass storage or external facilities.

12.6 SOURCE CODE EDITING

The many different mass storage devices, terminals and user preferences make it impossible to provide more than a start at putting source code onto FORTH screens. Two useful words for manipulating character data are already supplied in AIM 65/40 FORTH, namely (LINE) and .LINE . The following code defines two useful words that take advantage of (LINE) and .LINE , and a third word useful for initializing screens used for text. These three words (P , LIST , and WIPE allow data to be typed into a line of a screen and show the techniques involved in creating an editor appropriate to a particular setup. The example screen format shown results from using these words.

```
SCR # 13
  0 ( SOME PRIMITIVE WORDS TO PUT TEXT IN SCREENS.)
  1    BASE @ ( SAVE CURRENT BASE)    DECIMAL ( FOR THIS SCREEN)
  2 (  L ---     PUT TEXT INTO LINE # L VIA: L EDIT TEXTTEXTCR)
  3 : P    SCR @ (LINE) OVER SWAP BLANKS ( CLEAR LINE)
  4    0 WORD ( PARSE TEXT)  HERE COUNT 64 MIN ( 64 CH. LINES)
  5    ROT SWAP CMOVE ( MOVE TEXT)   UPDATE ;
  6
  7 (  S ---     S LIST  LISTS SCREEN # S )
  8 : LIST    DUP CR ." SCR # " . ( PRINT SCREEN AND SAVE) SCR !
  9    16 0 DO  CR I 3 .R SPACE   I SCR @ .LINE    LOOP CR ;
 10
 11 (  S ---     S WIPE BLANKS & WRITES SCREEN # S.  BE CAREFUL)
 12 : WIPE    B/SCR * B/SCR BOUNDS ( SCREEN # TO BLOCK RANGE)
 13    DO  I BLOCK  B/BUF BLANKS  UPDATE  LOOP FLUSH ;
 14
 15  BASE ! ( RESTORE PREVIOUS BASE.)     ;S
```

The words LIST and P work together in that a screen should be listed before text is placed in it with P . The act of listing a screen makes that screen the current screen and operations are directed to it.

The word LIST uses .LINE to output 16 lines of 64 characters each. LIST also prints the screen number and the number of each line, for reference when placing text in that screen. Given the line number as a parameter, the word P fetches the current screen number then places blanks in that line

before moving the following text string into it. The phrase 0 WORD parses out the following text to the carriage return and moves it to HERE with the character count in the first position. The phrase HERE COUNT ROT SWAP gets the address and count of the text, positioned correctly along with the address of the destination, so that CMOVE can be used to move the text. Once the text is in the proper buffer, UPDATE flags the buffer as having new data in it, and that data will automatically be written to mass storage if the buffer is needed

The word WIPE takes the screen number left on the stack and fills all of its blocks with spaces, thus preparing the screen for editing. Because WIPE overwrites anything written in the screen, it must be used with caution.

The words are used like this:

```
10 WIPE
```

places blanks in screen 10, and

```
10 LIST
```

verifies this.

To enter text, use P like this:

```
3 P THIS LINE OF TEXT GOES ON LINE 3.
```

This text will be placed on line 3, and the rest of line 3 will be blanked, in case there was old text on it.

Use P to place text into screens to make a simple editor. Test these words by loading the screens and trying them out. Then use the simple editor to make a enhanced editor that takes advantage of any features that your particular setup has.

Before you use P with a TTY or CRT, the input buffer size should be set to the display line length. The phrase

    80 UC/L

will do this, with the number being the number of characters per line.

After a screen has been created or edited, the new information must be written to the disk before that screen is compiled. This can be done with a FLUSH before the LOAD .

APPENDIX A

AIM 65/40 FORTH FUNCTIONAL SUMMARY

This appendix contains a summary of the AIM 65/40 FORTH word definitions, grouped by area of primary function. Consult Appendix B for the detailed definition of each word.

Stack Notation

The stack operation is denoted in the parentheses. The symbols on the left indicate the order in which input parameters must be placed on the stack prior to FORTH word execution. Three dashes (---) indicates the FORTH word execution point. Any parameters left on the stack after execution are listed on the right. The top of the stack is to the right.

Symbol Definition

n,n1,...	16-bit signed number
d,d1,...	32-bit signed number
u,u1,...	16-bit unsigned number
ud,ud1,...	32-bit unsigned number
addr,addr1,...	address
b	8-bit byte (with eight high bits zero)
c	7-bit ASCII character value (with nine high bits zero)
f	Boolean flag (zero - false, non-zero = true)
ff	Boolean false flag (value = zero)
tf	Boolean true flag (value = non-zero)

A.1 STACK MANIPULATION

DUP	(n --- n n)	Duplicate the number on the stack.
2DUP	(d --- d d) or (n1 n2 --- n1 n2 n1 n2)	Duplicate the top double number (or the top two numbers) on the stack.
DROP	(n ---)	Delete the top number on the stack.
2DROP	(d ---) or (n1 n2 ---	Delete the top double number (or the top two numbers) on the stack.
SWAP	(n1 n2 --- n2 n1)	Exchange the top two numbers on the stack.
OVER	(n1 n2 --- n1 n2 n1)	Copy second number on the stack to the top.
ROT	(n1 n2 n3 n2 n3 n1)	Rotate the third number on the stack to the top.
-DUP	(n --- n ?)	Duplicate the top number on the stack only if it is non-zero.
>R	(n ---	Move top item to Return Stack.
R>	(--- n)	Retrieve item from Return Stack.
R	(--- n)	Copy top of Return Stack onto stack.
PICK	(n --- nth)	Copy the nth item to top.
SP@	(--- addr)	Return address of stack top position.
RP@	addr)	Return address of the return stack pointer.
BOUNDS	(addr n --- addr r + n addr)	Convert start addr and count to start and stop addresses.
.S	---)	Display stack contents without modifying the stack.

A.2 NUMERAL REPRESENTATION

DECIMAL	( )	Set decimal base.
HEX	( )	Set hexadecimal base.
BASE	(--- addr)	System variable containing number base.
DIGIT	(---)	Convert ASCII to binary.
0	(--- 0)	The number zero.
1	(--- 1)	The number one.
2	(--- 2)	The number two.
3	(--- 3)	The number three.
4	(--- 4)	The number four.

A.3 ARITHMETIC AND LOGICAL

+	(n1 n2 --- sum)	Add two 16-bit numbers.
D+	(d1 d2 --- sum)	Add two 32-bit numbers.
-	(n1 n2 --- diff)	Subtract (n1-n2).
*	(n1 n2 --- prod)	Multiply.
/	(n1 n2 --- quot)	Divide (n1/n2).
MOD	(n1 n2 --- rem)	Modulo (i.e., remainder from division).
/MOD	(n1 n2 --- rem quot)	Divide, giving remainder and quotient.
*/MOD	(n1 n2 n3 --- rem quot)	Multiply, then divide (n1*n2/n3), with double intermediate.
*/	(n1 n2 n3 --- quot)	Like */MOD , but give quotient only.
U*	(u1 u2 --- ud)	Unsigned multiply leaves double product.
U/	(ud u1 --- u2 u3	Unsigned remainder and quotient from double dividend.
M*	(n1 n2 d)	Signed multiplication leaving double product.
M/	(d n1 n2 n3)	Signed remainder and quotient from double dividend.
M/MOD	(ud1 u2 u3 ud4)	Unsigned divide leaving double quotient and remainder from double dividend and single divisor.
MAX	(n1 n2 --- max)	Maximum.
MIN	(n1 n2 --- min)	Minimum.
+-	(n1 n2 --- n3)	Set sign, n3 = n1 times the sign of n2.
D+-	(d1 n --- d3)	Set sign of double number.
ABS	(n --- absolute)	Absolute value.
DABS	(d --- absolute)	Absolute value of double number.
NEGATE	(n --- -n)	Change sign.
DNEGATE	(d --- -d)	Change sign of double number.
S -> D	(n d)	Sign extend single number to double number.
1+	(n1 --- n1+1)	Increment by 1.
2+	(n1 --- n1+2)	Increment by 2.
1-	(n1 --- n1-1)	Decrement by 1.
2-	(n1 --- n1-2)	Decrement by 2.
AND	(n1 n2 --- and)	Logical AND (bitwise).
OR	(n1 n2 --- or)	Logical OR (bitwise).
XOR	(n1 n2 --- xor)	Logical exclusive OR (bitwise).

A.4 COMPARISON OPERATORS

	(n1 n2 --- f)	True if n1 less than n2.
	(n1 n2 --- f)	True if n1 greater than n2.
	(n1 n2 --- f)	True if top two numbers are equal.
Ø<	(n --- f)	True if top number negative.
Ø=	(n --- f)	True if top number zero (i.e., reverses truth value).
U<	u1 u2 --- f)	True if u1 less than u2.
NOT	f --- f')	Reverse Boolean value (same as Ø =).

A.5 CONTROL STRUCTURES

DO ... LOOP	(end+1 start --- ... loop)	Set up loop, given index range.
DO ... n +LOOP	(end+1 start --- ... n +loop)	Like DO...LOOP , but and stack value (instead of always '1') to index.
I	(--- index)	Place current index value on stack.
LEAVE	(---)	Terminate loop at next LOOP or +LOOP .
BEGIN ... UNTIL	BEGIN ... f UNTIL	Loop back to BEGIN until true at UNTIL .
BEGIN ... WHILE ... ... REPEAT	BEGIN ... f WHILE ... REPEAT	Loop while true at WHILE ; REPEAT loops unconditionally to BEGIN .
BEGIN ... AGAIN		Unconditional loop.
IF ... THEN	if: (f ---)	If top of stack true, execute.
IF ... ELSE ... THEN	if: (f ---)	Same, except that if top stack false, execute ELSE clause.
END		Alias for UNTIL .
ENDIF		Alias for THEN .

A.6 MEMORY

@	(addr --- n)	Replace word address by contents.
!	(n addr ---)	Store second word at address on top.
C@	(addr --- b)	Fetch one byte only.
C!	(b addr ---)	Store one byte only.
?	(addr ---)	Print contents of address.
+!	(n addr ---)	Add second number on stack to contents of address on top.
CMOVE	from to n --	Move n bytes in memory.
FILL	addr n b ---	Beginning at addr, fill n bytes in memory with b.
ERASE	addr n ---	Beginning at addr, fill n bytes in memory with zeroes.
BLANKS	(addr n ---)f	Beginning at addr, fill n bytes in memory with blanks.
TOGGLE	addr b ---	Exclusively OR byte at addr with byte b.

A. INPUT-OUTPUT

-CR		Output a carriage return and line feed to the AIM 65/40 printer, but not to the display.
CR		Output a carriage return and line feed to the AIM 65/40 printer and display.
SPACE	(---	Type one space.
SPACES	(n -	Type n spaces.
CLRLINE	(---	Output a CTRL B to the AIM 65/40 printer and display.
."	---)	Print message (terminated by ").
DUMP	addr n	Dump n words starting at address using current base.
TYPE	(addr n ---	Type string of n characters starting at address.
?TERMINAL	(--- f)	True if any key is depressed.
KEY	(--- c)	Read key, put ASCII value on stack.
EMIT	(c ---)	Output ASCII value from stack.
EXPECT	(addr n ---	Read n characters (or until carriage return) from input to address.
WORD	(c ---)	Read the next text character string.
IN	(--- addr)	User variable containing current offset within input buffer.
BL	(--- c)	Put a SPACE character (ASCII $20) on the stack.
C/L	(--- n)	Maximum number of characters/line.
TIB	(--- addr)	Terminal Input Buffer start addr.
QUERY	(-	Input text from terminal.
ID.	(addr ---)	Print <name> given name field address (NFA).
HANG	(---)	Wait for key stroke.

A.8 OUTPUT FORMATTING

NUMBER	(addr --- d	Convert string at address to double-precision number.
<#	---)	Start output string.
#	|d| --- |d|	Convert next digit of double-precision number and add character to output string.
#S	(|d| --- 0 0	Convert all significant digits of double-precision number to output string.
SIGN	(n |d| --- |d|)	Insert sign of n into output string.
#>	(|d| --- addr u)	Terminate output string (ready for TYPE).
HOLD	(c ---)	Insert ASCII character into output string.
HLD	(--- addr)	Hold pointer, user variable.
-TRAILING	(addr n1 --- addr n2)	Suppress trailing blanks.
.LINE	(line SCR ---)	Display line of text from mass storage.
COUNT	(addr1 --- addr+1 n)	Count and address of message text.
	(n ---)	Print number ASCII string.
.R	(n fieldwidth --)	Print number ASCII string right-justified in field.
D	(d ---)	Print double number ASCII string.
D.R	(d fieldwidth --)	Print double number ASCII string right-justified in field.
DPL	(--- addr)	Address of number of digits to the right of decimal point.

A.9 MONITOR & CASSETTE I/O

COLD	(---)	AIM 65/40 FORTH cold start.
MON	(---)	Exit to AIM 65/40 Monitor.
CLOSE	(---)	Close audio tape file.
?IN	(---)	Set active input device.
?OUT	(---)	Set active output device.
GET	c)	Input a character from the active input device.
PUT	(c	Output a character to the active output device.
READ	(addr n	Input n characters from active input device to addr.
WRITE	(addr n -)	Output n characters at addr to active output device.
SOURCE		Interpret input from active input device through AIM 65/40 Monitor.
FINIS	(--	End of file marker for input via SOURCE .

A.10 COMPILER-TEXT INTERPRETER

-->	(---)	Interpret next screen.
;S	(---)	Stop interpretation.
[COMPILE]	(<name> --	Force compilation of IMMEDIATE word.
LITERAL	(n --- n)	Compile a number into a literal.
DLITERAL	(d --- d)	Compile a double number into a literal.
EXECUTE	(addr ---	Execute the definition CFA on top of stack.
[		Suspend compilation, enter execution.
]	(-	Resume compilation.

A.11 DICTIONARY CONTROL

CREATE	(---	Create a dictionary header.
FORGET	(<name> --	FORGET all definitions from <name>.
HERE	(--- addr	Returns address of next unused byte in the dictionary.
ALLOT	(n ---	Leave a gap of n bytes in the dictionary.
TASK	(---)	A dictionary marker null word.
'	(<name> addr	Find the PFA of <name> in the dictionary.
-FIND	found: (<name> --- PFA b tf) <name> not found: (<name> --- ff)	Search dictionary for <name>.
DP	(n --- addr)	User variable containing the the dictionary pointer.
C,	(b ---)	Compiles byte into dictionary.
,	(n ---)	Compile a number into the dictionary.
PAD	(--- addr)	Pointer to temporary buffer.
IMMEDIATE	(<name>	Forces execution when compiling.
INTERPRET		The Text Interpreter executes or compiles.
LATEST	addr)	Leave name field address (NFA) of top word in CURRENT .
LIT	(--- n	Place 16-bit literal on the stack.
CLIT	(--- b)	Place byte literal on the stack.
LITERAL	(n ---)	Compile a 16-bit literal.
SMUDGE	(---)	Toggle name SMUDGE bit.
STATE	(--- addr)	User variable containing compilation state.

A.12 DEFINING WORDS

: <name>	(---)	Begin colon definition of <name>.
;	(---)	End colon definition.
VARIABLE	Compilation: (n --- <name>) Execution: (<name> --- addr)	Create a variable named <name> with initial value n; returns address when executed.
CONSTANT	Compilation: (n --- <name>) Execution: (<name> --- n)	Create a constant named <name> with value n; returns value when executed.
CODE <name>	(---)	Begin definition of assembly-language primitive operation named <name>.
;CODE	(---)	Used to create a new defining word, with execution-time "code routine" for this data type in assembly.
<BUILDS... DOES>	Compilation: <BUILDS ... Execution: ... DOES> ...	Used to create a new defining word, with execution-time routine for this data type in higher-level FORTH.
USER	Offset user <name>	Create a user variable.

A.13 VOCABULARIES

CONTEXT	(--- addr)	Returns address of pointer to CONTEXT vocabulary.
CURRENT	(--- addr)	Returns address of pointer to CURRENT vocabulary.
FORTH	(---)	Main FORTH vocabulary (execution of FORTH sets CONTEXT vocabulary).
ASSEMBLER	(---)	Assembler vocabulary; sets CONTEXT .
DEFINITIONS	(<name> ---)	Sets CURRENT vocabulary to CONTEXT .
VOCABULARY	(--- <name>)	Create new vocabulary named <name>.
VLIST	(---)	Print names of all words in CONTEXT vocabulary.
VOC-LINK	(--- addr)	Most recently defined vocabulary.

A.14 MASS STORAGE

LOAD	(screen ---)	Load editing screen into buffer and compile or execute. Automatically saves prior buffer contents if necessary.
BLOCK	(block --- addr)	Load editing screen into buffer and compile or execute. Automatically stores prior contents of buffer if necessary.
B/BUF	(--- n)	System constant giving mass storage block size in bytes.
B/SCR	(--- n)	Number of blocks/editing screen.
BLK	(--- addr)	System variable containing current block number.
SCR	(--- addr)	System variable containing current screen number.
UPDATE	(---)	Mark last buffer accessed as updated.
FLUSH	(---)	Write all updated buffers to disk.
EMPTY-BUFFERS	(---)	Erase all buffers.
+BUF	(addr1 --- addr2 f)	Increment buffer address.
BUFFER	(n --- addr)	Fetch next memory buffer.
R/W	(addr blk f ---)	User read/write linkage.
USE	(--- addr)	Variable containing address of next buffer.
PREV	(--- addr)	Variable containing address of latest buffer.
FIRST	(--- n)	Leaves address of first block buffer.
OFFSET	(--- addr)	User variable block offset to mass storage.

A.15 MISCELLANEOUS AND SYSTEM

(<comment>)(---)		Begin comment, terminate by right parentheses on same line.
CFA	(pfa --- cfa)	Alter parameter field address to code field address.
NFA	(pfa --- nfa)	Alter parameter field address to name field address.
PFA	(nfa --- pfa)	Alter name field address to parameter field adddress.
LFA	(pfa --- lfa)	Alter parameter field address to link field address.
LIMIT	(--- n)	Top of memory.
QUIT	(---)	Clear Return Stack and return to terminal.

A.16 SECURITY

!CSP	(---)	Store stack position into check stack pointer.
?COMP	(---)	Error if not compiling.
?CSP	(---)	Check stack position.
?ERROR	(---)	Outputs error message.
?EXEC	(---)	Not executing error.
?PAIRS	(---)	Conditional not paired error.
?STACK	(---)	Stack out of bounds error.
CSP	(---)	User variable for check stack pointer.
ABORT	(---)	Error ...operation terminates.
ERROR	(line --- in blk)	Execute error notification and restart system.
MESSAGE	(n ---)	Displays message number n.
WARNING	(--- addr)	Flag for to message routine.
FENCE	(--- addr)	Prevents FORGET below this point.
WIDTH	(--- addr)	Controls the number of significant characters of <name>.

A.17 PRIMITIVE

(."		Run-time procedure compiled by ." .
(;CODE)	(---)	Run-time procedure compiled by ;CODE .
(+LOOP)	(n ---)	Run-time procedure compiled by +LOOP .
(ABORT)		Run-time procedure compiled by ABORT .
(DO)	(limit+1 start ---)	Run-time procedure compiled by DO .
(FIND)	(addr1 addr2 - pfa b ff) (addr1 addr2 - ff)	Searches the dictionary
(LINE)	(n1 n2 ---) addr count)	Virtual storage line primitive.
(LOOP)	(---)	Run-time procedure compiled by LOOP .
(NUMBER)		Converts ASCII to numeric.
ØBRANCH	f ---)	Run-time conditional branch.
BRANCH	(---)	Run-time unconditional branch.
CLIT	(---)	Indicates single character literal.
ENCLOSE	(add c --- addr 1 n2 n3)	Text scanning by WORD .
R0	(--- addr)	Location of Return Stack base.
S0	(--- addr)	Location of Parameter Stack base.
RP!	(---)	Initializes Return Stack.
SP!		Initializes Parameter Stack.

APPENDIX B

AIM 65/40 FORTH GLOSSARY

This glossary contains the definition of all words in the AIM 65/40 FORTH vocabulary. The definitions are presented in ASCII sort order

Stack Notation

The first line of each entry shows a symbolic description of the action of the procedure on the parameter stack. The symbols on the left indicate the order in which input parameters have been placed on the stack. Three dashes "---" indicate the execution point; any parameters left on the stack after execution are listed on the right. In this notation, the top of the stack is to the right.

Symbol Definition

addr,addr1	memory address
b	8-bit (with high eight bits zero)
c	7-bit ASCII character (with high nine bits zero)
d,d1,...	32-bit signed double integer, most significant portion with sign on top of stack
flag	Boolean flag (0=false, non-zero=true)
ff	Boolean false flag (value = 0)
n,n1,...	16-bit signed integer number
u,u1,...	16-bit unsigned integer number
ud,ud1,...	32-bit unsigned number
tf	Boolean true flag (value = non-zero)

Pronunciation

The natural language pronunciation of FORTH names is given in double quotes (").

Integer Format

Unless otherwise noted, all references to numbers are for 16-bit signed integers. The high byte of a number is on top of the stack, with the sign in the left-most bit. For 32-bit signed double numbers, the most significant part (with the sign) is on top.

All arithmetic is implicitly 16-bit signed integer math, with error and underflow indication unspecified.

Capitalization

Word names as used within the glossary are conventionally written in upper case characters. Lower case is used when reference is made to the run-time machine codes (not directly accessible), e.g., VARIABLE is the user word to create a variable. Each use of that variable makes use of a code sequence 'variable' which executes the function of the particular variable.

Attributes (ATTR)

Capital letters show definition characteristics

C May only be used w[illegible]in a colon-defini[illegible]n. A digit indicates number [illegible] memory addresses u[illegible], if other than one

E Intended for execution only.

I Indicates that the word is IMMEDIATE and will execute during compilation, unless special action is taken

P Has precedence bit set. Will execute even when compiling.

U A user variable.

Group Key Words (GROUP)

The following key words identify the functional group (see Appendix A) that each word is most related to.

STACK	Stack Manipulation
NUMERIC	Numeric Representation
ARITHMETIC	Arithmetic and Logical
COMPARISON	Comparison Operators
CONTROL	Control Structures
MEMORY	Memory
I/O	Input/Output
FORMAT	Output Formatting
MONITOR	Monitor and Cassette Input/Output
COMPILER	Compiler - Text Interpreter
DICTIONARY	Dictionary Control
DEFINING	Defining Words
VOCABULARY	Vocabularies
MASS	Mass Storage
MISC	Miscellaneous
SECURITY	Security/Error Detection
PRIMITIVE	Primitives
ASSEMBLER	Assembler Dictionary
PARAMETER	Parameter Used in FORTH

WORD | STACK NOTATION/DEFINITION | GROUP | ATTR

!   n addr ---   MEMORY
"store"
Stores 16-bit number n into addr.

!CSP   ---   SECURITY
"store CSP"

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

ud1 --- ud2   FORMAT
"sharp"
Generates the next ASCII character placed in an output string from ud1. Result ud2 is the quotient after division by BASE, and is maintained for further processing. Use between <# and #> . See #S .

#>   d --- addr n   FORMAT
"sharp-greater"
Terminates numeric output conversion by dropping d, leaving the text address and character count n suitable for TYPE .

#S   ud --- Ø Ø   FORMAT
"sharp-s"
Converts all digits of a ud adding each to the pictured numeric output text, until the remainder is zero, A single zero is added to the output string if the number was initially zero. Use only between <# and #> .

'   --- addr   DICTIONARY   I
"tick"
Used in the form:

' <name>

If executing, leaves the parameter field address of the next word accepted from the input stream. If compiling, compiles this address as a literal; later execution will place this value on the stack.

If the word is not found after a search of CONTEXT and FORTH vocabularies an error message is displayed.

WORD | STACK NOTATION/DEFINITION | GROUP | ATTR

(MISC
"paren"
Used in the form:

(cccc)

Accepts and ignores comment characters from the input stream, until the next right parenthesis. As a word, the left parenthesis must be followed by one blank. It may be freely used while executing or compiling. An error condition exists if the input stream is exhausted before the right parenthesis.

(.")   PRIMITIVE   C

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

(;CODE)   PRIMITIVE   C

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

(+LOOP)   PRIMITIVE   C

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

(ABORT)   PRIMITIVE

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

(DO)   limit +1 start ---   PRIMITIVE   C

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

(FIND) PRIMITIVE

addr1 addr2 --- pfa byte tf (ok)
addr1 addr2 --- ff (bad)

Searches the dictionary starting at the name field address addr2, matching to the text at addr1. Returns parameter field address, length of name field byte and Boolean true for a good match. If no match is found, only a Boolean false is left. See -FIND .

(LINE) n1 n2 --- addr count PRIMITIVE

Converts the line number n1 and the screen number n2 to the disk buffer address containing the data. A count of 64 indicates the full line text length. See .LINE .

(LOOP) PRIMITIVE C

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

(NUMBER) d1 addr1 --- d2 addr2 PRIMITIVE

Converts the ASCII text beginning at addr1+1 with regard to BASE . The new value is accumulated into d1, being left as d2. addr2 is the address of the first unconvertable digit. See NUMBER .

***** n1 n2 --- n3 ARITHMETIC
"times"
Multiples n1 by n2 and leaves the product n3.

***/** n1 n2 n3 --- n4 ARITHMETIC
"times-divide"
Multiplies n1 by n2, divides the result by n3 and leaves the quotient n4. n4 is rounded toward zero. The product of n1 times n2 is maintained as an intermediate 32-bit value for a greater precision than the otherwise equivalent sequence:

n1 n2 * n3 /

***/MOD** n1 n2 n3 --- n4 n5 ARITHMETIC
"times-divide-mod"
Multiplies n1 by n2, divides the result by n3 and leaves the remainder n4 and quotient n5. A 32-bit intermediate product is used as for */ . The remainder has the same sign as n1.

+ n1 n2 --- n3 ARITHMETIC
"plus"
Adds n1 to n2 and leaves the arithmetic sum n3.

+! n addr --- MEMORY
"plus store"
Adds n to the 16-bit value at the address, by the convention given for +.

+- n1 n2 --- n3 ARITHMETIC
"plus-minus"
Applies the sign of n2 to n1, which is left as n3.

+BUF addr1 --- addr2 flag MASS
"plus-buf"
Advances the virtual storage buffer address (addr1) to the next buffer address (addr2). Boolean flag is false when addr2 is the buffer presently pointed to by variable PREV .

+LOOP n1 --- (run-time) CONTROL IC
addr n2 --- (compile-time)
"plus-loop"
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 compared to the limit. The branch back to DO occurs until the new index is equal to or greater than the limit (n1 > Ø), or until the new index is equal to or less than the limit (n1 < Ø). Upon exiting the loop, the parameters are discarded and execution continues. Index and limit are signed integers in the range <-32,768..32,767>.

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR

+LOOP (Cont.)

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

, — n --- — DICTIONARY

"comma"
Stores n into the next available dictionary memory cell, advancing the dictionary pointer.

- — n1 n2 --- n3 — ARITHMETIC

"minus"
Substracts n2 from n1 and leaves the difference n3.

--> — MASS — I

"next-screen"
Continues interpretation with the next virtual storage screen.

-CR — INPUT/OUTPUT

"carriage-return"
Issues a carriage return and line feed to the printer but not to the display. Calls AIM 65 Monitor subroutine CRCK.

-DUP — STACK

n1 --- n1 (if zero)
n1 --- n1 n1 (non-zero)

"minus-dup"
Reproduces n1 only if it is non-zero. This is usually used to copy a value just before IF , to eliminate the need for an ELSE clause to drop it.

-FIND — DICTIONARY

--- pfa b tf (found)
--- ff (not found)

"dash-find"
Accepts the next text word (delimited by blanks) in the input stream to HERE , and searches the CONTEXT and then CURRENT vocabularies for a matching entry. 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.

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR

-TRAILING — addr n1 --- addr n2 — FORMAT

"dash-trailing"
Adjusts the character count n1 of a text string beginning address to suppress the output of trailing blanks. The characters at addr+n1 to addr+n2 are blanks. An error condition exists if n1 is negative.

n --- — INPUT/OUTPUT

"dot"
Displays the number on the top of a stack. The number is converted from a signed 16-bit two's complement value according to the numeric BASE . The sign is displayed only if the value is negative. A trailing blank is displayed after the number. Also see D. .

INPUT/OUTPUT — I

"dot-quote"
Used in the form:

." cccc"

Accepts the following text from the input stream, terminated by " (double-quote). If executing, transmits this text to the selected output device. If compiling, compiles so that later execution will transmit the text to the selected output device. At least 127 characters are allowed in the text. If the input stream is exhausted before the terminating double-quote, an error condition exists.

.LINE — n1 n2 --- — FORMAT

"dot-line"
Displays a line of text from mass storage by its line number n1 and screen number n2. Trailing blanks are suppressed.

.R — n1 n2 --- — FORMAT

"dot-R"
Displays number n1 right justified n2 places. No trailing blank is printed.

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR
.S	"dot-S" Displays the contents of the stack without altering the stack. This word is very useful in determining the stack contents during debugging programs and learning FORTH.	STACK	
/	nl n2 --- n3 "divide" Divides nl by n2 and leave the quotient n3. n3 is rounded toward zero. The remainder is lost.	ARITHMETIC	
/MOD	nl n2 --- n3 n4 "divide-mod" Divides nl by n2 and leaves the quotient n4 and remainder n3. n3 has the same sign as nl.	ARITHMETIC	
Ø	--- Ø "zero" The number zero is placed on top of the stack.	NUMERIC	
Ø<	n --- flag "zero-less" Leaves a true flag (1) if the number is less than zero (negative), otherwise leaves a false flag (Ø). The number is lost.	COMPARISON	
Ø=	n --- flag "zero-equals" Leaves a true flag (1) if the number is equal to zero, otherwise leaves a false flag (Ø). The number is lost.	COMPARISON	
ØBRANCH	flag --- "zero-branch" The run-time procedure to conditionally branch. If the flag is false (zero), the following in-line parameter is added to the interpretive pointer to branch ahead or back. Compiled by IF UNTIL, and WHILE .	PRIMITIVE	C

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR
1	--- 1 "one" The number one is placed on top of the stack.	NUMERIC	
1+	n --- n+1 "one-plus" Increments n by one according to the operation of + .	ARITHMETIC	
1-	n --- n-1 "one-minus" Decrements n by one according to the operation of - .	ARITHMETIC	
2	--- 2 "two" The number two is placed on top of the stack.	NUMERIC	
2+	n --- n+2 "two-plus" Increments n by two according to the operation of + .	ARITHMETIC	
2-	n --- n-2 "two-minus" Decrements n by two, according to the operation of - .	ARITHMETIC	
2DROP	d --- or nl n2 --- "two-drop" Drops the top double number on the stack.	STACK	
2DUP	d --- d d or nl n2 --- nl n2 nl n2 "two-dup" Duplicates the top double number on the stack.	STACK	
3	--- 3 "three" The number three is placed on top of the stack.	NUMERIC	
4	--- 4 "four" The number four on top of the stack.	NUMERIC	

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR

DEFINING E

"colon"
A defining word used in the form:

: <name> ... ;

Selects the CONTEXT vocabulary to be identical to CURRENT . Creates a dictionary entry for <name> in CURRENT , and sets the compile mode. Words thus defined are called 'colon-definitions'. The compilation addresses of subsequent words from the input stream which are not immediate words are stored into the dictionary to be executed when <name> is later executed. IMMEDIATE words are executed as encountered.

If a word is not found after a search of the CONTEXT and FORTH vocabularies conversion and compilation of a literal number is attempted, with regard to the current BASE ; that failing, an error condition exists.

DEFINING

"semi-colon"
Terminates a colon-definition and stops further compilation. If compiling from mass storage and the input stream is exhausted before encountering ; an error condition exists.

;CODE

DEFINING

"semi-colon-code"
Used in the form:

: <name> .... ;CODE <assembly code>
END-CODE

Stops compilation and terminates a new defining word <name> by compiling (;CODE) . Sets the CONTEXT vocabulary to ASSEMBLER , assembling to machine code the following mnemonics.

When <name> is later executed in the form:

<name> <namex>

to define the new <namex>, the code field address of <namex> will contain the address of the code sequence following the ;CODE in <name>. Execution of any <namex> will cause this machine code sequence to be executed.

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR

;S

COMPILER

"semi-colon-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 --- flag COMPARISON

"less-than"
Leaves a true flag (1) if n1 is less than n2; otherwise leaves a false flag (Ø).

<#

d --- d FORMAT

"less-than-sharp"
Initializes the pictured numeric output format using the words:

<# # #S HOLD SIGN #>

specifies the conversion of a double-precision number into an ASCII character string stored in right-to-left order, producing text at PAD .

<BUILDS

DEFINING

Used within a colon-definition:

: <name> <BUILDS ... DOES> ... ;

each time <name> is executed, <BUILDS defines a new word with a high-level execution procedure. Executing <name> in the form:

<name> <namex>

uses <BUILDS to create a dictionary entry for <namex> with a call to the DOES> part for <namex> When nnnn is later executed, it has the address of its parameter area on the stack and executes the words after DOES> in <name>. <BUILDS and DOES> allow run-time procedures to written in high-level rather than in assembler code (as required by ;CODE).

=

n1 n2 --- flag COMPARISON

"equals"
Leaves a true flag (1) if n1 is equal to n2; otherwise leaves a false flag (Ø).

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR

n1 n2 --- flag — COMPARISON
"greater-than"
Leaves a true flag (1) if n1 is greater than n2; otherwise a false flag (0).

>R — n --- — STACK
"to-R"
Removes a number from the computation stack and places it as the most accessible number on the return stack. Use should be balanced with R> in the same definition.

addr -- — STACK
"question-mark"
Displays the value contained at the address on the top of the stack in free format according to the current BASE. Uses the format of . .

?COMP — SECURITY
Issues error message if not compiling.

?CSP — SECURITY
Issues error mes: : stack posi ion differs from value saved in (

?ERROR — SECURITY
Issues error message #1 (STACK EMPTY), if the Boolean flag is true.

?EXEC — SECURITY
Issues an error message if not executing.

?IN — MONITOR
Calls the AIM 65/40 Monitor subroutine WHEREI to set the active input device.

?OUT — MONITOR
Calls the AIM 65/40 Monitor subroutine WHEREO to set the active output device.

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR

?PAIRS — n1 n2 --- — SECURITY
Issues error message #19 (CONDITIONALS NOT PAIRED) if n1 does not equal n2. The message indicates tha compiled conditionals do not match.

?STACK — SECURITY
Issues error message #7 (FULL STACK) if the stack is out of bounds.

?TERMINAL — --- flag — INPUT/OUTPUT
Tests the terminal keyboard for actuation of any key. Generates a Boolean value. A true flag (1) indicates actuation, whereas a false flag (0) indicates non-actuation.

@ — addr --- n — MEMORY
"fetch"
Leaves the 16-bit contents of the address on top of the stack.

ABORT — SECURITY
"abort"
Clears the stacks and enters the execution state. Returns control to the AIM 65/40 keyboard.

ABS — n --- u — ARITHMETIC
"absolute"
Leaves the absolute value of n as u

AGAIN — addr n --- (co -time) — CONTROL
"again"
Used in a :olon-defini in the form:

BEGIN AGAIN

At run-time, AGAIN forces execution to return to the corresponding BEGIN . There is no effect on the stack. Execution cannot leave this loop (unles 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.

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR
ALLO'	n --- "allot" Adds the signed number to the dictionary pointer DP . May be used to reserve dictionary space or re-origin memory. n is the number of bytes.	COMPILER	
AND	n1 n2 --- "and" Leaves the bit-wise logical AND of n1 and n2 as n3.	ARITHMETIC	
ASSEMBLER	"assembler" Sets the vocabulary to ASSEMBLER .	VOCABULARY	
B/BUF	--- n "bytes-per-buffer" Leaves the number of bytes (default value = 128) per data buffer, the byte count read from mass storage by BLOCK . The actual buffer size is four bytes larger than this value.	MASS	
B/SCR	--- n "blocks per screen" Leaves the number of blocks (default value 8) per FORTH screen. By convention, an editing screen is 1024 bytes organized as 16 lines of 64 characters each.	MASS	
BASE	--- addr "base" Leaves the address of the variable containing the current number base used for input and output conversion. The range of BASE is 2 through 70.	NUMERIC	U

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR
BEGIN	--- addr n (compile-time) "begin" Occurs in a colon-definition in form: BEGIN ... flag UNTIL BEGIN ... AGAIN BEGIN ... flag WHILE ... REPEAT At run-time, BEGIN marks the start of a word sequence for repetitive execution. A BEGIN-UNTIL loop will be repeated until flag is true. A BEGIN-WHILE-REPEAT loop will be repeated until flag is false. The words after UNTIL or REPEAT will be executed when either loop is finished. flag is always dropped after being tested. The BEGIN-AGAIN loop executes indefinitely. At compile-time, BEGIN leaves its return address and n for compiler error checking.	CONTROL	
BL	--- cha "blank" A constant th ves the ASCI: character value fo "blank", i.e.	INPUT/OUTPUT	
BLANKS	addr n --- "blanks" Fills an area of memory beginning at addr with the ASCII value for "blank", the number of bytes specified by count n will be blanked.	MEMORY	
BLK	--- addr "b-l-k" Leaves the address of a user variable containing th number of the mass storage block being interpreted as the input stream. If the content is zero, the input stream is taken from the terminal.	MASS	
BLOCK	n --- addr "block" Leaves the first address of the block buffer containing block n. If the block is not already in memory, it is transferred from mass storage to whichever buffer was least recently accessed. If the block occupying that buffer has been marked as updated, it is rewritten onto mass storage before block n is read into the buffer. If correct mass storage read or write is not possible, an error condition exists. Only data within the latest block	MASS	

WORD	**STACK NOTATION/DEFINITION**	**GROUP**	**ATTR**

BLOCK (Cont. referenced by BLOCK is valid by byte address, due to sharing of the block buffers. n is an unsigned number. Also see BUFFER , R/W , UPDATE and FLUSH .

BOUNDS addr n --- add +n addr — ARITHMETIC
"bounds"
Bounds is equivalent to OVER + SWAP . It is used to convert addr and count to a start and stop address for a loop.

BRANCH — CONTROL
"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 , and REPEAT .

BUFFER n --- addr — MASS
"buffer"
Obtains the next block buffer, assigning it to block n. The block is not read from mass storage. If the previous contents of the buffer is marked as UPDATED, it is written to the mass storage. If correct writing to mass storage is not possible, an error condition exists. The address left is the first byte within the buffer for data storage.

C! n addr --- — MEMORY
"c-store"
Stores the least significant 8-bits of n into the byte at the address.

C, n --- — DICTIONARY
"c-comma"
Stores 8 bits of n into the next available dictionary byte, advancing the dictionary pointer.

C/L --- n — INPUT/OUTPUT
"characters/line"
Leaves the number of characters (default value = 80) per input line.

C@ add --- byte — MEMORY
"c-fetch
Leaves t 3-bit contents the byte at the address on the t of the stack i e low order byte. The high ord byte is zero.

WORD	**STACK NOTATION/DEFINITION**	**GROUP**	**ATTR**

CFA pf. -- cfa — MISC
"c-f-a"
Convert: e parameter field address (pfa) of a definit to its code field address (cfa).

CLIT --- b — STACK
"c-lit"
Compiled within system object code to indicate that the next byte is a single character literal (i.e., in range 0-255). Used only in system code (not by application program, i.e. user). Application programs use LITERAL , which uses CLIT or LIT as appropriate.

CLOSE — MONITOR
"close"
Closes tape Sets the active input drive to keyboard and /e output device :o Display/ Printer.

CLRLINE --- — INPUT/OUTPUT
"clear-line"
Outputs a CTRL B to the AIM 65/40 printer and display to clear to the end of the line.

CMOVE addr1 addr2 n --- — MEMORY
"c-move"
Moves n bytes from memory area beginning at address addr1 to memory area starting at addr2. The contents of addr1 is moved first proceeding toward high memory. If n is zero or negative, nothing is moved.

CODE — ASSEMBLER
"code"
A defining word used in the form:

CODE <name> ... <assembly code> ... END-CODE

To set CONTEXT to the ASSEMBLER vocabulary and to create a dictionary entry for <name>. When <name> is later executed the machine code in this parameter field will execute.

WORD	STACK NOTATION/DEFINITION	GROUP ATTR

COLD — MONITOR

"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. Performs the same functions as entering FORTH from the AIM 65/40 Monitor via the 5 key.

COMPIL — COMPILER

"compile"
When the word containing COMPILE executes, the compilation address of the next non-immediate word following COMPILE is copied (compiled) into the dictionary. This allows specific compilation situations to be handled in addition to simply compiling an execution address (which the interpreter already does).

CONSTANT — n --- <name> (compile-time) DEFINING
<name> --- n (run-time)

"constant"
A defining word used in the form:

n CONSTANT <name>

to create a dictionary entry for <name>, leaving n in its parameter field. When <name> is later executed, it will push the value of n to the stack.

CONTEXT — --- addr — DICTIONARY

"context"
Leaves the address of a user variable pointing to the vocabulary in which dictionary searches are to made, during interpretation of the input stream.

COUNT — addr --- addr+1 n — FORMAT

"count"
Leaves the address addr+1 and the character count n of text beginning at addr. The first byte at addr must contain the character count n. The actual text starts with the second byte. The range of n is Ø-255. Typically COUNT is followed by TYPE .

CR — INPUT/OUTPUT

"carriage-return"
Transmits a carriage) and line feed (LF) to the active output

WORD	STACK NOTATION/DEFINITION	GROUP ATTR

CREAT — DICTIONARY

"create"
A defining word used in the form

CREATE <name>

Creates a dictionary entry for <name> without allocating any parameter field memory. When <name> is subsequently executed, the address of the first byte of <name>'s parameter field is left on the stack. The code field contains the address of the word's parameter field. The new word is created in the CURRENT vocabulary.

CSP — ---- addr — SECURITY

"c-s-p"
Leaves the address of a user variable temporarily storing the check stack pointer (CSP) position, for compilation error checking.

CURRENT — --- addr — DICTIONARY

"current"
Leaves the address of a user variable pointing to the vocabulary into which new word definitions are to be entered.

D+ — d1 d2 --- d3 — ARITHMETIC

"d-plus"
Adds double precision numbers d1 and d2 and leaves the double precision number sum d3.

D+- — d1 n --- d2 — ARITHMETIC

Applies the sign of n to the double precision number d1 and leaves it as double precision number d2.

D. — d --- — FORMAT

"d-dot"
Displays a signed double-precision number from a 32-bit two's complement value. The high-order 16 bits are most accessable on the stack. Conversion is performed according to the current BASE . A blank follows.

D.R — d n --- — FORMAT

"d-dot-r"
Displays a signed double-precision number d right aligned in a field n characters wide.

DABS d --- ud ARITHMETIC
"d-abs"
Leaves the absolute value ud of a double number.

DECIMAL NUMERIC
"decimal"
Sets the numeric conversion BASE to decimal (base 10) for input-output.

DEFINITIONS VOCABULARY
"definitions"
Used in the form:

cccc DEFINITIONS

Sets CURRENT to the CONTEXT vocabulary so that subsequent definitions will be created in the vocabulary previously selected at CONTEXT .

DIGIT NUMERIC
char n1 --- n2 tf (valid conversion)
char n1 --- ff (invalid conversion)
"digit"
Converts the ASCII character (using BASE n1) to its binary equivalent n2, accompanied by a true flag (1). If the conversion is invalid, leaves only a false flag (0).

DLITERAL d --- d (executing) COMPILER
d --- (compiling)
"d-literal"
If compiling, compiles a stack double number into a literal. Later execution of the definition containing the literal will push it to the stack.

If executing, the number will remain on the stack.

DNEGATE d1 --- -d1 ARITHMETIC
"d-negate"
Leaves the two's complement of a double precision number.

DO n1 n2 --- (run-time) CONTROL
addr n --- (compile-time

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. At the +LOOP the index is modified by a positive or negative value. Until the new index equals or exceeds the limit, execution loops back to just after DO ; otherwise the loop parameters are discarded and execution continues ahead. Both n1 and n2 are determined at run-time and may be the resuult of other operations.

Loops may be nested. Within a loop I will copy the current value of the index to the stack. See I , LOOP , +LOOP , LEAVE .

At compile-time within the colon-definition, DO compiles (DO) and leaves the following addr and n for later error checking.

DOES>
"does"
Defines t
defining
Used in t

: <name> ... <BUILDS ...
DOES> ... ;
and then <name> <namex>.

Marks the termination of the defining part of the defining word <name> and begins the definition of the run-time action for words that will later be defined by <name>.

DOES> alters the code field and first parameter of the new word to execute the sequence of compiled word addresses following DOES> . Used in combination with <BUILDS . The execution of the DOES> part begins with the address of the first parameter of the new word <namex> on the stack. Upon execution of <name> the sequence of words between DOES> and ; will be executed, with the address of <namex>'s parameter field on the stack. This allows interpretation using this area or its contents.

WORD | **STACK NOTATION/DEFINITION** | **GROUP** | **ATTR**

DOES> (Cont.)

Typical uses include the FORTH assembler, multi-dimensional arrays, and compiler generation.

DP — addr — COMPILER — U

"d-p"

Leaves the address of user variable, the dictionary pointer, which points to address the next free memory address above the dictionary. The value may be read by HERE and altered by ALLOT .

DPL — ---- addr — FORMAT — U

"d-p-l"

Leaves the address of user variable containing the number of digits to the right of the decimal on double integer input. It may also be used hold output column location of a decimal point, in user generated formating. The default value on single number input is -1.

DROP — n --- — STACK

"drop"

Drops the number on top of the stack from the stack.

DUMP — addr n --- — INPUT/OUTPUT

"dump"

Displays the contents of n memory locations beginning at addr. Both addresses and contents are shown in the current numeric base. DUMP outputs 8 bytes on a line.

DUP — n --- n n — STACK

"dup"

Duplicates the value on the stack.

ELSE — addr1 n1 addr2 n2 (compiling) — CONTROL — I

"else"

Occurs within a colon-definition in the form:

IF ... ELSE ... THEN

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 THEN . It has no stack effect.

WORD | **STACK NOTATION/DEFINITION**

ELSE (Cont.)

At compile-time, ELSE emplaces BRANCH reserving a branch offset, 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 at addr1. See IF and THEN .

EMIT — char --- — INPUT/OUTPUT

"emit"

Transmits an ASCII character to the selected output device. See KEY .

EMPTY-BUFFERS — MASS

"empty-buffers"

Marks all block-buffers as empty, not necessarily affecting the contents. Updated blocks are not written to the mass storage. This is also the required initialization procedure before first use of the mass storage.

ENCLOSE — addr char --- addr n1 n2 n3 — PRIMITIVE

"enclose"

The text scanning primitive used by WORD. From the text address addr and an ASCII delimiting character is determined the byte offset to the first non-delimiter character n1, the offset to the first delimiter after the text n2, and the offset to the first character not included n3. This procedure will not process past an ASCII 'null', treating it as an unconditional delimiter.

END — CONTROL

"end"

This is an 'alias' or duplicate definition for UNTIL .

ENDIF — addr n --- (compile) — CONTROL

"end-if"

An alias for THEN . See THEN .

ERASE — addr n --- — MEMORY

"erase"

Clears a region of memory to zero from addr over n addresses.

WORD | **STACK NOTATION/DEFINITION** | **GROUP** **ATTR**

ERROR line --- in blk SECURITY
"error"
Executes error notification and restart of system. WARNING is first examined. If WARNING = 1, the text of line n, relative to screen 4 of drive 0 is printed. This line number may be positive or negative, and beyond just screen 4. If WARNING = 0, n is just printed as a message number (non-disk installation). If WARNING = -1, the definition (ABORT) is executed, which executes the system ABORT . The user may cautiously modify this execution by altering (ABORT) . AIM 65/40 FORTH saves the contents of IN and BLK to assist in determining the location of the error. Final action is execution of QUIT .

EXECUTE addr --- COMPILER
"execute"
Executes the definitio[n whos]e code field address is on the stack. The code [field] address is also called the compilation address.

EXPECT addr count --- INPUT/OUTPUT
"expect"
Transfers characters from the terminal beginning at addr, upwards until a "return" or the count of n characters has been received. Takes no action for n = zero or less. One or more nulls are added at the end of the text.

FENCE --- addr SECURITY U
"fence"
Leaves the address of a user variable containing an address below which FORGETting is trapped. To forget below this point the user must alter the contents of FENCE .

FILL addr n b --- MEMORY
"fill"
Fills n bytes, beginning at addr with the byte pattern b.

FINIS MONITOR
"finis"
Marks the end of the input data stream into the compiler.

FIRST --- n MASS
"first"
Leaves the first (lowest address of the data (or mass storage buffer.

WORD | **STACK NOTATION/DEFINITION** | **GROUP** **ATTR**

FLUS MASS
"flush"
Writes all blocks to mass storage that have been flagged as UPDATEd . An error condition results i writing to mass storage is not completed.

FORGE DICTIONARY
"forget"
Executes in the form:

FORGET <name>

Delete from the dictionary <name> (which is in th CURRENT vocabulary) and all words added to the dictionary after <name>, regardless of their vocabulary. An error message will occur if the CURRENT and CONTEXT vocabularies are not currently the same. Failure to find <name> in CURRENT or FORTH is an error condition.

FORTH VOCABULARY
"forth"
The name of the primary vocabulary. Execution makes FORTH the CONTEXT vocabulary.

New definitions become a part of FORTH until a differing CURRENT vocabulary is established.

User vocabularies conclude by "chaining" to FORTH, so it should be considered that FORTH is 'contained' within each user's vocabulary.

GET --- char MONITOR
"get"
Leaves the ASCII value of the next available character from the active input device and outputs the character to the active output device.

HANG INPUT/OUTPUT
"hang"
Waits until a key is depressed then continues

HERE --- addr DICTIONARY
"here"
Leaves the addres[s of the] next available dictionary location.

HEX

NUMERIC

"hex"
Sets the numeric conversion BASE to sixteen (hexadecimal).

HLD

--- addr FORMAT

"hold"
Leaves the address of user variable which holds the address of the latest character of text during numeric output conversion.

HOLD

char FORMAT

"hold"
Used between <# and #> to insert an ASCII character into a pictured numeric output string.

I

-- CONTROL

"i"
Used within a DO-LOOP to copy the loop index from the return stack to the stack.

ID.

nfa INPUT/OUTPUT

"i-d-dot"
Print a definition's name from its name field address. See NFA.

IF

flag (run-time) CONTROL
addr n (compile)

"if"
Used in a colon-definition in form:

```
IF ... THEN
IF ... ELSE ... THEN
```

At run-time, IF selects execution based on a Boolean flag. If flag is true, the words following IF are executed and the words following ELSE are skipped. The ELSE part is optional.

If flag is false, the words between IF and ELSE, or between IF and THEN (when no ELSE is used), are skipped. IF-ELSE-THEN conditionals may be nested.

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

WORD STACK NOTATION/DEFINITION GROUP ATTR

IMMEDIATE

COMPILER

"immediate"
Marks the most recently made dictionary entry as a word which will be executed when encountered rather than being compiled.

IN

--- addr INPUT/OUTPUT

"in"
Leaves the address of 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 .

INTERPRET

COMPILER

"interpret"
The outer text interpreter which sequentially executes or compiles text from the input stream (terminal or mass storage) depending on STATE . If the word name 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 .

KEY

--- char INPUT/OUTPUT

"key"
Leaves the ASCII value of the next available character from the active input device.

LATEST

addr COMPILER

"latest"
Leaves the name field address of the top-most word in the CURRENT vocabulary.

LEAVE

CONTROL

"leave"
Forces termination of a DO-LOOP at the next opportunity 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.

WORD | STACK NOTATION/DEFINITION | GROUP ATTR

LFA pfa --- lfa DICTIONARY

"l-f-a"

Converts the parameter field address (pfa) of a dictionary definition to link field address (lfa).

LIMIT MISC

Leaves the highest address plus one available in the data (or mass storage) buffer. Usually this is the highest system memory.

LIT n COMPILER

"lit"

Within a colon-definition, LIT is automatically compiled before each 16-bit literal number encountered in input text. Later execution of LIT causes the contents of the next dictionary address to be pushed to the stack.

LITERAL n --- (compiling) COMPILER

"literal"

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

```
: xxx [ calculate ] LITERAL ;
```

Compilation is suspended for the compile time calculation of a value. Compilation is then resumed and LITERAL compiles this value into the definition.

LOAD n --- MASS

"load"

Begins interpretation of screen n by making it the input stream; preserves the locators of the present input stream (from IN and BLK).

If interpretation is not terminated explicitly it will be terminated when the input stream is exhausted. Control then returns to the input stream containing LOAD , determined by the input stream locators IN and BLK .

WORD | STACK NOTATION/DEFINITION | GROUP ATTR

LOOP addr n --- (compiling) CONTROL

"loop"

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. The branch back to DO occurs until the index equals or exceeds the limit; at that time, the parameters are discarded and execution continues ahead.

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

M* n1 n2 --- d ARITHMETIC

"m-times"

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

M/ d n1 --- n2 n3 ARITHMETIC

"m-divides"

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 ARITHMETIC

"m-divide-mod"

An unsigned mixed magnitude math operation which leaves a double quotient ud4 and remainder u3, from a double dividend ud1 and single divisor u2.

MAX n n2 --- max ARITHMETIC

"max"

Leaves the greater of two numbers.

MESSAGE n --- SECURITY

"message"

Displays on the selected active device the text of line n relative to screen 4 of drive Ø. 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 displayed as a number (no mass storage).

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR
MIN	n1 n2 --- n3 "min" Leaves the smaller number n3 of two numbers, n1 and n2.	ARITHMETIC	
MOD	n1 n2 --- n3 "mod" Leaves the remainder n3 of n1 divided by n2, with the same sign as n1.	ARITHMETIC	
MON	"mon" Exits to the AIM 65 Monitor, leaving a re-entry to FORTH.	MONITOR	
NEGATE	n --- -n "negate" Leaves the two's complement of a number, i.e. the difference of Ø less n.	ARITHMETIC	
NFA	pfa --- nfa Converts the parameter field address (pfa) of a definition to its name field address (nfa).	DICTIONARY	
NOT	flag --- "not" Leaves a true flag (1) if the number is equal to zero, otherwise leaves a false flag. Same as Ø= .	COMPARISON	
NUMBER	addr --- d "number" Converts a character string left at addr with a preceeding count, to a signed double precision number, using the current number 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.	FORMAT	
OFFSET	--- addr "offset: Leaves the address of user variable which contains a block offset to mass storage. The content of OFFSET is added to the stack number by BLOCK . Messages by MESSAGE are independent of OFFSET . See BLOCK and MESSAGE .	MASS	U

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR
OR	n1 n2 --- n3 "or" Leaves the bit-wise logical or of two 16 bit values.	ARITHMETIC	
OVER	n1 n2 --- n1 n2 n1 "over" Copies the second stack value, placing it as the new top of stack.	STACK	
PAD	--- addr Leaves the address of a scratch area used to hold character strings for intermediate processing. The maximum capacity is 64 characters.	DICTIONARY	
PFA	nfa --- pfa "p-f-a" Converts the name field address (nfa) of a dictionary definition to its parameter field address (pfa).	DICTIONARY	
PICK	n --- nth "pick" Returns the contents of the nth stack value, not counting n itself. An error conditions results for n less than one. 2 PICK is equivalent to OVER .	STACK	
PREV	--- addr "prev" Leaves the address of a user variable containing the address of the disk buffer most recently referenced. The UPDATE command marks this buffer to be later written to mass storage.	MASS	U
PUT	(c ---) "put" Transmits an ASCII character to the active output device (see ?OUT).	MONITOR	
QUERY	"query" Accepts input of up to 8Ø characters of text, (or until a 'return") from the keyboard into the terminal input buffer (TIB) . WORD may be used to accept text from this buffeer as the input stream, by setting IN and BLK to zero.	INPUT/OUTPUT	

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR

QUIT — MISC

"quit"
Clears the return stack, stops compilation, and returns control to the keyboard. No message is given.

R — --- n — STACK

"r"
Copies the top of the return stack to the computation stack.

R/W — addr blk flag --- — MASS

"r-slash-w)
The mass storage read-write linkage. addr specifies the source or destination block buffer, blk is the sequential number of the referenced block; and flag specified read or write (flag = Ø is write and flag = 1 is read). R/W determines the location on mass storage, performs the read-write and performs any error checking. R/W executes the cfa found in UR/W . On cold start this is the address of (ABORT) .

R> — --- n — STACK

"r-from"
Removes the top value from the return stack and leaves it on the computation stack. See >R and R .

RØ — --- addr — PRIMITI /E — U

"r-zero"
Leaves the address c variable contai ing the initial value of the stack pointer. See RP! .

READ — addr n --- — MONITOR

"read"
Inputs n characters from the active input device and stores the ASCII value starting at addr.

REPEAT — addr n --- (compiling) — CONTROL

"repeat"
Used within a colon-definition in the form:

BEGIN ... WHILE ... REPEAT

At run-time, RE rces an unconditional branch back to just aft orresponding BEGIN .

At compile-time, compiles BRANCH and the offset from HER dr. n is used for error testing.

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR

ROT — n2 n3 n2 n3 n1 — STACK

"rote
Rotat the top t values on the stack, bringin the t rd to the .

RP! — PRIMITIVE

"r-p-store"

Initializes the return stack pointer from user variable RO .

RP@ — --- addr — STACK

"r-p-fetch"
Leaves the address of a variable containing the return stack pointer.

S->D — n --- d — ARITHMETIC

"s-to-d"
Extends the sign of single number n to form double number d.

SØ — --- addr — PRIMITIVE

"s-zero"
Leaves the address user variable that contains the initial value for the parameter stack pointer. See SP!

SCR — --- addr — MASS

"s-c-r"
Leaves the address of user variable containing the screen number most recently referenced by LIST .

SIGN — n d --- d — FORMAT

"sign"
Inserts the ASCII "-" (minus sign) into the pictured numeric output string if n is negative. n is discarded, but double number d is maintained. Must be used between <# and #> .

SMUDGE — DICTIONARY

"smudge"
Used during word definition to toggle the "smudge bit" in a definitions name field. This prevents an uncompleted definition from being found during dictionary searches, until compiling is completed without error.

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR

SOURCE — MONITOR

"source"
A procedure which identifies the active input device for batch compilation. The procedure is:

SOURCE <RETURN> IN = [INPUT DEVICE CODE]

If the device code = M, compilation starts at the top of the AIM 65/40 Editor Text buffer. Compilation continues until FINIS is encountered.

SP! — STACK

"s-p-store"
Initializes the stack pointer from S0 .

SP@   --- addr — STACK

"s-p-fetch"
Returns the address of the top of the stack as it was before SP@ was executed. (e.g., 1 2 SP@ @ . . . would type 2 2 1)

SPACE — INPUT/OUTPUT

Transmits an ASCII blank to the active output device.

SPACES   n ---

"spaces"
Transmit n ASCII blanks to the active output device.

STATE   --- addr — COMPILER — U

"state"
Leaves the address of user variable containing the compilation state. A non-zero value indicates compilation.

SWAP   n1 n2 --- n2 n1 — STACK

"swap"
Exchanges the top two values on the stack.

TASK — DICTIONARY

"task"
A no-operation word which can mark the boundary between applications. By forgetting TASK and re-compiling, an application can be discarded in its entirety. Its definition is : TASK ; .

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR

THEN — CONTROL

"then"
Used within a colon-definition, in the form:

```
IF . . . ELSE . . . THEN  or
IF       THEN
```

THEN is the point where execution resumes after ELSE or IF (when no ELSE is present).

TIB   --- addr — INPUT/OUTPUT

"t-i-b"
Leaves the address of user variable containing the starting address of the terminal input buffer.

TOGGLE   addr b --- — MEMORY

"toggle"
Complements the contents of addr by the 8-bit pattern byte.

TYPE   addr n --- — INPUT/OUTPUT

"type"
Transmits n characters beginning at addr to the active output device. No action takes place for n less than one.

U*   un1 un2 --- ud — ARITHMETIC

"u-times"
Performs and unsigned multiplication of un1 by un2, leaving the unsigned double number product of two unsigned numbers.

U-CR   --- addr — PARAMETER

"u-dash-carriage return"
Leaves the address of the user variable containing the code field address of the U-CR orphan word.

U/   ud u1 --- u2 u3 — ARITHMETIC

"u-divide"
Performs the unsigned division of double number ud by u1, leaving the unsigned remainder u2 and unsigned quotient n3 from the unsigned double dividend ud and unsigned divisor u1.

U<   un1 un2 --- flag — ARITHMETIC

"u-less-than"
Leaves the flag representing the magnitude comparison of un1 < un2 where un1 and un2 are treated as 16-bit unsigned integers.

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR
U?IN	--- addr "u-question mark-in" Leaves the address of the user variable containing the code field address of the U?IN orphan word.	PARAMETER	U
U?OUT	--- addr "u-question mark-out" Leaves the address of the user variable containing the code field address of the U?OUT orphan word.	PARAMETER	U
U?TERMINAL	--- addr "u-question mark-terminal" Leaves the address of the user variable containing the code field address of the U?TERMINAL orphan word.	PARAMETER	U
UABORT	--- addr "u-abort" Leaves the address of the user variable containing the code field address of the ABORT word.	PARAMETER	U
UB/BUF	--- addr "u-bytes-per-buffer" Leaves the address of the user variable containing the number of bytes per buffer.	PARAMETER	U
UB/SCR	--- addr "u-blocks-per-screen" Leaves the address of the user variable containing the number of blocks per screen.	PARAMETER	U
UC/L	--- addr "u-characters-per-line" Leaves the address of the user variable containing the number of characters per line.	PARAMETER	U
UCLOSE	--- addr "u-close" Leaves the address of the user variable containing the code field address of the UCLOSE orphan word.	PARAMETER	U
UCR	--- addr "u-carriage return" Leaves the address of the user variable containing the code field address of the UCR orphan word.	PARAMETER	U

WORD	STACK NOT TIO TION	GROUP	ATTR
UEMIT	--- "u-emit" Leaves th ss of the user variable containing the code ddress of the EMIT output word.	PARAMETER	U
UFIRST	--- "u-first" Leaves th s of the user variable containing the first the data (or mass storage) buffer.	PARAMETER	U
UKEY	--- "u-key" Leaves th s of the user variable containing code fiel s of the the KEY input word.	PARAMETER	U
ULIMIT	--- "u-limit" Leaves th s of the user variable containing the last plus one of the data (or mass storage)	PARAMETER	U
UNTIL	f (run-time) addr (compile-time) "until" Occurs w olon-definition in the form: BEG UNTIL At run-t flag is true, the loop is terminated. If flag , execution returns to the first word after B BEGIN - UNTIL structures may be nested. At compi UNTIL compiles ØBRANCH and an offset f to addr. n is used for error tests.	CONTROL	
UPDATE	"update" Marks th ecently referenced block (pointed to by PREV tered. The block will subsequently be trans utomatically to mass storage, should its buff quired for storage of a different bloc	MASS	

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR

UR/W --- addr PARAMETER

"u-read-write"
Leaves the address of the user variable containing the code field address of the mass storage I/O word. Initialized to (ABORT) on a cold start.

USE --- addr MASS U

"use"
Leaves the address of user variable containing the address of the block buffer to use next, as the least recently written.

USER n --- DEFINING

"user"
A defining word used in the form:

n USER <name>

which creates a user variable <name>. The parameter field of <name> contains n as a fixed offset relative to the user pointer register UP for this user variable. When <name> is later executed, it places the sum of its offset and the user area base address on the stack as the storage address of that particular variable. Offsets of $6Ø to $7F are available. See Appendix G.

VARIABLE n --- <name> (compute-time) DEFINING
<name> --- (run-time)

"variable"
A defining word executed in the form:

n VARIABLE <name>

to create a dictionary entry for <name> and allot two bytes for storage in the parameter field. When <name> is later executed, it will place the storage address on the stack.

VOC-LINK ---- addr VOCABULARY

"voc-link"
Leaves the address of user variable containing the address of a field in the definition of the most recently created vocabulary. All vocabulary names are linked by these fields to allow control for FORGETting through multiple vocabularies.

WORD	STACK NOTATION/DEFINITION	GROUP	ATTR

VOCABULARY VOCABULARY

"vocabulary"
A defining word used in the form:

VOCABULARY <name>

to create (in the CURRENT vocabulary) a dictionary entry for <name>, which specifies a new ordered list of word definitions. Subsequent execution of <name> will make it the CONTEXT vocabulary. When <name> becomes the CURRENT vocabulary (see DEFINITIONS), new definitions will be created in that list.

New vocabularies 'chain' to FORTH. This is, when a all of dictionary search through a vocabulary is exhausted, FORTH will be searched.

VLIST VOCABULARY

"v-list"
Lists the names of the definitions in the CONTEXT vocabulary. Depression of any key will terminate the listing.

WARNING --- addr SECURITY U

"warning"
Leaves the address of user variable containing a value controlling messages. If value = 1 mass storage is present and screen 4 of drive Ø is the base location for messages. If value = Ø, no disk is present and messages will be presented by number. If value = -1, execute (ABORT) for a user specified procedure. See MESSAGE and ERROR .

WHILE flag --- (run-time) CONTROL
addr1 n1 -> addr1 n1 addr2 n2

"while"
Occurs in a colon-definition in the form:

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

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

At compile-time, WHILE emplaces (ØBRANCH) and leaves addr2 of the reserved offset. The stack values will be resolved by REPEAT .

WIDTH --- addr SECURITY U
"width"
Leaves the address of user variable containing the maximum number of letters saved in the compilation of a definitions name. It must be 1 through 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 char --- COMPILER
"word"
Receives characters from the input stream until the non-zero delimiting character in the stack is encountered or the input stream is exhausted, ignoring leading delimiters. The characters are stored as a packed string with the character count in the first character position. The actual delimiter encountered (char or null) is stored at the end of the text but not included in the count. If the input stream was exhausted as WORD is called, then a zero length will result.

WRITE addr n --- MONITOR
"write"
Outputs n characters to the active output device starting at addr.

XOR n1 n2 --- n3 ARITHMETIC
"x-or"
Leaves the bit-wise logical exclusive or of two values.

COMPILER
"left-bracket"
Ends the compilation mode. The text from the input stream is subsequently executed. See] .

[COMPILE] COMPILER
"bracket compile"
Used in a colon-definition in form:

```
[COMPILE] <name>
```

Forces compilation of the following word. This allows compilation of an IMMEDIATE word when it would otherwise be executed.

COMPILER
"right bracket"
Sets the compilation mode. The text from the input stream is subsequently compiled. See [.

APPENDIX C

AIM 65/40 FORTH ASSEMBLER FUNCTIONAL SUMMARY

This appendix contains a summary of the AIM 65/40 FORTH Assembler word definitions grouped by area of primary function. Consult appendix D for the detail definition of each word.

Stack Notation

The stack operation is denoted in parenthesis. The symbols on the left indicate the order in which input parameters must be placed on the stack prior to FORTH word execution. Three dashes (---) indicate the FORTH word execution point. Any parameters left on the stack after execution are listed on the right. The top of the stack is to the right.

Symbol Definition

A/T	Assembly-time
R/T	Run-time
H/B	High-byte
L/B	Low-byte
addr, addr1,..	Address

C.1 OP-CODES

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

C.2 ADDRESS MODES

.A	---	Accumulator address mode.
#	---	Immediate address mode.
,X	---	Indexed X address mode.
,Y	---	Indexed Y address mode.
X)	---	Indexed Indirect X address mode.
)Y	---	Indirect Indexed Y address mode.
	---	Indirect Absolute address mode.

C.3 CONDITIONAL SPECIFIERS

Ø<	A/T:	--- cc	Branch on negative (N=1).
Ø=	A/T:	--- cc	Branch on zero (Z=1).
VS	A/T:	--- cc	Branch on overflow (V=1).
CS	A/T:	--- cc	Branch on carry (C=1).
NOT	A/T:	cc1 --- cc2	Reverse the condition code.

C.4 CONTROL

BEGIN,	A/T:	--- addrB 1	At A/T, leaves the dictionary pointer address and the value 1 for later testing of conditional pairing.
	R/T:	---	At R/T, marks the beginning of a repeatedly executed assembly sequence.
UNTIL,	A/T:	addrB 1 cc --	At A/T, assembles a conditional branch instruction to addrB (BEGIN, point) based on condition code cc.
	R/T:	---	At R/T, conditionally branches to the BEGIN, point (if cc is false) or continues ahead (if cc is true).
AGAIN,	A/T:	addrB 1 ---	At A/T, assembles a JMP instruction to addrB (BEGIN, point)
	R/T:	---	At R/T, jumps to the BEGIN, point.

REPEAT,	A/T:	addrB 1 addrW 3 ---	At A/T, assembles a JMP instruction to the BEGIN, point
	R/T:	---	At R/T, jumps to the BEGIN, point.
WHILE,	A/T:	addrB 1 --- addrB 1 addrW 3	At A/T, assembles a conditional branch instruction to the instruction following REPEAT, based on the condition code cc.
	R/T:	---	At R/T, conditional branches to the point following REPEAT, if cc is false, or continues ahead if cc is true.
IF,	A/T:	--- addr 2	At A/T, creates an unresolved forward conditional branch based on cc and leaves addr for resolution by ELSE, or THEN,.
	R/T:	cc --- addr 2	At R/T, conditionally branches to the ELSE, point (or THEN, point if ELSE is not present) if cc is false, or continues ahead if cc is true.
ELSE,	A/T:	addr1 2 --- addr2 2	At A/T, assembles a forward JMP instruction to THEN, and resolves the forward conditional branch from IF, .
	R/T:	---	At R/T, marks the start of an assembly sequence conditionally branched to from IF, if cc is false.
THEN,	A/T:	addr 2 ---	At A/T, marks the conclusion of a conditional structure started by IF, and resolves the forward conditional branch from IF, (if ELSE, is not present).
	R/T:	---	At R/T, marks the conclusion of a conditional structure started by 'IF, .
ENDIF,	A/T:	addr 2 ---	Alias for THEN, .
	R/T:	---	

C.5 RETURN

PUSH	A/T:	addr	At A/T, leaves the address of the R/T return point which will add the accumulator (H/B) and the top machine stack byte (L/B) to the data stack.
	R/T		
PUT	A/T:	--- addr	At A/T, leaves the address of the R/T return point which will write the accumulator (H/B) and the top machine stack byte (L/B) to replace the existing top data stack 16-bit value (n1).
	R/T:	n2	
POP	A/T	addr	At A/T, leaves the address of the R/T return point which will pull a 16-bit value from the data stack and continue interpretation.
	R/T:		
PUSHØA	A/T	addr	At A/T, leaves the address of the R/T return point which will push a zero (H/B) and the accumulator (L/B) onto the data stack.
	R/T		
PUTØA	A/T:	addr	At A/T, leaves the address of the R/T return point which will write a zero (H/B) and the accumulator (L/B) to replace the existing data stack 16-bit Value (n1).
	R/T:	n1 n2	
POPTWO	A/T	--- addr	At assembly-time, leaves the address of the run-time return point which will pull two 16-bit values from the data stack and continue interpretation.
	R/T	n1 n2	
NEXT	A/T	addr	At assembly-time, leaves the address of the FORTH inner-interpreter.

C.6 STACK

BINARY	A/T:		At A/ leaves the addr s of a return poin which, at R/T, will ull two 16-bit valu from the stack and sh the accumu- lato (H/B) and the top mach e stack byte (L/B) to t data stack.
	R/T:	n1 n2 --	
RP)	A/T:	1Ø1 (hex)	At A/T, used to address the bottom of the Return Stack.
TOP	A/T	Ø	At A/T, used to address the top item on the data stack.
SEC	A/T		At A/T, use to address the second tem on the data stack

C.7 REGISTERS

N	A/T:	-- addr	Leaves the address of a nine-byte work space in page zero.
SETUP	A/T:	addr	Leaves the address of a utility routine to move items from the stack to the N area on z-page.
IP	A/T:	addr	Leaves the address of the pointer to the next FORTH execution address in a colon-definition to be interpreted.
UP	A/T:	addr	Leaves the address of the pointer to the base of the user area.
W	A/T:	addr	Leaves the address of the pointer to the code field of the FORTH word being executed.
XSAVE	A/T	addr	Leaves the address of a temporary buffer for saving the X register.

C.8 INTERRUPT

INTFLAG	A/T:	--- addr	At A/T, leaves the address of the interpretive flag byte on page zero.
	R/T:		
INTVECT	A/T:	--- addr	At A/T, leaves the address of the interpretive interrupt vector.
	R/T:		

C.9 MISCELLANEOUS

END-CODE	A/T:	---	Marks the end of a CODE-definition.
MEM	A/T:		Sets MODE to direct memory addressing on z-page.

APPENDIX D

AIM 65/40 FORTH ASSEMBLER GLOSSARY

This glossary contains the definitions of all words in the AIM 65/40 FORTH ASSEMBLER vocabulary with exception of the op-codes. The definitions are presented in ASCII sort order.

Stack Notation

The first line of each entry shows a symbolic description of the action of the procedure on the parameter stack. The symbols on the left indicate the order in which input parameters have been placed on the stack. Three dashes "---" indicate the execution point; any parameters left on the stack after execution are listed on the right. In this notation, the top of the stack is to the right.

Symbol Definition

addr,addr1,...	memory address
cc,cc1,...	condition code
n,n1,...	16-bit signed number

Pronunciation

The natural language pronunciation of FORTH names is given in double quotes (").

Capitalization

Word names as used within the glossary are conventionally written in upper-case characters. Lower case is used when reference is made to the run-time machine codes (not directly accessible), e.g., VARIABLE is the user word to create a variable. Each use of that variable makes use of a code sequence 'variable' which executes the function of the particular variable.

Group Key Words (GROUP)

ADDRESS	Addressing Mode
OP-CODE	Operation Code
CONTROL	Control Structures
STACK	Stack Addressing
REGISTER	Assembly Register
CONDITION	Conditional Specifier
RETURN	Return of Control
INTERRUPT	Interrupt Processing

STACK NOTATION/DEFINITION **GROUP**

"immediate" ADDRESS
Specifies 'immediate' addressing mode for the next op-code generated.

"indirect" ADDRESS
Specifies 'indirect absolute' addressing mode for the next op-code generated.

--- ADDRESS
"indirect indexed Y"
Specifies 'indirect indexed Y' addressing mode for the next op-code generated.

--- ADDRESS
"indexed x"
Specifies 'indexed X' addressing mode for the next op-code generated.

--- ADDRESS
"indexed Y"
Specifies 'indexed Y' addressing mode for the next op-code generated.

--- ADDRESS
"accumulator" ,
Specifies accumulator addressing mode for the next op-code generated.

--- cc (assembly-time) CONDITION
"zero-less"
Specifies that the immediately following conditional will branch based on the processor negative flag status bit being negative (N=1), i.e., less than zero. The flag cc is left at assembly-time; there is no run-time effect on the stack.

--- cc (assembly-time) CONDITION
"zero-equals"
Specifies that the immediately following conditional will branch based on the processor zero flag status bit being equal to one (Z=1); i.e. equal to zero. The flag cc is left at assembly-time; there is no run-time effect on the stack.

WORD — **STACK NOTATION/DEFINITION** — GROUP

AGAIN, — CONTROL

addr 1 --- (assembly-time)
--- (run-time)
"again"
Occurs in a CODE-definition in the form:

BEGIN, . . . AGAIN,

At assembly-time, AGAIN, assembles a JMP instruction to addr. The number 1 is issued for error checking.

At run-time, AGAIN, branches unconditionally to its matching BEGIN, .

BEGIN, — CONTROL

--- addr 1 (assembly-time)
--- (run-time)
Occurs in a CODE-definition in the form:

BEGIN, . . . cc UNTIL,

At assembly time, BEGIN , leaves the dictionary pointer address addr and the value 1 for later testing of conditional pairing by UNTIL, or AGAIN, .

At run-time, BEGIN, marks the start of an assembly sequence repeatedly executed. It serves as the return point for the corresponding UNTIL, . When reaching UNTIL, a branch to BEGIN, will occur if the processor status bit given by cc is false; otherwise execution continues ahead.

BINARY, — RETURN

--- addr (assembly-time
n1 n2 --- (n) (run-time)
"binary"
At assembly-time constant which leaves the machine address of a return point which, at run-time, will pull two 16-bit values from the stack and push the accumulator (high-byte) and the top machine stack byte (as low-byte) to the data stack.

CS — CONDITION

--- cc (assembly-time)
"carry-set"
Specifies that the immediately following conditional will branch based on the processor carry status flag being set (C=1). The flag cc is left at assembly-time; there is no run-time effect on the stack.

WORD — **STACK NOTATION/DEFINITION** — **GROUP**

ELSE — CONTROL

addr1 2 --- addr2
2 (assembly-time) --- (run-time)
"else"
Occurs within a CODE-definition in the form:

cc IF, <true part> ELSE, <false part>
THEN,

At assembly-time, ELSE, assembles a forward jump to just after THEN, and resolves a pending forward conditional branch from IF, . The value 2 is used for error checking of conditional pairing.

At run-time, if the condition code specified by cc is false, execution will skip to the machine code following ELSE, .

END-CODE — MISC

"end-code"
An error check word marking the end of a CODE-definition. Successful execution to and including END-CODE will unsmudge the most recent CURRENT vocabulary definition, making it available for execution. END-CODE also exits the ASSEMBLER making CONTEXT the same as CURRENT .

ENDIF — CONTROL

addr 2 --- (assembly-time)
--- (run-time)
"end-if"
Another name for THEN, .

IF — CONTROL

--- addr 2 (assembly-time)
cc --- addr 2 (run-time)
"if"
Occurs within a code definition in the form:

cc IF, <true part> ELSE, <false part>
THEN,

At assembly-time IF, creates an unresolved forward branch based on the condition code cc, and leaves addr and 2 for resolution of the branch by the corresponding ELSE, or THEN, . Conditionals may be nested.

At run-time, IF, branches based on the condition code cc (Ø< or Ø= or CS). If the specified processor status is true, execution continues ahead, otherwise branching occurs to just after ELSE, (or THEN, when ELSE, is not present). At ELSE, execution resumes at the corresponding THEN, .

WORD — **STACK NOTATION/DEFINITION** — **GROUP**

INTFLAG

--- add (assembly-time   INTERRUPT
--- (run-time)

"interrupt fl '
A constant wh value is the ss of the interpretive :errupt flag by zero page It is used in code-definitio the form:

```
INTFLAG LDA,  (MOVE THE INTFLAG BYTE TO A)
```

Bit 7 of INTFLAG is the interpretive interrupt bit. 1 means interrupt. Bit 6 of INTFLAG is the interpretive interrupt mask bit. If bit 6 is on, bit 7 is not tested for an interpretive interrupt. If bit 6 is off and bit 7 is on, the word whose code field address is in INTVECT will be executed on return to NEXT . After that word finishes, regular FORTH word execution continues.

INTVECT

--- addr (assembly-time)   INTERRUPT
--- (run-time)

"interrupt vector"
A constant whose value is the address of the interpretive interrupt vector. This vector must contain the code field address of the FORTH word to execute on interpretive interrupt.

IP

--- addr (assembly-time)   REGISTER

"i-p"
Used in a CODE-definition in the form:

```
IP STA,  or  IP )Y LDA,
```

At assembly-time, a constant which leaves the address of the pointer to the next FORTH execution address in a colon-definition to be interpreted.

At run-time, NEXT moves IP ahead within a colon-definition. Therefore, IP points just after the execution address being interpreted. If an in-line data structure has been compiled (i.e., a character string), indexing ahead by IP can access this data:

```
IP STA,  or  IP )Y LDA,
```

loads the third byte ahead in the colon-definition being interpreted.

WORD — **STACK NOTATION/DEFINITION** — **GROUP**

MEM

---   MISC

"memo y"
Used within the mbler to set E to the defau.t value f rect memory ad ssing on z-pag .

MODE

--- addr   ADDRESS

"mode"
System variable used to determine the assembler addressing mode.

N

--- addr (assembly-time)   REGISTER

"n"
Used in a CODE-definition in the form:

```
N 1 - STA,  or  N 2+ )Y ADC,
```

A constant which leaves the address of a 9 byte workspace in z-page. Within a single CODE-definition, free use may be made over the range N-1 thru N+7. See SETUP .

NEXT

--- addr (assembly-time)   RETURN

"next"
A constant which leaves the machine address of the FORTH address interpreter. All CODE-definitions must return execution to NEXT, or include code that returns to NEXT (i.e., PUSH , PUT , PUSHØA , PUTØA , BINARY , POP , POPTWO).

NOT

cc1 --- cc2 (assembly-time)   CONDITION

"not"
When assembling, reverse the condition code for the following conditional. For example:

```
Ø= NOT IF,  <true part>  THEN,
```

will branch based on "not equal to zero".

POP

--- addr (assembly-time)   RETURN
n --- (run-time)

"pop"
A constant which leaves (during assembly) the machine address of the return point which, at run-time, will pull a 16-bit value from the data stack and continue interpretation.

WORD	STACK NOTATION/DEFINITION	GROUP

POPTWO — RETURN

--- addr (assembly-time)
n1 n2 --- (run-time)

"pop-two"
At assembly time, constant which leaves machine address of the return point which, at run-time, will pull two 16-bit values from the data stack and continue interpretation.

PUSH — RETURN

--- addr (assembly-time)
--- n (run-time)

"push"
At assembly-time, constant which leaves the machine address of the return point which, at run-time, will add the accumulator (as high-byte) and the top machine stack byte (as low-byte) to the data stack.

PUSHØA — RETURN

--- addr (assembly-time)
--- n (run-time)

"push-Ø-a"
At assembly-time, constant which leaves the machine address of the return point which, at run-time, will add a zero (as high byte) and the accumulator (as low byte) to the data stack.

PUT — RETURN

--- addr (assembly-time)
n1 --- n2 (run-time)

"put"
At assembly time, constant which leaves the machine address of the return point which, at run-time, will write the accumulator (as high-byte) and the top machine stack byte (as low-byte) over the existing data stack 16-bit value (n1).

PUTØA — RETURN

--- addr (assembly-time)
n1 --- n2 (run-time)

"put-zero-a"
At assembly-time, constant which leaves the machine address of the return point which, at run-time, will write a zero (as high-byte) and the accumulator as low-byte) over the existing data stack 16-bit value (n1).

WORD	STACK NOTATION/DEFINITION	GROUP

REPEAT — CONTROL

addrB 1 addrW 3 ---
(assembly-time) --- (run-time)

"repeat"
Occurs in a code definition in the form:

BEGIN, ... cc WHILE, ... REPEAT,

At assembly-time, REPEAT, assembles as JMP instruction to the instruction immediately following the BEGIN, word.

At run-time REPEAT, unconditionally branches back to its matching BEGIN, .

RP) — STACK

--- 1Ø1 (assembly-time)

"return-pointer"
Used in a CODE-definition in the form:

RP) LDA, or RP) 3+ STA,

Addresses the top byte of the return stack (containing the low byte) by selecting the ,X mode and leaving n=$1Ø1. n may be modified to another byte offset. Before operating on the return stack the X register must be saved in XSAVE and TSX, executed. Before returning to NEXT, the X register must be restored.

SEC — STACK

--- 2 (assembly-time)

"second"
Used in a CODE-definition in the form:

SEC LDA, or SEC 1+ STA,

Addresses the second 16-bit item on the data stack by selecting the , X,X address mode and leaving 2 on the stack.

SETUP — STACK

--- addr (assembly-time)

"setup"
A constant whose value is the address of a utility routine to move items from the stack to the N area of zero page. The number of items to move (1, 2, 3 or 4 only) is in the A register.

WORD	STACK NOTATION/DEFINITION	GROUP

THEN

addr 2 --- (assembly-time) CONTROL
--- (run-time)
"then"
Occurs in a CODE-definition in the form:

```
cc IF, <true part> ELSE, <false part> THEN,
```

At assembly-time, THEN, marks the conclusion of a conditional structure. The conditional branch instructions generated by IF, and the JMP instruction generated ELSE, point to the instruction immediately following THEN, . When assembling, addr and 2 are used to resolve the pending forward branch to THEN, .

At run-time THEN, marks the conclusion of a conditional structure. Execution of either the true part or false part resumes following THEN, .

TOP

--- 0 (assembly-time) STACK
"top"
Used during code assembly in the form:

```
TOP LDA, or TOP 1+ X) STA,
```

Addresses the top of the data stack (containing the low byte) by selecting the ,X mode and leaving n=0, at assembly-time. This value of n may be modified to another byte offset into the data stack. Must be followed by a multi-mode op-code mnemonic.

UNTIL,

addr 1 cc --- (assembly-time) CONTROL
--- (run-time)
"until"
Occurs in a CODE-definition in the form:

```
BEGIN, ... cc UNTIL,
```

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.

At run-time, UNTIL, controls the conditional branching back to BEGIN, . If the processor status bit specified by cc is false, execution returns to BEGIN, ; otherwise execution continues ahead.

WORD	STACK NOTATION/DEFINITION	GROUP

UP

--- addr (assembly-time) REGISTER
"user pointer"
Used in a CODE-definition in the form:

```
UP LDA, or UP )Y STA,
```

A constant which leaves the address of the pointer to the base of the user area. The instructions

```
HEX 12 # LDY, UP )Y LDA,
```

will load the low byte of the sixth user variable, DP.

VS

--- cc (assembly-time) CONDITION
"overflow set"
Specifies that the immediately following conditional will branch based on the processor status overflow flag being on (V=1). The flag cc is left at assembly-time; there is no run-time effect on the stack.

W

--- addr (assembly-time)
Used in a CODE-definition in the form:

```
W 1+ STA, or W 1 - JMP, or W )Y ADC,
```

At assembly-time constant which leaves at assembly-time the address of the pointer to the code field (execution address) of the FORTH dictionary word being executed. Indexing relative to W can yield any byte in the definitions parameter field. For example, the instructions

```
2 # LDY, W )Y LDA,
```

will fetch the first byte of the parameter field.

WHILE,

addrB 1 --- addrB 1 addrW 3 CONTROL
(assembly-time) --- (run-time)
"while"
Occurs in a CODE-definition in the form:

```
BEGIN, ... cc WHILE, ... REPEAT,
```

At assembly-time, WHILE, assembles a conditional relative branch instruction to the instruction immediately following the REPEAT, based on the condition code cc.

WORD	STACK NOTATION/DEFINITION	GROUP
WHILE, (Cont.)	At run-time WHILE, controls the conditional branching to just past REPEAT, . If the processor status bit specified by cc is true, WHILE, continues execution through to REPEAT, which then branches back to BEGIN, . If cc is false a jump is made to just after REPEAT, and execution continues.	
X)	"indexed indirect X" Specifies "indexed indirect X" addressing mode for the next op-code generated.	ADDRESS
XSAVE	--- addr (assembly-time) "x-save" Used in a CODE-definition in the form: XSAVE STX, or XSAVE LDX, A constant which leaves the address at assembly time of a temporary buffer for saving the X register. Since the X register indexes to the data stack in z-page, it must be saved and restored when used for other purposes.	REGISTER

APPENDIX E

ERROR MESSAGES AND RECOVER

E.1 STANDARD ERROR MESSAGE

The standard FORTH error message is "?" . This question mark is output along with the most recently interpreted word when that word can not be found in the dictionary and will not convert into a number in the current BASE . For example:

```
ROCKWELL AIM 65/40

{5}
AIM 65/40 FORTH V1.4

QUERTY
QUERTY ?

ABC
ABC ?

HEX OK

ABC OK
DECIMAL . <RETURN> 2748 OK
```

Upon initialization, QUERTY and ABC were not in the dictionary, therefore, the ? error message was displayed when they were entered. After the number base of the I/O was changed to HEX , however, ABC became a valid number. ABC was then accepted as a valid number upon the record entry attempt, converted to internal two's complement binary format, and stored on the stack. The number was then removed from the stack and displayed in decimal.

E.2 STANDARD ERROR MESSAGE WORD

AIM 65/40 FORTH has a standard error message word

?ERROR

which takes two items from the stack:

t n ?ERROR

where t is Boolean and n is the desired error number.

If the Boolean is false, nothing happens; but if it is true, one of three things happen depending on the value of the user variable WARNING . If WARNING is zero, the number n is printed as an error message. If WARNING is greater than zero, a disk is assumed to be in use. Then n becomes the line number relative to line 0, screen 4 of drive 0 and that line number is displayed in ASCII. The line number may be negative, zero or positive and greater than fifteen. The line number is simply an offset from line 0 screen 4. If WARNING is less than zero, the word ABORT is executed.

E.3 AIM 65/40 FORTH ERROR DEFINITIONS

The error conditions detected by AIM 65/40 FORTH are listed in Table E-1. For increased utility the two most common errors are given in English. These are error message 1, STACK EMPTY , and warning message 4, NOT UNIQUE .

The last action of error messages processing is to clear the stacks and execute QUIT . However, the warning message 'NOT UNIQUE' is simply output, it has no effect on the stacks and execution continues normally.

Error message number 3 is slightly different in that it prints the name of the code word being defined, the name of the assembler op-code word being interpreted, and the message number or message.

Table E-1. AIM 65/40 FORTH Error Message

Number	Message	Definition	Action
0	?	Echoed word was the last one interpreted. Name is not in the dictionary and is not a number.	Define the named item. Check number conversion base.
1	STACK EMPTY	Parameter stack is empty.	Don't pull more items off of the stack than are there.
2	DICTIONARY FULL	The dictionary space is used up. FIRST HERE - is less than $A0.	Increase space for dictionary by FORGETing entries or moving FIRST .
3	HAS INCORRECT ADDRESS MODE	The address mode for that assembler op-code is incorrect.	Use a correct address mode. See R6502 Programming Manual.
4	NOT UNIQUE	The dictionary entry <name> just created is not unique.	Be aware that the new definition of <name> obscures the old one and all future references to <name> will be to the new entry (often an advantage).
5	--	Not assigned	--

Table E-1. AIM 65/40 FORTH Error Message (Continued)

Number	Message	Definition	Action
6	DISC RANGE?	The disk block asked for is out of range.	This is available for the user to put in his definition of R/W.
7	FULL STACK	The parameter stack is full (more than 65 items).	Remove some stack item. DROP or output.
8	DISC ERROR!	There has been a disk error.	This is available for the user's R/W definition.
9-16	--	Not assigned	--
17	COMPILATION ONLY	The word just interpreted must be used in a definition.	Don't use compilation words interpretively.
18	EXECUTION ONLY	The word just interpreted must be used outside of a definition.	Don't use interpretive words in a definition.
19	CONDITIONALS NOT PAIRED	Omitted word or incorrect nesting of conditionals.	Pair conditionals correctly.
2Ø	DEFINITION NOT FINISHED	The current definition is not yet finished.	Finish definition.

Table E-1. AIM 65/40 FORTH Error Message (Continued)

Number	Message	Definition	Action
21	IN PROTECTED DICTIONARY	The word in question is below the FENCE	Cease trying to FORGET a protected word or move FENCE.
22	USE ONLY WHEN LOADING	Incorrect use of the word -->	Use the word --> only while loading.

APPENDIX F

PAGE ZERO and ONE MEMORY MAP

Hex Address	No. Bytes	Cold Start Hex Value	Warm Start Hex Value	Parameter Name	Parameter Description
ØØØ-ØØF	16				Stack overflow.
Ø1Ø-Ø91	13Ø				Parameter Stack.
Ø92-Ø93	2				Error storage for BLK .
Ø94-Ø95					Error storage for IN
Ø96-Ø9E				N	N Area temporary buffer.
Ø9F-ØAØ		98 C2	98 C2	IP	Interpretive Pointer -- initialized to STRTUP .
ØA1	1	6C	6C	W-1	Op-Code for Indirect JMP .
ØA2-ØA3				W	Working address for jump to next FORTH word.
ØA4-ØA5	2	ØØ Ø7	ØØ Ø7	UP	User area pointer
ØA6	1	-	-	XSAVE	X register temporary storage.
ØA7	1	ØØ	-	INTFLAG	Interrupt flag.
ØA8-ØA9	2	45 D2	-	INTVECT	Interrupt vector. Initialized to ABORT .
ØAA-ØEF	69				Unused by FORTH.
ØFØ-ØFF	16				Used by AIM 65/4Ø I/O ROM and Monitor
1ØØ-1FF	256				FORTH and R65Ø2 return stack.

APPENDIX G

USER VARIABLES RAM MAP

Hex Address	No. Bytes	Cold Start Value	Warm Start Value	Parameter Name	Parameter Description
ØØ-7Ø1	2	5Ø C3		U?IN	CFA of U?IN orphan word.
Ø2-7Ø3	2	5A C3		U?OUT	CFA of U?OUT orphan word.
Ø4-7Ø5	2	F2 C3		UCLOSE	CFA of UCLOSE orphan word.
Ø6-7Ø7	2	33 C3		UKEY	CFA of UKEY orphan word.
Ø8-7Ø9	2	2Ø C3		UEMIT	CFA of UEMIT orphan word.
ØA-7ØB	2	FD C3		U-CR	CFA of U-CR orphan word.
ØC-7ØD	2	3B C3		U?TERMINAL	CFA of U?TERMINAL orphan word.
ØE-7ØF	2	47 C3		UCR	CFA of UCR orphan word.
1Ø-711	2	1F ØØ		WIDTH	No. of letters in name.
12-713	2	ØØ ØØ		WARNING	Error message action switch.
14-715	2	ØØ Ø8		FENCE	Forget protection point.
16-717	2	ØB Ø8		DP	Dictionary Pointer.
18-719	2	2A Ø7		VOC-LINK	Last VOC field.
1A-71B	2	4Ø ØØ		UC/L	No. of characters/line.
1C-71D	2	8Ø ØØ		UB/BUF	No. of bytes/disk buffer.
1E-71F	2	Ø8 ØØ		UB/SCR	No. of buffers/screen.
2Ø-721	2	81 AØ			FORTH chain head.
22-723	2	ØØ Ø8			FORTH vocabulary pointer.
24-725	2	ØØ ØØ			FORTH vocabulary line.
26-727	2	81 AØ			ASSEMBLER chain head.
28-729	2	C3 DF			ASSEMBLER vocabulary pointer.
2A-72B	2	24 Ø7			ASSEMBLER vocabulary link.
2C-72D	2	45 D2		UABORT	CFA of UABORT orphan word.
2E-72F	2	45 D2		UR/W	CFA of UR/W orphan word.
3Ø-731	2	92 ØØ	92 ØØ	SØ	Parameter Stack base address.
32-733	2	FF Ø1	FF Ø1	RØ	Return Stack base address.
34-735	2	8Ø Ø7	8Ø Ø7	TIB	Terminal Input Buffer pointer.
36-737	2	ØØ 4Ø		USE	Mass storage buffer to use.
38-739	2	ØØ 4Ø		PREV	Mass storage buffer just used.

Hex Address	No. Bytes	Cold Start Value	Warm Start Value	Parameter Name	Parameter Description
73A-73B		00 40		UFIRST	Start of mass storage buffer.
73C-73D		00 40		ULIMIT	End of mass storage buffer.
73E-73F		00 00		BLK	Number of current block.
740-741				IN	Byte offset in current input stream.
742-743	2		-	SCR	Most recently listed screen.
744-745	2		-	OFFSET	Block offset to disk drives.
746-747	2	22 07	22 07	CONTEXT	CONTEXT vocabulary pointer
748-749	2	22 07	22 07	CURRENT	CURRENT vocabulary pointer
74A-74B	2	00 00	00 00	STATE	Contains state of computation
74C-74D	2	0A 00	-	BASE	Current I/O base number.
74E-74F	2		-	DPL	Number of decimals in double-precision input.
750-751	2			CSP	Check Stack Pointer.
752-753	2			HLD	Address of current output.
754-755	2			MODE	ASSEMBLER addressing mode.
756-77F	42				User available.
780-7FF	128				Terminal Input Buffer

APPENDIX H

ASCII CHARACTER SET

HEX	DEC	ASCII	HEX	DEC	ASCII	HEX	DEC	ASCII	HEX	DEC	ASCII
00	0	NUL	20	32	SP	40	64	@	60	96	
01	1	SOH	21	33	!	41	65	A	61	97	a
02	2	STX	22	34	"	42	66	B	62	98	b
03	3	ETX	23	35	#	43	67	C	63	99	c
04	4	EOT	24	36	$	44	68	D	64	100	d
05	5	ENQ	25	37	%	45	69	E	65	101	e
06	6	ACK	26	38	&	46	70	F	66	102	f
07	7	BEL	27	39	'	47	71	G	67	103	g
08	8	BS	28	40	(	48	72	H	68	104	h
09	9	HT	29	41	)	49	73	I	69	105	i
0A	10	LF	2A	42	*	4A	74	J	6A	106	j
0B	11	VT	2B	43	+	4B	75	K	6B	107	k
0C	12	FF	2C	44	,	4C	76	L	6C	108	l
0D	13	CR	2D	45	-	4D	77	M	6D	109	m
0E	14	SO	2E	46	.	4E	78	N	6E	110	n
0F	15	SI	2F	47	/	4F	79	O	6F	111	o
10	16	DLE	30	48	0	50	80	P	70	112	p
11	17	DC1	31	49	1	51	81	Q	71	113	q
12	18	DC2	32	50	2	52	82	R	72	114	r
13	19	DC3	33	51	3	53	83	S	73	115	s
14	20	DC4	34	52	4	54	84	T	74	116	t
15	21	NAK	35	53	5	55	85	U	75	117	u
16	22	SYN	36	54	6	56	86	V	76	118	v
17	23	ETB	37	55	7	57	87	W	77	119	w
18	24	CAN	38	56	8	58	88	X	78	120	x
19	25	EM	39	57	9	59	89	Y	79	121	y
1A	26	SUB	3A	58		5A	90	Z	7A	122	z
1B	27	ESC	3B	59		5B	91	[	7B	123	{
1C	28	FS	3C	60		5C	92	\	7C	124	
1D	29	GS	3D	61		5D	93	]	7D	125	
1E	30	RS	3E	62		5E	94	↑	7E	126	~
1F	31	VS	3F	63		5F	95	←	7F	127	DEL

NUL	- Null	DLE	- Data Link Escape
SOH	- Start of Heading	DC	- Device Control
STX	- Start of Text	NAK	- Negative Acknowledge
ETX	- End of Text	SYN	- Synchronous Idle
EOT	- End of Transmission	ETB	- End of Transmission Block
ENQ	- Enquiry	CAN	- Cancel
ACK	- Acknowledge	EM	- End of Medium
BEL	- Bell	SUB	- Substitute
BS	- Backspace	FSC	- Escape
HT	- Horizontal Tabulation	FS	- File Separator
LF	- Line Feed	GS	- Group Separator
VT	- Vertical Tabulation	RS	- Record Separator
FF	- Form Feed	US	- Unit Separator
CR	- Carriage Return	SP	- Space (Blank)
SO	- Shift Out	DEL	- Delete
SI	- Shift In		

APPENDIX I

FORTH STRING WORDS

This appendix defines FORTH words that can be created to handle character string data. The FORTH words defined are similiar to string handling functions provided in AIM 65/40 BASIC. The defined words are based on, and extend, functions described by Ralph Deane in an article entitled "A Proposal On Strings for FORTH," published in Dr. Dobbs Journal of Computer Calisthenics & Orthodonia, November/December 1980 (See Appendix N).

The following string handling words can be implemented using the colon-definitions listed in Table I-1:

FORTH Word	Function
STRING	Define a string
"	Enter text
S!	Store entire string
SUB	Substitute part of string
MID$	Get m characters of string
LEFT$	Get left-most n characters of string
RIGHT$	Get right-most n characters of string
VAL	Convert string to numeric value
STR$	Convert numeric to string
LEN	Get current length of string
MLEN	Get maximum length of string
S+	Add strings
S=	Compare strings

The easiest way to implement these functions is to enter the colon-definitions shown in Table I-1 into the AIM 65/40 Text Editor and to batch compile.

Table I-1. FORTH String Words

```
: SRCH
  DUP BEGIN DUP
  C@ SWAP 1+ SWAP
  Ø= END SWAP - 1- ;

: STRING
  <BUILDS ABS
  255 MIN 1 MAX DUP
  C,
  Ø DO 32 C, LOOP Ø C,
  DOES> 1+ DUP SRCH ;

  Ø VARIABLE IB
  254 ALLOT

: (")
  R COUNT DUP 1+
  R> + >R ;

: "
  34 STATE @ IF
  COMPILE (") WORD
  HERE C@ 1+ ALLOT
  ELSE WORD HERE COUNT
  IB SWAP ROT OVER IB
  SWAP 1+ CMOVE 2DUP
  + Ø SWAP C! THEN ;
  IMMEDIATE

: VAL
  OVER + BL SWAP
  C! 1- NUMBER ;

: STR$
  SWAP OVER DABS
  <# #S SIGN #> ;

: MLEN
  DROP 1- C@ ;

: S!
  DROP DUP 1- C@
  ROT MIN 1 MAX 2DUP
  + Ø SWAP C! CMOVE ;

: LEN
  SWAP DROP ;
```



```
: MID$
  SWAP >R ROT
  MIN 1 MAX SWAP OVER
  MAX OVER - 1+ SWAP
  R> + 1- SWAP OVER
  SRCH MIN ;

: LEFT$
  >R >R 1 SWAP
  R> R> MID$ ;

: RIGHT$
  >R >R 256
  R> R> MID$ ;

: S+
  ROT >R ROT R>
  SWAP OVER IB SWAP
  CMOVE SWAP OVER +
  255 MIN DUP >R OVER
  - SWAP IB + SWAP
  CMOVE R> Ø OVER IB
  + C! IB SWAP ;

: SUB
  ROT MIN 1 MAX
  CMOVE ;

: S=
  ROT OVER
  = IF 1 SWAP Ø DO
  DROP OVER
  C@ OVER C@ = IF 1+
  SWAP 1+ SWAP 1 ELSE
  Ø LEAVE THEN LOOP
  ELSE DROP Ø THEN
  SWAP
  DROP SWAP DROP ;
```

I.1 COMPILATION PROCEDURE

The procedure to enter and compile is

```
{E} (Enter from AIM 65/40 Monitor
EDIT FROM=2000 TO=4000 IN=<RETURN
: SRCH
  .
  .                        (Enter from Table I-1)
  .
: S= .... ;
." CR DONE"
FINIS
<RETURN>

*END*
={Q}

{5}
 AIM 65/40 FORTH V1.4
SOURCE <RETURN> IN=M
DONE
```


I.2 WORD DESCRIPTIONS

Each of the string words are described below. Note that there are two words, SRCH and (") , and a variable area, IB , that are used internally by the string functions and are not described.

STRING

STRING creates a word in the dictionary up to 255 characters. The string is initialized to all spaces with a zero at the end and the maximum length at the beginning. For example,

```
30 STRING A$
```

Creates a string named A$ which has room for 30 characters. When the name A$ is executed, the current length and the address of the text is put on the stack in the order required for the word TYPE .

"

" enters text into an intermediate buffer called IB , if used in the immediate mode. In the compile mode, the text is put into the dictionary. In either case the length and text address is left on the stack. Text is terminated by another " .

S!

S! moves the entire string text from one string to another, for example,

```
" COWS EAT CORN" A$ S!
```

puts the text "COWS EAT CORN" into the string A$.

Also as an example, define another string BEST and move A$ into it

```
40  STRING BEST
A$  BEST  S!
```

MID$

MID$ gets the m characters of a string starting at the nth character position, for example,

```
6  3  A$  MID$  TYPE
```

will print the word EAT .

LEFT$

LEFT$ gets the left-most n characters of a string, for example,

```
3  BEST  LEFT$  TYPE
```

will print the word COW .

RIGHT$

In like manner RIGHT$ gets the right-most n characters of a string. The sequence

```
10  A$  RIGHT$  BEST  S!
```

makes the string BEST now contain the word CORN verified by

```
BEST  TYPE
```

VAL

VAL converts a string to a double-precision number, for example,

```
" 128"  VAL  D.
```

gives

```
128
```

STR$

Conversely, STR$ converts a double-precision number into text. The sequence

```
567.  STR$  A$  S!
```

makes the string A$ equal to "567.".

LEN

LEN returns the current length of a string, such as

```
A$  LEN  .  <return> 3
```

MLEN

MLEN return th maximum length of a string, such as

```
A$ MLEN . <RETURN> 30
```

SUB

SUB allows substitution of characters in a string, for example,

```
" COWS EAT CORN" A$ S!
" ATE" 6 3 A$ MID$ SUB
```

replaces EAT with ATE in string A$.

S+

S+ adds strings together and puts the result in IB, for example,

```
" AND HAY" BEST S!
A$ BEST S+ BEST S!
```

adds BEST to A$. Verify by

```
BEST TYPE
```

and get

```
COWS EAT CORN AND HAY
```

S=

S= compares ings to see they are equal in length and te: If so, a s returned on the stack, else a

APPENDIX

USER 24-HOUR CLOCK PROGRAM IN FORTH

This appendix describes a 24-hour clock program written in FORTH using either machine level or interpretive interrupt handling. The 24-hour clock is maintained under interrupt control, using Timer 1 in the AIM 65/40 SBC Module User R6522 VIA. The program allows you to initialize the clock, enter a message that will be displayed with the time, and display the time either upon command or continuously.

HOW TO OPERATE THE PROGRAM

The 24-hour clock program is compiled into FORTH words as described in the next section. Once compiled you must be in FORTH to command the 24-hour clock functions. Once initiated however, the clock will continue to run as long as the User VIA is not reset, the User R6522 Timer 1 operating mode is no altered, or the IRQ Priority Latch mask (PRIRTY at address $FF80) is not altered to inhibit the $\overline{IRQ}$ interrupt from the User VIA ($\overline{IRQ3}$ on the AIM 65/40 SBC Module --refer to Section 2.7 in the AIM 65/40 System User's Manual).

Once FORTH has been entered, the program compiled and initialized with a time value, control may be returned to the AIM 65/40 Monitor. Be sure to re-enter FORTH with the 6 key, however, or the program will have to be re-compiled.

The 24-hour clock functions are entered from FORTH using any of four keys. These four keys are defined as FORTH words and are entered into the FORTH vocabulary. The keys, their functions and the associated operating procedure is:

M Key Allows a message of up to 30 characters to be displayed preceding the time value. Enter the message as follows

(1) Type M.

(2) Press <RETURN>.

(3) Type a message up to 30 characters long.

(4) Press <RETURN> (do not press <RETURN> if exactly 30 characters are entered). An example is:

```
M<RETURN> AIM 65/40 FORTH TIME
<RETURN>OK
```

T Key Allows the initial time value to be entered. Enter it as follows:

(1) Type in the time in the format HH.MM.SS (<u>not</u> HH:MM:SS).

(2) Press <SPACE>.

(3) Type T.

(4) Press <RETURN>.

For example:

```
16.05.00<SPACE>T<RETURN>OK
```

D Key Causes the message and time to be displayed (and printed if the printer is ON) once each time D is typed. The display format is:

```
<MESSAGE>HH:MM:SS
```

The time is displayed immediately after the message, for example,

```
D<RETURN>
AIM 65/40 FORTH TIME 16:05:10
```

The system remains in the FORTH command mode.

C Key Causes the message and time to be continuously displayed. For example,

```
C<RETURN>
FORTH TIME 16:05:30
```

Press a key to terminate the display (although the clock will continue to run). The key will also be interpreted as a FORTH command or data character.

J.2 HOW TO COMPILE THE PROGRAM

a. Load the program listed in Figure J-1 or Figure J-3 into the AIM 65 Text Editor and compile it. The program listed in Figure J-1 contains a machine level interrupt handler (see Section 7.2) while the program listed in Figure J-3 contains an interpretive interrupt handler (see Section 7.3). The load and compile procedure is:

```
{E}
EDIT FROM=2000 TO=3FFF IN=<RETURN>

( 24-HOUR CLOCK)
HEX
CODE DISABLE
  .
  .                          (Figure J-1 or J-3 Program)
  .
: D
  CR D QUIT ;
." CR DONE"
FINIS
<RETURN>

={Q}

{5}
 AIM 65/40 FORTH V1.4
SOURCE <RETURN> IN=M
D NOT UNIQUE
DONE
OK
```

b. Run a VLIST and verify that the compiled words are entered into the FORTH vocabulary as listed in Figure J-2 or J-4.

```
( 24-HOUR CLOCK USING IRQ INTERRUPTS )
HEX 022D CONSTANT UIRQAM
FEA0 CONSTANT IRQRTN
FFA4 CONSTANT UT1
FFAB CONSTANT UACR
FFAD CONSTANT UIFR
FFAE CONSTANT UIER
C34F CONSTANT PERIOD

0 VARIABLE DAY# ( 2 BYTES)
0 VARIABLE TICKS ( 4 BYTES) 0 ,

CODE DISABLE ( DISABLE USER VIA INT)
7F # LDA,
UIER STA,
NEXT JMP,
END-CODE
DISABLE

ASSEMBLER
HERE PHA, ( SAVE IRQ VECTOR)
CLC,
5 # LDA, ( 50 MS)
TICKS 3 + ADC,
TICKS 3 + STA,
64 # CMP, ( AT 100?)
 CS IF, ( >= 100)
 0 # LDA,
 TICKS 3 + STA,
 TICKS 2+ INC,
 TICKS 2+ LDA,
 3C # CMP,
  CS IF, ( >= 60)
  0 # LDA,
  TICKS 2+ STA,
  TICKS 1+ INC,
  TICKS 1+ LDA,
   3C # CMP,
   CS IF, ( >= 60)
   0 # LDA,
   TICKS 1+ STA,
   TICKS INC,
   TICKS LDA,
   18 # CMP,
    CS IF, ( >= 24)
    0 # LDA,
    TICKS STA,
    DAY# INC,
     0= IF,
     DAY# 1+ INC,
     THEN,
    THEN,
   THEN,
  THEN,
 THEN,
UIFR LDA, ( CLEAR USER VIA IRQ)
UIFR STA,
PLA,
IRQRTN JMP, ( RETURN TO I/O ROM)
```

Figure J-1. 24-Hour Clock Program Using a Machine Level Interrupt Handle


```
UIRQAM !   SET IRQ VECTOR)

FORTH
: INIT ( INITIALIZE THE USER VIA)
 40 UACR C! ( SET T1 FREE-RUN MODE)
 PERIOD UT1 ! ( LOAD T1 VALUE = 1/100 SEC
 C0 UIER C! ; ( ENABLE USER VIA T1 INT)

DECIMAL
: :DD ( TYPE M OR S)
 S->D <# # # 58 ( :) HOLD #> TYPE ;

: .T ( PRINT TIME)
 TICKS C@ ( HRS) 2 .R TICKS 1+ C@ ( M)
 :DD ( SAVE & PRINT SEC. )
 TICKS 2+ C@ DUP :DD
 BEGIN ( WAITING)
 TICKS 2+ C@
 OVER = NOT UNTIL DROP ;

: M ( ENTER 30 CHAR MESSAGE)
 PAD 1+ DUP 30 EXPECT PAD
 BEGIN
   1+ DUP C@ 0=
 UNTIL ( NULL FOUND)
 PAD 1+ - ( # OF CHARACTERS)
 PAD C! ( FOR TYPE) ;

: .M ( PRINT MESSAGE)
 PAD COUNT 30 MIN TYPE ;

: D
 DECIMAL .M .T ;

: T! ( SET TIME)
 100 U/ ( GET SEC)
 100 /MOD ( MIN HRS)
 TICKS C! ( LOAD HRS)
 TICKS 1+ C! ( MIN)
 TICKS 2+ C! ( & SEC)
 0 TICKS 3 + C! ( ZERO 100THS) ;

: T ( SET TIME & GO)
 T! INIT ;

: C ( CONTINUOUSLY DISPLAY MSG & TIME)
 BEGIN 24 EMIT ( BLANK CURSOR )
   13 EMIT ( STAY ON LINE) D ?TERMINAL
 UNTIL 23 EMIT ( RESTORE CURSOR ) QUIT

: D ( DISPLAY MSG & TIME ONCE)
 CR D QUIT ;

CR ." DONE
FINIS
 *END*
```

Figure J-1. 24-Hour Clock Program Using a Machine Level Interrupt Handler (Cont'd)

Figure J-2. VLIST of 24-Clock Program
Using a Machine Level Interrupt Handler


```
( 24-HOUR CLOCK USING FORTH INTERRUPTS
HEX 022D CONSTANT UIRQAM
FEA0 CONSTANT IRQRTN
FFA4 CONSTANT UT1
FFAB CONSTANT UACR
FFAD CONSTANT UIFR
FFAE CONSTANT UIER
C34F CONSTANT PERIOD

0 VARIABLE DAY# ( 2 BYTES)
0 VARIABLE TICKS ( 4 BYTES) 0 ,

CODE DISABLE ( DISABLE USER VIA INT)
7F # LDA,
UIER STA,
NEXT JMP,
END-CODE
DISABLE

( MACHINE CODE INTERRUPT SERVICE )
ASSEMBLER
HERE PHA, ( SAVE IRQ VECTOR)
80 # LDA, ( SET INT REQUEST)
INTFLAG ORA,
INTFLAG STA,
UIFR LDA, ( CLEAR USER VIA IRQ)
UIFR STA,
PLA,
IRQRTN JMP, ( RETURN TO I/O ROM)

CODE ARM ( RETURN FROM FORTH INTERRUPTS)
BF # LDA, ( RESET INT REQUEST BIT )
INTFLAG AND,
INTFLAG STA,
' ;S JMP, ( RESTORE INTERRUPTED IP )
END-CODE

UIRQAM ! ( SET IRQ VECTOR)

FORTH
: INIT ( INITIALIZE THE USER VIA)
 40 UACR C! ( SET T1 FREE-RUN MODE)
 C34F UT1 ! ( LOAD T1 VALUE = 1/100 SEC)
 C0 UIER C! ; ( ENABLE USER VIA T1 INT)

DECIMAL
: +!L ( INCREMENT / STORE / LIMIT CHECK)
 OVER +! ( ADD INC. )
 SWAP OVER C@ < DUP
 IF 0 ROT C!
   ELSE SWAP DROP
 THEN ;
```

Figure J-3. 24-Hour Clock Program
Using an Interpretive Interrupt Handler

```
: T+ ( FORTH LEVEL INTERRUPT SERVICE )
 99 TICKS 3 + 5 +!L ( 1/100 SEC COUNT)
 IF 59 TICKS 2+ 1 +!L ( SECONDS )
   IF 59 TICKS 1+ 1 +!L ( MINUTES )
     IF 23 TICKS 1 +!L ( HOURS )
       IF 23 TICKS 1 +!L ( HOURS )
       THEN
     THEN
   THEN
 THEN
 ARM [ SMUDGE  ( SIMILAR TO ; )

' T+ CFA ( PUT ADDRESS ON STACK )
ASSEMBLER INTVECT ! ( SAVE INT VECTOR)

FORTH
: :DD ( TYPE M OR S)
 S->D <# # # 58 ( : ) HOLD #> TYPE ;

: .T ( PRINT TIME)
 TICKS C@ ( HRS) 2 .R
 TICKS 1+ C@ :DD ( MIN)
 TICKS 2+ C@ DUP :DD ( SAVE & DISP SEC)
 BEGIN ( WAITING FOR A SECOND CHANGE)
   TICKS 2+ C@
   OVER = NOT
 UNTIL DROP ;

: M ( ENTER 30 CHAR MESSAGE)
 PAD 1+ DUP 30 EXPECT PAD
 BEGIN
   1+ DUP C@ 0=
 UNTIL ( NULL FOUND)
 PAD 1+ - ( # OF CHARACTERS)
 PAD C! ( FOR TYPE) ;

: .M ( PRINT MESSAGE)
 PAD COUNT 30 MIN TYPE ;

: D
 DECIMAL .M .T ;
```

Figure J-3. 24-Hour Clock Program
Using an Interpretive Interrupt Handler (Cont'd)


```
: T! ( SET TIME)
 100 U/ ( GET SEC)
 100 /MOD ( MIN HRS)
 TICKS C! ( LOAD HRS)
 TICKS 1+ C! ( LOAD MIN)
 TICKS 2+ C! ( LOAD SEC)
 0 TICKS 3 + C! ; ( ZERO 100THS SEC

: T ( SET TIME & GO)
 T! INIT ;

: C ( CONTINUOUSLY DISPLAY MSG & TIME)
 BEGIN  24 EMIT ( BLANK CURSOR )
   13 EMIT ( STAY ON SAME LINE) D ?TERMINAL
 UNTIL 23 EMIT ( RESTORE CURSOR ) QUIT ;

: D ( DISPLAY MSG & TIME ONCE)
 CR D QUIT ;

CR ." DONE"
FINIS
 *END*
```

Figure J-4. VLIST of 24-Hour Program
Using an Interpretive Interrupt Handler

APPENDIX

UTILITY EXAMPLES

K.1 MEASURING FORTH WORD EXECUTION TIME

It is often desired to know how long it takes for a FORTH word to execute, expecially in time critical aplications. The following words measure such execution time in AIM 65/40 clock cycles, i.e., microseconds.

```
HEX
: ON  FFFF  FFA4 ! ;
: OFF  FFA4 @ 12B + DUP CR
     IF FFFF DNEGATE D.
     ELSE .  THEN ;
```

The word ON initializes and starts Timer 1 in the AIM 65/40 SBC module User 6522 VIA. The word OFF displays the number of cycles elapsed from the start of the timer minus ON and OFF word overhead. Use these words as shown in the following colon-definition example to measure the execution time of a FORTH word, in this case DUP .

```
DECIMAL OK
: TDUP ON DUP OFF ;
OK
TDUP
76 OK
```

Using this technique, the execution time of most AIM 65/40 or other FORTH words defined using colon- or CODE-definitions can be measured. Set up and run similiar colon-definition words as needed for your application.

Many problems can be programmed in FORTH using different combinations of FORTH words with differing resultant execution speed. If speed is important, measure the execution time of each approach to decide which solution to use.

If the execution time of a FORTH word defined in high level, i.e., colon-definitions, is too long, redefine portions, or all, of the word in assembly code, i.e., colon-definitions, then remeasure. Comparing the execution time of the word defined in assembly code versus FORTH will show the performance improvement. For cases where the execution time exceeds the 16-bit counter capacity, other timing words can easily be defined to accumulate the time.

K.2 AIM 65/40 ROM CHECK-SUM PROGRAM

The object code bit pattern in the AIM 65/40 Monitor and FORTH ROMs (as well as other PROM/ROMs) can be easily verified by performing a check-sum on them using the FORTH word CHECK-SUM described below. The check-sum value is displayed on the second line.

a. Definition

```
AIM 65/40 FORTH V1.4
OK
: CHECK-SUM  ( ADDR COUNT ---.  32-bit CHECKSUM)
  0 S->D                       ( ACCUM.)
  ROT                          ( GET COUNT)
  0 DO                         ( COUNT TIMES)
  3 PICK                       ( GET BASE)
  I +                          ( FORM ADDR.)
  C@ S-.D                      ( 32 BITS)
  D+                           ( SUM IT)
  LOOP
  CR D.                        ( PRINT SUM)
  DROP ;                       ( BASE ADDR.)
```

b. Execution

```
HEX OK
A000 1000 CHECK-SUM            (R32U5-11 Monitor ROM)
7374D OK
B000 1000 CHECK-SUM            (R32U6-11 Monitor ROM)
6EA06 OK
C000 1000 CHECK-SUM            (R32P0-11 FORTH ROM)
82647 OK
D000 1000 CHECK-SUM            (R32P1-11 FORTH ROM)
8C1D3 OK
F000 0F7F CHECK SUM            (R32T3-12 I/O ROM)*
74508 OK
FFE0 001F CHECK-SUM            (R32T3-12 I/O ROM)*
117B OK
```

---

*Skip AIM 65 SBC Module I/O ($FF80-$FFDF)

APPENDIX L

AIM 65/40 FORTH VERSUS FIG-FORTH

This table is a comparison of AIM 65/40 FORTH V1.4 and the FIG-FORTH model from which it is derived.

a. Words in AIM 65/40 FORTH V1.4 that are not in FIG-FORTH 1.0:

Word Name

-CR
.S
1-
2-
2DROP
2DUP
4
?IN
?OUT
ASSEMBLER
BOUNDS
C/L
CLIT
CLOSE
CODE
CURRENT
DNEGATE
FINIS
FLUSH
GET
HANG
NEGATE
NOT
PICK
READ
PR@
SOURCE
U<
UABORT
UB/BUF
UB/SCR
UC/L
UEMIT
UFIRST
UKEY
ULIMIT
UR/W
WRITE

b. The following words are in FIG-FORTH 1.Ø but are not in AIM 65/4Ø FORTH V1.4 (however, some of the words are in the AIM 65/4Ø FORTH Assembler vocabulary):

Word Name	Where Used	Comment
+ORIGIN	system	
?LOADING	system	
BACK	system	
BLOCK-READ	user disk word	
BLOCK-WRITE	user disk word	
DLIST	duplicate name	(VLIST)
DMINUS	new name	(DNEGATE)
DRØ	disk	
DR1	disk	
FLD	not used	
INDEX	disk	
LIST	disk	
MINUS	new name	(NEGATE)
MOVE	N/A	(word addressing computers)
NEXT	AIM 65/4Ø FORTH Assembler	
OUT	not used	
POP	AIM 65/4Ø FORTH Assembler	
PUSH	AIM 65/4Ø FORTH Assembler	
PUT	AIM 65/4Ø FORTH Assembler	
R#	system	
TRAVERSE	system	
TRIAD	disk	
X	system	(null)

APPENDIX M

FORTH AND THE RM 65 FDC MODULE

This appendix describes the actual code used to interface the RM 65 FDC module with AIM 65/4Ø FORTH. This example uses a single 5-inch disk drive with one side and double-density recording and operates with the RM 65 FDC module PROM R32N5 (dated 1/4/82). The example is entered into the Text Editor and compiled using the SOURCE word. For a detailed description of the code words, refer to Section 12. There are seven major words created which supplement the use of the FDC module:

INIT

Initializes the FDC module and turns ON drive one, side one, in single-density mode.

MOTORON

Turns ON the drive from SRCDSK, side from SRCSID, and density from SRCDEN.

MOTOROFF

Turns OFF the selected drive

FORMAT

Initializes the disk in the selected drive. A formatted disk will have all sectors filled with $E5 which on the AIM 65/4Ø microcomputer is displayed as a blinking "E.", and printed as "e".

WIPE n ---

Clears screen n by filling it with null characters ($ØØ).

LIST n ---

Lists screen n as 16 numbered lines (Ø to F) with 64 characters each.

EDIT n ---

Places the text following EDIT (up to 64 characters) into line n on the current screen (i.e., the last screen accessed with LIST or WIPE).

Other words that are present in FORTH and are also useful in the creation and execution of source code include:

LOAD n ---

Compiles source code into the dictionary starting at screen n , line 0 and continuing until a ;S is encountered. The ;S should be within four lines of the last line of source code. More than one sequential screen is loaded by using --> to point to the next screen.

EMPTY-E FFERS

Marks ll RAM block buffers as empty and fills them with lls ($00).

FLUSH

Writes out any updated RAM block buffers to the disk.

Figure M-1 lists the FDC interface program in FORTH.


```
( AIM 65/40-FORTH  DOUBLE DENSITY DISK ROUTINES )
HEX FORGET TASK ( FDC RAM $4A0-$55F )
1 CONSTANT S# ( ONLY 1 SCREEN NEEDED )
100 UB/BUF ! ( 256 FOR DOUBLE DENSITY )
4 UB/SCR !  ( 4  FOR DOUBLE  DENSITY )
LIMIT B/BUF 4 + B/SCR * S# * - UFIRST ! ( TOP OF RAM )
0 OFFSET ! ( OFFSET NOT NEEDED WITH 1 DRIVE)
FIRST DUP USE ! PREV !  ( SET UP FIRST BUFFER )
EMPTY-BUFFERS ( CLEAR OUT THE BUFFER AREA )
CODE INIT1 XSAVE STX, 8000 JSR, ( CALL INIT )
 ( SET DRIVE PARAMETERS IN SRCDRV, SRCSID, & SRCDEN )
 0 # LDA, 4A5 STA, ( DRIVE ONE INTO SRCDRV )
 0 # LDY, 4AF STY, ( SIDE ONE INTO SRCSID )
 0 # LDX,  4B1 STX, ( DOUBLE DENSITY INTO SRCDEN  )
 83E0 JSR,  ( MOTON ) XSAVE LDX, NEXT JMP, END-CODE
: INIT FD46 4FB ! ( UIRQBM > IRQOUT ) 83C9 22B
 !  ( SET UP IRQHAN> INIT1 ;
: SIZEOK OVER 230 < ; ( 16 SECTOR * 35 TRACK )
: BBUF DUP 4F1 ( RDBUF ) ! 4F3 ( WRTBUF ) ! ;
: T&S SWAP 10 /MOD ; ( LEAVE TRACK & SECTOR )
CODE SEEK XSAVE STX, TOP LDA, 8104 JSR, ( CALL SEEK )
 XSAVE LDX, 99 # AND, PUSH0A JMP, END-CODE
CODE DREAD XSAVE STX, TOP LDA, 84BF JSR, ( CALL RDSEC )
 XSAVE LDX, BD # AND, PUSH0A JMP, END-CODE
CODE DWRITE XSAVE STX, TOP LDA, 8500 JSR, ( CALL WRTSEC
 XSAVE LDX, FD # AND, PUSH0A JMP, END-CODE
: INTDIS FF FF80 C! ( MASK OUT ALL IRQ BUT FDC ) ;
: INTENB 00 FF80 C! ( RESTORE THE IRQ MASK ) ;
: DERROR INTENB CR ." DISK ERROR - " ; ( RECOVER & PRINT
: DATA ( FETCH A BYTE ) ROT BBUF T&S SEEK DUP
  IF DERROR ." SEEK A=" .  ( SEEK ERROR )  ELSE DROP
    THEN DROP 1+ ( SECTOR 1 TO 16 ) SWAP
  IF DREAD DUP   IF DERROR ." READ A=" . ( READ ERROR )
      ELSE DROP   THEN ( DO NOTHING )
    ELSE DWRITE DUP   IF DERROR ." WRITE A=" . ( ERROR )
      ELSE DROP   THEN ( DO NOTHING )   THEN  DROP ;
: DISK SIZEOK   IF INTDIS DATA INTENB
   ELSE CR ." BLOCK TOO LARGE ERROR " ABORT     THEN ;
' DISK CFA UR/W ! ( STORE INTERFACE WORD )
( UTILITIES THAT MUST BE AVAILABLE TO USER )
CODE FORMAT XSAVE STX, 8095 JSR, ( CALL FORMAT )
 XSAVE LDX, NEXT JMP, END-CODE
CODE MOTOROFF XSAVE STX, 8478 JSR, ( CALL MOTOFF )
 XSAVE LDX, NEXT JMP, END-CODE
CODE MOTORON XSAVE STX, 8478 JSR, ( CALL MOTOFF )
 4A5 LDA, ( SRCDSK ) 4AF LDY, ( SRCSID )
 4B1 LDX, ( SRCDEN ) 83E0 JSR, ( CALL MOTON )
 XSAVE LDX, NEXT JMP, END-CODE
: EDIT SCR @  (LINE)  OVER SWAP BLANKS ( CLEAR OUT LINE
 0 WORD ( PARSE TEXT ) HERE COUNT 40 MIN ( 64 CHAR LINES
 ROT SWAP CMOVE ( MOVE TEXT ) UPDATE ( MARK BUFFER ) ;
: LIST  DUP CR ." SCR # " . ( PRINT SCREEN AND SAVE ) SCR !
 10 0 DO CR I 3 .R SPACE I SCR @  .LINE  LOOP  CR ;
: WIPE  B/SCR * B/SCR BOUNDS ( SCREEN # TO BLOCK RANGE )
 DO  I BLOCK  B/BUF BLANKS UPDATE  LOOP  FLUSH ;
: TASK ; ( THROUGH WITH CODE ) ( FINIS )
```

Figure M-1. AIM 65/40 FORTH Floppy Disk Example

APPENDIX N

SELECTED BIBLIOGRAPH

Anderson, A. and Wasson, P. FORTH-79 Tutorial and Reference Manual, MicroMotion, 12077 Wilshire Blvd, Suite 506, West Los Angeles, CA, February 1981.

Bartoldi, P, "Stepwise Development and Debugging Using a Small Well-Structured Interactive Language for Data Acquisition and Instrument Control," Proceedings of the International Symposium and Course on Mini and Microcomputers and their Applications.

Brodie, L., "Starting FORTH" Prentice-Hal Englewood Cliffs, N.J., 1981.

Cassady, J. J., "Stacking Strings in FORTH", BYTE, February 1981, pages 152-162.

Deane, R., "A Proposal on Strings for FORTH", Dr. Dobb's Journal of Computer Calisthenics & Orthodontia, November/December 1980, pages 40-43.

Dessey, R. and M. K. Starling, "Forth Generation languages for Laboratory Applications", American Laboratory, February 1980, pages 21-36.

Ewing, M. S., The Caltech FORTH Manual, California Institute of Technology, Pasadena CA, 1978.

Ewing, M. S., and W. H. Hammond, "The FORTH Programming System," Proceedings of the Digital Equipment Computer Users Society (DECUS), San Diego, CA, November 1974, page 477.

FORTH Interest Group, fig-FORTH Installation Manual Glossary Model", May 1979, Box 1105, San Carlos, CA, 94070.

FORTH Interest Group, "FORTH Dimensions" a bimonthly newsletter, c/o FORTH Interest Group.

Harris, K., "FORTH Extensibility or How to Write a Compiler in 25 Words or Less", BYTE, August 1980, pages 164 - 184.

Hicks, S. M., "FORTH's Forte is Tighter Programming", Electronics, March 15, 1979, pages 115-118.

James, J. S., "FORTH for Micro Computers", Dr. Dobb's Journal of Computer Calisthenics & Orthodontia, May 1978; also in ACM SIGPLAN Notices, October 1978.

James, J. S., "What Is FORTH? A Tutorial Introduction", BYTE, August 1980, pages 100-126.

Mannoni, M., "FORTH - An Extensible Path to Efficient Programs", Electronic Design, July 19, 1980, pages 175-178

Moore, C. H., "FORTH: a New Way to Program a Minicomputer", Astronomy and Astrophysics Supplement, 1974, number 15, pages 497-511.

Phillips, J. B. "Threaded Code for Laboratory Computers", Software Practice and Experience, Vol. 8, 1978, pages 257-263.

Rather, E. D., and C. H. Moore, "The FORTH Approach to Operating Systems", ACM 1976 Proceedings, Association for Computing Machinery, 1976.

Rather, E. D., and C. H. Moore, and J. M. Hollis, "Basic Principles of FORTH Language as Applied to a PDP-11 Computer", Computer Division Internal Report No. 17, National Radio Astronomy Observatory, Charlottesville, VA; Kitt Peak National Observatory, Tucson, AZ, March 1974.

ELECTRONIC DEVICES DIVISION REGIONAL ROCKWELL SALES OFFICES

HOME OFFICE

Electronic Devices Division
Rockwell International
3310 Miraloma Avenue
P.O. Box 3669
Anaheim, California 92803
(714) 632-3729
TWX 910 591-1698

UNITED STATES

Electronic Devices Division
Rockwell International
1842 Reynolds
Irvine, California 92714
(714) 632-3710
TWX 910 505-2518

Electronic Devices Division
Rockwell International
921 Bowser Road
Richardson, Texas 75080
(214) 996-6500
Telex 73-307

Electronic Devices Division
Rockwell International
10700 West Higgins Rd., Suite 102
Rosemont, Illinois 60018
(312) 297-8862
TWX 910 233-0179 (RI MED ROSM)

Electronic Devices Division
Rockwell International
5001B Greentree
Executive Campus, Rt. 73
Marlton, New Jersey 08053
(609) 596-0090
TWX 710 940-1377

EUROPE

Electronic Devices Division
Rockwell International GmbH
Fraunhoferstrasse 11
D-8033 Munchen-Martinsried
West Germany
(089) 859-9575
Telex 0521/2650 rimd d

Electronic Devices Division
Rockwell International
Heathrow House, Bath Rd.
Cranford, Hounslow,
Middlesex, England
(01) 759-9911 Ext. 35
Telex 851-25463

Electronic Devices Division
Rockwell International
Via Mac Mahon 50
20155 Milano
Italy
0039/2/389125
Telex 331212 intrad I

FAR EAST

Electronic Devices Division
Rockwell International Overseas Corp.
Itohpia Hirakawa-cho Bldg.
7-6, 2-chome, Hirakawa-cho
Chiyoda-ku, Tokyo 102, Japan
(03) 265-8806
Telex J22198

Rockwell International