FORTH/APPLE-II-PLUS USER MANUAL

With Applications to Data Sampling and Processing

by

T. K. STANTON
Project Associate
Marine Studies Center
University of Wisconsin

LEE POWELL
Electrical Engineer
Geophysical and Polar Research Center
University of Wisconsin

WISCONSIN REPORT NO. 82-1
ABSTRACT

This manual is written to aid the beginner in using the Forth Programming Language with the Apple-II-Plus Computer. Special attention is focused toward a low-cost data sampling and processing system. The version of Forth described was originally written by William Graves of Softtape and was modified with additions by Lee Powell, U.W. Along with many examples, the manual explains file manipulations, number manipulations and the programming language. Finally, a data sampling-processing system that employs the Apple Computer and Forth Language is described. In this system, integration of assembler coding with the Forth Language provides a sampling rate of up to 26 kHz.

Research made possible through The Office of Naval Research
Contract N00014-79-C-0703
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.</td>
<td>Introduction</td>
<td>i</td>
</tr>
<tr>
<td>0.1</td>
<td>This Manual</td>
<td>i</td>
</tr>
<tr>
<td>0.2</td>
<td>Forth Language</td>
<td>i</td>
</tr>
<tr>
<td>0.2.1</td>
<td>General</td>
<td>i</td>
</tr>
<tr>
<td>0.2.2</td>
<td>This Version of Forth</td>
<td>ii</td>
</tr>
<tr>
<td>0.2.3</td>
<td>This Apple</td>
<td>ii</td>
</tr>
<tr>
<td>1.</td>
<td>File Manipulations</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Top Level Flow</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>File Transfer Commands—Summary</td>
<td>1</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Program (File) Structure—General Remarks</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>File Transfer Commands—Individual Paths</td>
<td>7</td>
</tr>
<tr>
<td>1.3.1</td>
<td>To Boot—Path 1</td>
<td>7</td>
</tr>
<tr>
<td>1.3.2</td>
<td>Creating Programs</td>
<td>8</td>
</tr>
<tr>
<td>1.3.2.1</td>
<td>Introduction</td>
<td>8</td>
</tr>
<tr>
<td>1.3.2.2</td>
<td>Creating &quot;Permanent&quot; Programs—Path 2</td>
<td>8</td>
</tr>
<tr>
<td>1.3.2.3</td>
<td>Creating &quot;Temporary&quot; Programs—Path 8</td>
<td>9</td>
</tr>
<tr>
<td>1.3.3</td>
<td>Storing Programs—Path 4</td>
<td>11</td>
</tr>
<tr>
<td>1.3.4</td>
<td>Modifying Program Stored In Disk—Path 3</td>
<td>12</td>
</tr>
<tr>
<td>1.3.5</td>
<td>Loading Program From Buffer—Path 5</td>
<td>12</td>
</tr>
<tr>
<td>1.3.6</td>
<td>Path 7</td>
<td>13</td>
</tr>
<tr>
<td>1.3.7</td>
<td>Loading Program From Disc For Immediate Execution—Path 6</td>
<td>13</td>
</tr>
<tr>
<td>1.3.8</td>
<td>Using Only The Existing Dictionary—Path 8</td>
<td>13</td>
</tr>
<tr>
<td>1.3.9</td>
<td>Execute Verbs—Path 9</td>
<td>14</td>
</tr>
</tbody>
</table>
1.3.10 Leaving System ................................................. 14
1.3.11 Example Paths ................................................. 15
1.4 Editor .......................................................... 21
1.5 Two-Drive System .............................................. 21
1.5.1 Directives .................................................... 21
1.5.2 Transferring Files Between Drives ......................... 22
1.6 Useful Keyboard Commands .................................... 22

2. Number Manipulations .......................................... 24
2.1 Introduction ................................................... 24
2.2 Number/Address Transfer Flow ................................ 24
2.3 Number/Address Transfer Commands ......................... 24
2.4 Examples ........................................................ 28

3. Program Language .............................................. 29
3.1 Introduction ................................................... 29
3.2 Structure ....................................................... 29
3.3 Versatility in Writing ....................................... 29
3.4 Comments ........................................................ 30
3.4.1 One-Line .................................................... 30
3.4.2 Multi-Line ................................................... 30
3.5 Mathematical Operations ...................................... 31
3.5.1 With Output ................................................ 31
3.5.2 With Input and Output .................................... 31
3.5.3 Data Stack Usage ........................................... 33
3.6 Alphanumeric (String) Output ................................ 33
3.7 Keyboard I/O--More Versatile Usages ...................... 34
3.7.1 Introduction ................................................ 34
3.7.2 Prompt-Entering Numbers to Stack ....................... 34
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.7.3</td>
<td>Prompt-Assigning Addresses to Input Numbers</td>
<td>35</td>
</tr>
<tr>
<td>3.8</td>
<td>Conditional Statements</td>
<td>36</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Introduction</td>
<td>36</td>
</tr>
<tr>
<td>3.8.2</td>
<td>Do-Loop</td>
<td>36</td>
</tr>
<tr>
<td>3.8.2.1</td>
<td>Single</td>
<td>36</td>
</tr>
<tr>
<td>3.8.2.2</td>
<td>Non-Unity Index Increments</td>
<td>37</td>
</tr>
<tr>
<td>3.8.2.3</td>
<td>Nested</td>
<td>37</td>
</tr>
<tr>
<td>3.8.2.4</td>
<td>Leaving Loop Prematurely</td>
<td>39</td>
</tr>
<tr>
<td>3.8.3</td>
<td>Begin-End Loop</td>
<td>40</td>
</tr>
<tr>
<td>3.8.3.1</td>
<td>Finite Loop</td>
<td>40</td>
</tr>
<tr>
<td>3.8.3.2</td>
<td>Infinite Loop</td>
<td>41</td>
</tr>
<tr>
<td>3.8.4</td>
<td>If-Else-Then</td>
<td>41</td>
</tr>
<tr>
<td>3.9</td>
<td>Arrays</td>
<td>42</td>
</tr>
<tr>
<td>3.9.1</td>
<td>Introduction</td>
<td>42</td>
</tr>
<tr>
<td>3.9.2</td>
<td>DIM—Multidimensional, Slow</td>
<td>42</td>
</tr>
<tr>
<td>3.9.2.1</td>
<td>Elementary Setup</td>
<td>42</td>
</tr>
<tr>
<td>3.9.2.2</td>
<td>Automatic Setup</td>
<td>43</td>
</tr>
<tr>
<td>3.9.3</td>
<td>One Dimension—Fast</td>
<td>43</td>
</tr>
<tr>
<td>3.9.4</td>
<td>More Array Examples</td>
<td>44</td>
</tr>
<tr>
<td>3.10</td>
<td>Comparison Verbs</td>
<td>44</td>
</tr>
<tr>
<td>3.11</td>
<td>A/D and D/A Conversions</td>
<td>45</td>
</tr>
<tr>
<td>3.11.1</td>
<td>Introduction</td>
<td>45</td>
</tr>
<tr>
<td>3.11.2</td>
<td>Channel Addresses</td>
<td>45</td>
</tr>
<tr>
<td>3.11.3</td>
<td>A/D Converter</td>
<td>45</td>
</tr>
<tr>
<td>3.11.3.1</td>
<td>Sampling a Channel</td>
<td>45</td>
</tr>
<tr>
<td>3.11.3.2</td>
<td>Sampling/Printing Continuously</td>
<td>46</td>
</tr>
<tr>
<td>3.11.3.3</td>
<td>Sample-Store</td>
<td>47</td>
</tr>
</tbody>
</table>
FIGURES

Figure 1. Flow of File ........................................... 2
Figure 2. File Transfer Commands—Flow Diagram .................. 3
Figure 3. Flow Chart of File Transfer Commands .................. 4
Figure 4. Number/Address Transfer Flow .......................... 25
Figure 5. Number/Address Transfer Commands .................... 26
Sec. 3.14.2 Sonar Echo Recording Setup .......................... 57
Figure 6. Preprocessing Setup .................................... 59
Sec. 3.14.3.1 Preprocessing Signals ............................... 60
Figure 7. Test Setup .............................................. 62
Figure 8. Flow Chart .............................................. 63
0. **INTRODUCTION**

0.1 **This Manual**

This manual is written to aid the beginner in using the FORTH programming language on the APPLE-II-Plus computer (disk drive). The manual is by no means a comprehensive presentation of either the FORTH language or the APPLE computer but should serve as a tool for one who wishes to learn the basic building blocks of the system.

This manual is divided into three major sections—

1) File manipulations
2) Number manipulations
3) Programming language

Sections 1) and 2) are effectively summarized in flow diagrams on Figures 2 and 3 and in sections 2.2 and 2.3.

It is expected that once this manual is read once, subsequent references to it by the user will be made to those two figures and two sections as well as key examples in the sections 1.3.11, 2.4, and 3.

0.2 **Forth Language**

0.2.1 **General**

The FORTH language is a so-called "low level"-"high level" language—i.e. it is indeed a high level language, but at a lower level than languages such as BASIC, FORTRAN or PASCAL. Its structure and commands are similar to the ASSEMBLER language.

The advantages of this language are that it is fast, structured, and easily interfaceable with ASSEMBLER. Its stack-oriented
organization is also quite effective in certain applications.

The major disadvantage of this language is that since it is at a relatively low level, the programming of the language tends to be awkward—especially for the beginner. However, when computational speed is of the essence, this disadvantage is offset.

0.2.2 This Version of FORTH

This version of the FORTH language was originally written by William Graves of Softape and its capabilities were greatly increased with additions by Lee Powell, University of Wisconsin. The original version is presented in the FORTH Reference Manual written by Graves. One of the major modifications made by Powell was the addition of the ASSEMBLER language to the FORTH operating system.

0.2.3 This APPLE

Type: APPLE -II-Plus
Made by: Apple Computer, Inc.
Cards: Language (Apple Computer, Inc.)
A/D and D/A (Mountain Hardware)
Disk Drive (Apple Computer, Inc.)
I/O devices: CRT Monitor (Sony)
2 Disk Drives (Apple Computer, Inc.)
A/D and D/A Channel Access Box (Built at UW)
Disk: FORTH Operating System created by William Graves, Softtape.
Note: Although the above system is described in this manual, one could still use FORTH on this computer by using only 1 disk drive, its associated card, a monitor (or printer) and the FORTH disk.
1 FILE MANIPULATIONS

1.1 Top Level Flow

In creating, modifying, storing, coding and executing files, the files must be manipulated between and/or within three major locations:

1) BUFFER- Used primarily for editing.

2) DICTIONARY (DICT)- Contains compiled verbs ready for execution. A set of FORTH verbs always exists in the DICT. Additional verbs are added temporarily by the user in the context of programs.

3) DISC- Stores accessible copy of all desired files.

Figure 1 illustrates the flow of files between these locations.

1.2 File Transfer Commands—Summary

File flow can be controlled via certain file transfer commands. These commands are given in Figure 2.

Figure 3 represents a flow chart which summarizes most typical file manipulations. Branch (decision) points are included with common paths that can be followed.
Figure 1. FLOW OF FILE
Figure 2. FILE TRANSFER COMMANDS--FLOW DIAGRAM

- "FILENAME" is the name of the file
- The character "_" denotes a required space
Figure 3. FLOW CHART OF FILE TRANSFER COMMANDS

1. **BOOT DISC**
   - 'Put (or leave) FORTH disc in Drive 1
     (Door closed) Turn on computer (or off then on)

2. (Create "permanent" program for possible disc storage)

3. (Modifying program stored in disc)

4. (Store new program on disc)

5. **LOAD**
   - (Compile program for execution)

6. (Load disc-stored program for immediate execution)

7. (Create and execute "temporary" programs or execute permanent dictionary words)

8. **CREATE TEMPORARY PROGRAM?**
   - Yes
     - **EXECUTE VERBNAME**
     - Yes
     - **SAVE DISC IN DRIVE**
     - No
   - No
     - **BACK TO OTHER NODE?**
     - Yes
     - **EXECUTE VERBNAME**
     - No

9. **VERBNAME**
   - (Execute verb)

10. **LEAVE COMPUTER ON**
    - Yes
    - **SAVE DISC IN DRIVE**
    - No

Note: _fn means FILENAME
     _ denotes required space
1.2.1 Program (File) Structure--General Remarks

Before learning about the control commands, it will be useful to understand a few things about the program structure. To each program (or file) is associated a Filename. This name serves as an address when the program is read, written, or compiled. Each program can contain one or more Verbs with appropriate variable and constant definitions and control statements outside the verbs. Once the program is compiled, it is each individual verb that is executed, not the entire program. Thus transfer statements apply to files, execution statements apply to verbs. A typical file structure is as follows:

VARIABLE, CONSTANT DEFINITIONS
CONTROL STATEMENTS (eg, BLOAD_fn)
COMMENTS

:_VERBNAME 1
  .
  .
  .
  (Action--can use Dict verbs, previously defined verbs only.)
FILE
CALLED
  .
FILENAME
  ;

:_VERBNAME 2
  .
  .
  .
  (Action--can use dict verbs, previously defined verbs, and Verb 1 only.)
  .
  .
  ;

:_VERBNAME N
  .
  .
(Action—can use dict verbs, previously defined verbs, and Verbs 1, 2, ..., N only.)

Note: _ means space

"_"VERBNAME N . . . (Action) . . . ; defines Verb N

where ":" begins compilation of the verb and ";" ends compilation.

(: and ; are system verbs.)

Note also that each verb can only use verbs previously defined in the text (the compiler works top down). Thus in very general terms, Verbs 1 through N-1 can be looked at as subroutines to Verb N.
1.3 File Transfer Commands—Individual Paths

1.3.1 To Boot—Path 1

1) **INSERT** (or if already in, **LEAVE**) **FORTH DISC** in Disk Drive 1. The disc goes in with the label on the top side—the label end goes in last.

2) **CLOSE** (or if already closed, **LEAVE CLOSED**) drive door.

3) **TURN ON APPLE** (Booting)
   
   —or—

   if it was already on, type

   6_PR#

   If that doesn't work, (say, if the system is "crashed") then turn it off then back on again.

4) Wait until the drive light is off and the "#" prompt appears which indicates the system was properly booted.
1.3.2 Creating Programs

1.3.2.1 Introduction

Two types of programs can be created: "PERMANENT" and TEMPORARY. The "permanent" type is one that can be created, edited, compiled, executed, and stored on disc if desired. The temporary can only be created, compiled, and executed. It can not be stored on disc or edited with the editor. The advantage of the latter-mentioned program is when a short program or single algorithm is to be tested, it is quite often convenient to do it in the temporary program mode.

1.3.2.2 Creating "Permanent" Programs--Path 2

This is done in the BUFFER using the EDITOR. To create a "permanent" program, first clear the buffer by typing

CLR

Then type

ADD

The commands allow you to ADD lines to the end of whatever is in the buffer. Since in this case the buffer is cleared, the first line entered after the ADD statement is assigned line #1.

Here is an example program to enter after ADD has been typed:
Note: as previously mentioned, a _ means space

Jumping ahead: To execute the above program, type

LOAD

then

TEST

the phrase THIS IS A TEST should then be printed.

1.3.2.3 Creating "Temporary" Programs--Path 8

Any "temporary" programs will remain stored in the dictionary until "forgotten" or the computer is shut off. To create a "temporary" program, simply type it in. The editor and buffer are not used in this case. As every line is entered, it will be compiled. So syntax errors are detected immediately. If an error is in a line and the computer detects it, your previous creating work will be erased and you will have to start over. Therefore be careful in entering each line.
To enter the program discussed in an earlier section, type

: _TEST (CR)
"_THIS IS A TEST" (CR)
S. (CR)
( CR)

Jumping ahead: to execute this program type

TEST

and as before, the phrase

THIS IS A TEST

should be printed.
1.3.3 Storing Programs--Path 4

Once it is decided that the program in the buffer should be stored on the disc type (after the prompt #)

```
DWRITE_fn
```

This statement transfers all of the information from the buffer to the disc and entitles it "fn" (filename). The stored information is not compiled. Note the distinction between a filename and a verbname (see program [file] structure--general section 1.2.1). To make sure it has been transferred just type

```
DOS: _CATALOG
```

which gives a listing of all filenames on the disc.
1.3.4 Modifying Program Stored in Disc--Path 3

To modify a program that is already stored on disc, first you must transfer that program from the disc to the buffer area. This is done by typing

\[ \text{DREAD}\_fn \]

You may want to first list the program that is done by typing the editor command

\[ \text{L}\_n\_m \]

Where \( n \) and \( m \) define the first and last lines, respectively, you wish to see (note: \( m-n \) should be \( \leq 2\phi \)).

Then you may want to use commands involving deleting, adding, inserting or editing lines. These commands are described in the editor section.

1.3.5 Loading Program from Buffer--Path 5

Once a program is written in the Buffer area, you may want to execute it. This is done by first compiling it and loading the compiled version into the dictionary. All of this is done simply by typing

\[ \text{LOAD} \]

If there is a syntax error, you will see an appropriate error message. Then to proceed, you must go back and edit the program and then load again.

Once there is no error message after loading, you can then execute it (see appropriate section).
1.3.6 **Path 7**

This path is taken when the user wishes to:

1) Create and execute "temporary" programs
2) Execute permanent dictionary verbs.

This is the most direct path to take after the system is booted. The user works directly out of the dictionary with the FORTH verbs as well as possibly some temporary verbs created.

1.3.7 **Loading Program from Disc for Immediate Execution--Path 6**

This path represents the case in which there is a file or disc that is already written and you wish to execute it without making any changes to it. The process first involves reading the file, compiling it, and writing it into the dictionary. This all is done in the command:

```
DLOAD_FILENAME
```

If the file compiles successfully, you may then execute the program (Path 7). If not, you need to take another path (3) for debugging.

1.3.8 **Using Only the Existing Dictionary--Path 8**

This path is taken when you want to get right to the dictionary and practice with the already existing verbs and/or write a "temporary" verb (see Path 7 for description of "temporary" verb) for execution.
1.3.9 **Execute Verbs--Path 9**

This path is taken once you are ready to execute verb(s) in the dictionary. These can be verbs you have created and compiled successfully and/or the permanent verbs that are part of the FORTH Operating System.

To execute the verb(s) type:

```
VERBNAME N_VERBNAME (N-1) - - - - -VERBNAME 2_VERBNAME 1
```

or

```
VERBNAME N (CR)
VERBNAME(N-1) (CR)

VERBNAME 1 (CR)
```

The verbs are numbered in reverse order to illustrate the order in which they probably should appear in the file (i.e. VERBNAME 1 first, etc.). Each verb can only use verbs previously compiled; therefore, Verb N can use Verbs 1 - (N-1) and so on.

1.3.10 **Leaving System--Path 10**

When leaving the computer, you may want to leave it turned on. If this is the case, leave a disc in the drive to prevent the drive mechanism from possibly running freely.
1.3.11 Example Paths

This section illustrates the following set of paths:

- Creating "permanent" file
- Storing it on disc
- Executing it
- Modifying file from disc
- Executing new version
- Storing new version on disc
- Creating "temporary" verb
- Executing "temporary" verb
- Using only the existing dictionary

First boot the computer as described in section 1.3.1.

CREATING "PERMANENT" FILE

Type

CLR

then

ADD

Now the prompts will be line numbers.

Type in

1: _TEST (CR)

2"_THIS IS A TEST"_S. (CR)

3; (CR)

4 (CR)

This is a 3-line file.

If a mistake was made in typing, see Section 1.4 (Editor).
STORING IT ON DISC

To store this file on disc under Filename "EXAMPLE1", type

DWRITE_EXAMPLE1

Note that now the file exists both in the Buffer and on disc. This file contains the verb named "TEST".

EXECUTING IT

To execute it, first compile it and load it into the dictionary by typing either

LOAD (from Buffer)

-or-

DLOAD_EXAMPLE1 (from disc)

ASIDE

Note that the verb "TEST" is now at the top of the dictionary. Type

LIST

to verify that and type

CR

several times to get to the bottom of the list.

To execute, type

TEST

The computer should respond with the message

THIS IS A TEST
MODIFYING FILE FROM DISK

At this point, since the file EXAMPLE1 is both in the Buffer and on Disk, it can be modified via two approaches—either going directly back to the Buffer and modifying it or by transferring the file from the Disk to the Buffer and then modifying it. For the sake of generality, the latter case will be done.

First type:

DREAD_EXAMPLE1

Note that this erases what was previously in the Buffer. To edit line 2, type

ED_2

Now push the → button until the cursor is over the A, now type

STILL A TEST"_S. (CR)

ASIDE

To verify the modification, type

L_1_1Φ

And the computer should display

1:_TEST

2"_THIS IS STILL A TEST"_S.

3;

If a mistake was made in typing, see Section 1.4 (Editor).
EXECUTING NEW VERSION

To execute this version, first compile it by typing

LOAD

ASIDE

It should now be on top of the dictionary list. To verify this type

LIST

And there should be two TESTs at the top—the most recently compiled
one at the very top.

To execute it, type

TEST

And the computer should respond with

THIS IS STILL A TEST

ASIDE

if you type

TEST

again, the computer should respond the same way.

Now type

FORGET_TEST
This removed the top test from the dictionary. If there were a verb(s) with a different name(s) on top of the two tests, it (they) would have been removed also by this command.

Now type

TEST

The computer should respond with

THIS IS A TEST

STORING NEW VERSION ON DISK

While all of the above commands have been issued, the modified file has remained in the buffer, to store it on the disk, type

DWRITE_EXAMPLE1

-or-

DWRITE_EXAMPLE2

depending on whether you wish to write over the old file EXAMPLE1 or create a new file named EXAMPLE2.

CREATING "TEMPORARY" VERB

To create a "temporary" verb, simply type it in carefully without using the editor. For example, type in

Prompt
↓
#: _TEST1 (CR)

"_THIS IS A TEST1". (CR)

; (CR)

If you made a typing mistake, everything is erased and you will have to start over. Note that this verb is being compiled line by line as you enter it.
EXECUTING "TEMPORARY" VERB

This is done by typing

TEST1

The computer will respond with

THIS IS A TEST1

ASIDE

To check out what is on the dictionary, type

LIST

and you will see that TEST1 is on top and TEST is second. To remove both of them, type

FORGET_TEST

Type

LIST

again to verify this.

USING ONLY THE EXISTING DICTIONARY

This path is convenient when you wish to try out some of the verbs offered in the dictionary. For example, to try out the "-" verb, type

Prompt
↓
#3_4_-
   (CR)

This action places the result (-1) on the stack To remove it from the stack and display the result, type

Prompt
↓
#.
   (CR)

The computer should respond with

-1
If you wish to leave the system, see section 1.3.10.

1.4 Editor

The editor package is always present, thus issuing an editor command will automatically allow interaction with the buffer. In some cases a (CR) is needed to get out of the buffer (otherwise it is done automatically).

Major editor commands are:

- **ADD** (CR) To add new lines to buffer
  (If other lines already exist, this adds the new lines after them.)

- **L_n** (CR) List line #n

- **L_n_m** (CR) Lists lines n through m

- **DEL_n** (CR) Delete line #n

- **DEL_n_m** (CR) Delete lines n through m

- **INS_n** (CR) Insert a line or set of lines before line #n

- **ED_n** (CR) Edit line #n

1.5 Two-Drive System

1.5.1 Directives*

When two drives are used, the commands DRIVE1 and DRIVE2 are used. When the system is initially booted, DRIVE1 is used by default. Then

*These verbs were created and entered into the FORTH system by Lee Powell
to use DRIVE2, type

```
DRIVE2 (CR)
```

and proceed as usual. From that point on, all commands will involve
the disc in DRIVE2 until the command

```
DRIVE1 (CR)
```

is issued which directs any subsequent commands back to DRIVE1. Thus
one can freely go back and forth between the two drives by these two
simple commands.

1.5.2 Transferring Files between Drives

To transfer a file from a disc in one drive to a disc in the other
drive type:

```
DRIVE_Ns (CR) , If the system is not already
directed toward that disc drive
DREAD_FILENAME (CR)
DRIVENd (CR)
DWRITE_FILENAME (CR)
```

Where:

\[
N_s = \text{source drive } \# \ (1 \text{ or } 2) \\
N_d = \text{destination drive } \# \ (1 \text{ or } 2)
\]

DRIVEN_s can be DRIVE1 or DRIVE2
DRIVEN_d can be DRIVE2 or DRIVE1

1.6 Useful Keyboard Commands

To list

```
LIST
L_n_m
DOS:_CATALOG
```

Lists dictionary--type (CR) to "turn page"
Lists lines n through m of buffer
Lists filenames from disk
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>To Delete</strong></td>
<td></td>
</tr>
<tr>
<td>DOS: DELETE_FILENAME</td>
<td>deletes file entitled &quot;filename&quot; from disk</td>
</tr>
<tr>
<td>FORGET_VERBNAME</td>
<td>Deletes all verbs on top of and including top most verb named &quot;verbname&quot; from dictionary</td>
</tr>
<tr>
<td>Clear buffer</td>
<td></td>
</tr>
<tr>
<td>CLR</td>
<td>Erases buffer memory</td>
</tr>
<tr>
<td>Clear screen</td>
<td></td>
</tr>
<tr>
<td>PAGE</td>
<td>Clears screen and places prompt at top left hand corner</td>
</tr>
<tr>
<td>Program Interrupt</td>
<td></td>
</tr>
<tr>
<td>CTRL C</td>
<td>Control C stops program during execution</td>
</tr>
</tbody>
</table>
2. NUMBER MANIPULATIONS

2.1 Introduction

Up to this point, manipulations of files have been discussed. Another set of important manipulations involves numbers. The numbers can be stored on a stack (one on top of another with no address) or in the dictionary with an associated address. In the dictionary, the number can have a constant-type address or a variable-type address. The main difference between the two is that the variable-type address is accessible which allows you to change the variable value under that address. The constant-type address is not accessible, therefore the value under its address remains constant.

2.2 Number/Address Transfer Flow

The number/address transfer flow is illustrated in Figure 4.

2.3 Number/Address Transfer Commands

The commands associated to this flow are in Figure 5.
Figure 4. Number/Address Transfer Flow.

- Keyboard/Display
- Dictionary
- Number Stack

Key operations:
- Define (←) for variables, constants
- Transfer (→) for constant, variable value change
- Transfer (➔) for unaddressed numbers
Figure 5. Number/Address Transfer Commands.

KEYBOARD/
DISPLAY

CV CONSTANT CN
VV VARIABLE VN
FORGET CN
FORGET VN (Delete)

V (Place # on stack)

(VN_I (Store stack # in Dict
under VN address)
VN @ (Load VV to stack)
CN (Load CV to stack)

NUMBER
STACK

DICTIONARY
Definitions

CV  Constant Value  (a number)
CN  Constant Name  (address)
VV  Variable Value  (a number)
VN  Variable Name  (address)
V   Number Value  (a number)

Note 1: Before any manipulations are done with constants or variables, they first must be defined. Furthermore, the definition must be outside the verb.

Note 2: ! and @ refer to 16-bit words. Replace these characters by C! and C@ respectively when dealing with 8-bit bytes (eg. with A/D converter).

Note 3: a) When a number is transferred from the number stack, it is removed from the top of the stack.
   b) When a number is transferred to the number stack, it is placed on top of the stack.
   c) When a number is transferred (via address) from the dictionary, the number and its associated address remain also in the dictionary.

Note 4: When a variable name is referenced, its address is placed on the stack. The character @ (or C@) replaces the address with its corresponding variable value. When a constant is referenced, its value is placed directly on the stack (no address).
2.4 Examples

The following commands illustrate number manipulations:

DEC  (set base to decimal)

5_CONSTANT_FIVE  (define)

6_VARIABLE_X  (define)

2  (place 2 on stack)

.  (print top number (2) from stack; response is 2)

3_FIVE_*_.  (print 3 times constant-five (5); response is 15)

X@.  (load variable X (6) to stack and print; response is 6)

X@FIVE_*_.  (load variable X (6) and constant five (5) to stack, multiply the two numbers and print answer)

8_X_!  (place 8 on stack, store it under variable name X, variable X is now equal to 8)

X@.  (load variable X (8) to stack and print; response is 8)

X@3+X_!  (load variable X (8) to stack, place 3 on stack, add 8 and 3, store sum (11) under variable name X)

X@.  (load variable X (11) to stack and print; response is 11)
3. **PROGRAM LANGUAGE**

3.1 **Introduction**

This chapter describes with examples the fundamental building blocks of the FORTH language. Included are I/O commands, mathematical operations, conditional statements, stack utilization, constants, variables, and arrays. There is also a general emphasis toward use of the A/D + D/A converter and game I/O ports.

3.2 **Structure**

See section 1.2.1.

3.3 **Versatility in Writing** (Example — string print)

Before getting too involved into programming, the versatility in writing a program will first be illustrated. For example, a simple program (written earlier) was written as:

```
1: TEST-VERB
2"_THIS IS A TEST"
3;.
4;
```

This can also be written as:

```
1:TEST-VERB"_THIS IS A TEST"_;.
```

Note: This shows the **least** number of spaces that need to be used — i.e. the _ indicates where a space must be entered.

**Execution**

When TEST-VERB is executed, the phrase

```
THIS IS A TEST
```

is printed.
3.4 Comments

Non-executable statements (comments) may appear within or outside a verb. A comment is defined by parentheses.

3.4.1 One-Line

Example:

(_COMMENT NUMBER ONE)

:\_TEST

(_COMMENT NUMBER TWO)

"_THIS IS A TEST"_S.

;

to execute this verb, type

TEST

the computer will respond with

THIS IS A TEST

and will ignore the comments. Note the required space after each parenthesis.

3.4.2 Multi-Line

Example:

:\_TEST

(_THIS IS A COMMENT

THAT OCCUPIES SEVERAL

LINES)

"_THIS IS A TEST"_S.

;
3.5 Mathematical Operations

3.5.1 With Output

The following program adds, subtracts, multiplies, and divides a set of numbers. The results are printed. For simplicity, the line numbers are deleted.

```plaintext
: _MATHOP
  DEC
  2 3 + _CR
  1 6 - _CR
  6 4 * _CR
  5 3 / _CR

The results are printed as:
  5
  -5
  24
  1

Note: 5/3 = 1

3.5.2 With Input and Output

This program allows the user to enter from the keyboard the numbers to be operated on:
```
MATHOPl
DEC
+._CR
-._CR
*._CR
/._CR
;

or

MATHOPl_DEC
+._CR-._CR*._CR/._CR;

To execute, type:

1_2_3_4_5_6_7_8_MATHOPl

The results are printed:

15   (7+8)
-1   (5-6)
12   (3*4)
0    (1/2)

The reason why the results show the operations to act in reverse order on the entered numbers is that the numbers 1-8 were put on a "stack" with 1 on the bottom and 8 on the top. The operation then proceeded from the top of the stack and worked down. First 7 and 8, then 5 and 6, etc.
3.5.3 Data Stack Usage

As mentioned on the previous page, one must be careful in knowing what is on the stack. This verb exemplifies stack usage:

:_MATHOP2_DEC

+_3*_.CR_;

To execute, type:

1_4_MATHOP2

The result is

15

Because the numbers 1 and 4 were first added together and the result (5) was then put on the stack. The number 5 is then taken off the stack to be multiplied with the number 3. The result (15) is then printed.

3.6 Alphanumeric (String) Output

As shown to this point, alphanumeric characters (string) may be printed out during execution of a verb. The form is as follows:

"---STRING---"_S.

Note the two required spaces. An example verb is:

:_TEST

"THIS IS A TEST"_S.

Type

TEST
to execute this verb. The phrase

THIS IS A TEST

will then appear as output.
3.7 Keyboard I/O - More Versatile Usages

3.7.1 Introduction

To this point I/O has been limited to the user entering numbers onto the stack, then executing the verb which acts on the numbers. This section illustrates two more levels of I/O capabilities-- the first utilizing a prompt command that tells the user how to put what numbers on the stack and the second appearing to do the same thing, but within the program, the entered numbers are assigned addresses so the subsequent operations do not depend so severely on where the number is located dictionary.

3.7.2 Prompt - Entering Numbers to Stack

This program consists of two verbs. The first is a prompt to instruct the user to enter the numbers and the second operates on the numbers:

```
:_PROMPT_DEC
"_THIS PROGRAM MULTIPLIES A AND B"

VERB
S._CR

PROMPT
"_ENTER A AND B THEN TYPE ""MULT"""

S._CR_;

VERB
:_MULT_*_.CR_;

MULT
```

To execute this, type

```
PROMPT
```

the computer will then type

```
THIS PROGRAM MULTIPLIES A AND B
```
ENTER A AND B THEN TYPE "MULT"

Then type

3_2_MULT

and the computer will print the result:

6

3.7.3 Prompt - Assigning Addresses to Input Numbers

This program uses the fact that one can address a number with the statement

n_VARIABLE_X

where X is the variable name and n is its value:

Ø_VARIABLE_X (defines and initializes X to be = to Ø)
Ø_VARIABLE_Y (defines and initializes Y to be = to Ø)
Ø_VARIABLE_Z (defines and initializes Z to be = to Ø)

:_PROMPT_DEC
"ENTER X Y AND Z THEN TYPE ""OP"""
S._CR_;
: OP
Z_!Y_!X_! (stores new values for Z, Y, and X)
X@Y@*_.CR (loads X, loads Y, multiplies, prints, CR)
X@Z@*_.CR (loads X, loads Z, multiplies, prints, CR)
Y@Z@*_.CR; (loads Y, loads Z, multiplies, prints, CR)

Note 1: The first 3 lines of the program are not within a verb.
"Variable" cannot be executed within a verb and must be defined outside it.

Note 2: n_X_! stores the number n under address X in dictionary. X_! removes the top number in the stack and stores under address X.
X_@ loads the number under address X onto stack.
To execute, type:

PROMPT

The computer answers with:

ENTER X Y AND Z THEN TYPE "OP"

You respond with:

3_4_5_OP

The results are printed:

12
15
20

3.8 Conditional Statements

3.8.1 Introduction

There are several sets of conditional statements in this language. They involve:

DO-LOOP
IF-ELSE-THEN
BEGIN-END

3.8.2 Do-Loop

3.8.2.1 Single

An example of a Do-Loop in a program is:

_:PRINT-I
5_Ø_DO
I_.CR
LOOP
;

Loops 5 times starting from Ø
(so I = Ø, 1, 2, 3, 4)
Prints value of I each loop and
returns carriage
To execute this verb, type

PRINT-I

The results are printed:

∅
1
2
3
4

Note: a_b_DO means loop a-b times starting at I = b.

Exception: if b > a, loop is executed once with I = b.

3.8.2.2 Non-Unity Index Increments

In the above-mentioned section index increments of 1 were implied in the Do-Loop commands. To change this increment value, use the structure:

A_B_DO

The system will then loop A-B times starting at B with index increments of C. The value C can be positive or negative. (If negative, make B > A).

3.8.2.3 Nested

Nested Do-Loops can use just the index I or employ an additional index J.
An example of using a single index in each loop is:

```
PRINT-2IS
4roe_DO
I_.CR
3_1DO
I_10*.CR
LOOP
LOOP:
```

To execute this verb, type

```
PRINT-2IS
```

The results are printed:

```
0
10
20
1
10
20
2
10
20
3
10
20
```

Note: The inner loop only "looped" twice each time since "I" could only equal 1 or 2 in that loop.

An example of using 2 indices is:

```
PRINT-IJ
5roe_DO
I_.CR
4_2DO
I_1J*.CR
LOOP
LOOP
```

Loop over

```
I = 0,1,2,3
```

Loop over

```
I = 1,2
```

Loop over

```
I = 0,1,2,3,4
```

Loop over

```
I = 2,3
```
Where in all applications J refers only to index of next outer loop.

To execute, type

PRINT-IJ

The results are printed

<table>
<thead>
<tr>
<th>J</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
</tr>
</tbody>
</table>

3.8.2.4 Leaving Loop Prematurely

If for some reason you wish to terminate a loop prematurely, the LEAVE command is used. This command can be used both in the +loop and -loop loops as long as the loop limit is not 0.

Example:

:_TEST
50_0_DO
I_.
LEAVE
I_.
LOOP
;

To execute this program, type

TEST

The computer will then display a

00

Because the loop is left once "LOOP" is encountered.
3.8.3 Begin-End Loop

The begin-end loop is effectively a "while" loop. It will loop continuously between the statements "begin" and "end" as long as the number read (and removed) off the stack during execution of the end statement is a 0. When it is non-zero at that point, the loop is exited.

3.8.3.1 Finite Loop

An example of a finite begin-end loop is:

```plaintext
:COMPARE-PRINT
BEGIN
DROP
DUP
5-
END
.
;
```

To execute this program, type

```
5_5_5_6_5_5_5COMPARE-PRINT
```

In this example, the program searches through the numbers you have placed on the stack and prints the first number that deviates from the # 5 (with exception to the initial top #--it is disregarded when drop is first encountered).

The result is printed:

6
3.8.3.2 **Infinite Loop**

If, for some application, an infinite loop is needed, simply place a "∅" directly before the end statement.

Example:

```
BEGIN
∅_END
```

3.8.4 **If-Else-Then**

The if-else-then logic is summarized as:

```
IF (do this if true)
ELSE (do this if false)
THEN
```

where:

- **TRUE NUMBER** --- Any non-zero integer
- **FALSE NUMBER** --- ∅

The following program illustrates the if-else-then logic:

```
: _TESTER
= _IF "_THESE NUMBERS ARE EQUAL" _S _CR
ELSE "_THESE NUMBERS ARE NOT EQUAL" _S _CR
THEN
```

where the "=" looks at the top two numbers of the stack and replaces them with a 1 if equal or a 0 if not equal.

To execute this program type

3_5_TESTER

The response is printed:

THESE NUMBERS ARE NOT EQUAL

3.9 Arrays

3.9.1 Introduction

FORTH does not have an array package. However, one is stored on disk under filename DIM and depending on need, others can be created fairly simply. DIM is a versatile, multidimensional array but is relatively slow to manipulate. Other arrays can be created that are much faster.

3.9.2 DIM-Multidimensional, Slow

3.9.2.1 Elementary Setup

An example of setting up DIM is:

DLOAD_DIM

LN1_LN2_ -- -- _LNn_N_DIM_ARRAYNAME

This sets up an N-dimensional array named arrayname with LN1, LN2, -- -- LN2 locations in the respective dimensions. The location address values lie in the range 0→LN1-1, 0→LN2-1, -- -- etc., respectively.

A real example is:
3.9.2.2 **Automatic Setup**

In some examples, it may be cumbersome to set up an array manually. The following program is useful in automatically setting up an array:

```
DLOAD_DIM
:.DIMSETUP
5_Ø_DO_1Ø_LOOP_5_;
```

Now execute by typing:

```
DIMSETUP
```

Then type

```
DIM_X
```

The array X has now been set up with 5 dimensions with 10 locations per dimension.

*Note:* The statement DIM X defines X in the dictionary. Be careful in setting up programs that use X by making sure it is first defined before those programs are loaded.

3.9.3 **One Dimensional-Fast**

The following verb sets up a fast, one-dimensional array:

```
:ARRAY_2* DTOP +!_;:
SWAP 2* +_;:
```

Once this verb is defined, the array can be set up by typing:
**N.ARRAY.ARRAYNAME**

Where N is the number of array elements (locations 0→N-1 and arrayname is the name of the array defined in this statement.

To store a number, V, into an array location, L, type

```
V_L_ARRAYNAME_!
```

where 0 ≤ L ≤ N-1

To fetch the number out of an array location and print it, type

```
L_ARRAYNAME_@.
```

### 3.9.4 More Array Examples

See examples in section 3.10.4—More Examples of A/D and D/A Conversions.

#### 3.10 Comparison Verbs

As discussed in the if-else-then, the "=" verb looks at the top 2 #s on the stack and replaces them with 1 if they are equal and a Φ if they are unequal. All comparison verbs are summarized as:

<table>
<thead>
<tr>
<th>VERB</th>
<th>NUMBER(S) IN STACK BEFORE VERB EXECUTED</th>
<th>NUMBER IN STACK AFTER VERB EXECUTED</th>
<th>DESCRIPTION true if:</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>a_b</td>
<td>1 if true, Φ if false</td>
<td>a=b</td>
</tr>
<tr>
<td>≤</td>
<td>a_b</td>
<td>&quot;</td>
<td>a≤b</td>
</tr>
<tr>
<td>≥</td>
<td>a_b</td>
<td>&quot;</td>
<td>a≥b</td>
</tr>
<tr>
<td>&lt;</td>
<td>a_b</td>
<td>&quot;</td>
<td>a&lt;b</td>
</tr>
<tr>
<td>&gt;</td>
<td>a_b</td>
<td>&quot;</td>
<td>a&gt;b</td>
</tr>
<tr>
<td>≠</td>
<td>a</td>
<td>&quot;</td>
<td>a≠Φ</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>a_b</td>
<td>&quot;</td>
<td>a≠b</td>
</tr>
</tbody>
</table>

Note: These verbs are also useful in the begin-end loop. For example, a logical true could be used to exit the loop.
3.11 A/D and D/A Conversions

3.11.1 Introduction

In some applications, it is necessary to access one or more of the 16 channels in each A/D and D/A converter. This is done by addressing the channel of interest then transferring the information from or to the respective A/D or D/A converter channel.

3.11.2 Channel Addresses

To each channel is a fixed address. These addresses for the A/D and D/A converter for slot #2 are:

<table>
<thead>
<tr>
<th>Channel#</th>
<th>Address</th>
<th>HEX</th>
<th>DEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C0A0</td>
<td>49312</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>C0A1</td>
<td>49313</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>C0A2</td>
<td>49314</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C0A3</td>
<td>49315</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>C0A4</td>
<td>49316</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>C0A5</td>
<td>49317</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>C0A6</td>
<td>49318</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>C0A7</td>
<td>49319</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>C0A8</td>
<td>49320</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>C0A9</td>
<td>49321</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>C0AA</td>
<td>49322</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>C0AB</td>
<td>49323</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>C0AC</td>
<td>49324</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>C0AD</td>
<td>49325</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>C0AE</td>
<td>49326</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>C0AF</td>
<td>49327</td>
<td></td>
</tr>
</tbody>
</table>

3.11.3 A/D Converter

3.11.3.1 Sampling a Channel

To sample a channel from the A/D converter, the command

CHANNEL-ADDRESS_C@
is used where C@ replaces the address number on the stack with the value from that channel. Note: a) that the computer fetches the second most recently addressed value, thus depending on the application, the above command may need to be entered twice, b) C@ loads lower 8-bits (1 Byte) of data from converter and @ loads 16-bits (2 Bytes) of data in applications not involving converter.

3.11.3.2 Sampling/Printing Continuously

An example program that continuously samples and prints the values from channels 0 and 1 is:

```plaintext
DEC
:_ADSAMPLER
49312_C@
BEGIN
49313_C@_"_"_"_S.
49312_C@_.CR
Ø_END
;
```

Note: a) This program prints the value from channel 0 in column 1 and the value from channel 1 in column 2. Since the computer will print the second most recently addressed data from the A/D converter, the order of channel 0 and channel 1 are reversed in the program in order to print the channels in the desired (0 then 1) order.

b) The sampled voltages lying in the maximum allowed -5v - 5v range will produce corresponding assigned values in the computer in the range Ø~255(DEC) where OV corresponds to 127 in computer.
To execute this program, type

ADSAMPLER

The results are

\[ \theta# \quad 1# \]
\[ \theta# \quad 1# \quad \text{Where } \theta# \text{ is value from channel } \theta \]
\[ \theta# \quad 1# \quad \text{1# is value from channel } 1 \]

3.11.3.3 Sample-Store

This program samples channel \( \theta \) then stores it under address "x".

\[ \_\_\_VAR_\_I\_\_X \]
\[ \_\_\_ADSAMPLERI \]
\[ 49312_\_C@ \]
\[ 49312_\_C@ \_X!_\_ ; \]

To execute it, type

ADSAMPLERI

The computer will respond with the usual prompt "#".

To check the value stored, type

\[ X\_\_\_X. \]

Note: Grounding the input to the sampled channel (OV) will produce a value of 127(DEC) in computer.

3.11.4 D/A Converter

3.11.4.1 D/A Output

In order to produce a signal at the output of the D/A converter, the command

\[ \_\_\_D_\_S_\_I_\_R_\_E_\_D_\_E_\_V_\_E_\_U_\_L_\_E_\_V_\_\_\_U_\_C_\_H_\_E_\_N_\_E_\_L_\_A_\_D_\_R_\_E_\_S_\_S_\_E_\_D_\_\_\_A\_\_D\_\_C\_\_E\_\_R\_\_E_\_ \]

\[ C! \]
is issued where \texttt{C!} directs the desired value to the address and leaves nothing on the stack. The desired value lies in the range (as discussed in the A/D section) \texttt{0 - 255(DEC)} which corresponds to a range in voltage output of \(-5\text{v} - 5\text{v}\). Again, \texttt{127} corresponds to \texttt{0V} output.

Note: As opposed to the A/D converter, only one command is required to transfer the value to the channel output.

3.11.4.2 Continuous Ramp Output

An example program that produces a repeating ramp at the output of a D/A channel is:

\begin{verbatim}
DEC :_RAMP BEGIN 256_Ø DO I_49312_C! I_49313_C! LOOP Ø_END ;
\end{verbatim}

To execute, connect channels \texttt{Ø} and \texttt{1} of the D/A converter to the X- and Y- inputs of an oscilloscope (in x-y mode), then type \texttt{RAMP}.

A continuously sweeping ramp function should then appear on the scope display.

3.11.5 More Examples of A/D and D/A Conversions

The following programs sample an A/D channel and eventually outputs the sampled value to a D/A channel. The first two examples
employ data arrays for intermediate storage while the last three examples utilize the stack. The sampling frequency for each algorithm is given.

** SAMPLE - STORE (DIM ARRAY, SLOW) - OUTPUT **

DEC
DLOAD DIM
100 1 DIM DATASAMPLES
: SAMPLEPLOT

110-Hz sampling
BEGIN
100 ø DO
sample-store
frequency
49312 C@ I DATASAMPLES !
LOOP
100 ø DO
Fetch-
I DATASAMPLES @ 49313 C!
output
LOOP ø END ;

** SAMPLE - STACK ONE - OUTPUT **

DEC
1.8-KHz sampling
: SAMPLEPLOT
frequency
BEGIN
49312 C@ 49313 C!
sample-output
ø END ;

** SAMPLE - STACK ALL - OUTPUT **

3.6-KHz sampling
: SAMPLEPLOT
frequency
BEGIN
100 0 DO
49312 C@
LOOP
100 0 DO
49313 C!
LOOP
Ø END;

** SAMPLE - SUM WITH STACK - OUTPUT **

DEC
: SAMPLE SUM PRINT

2.0-KHz
BEGIN
 samplingle
 frequency 10000 0 DO samples sums with top
 49312 C@ + of stack
 LOOP
 1 . prints 1
 Ø END;

Extra: This program tests how fast a number can be summed with the
stack as done in the previous example (no sampling done in this
case).

2.5-KHz
DEC
: SUMPRINT

summingle
frequency BEGIN
Ø
10000 Ø DO
51

1 + \hspace{10em} \text{sums 1 with top of stack}
LOOP
\hspace{10em} \text{prints sum}
\emptyset \text{ END ;}

3.11.5.1 \textbf{A/D and D/A Conversion Rates}

The sampling rates determined by the programs listed in the previous section are summarized as follows:

It takes 0.27 ms (3.6 KHz) to do each of the following:

1) sample (A/D) and place on stack
2) remove from stack and output (D/A)

And 0.4 ms (2.5 KHz) to add two numbers on the stack.

3.12 \textbf{Game I/O Ports}

3.12.1 \textbf{Introduction}

The APPLE-II-PLUS is not only accessible to the outside world through the A/D and D/A converter but also through the game I/O ports. These ports are either TTL (0-5v, 1-bit logic) input, TTL output or analog resistance input. The following ports are available:

3 Switches - TTL input
1 Strobe - TTL output
4 Paddles - analog resistance input
4 Annunciators - TTL output

3.12.2 \textbf{Channel Addresses}

The channels and their corresponding addresses are given in the following table:
3.12.3 Switches

Each switch is a TTL input. A ØV (or ground) input to a switch will produce a Ø in the sign bit location of the 8-bit byte associated to the switch input. A 5V (or open circuit) input will produce a 1 in the sign bit location.

An example program of sampling a switch port is as follows:

DEC
:SAMPLEPLOT

350-Hz
BEGIN
sampling
49249_C@

frequency
127>_IF_255_49312_C!
sample switch
compare to 127 (ØV) output 255 (5V) to a D/A converter or
ELSE_127_49312_C!
127 (ØV).

THEN_Ø_END_;
DEC

: _SAMPLEPLOT

3.6 KHz BEGIN
sampling 100_0 DO
frequency 49242_C@ sample switch and stack
LOOP
100_0 DO
127_> IF 255_49312_C! remove from stack and output
ELSE 127_49312_C! $V or 5V
THEN LOOP_0 END

3.12.4 Annunciator

Each annunciator is a TTL (0-5V, 1-bit logic) output. The output is turned on and off by separate commands (see address table in previous section). These commands take the form:

ANYNUMBER_ADDRESS_C!

where "anynumber" is used as a dummy number and is removed from the stack during the process.

An example program is:

DEC

: _TESTANØ

1-KHz BEGIN
loop rate Ø 49241_C! turns ANØ on
Ø 49240_C! turns ANØ off
Ø END

;
The output of the annunciator is a 0-5v rectangular wave.

3.12.5 Strobe

The strobe is very similar to the annunciators. It is a TTL output, however only one command is used to both turn it on then off again. As a result, the strobe pulse (~0.5 µs) is much shorter than that from the annunciator.

An example program is:

```
DEC
.TESTSTB
BEGIN
loop rate Ø.49216_C!  turns strobe on then off
Ø-END
;
```

Note: a) as with the annunciator, the "dummy number" must precede the channel address. This number is removed after the process.

b) strobe on = 0V

strobe off = +5V

3.12.6 Game-Port Conversion Rates

As determined by the previous programs, the conversion rates of the game ports are similar to those of the A/D and D/A converter. The sampling rate of the switch channels is 3.6 KHz (identical to A/D and D/A converter), the output rates of the annunciators and strobe are 1 KHz and 2 KHz respectively.
3.13 FORTH/ASSEMBLER Interface

3.13.1 Introduction

One of the major advantages of FORTH is the ease to which it can be interfaced with ASSEMBLER. For example, when the speed of ASSEMBLER is required in certain algorithms, these routines can be written in ASSEMBLER in the form of a verb. The command to execute each verb is then the same as if the verb were written in FORTH.

3.13.2 ASSEMBLER Enable/Disable*

To compile ASSEMBLER coding, the ASSEMBLER language must first be enabled by typing (or placing at the top of the coding)

ASSENA

It is the user's option to "disable" the ASSEMBLER once the ASSEMBLER has been compiled and/or executed by typing (or placing at the bottom of the coding)

ASSEND

Using the FORTH system with the ASSEMBLER "disabled" when the ASSEMBLER is not required allows it to compile faster. Thus it is a good practice to exercise the above option when possible.

3.13.3 ASSEMBLER Program Structure

The ASSEMBLER coding must begin and end and the commands CODE and NEXT_JMP, respectively. The program structure is as follows:

* The following commands were created and entered into the FORTH system by Lee Powell.
(definitions)

CODE_VERBNAME

(action)

NEXT_JMP,

3.13.4 Example

For an example of an ASSEMBLER program with an application of it, see section 3.14.

3.14 Comprehensive Data-Sampling Example

3.14.1 Introduction

The following example is a data processing system in which an APPLE-II-PLUS computer is used in sampling preprocessed sonar data from a two-channel (analog) magnetic tape. To optimize computer execution time and versatility, both ASSEMBLER and FORTH coding was employed. The tape was used to record (from a moving ship) sonar echoes from fish in one channel and the sonar transmitter trigger in the other. This setup is illustrated in the next section. The pre-processing involves dividing the echoes into 1-m depth bins, squaring, then integrating the signal within a bin. The signal is then passed on to the computer for final processing. This processing simply is the averaging of the integrated signals in each depth bin over each set of 200 transmitted pulses. The final results produce information on the mass of fish in each depth bin as a function of lateral position in the water.
3.14.2 Sonar Echo Recording Setup

[Diagram showing connections between Transceiver, Sonar Trigger, Sonar Echo, Tape Recorder, and Transducer]
3.14.3 **Preprocessing Setup**

Figure 6 is the preprocessing setup

where **Overrun Trigger** is a flag indicating the data has stopped coming off the tape (it senses the sonar trigger being shut off for an unusually long period of time).

**Sample Window** is a logic signal dictating the time window in which the computer can sample the integrated echo.

**1-m Reset** resets the integrated signal to zero every meter of depth (1.38 ms).

* This system was designed, built and tested by Lanny Stanfield, U.W.
Figure 6. Preprocessing Setup

- Tape Recorder
- Sonar Trigger
- Echo Signal
- 1-m Reset
- Sampling Logic
- Sample Window
- TO GAME I/O PORTS (TTL)
- Integrated Echo
- TO A/D CONVERTER
3.14.3.1 Preprocessing Signals

Triggers

- 0.4s
- Last Sonar Trigger
- Induced by Overrun Trigger
- Sonar Trigger
- Overrun Trigger

Remaining Signals

- 1.38 ms
- Sample Window
- 1-w Depth Reset
- Integrated Echo (Negative)
3.14.4 **APPLE Interface**

To integrate the APPLE computer with the preprocessing setup, the following connections are made:

- **SONAR TRIGGER** connected to SW0 (Switch 0)
- **OVERRUN TRIGGER** connected to SW1 (Switch 1)
- **SAMPLE WINDOW** connected to SW2 (Switch 2)
- **INTEGRATED ECHO** connected to Channel 0

Game I/O Ports

(TTL)

A/D converter

3.14.5 **Test Setup**

Figure 7 illustrates the test setup.

3.14.6 **Sampling Program**

3.14.6.1 **Flow Chart**

The flow chart is given in Figure 8.
Figure 7. Test Setup

"Sonar Trigger" -> "Fish Echo" -> "Bottom Echo"

Signal Generator

1.7-Hz Square Wave

Sine Wave

Signal Generator

12-Hz Sine Wave

Signal Generator

12-Hz Sine Wave

Signal Generator

Signal In -> Signal Out

Squaring Logic

Reset Signal

Overrun Sonar Sample
Trigger Trigger Window

TO GAME I/O PORTS (TTL)

TO A/D CONVERTER

Integrated Echo
Figure 8. Flow Chart

```
Begin  
  DEFINE PARAMETERS  
    
    SONAR TRIGGER ON  
      No  
        Loop up to 200 times  
          No  
            Loop 40 times  
              Yes  
                Sample window ON  
                  No  
                    Yes  
                      Sample integrated echo  
                        
                        Bottom like echo?  
                          Yes  
                            Leave loop  
                              
                              Transfer sampled data into (and sum with) large storage array  
                                
                                Overrun trigger ON  
                                  
                                  Rescale data  
                                    
                                    Print results  
                                      
                                      End
```

3.14.6.2 Coding

The coding is listed on the following pages. It is stored under 3 three files—

**ADASSEMBLERI** — General purpose algorithm in ASSEMBLER that performs sampling.

**PRINT3-2** — Printing routine — one of many. Each is in a separate file so choice can be made on what form of output is desired.

**EASTWARDINT1** — Major program that loads and uses ADASSEMBLER1 and prints.

To run this coding, type

```
DLOAD_EASTWARDINT2
START
```

The computer responds with

```
ENTER VALUE FOR INITIAL DEPTH(M) (SPACE)

THEN TYPE "INPUT"
```

Then type

```
DEPTH(M)_INPUT
```

And the sampling begins.
HEX
2 C059 CONSTANT ANU
3 28 XREG C!
4 0 VARIABLE DATA
5 64 DTOP +!
6 C040 CONSTANT STRB
7 C0A0 VARIABLE CHNL
8 C061 CONSTANT TRIG
9 C063 CONSTANT CLK
10 10 CONSTANT ADR
11 DATA ADR!
12 CODE A-D
13 0 # LDY,
14 XREG LDX,
15 BEGIN,
16 TRIG LDA,
17 NC END,
18 BEGIN,
19 TRIG LDA,
20 NS END,
21 BEGIN,
22 STRB STA,
23 BEGIN,
24 CLK LDA,
25 NC END,
26 BEGIN,
27 CLK LDA,
28 NS END,
29 CHNL @ LDA,
30 ANU STA,
31 ANU 1 - STA,
32 CHNL @ LDA,
33 ADR @Y STA,
34 INY,
35 DEX,
36 ZS END,
37 NEXT JMP,
PRINT3-2

1 : RUNPRINT
2 40 0 DO
3 SETNO @ 0 DO
4 J I SETDEPTH @ >=
5 IF " " S.
6 ELSE
7 I 40 * J + DATASTORE @
8 DUP 26 <
9 IF " ," ELSE DUP 51 <
10 ELSE DUP 76 <
11 IF " ;" ELSE DUP 101 <
12 IF " !" ELSE DUP 128 <
13 IF " *" ELSE " ?" THEN
14 THEN
15 THEN
16 THEN
17 THEN
18 THEN
19 THEN
20 THEN
21 THEN
22 THEN
23 THEN
24 S.
25 " " S. DROP
26 THEN
27 LOOP
28 CR
29 LOOP
30 ;
1 DEC
2 40 VARIABLE AVEDEPTH
3 40 VARIABLE MAXDEPTH
4 3 CONSTANT HALFWINDOW
5 0 VARIABLE SETNO
6 49250 CONSTANT TRIGOVER
7 ; ARRAY 2 * DTOP +! ;;
8 SWAP 2 * + ;
9 2000 ARRAY DATASTORE
10 40 ARRAY AVENO
11 50 ARRAY SETDEPTH
12 ; INITIALIZE
13 2000 0 DO
14 0 I DATASTORE 
15 LOOP
16 ;
17 INITIALIZE
18 DLOAD PRINT3-2
19 ASSENA
20 DLOAD ADASSEMBLER
21 ASSEND
22 DEC
23 ; START
24 " ENTER VALUE FOR INITIAL DEPTH(M)
25 (SPACE) THEN TYPE ""INPUT"" S. CR
26 ;
27 ; INPUT
28 AVEDEPTH 
29 MAXDEPTH 
30 " NOW TYPE ""RUN"" TO BEGIN SAMPLING" S. CR
31 ;
32 ; SCALEDATA
33 RUNPRINT
34 ;
35 ; SAMPLE
36 " INITIAL DEPTH= " S. AVEDEPTH @ .
37 " M" S. CR
38 " DEPTH WINDOW = " S. HALFWINDOW 2 * . " M" S. CR
39 BEGIN
40 AVEDEPTH @ SETNO @ SETDEPTH 
41 40 0 DO
42 0 I AVENO 
43 LOOP
44 200 0 DO
45 A-D
46 DEC
47 40 0 DO
48 I AVEDEPTH @ HALFWINDOW - <
49 IF
50 40 SETNO @ * I + DATASTORE @
51 DATA I + C@ +
52 40 SETNO @ * I + DATASTORE 
53 I AVENO @ 1 + I AVENO 
54 ELSE
55 DATA I + C@ 0 <>
IF I AVEDEPTH @ HALFWINDOW + <=
  IF 40 SETNO @ * I + DATASTORE @
  DATA I + C@ +
  40 SETNO @ * I + DATASTORE !
  I AVENO @ 1 + I AVENO !
ELSE
LEAVE
THEN
ELSE
I AVEDEPTH !
AVEDEPTH @ SETNO @ SETDEPTH @ <=
IF
ELSE
AVEDEPTH @ SETNO @ SETDEPTH !
THEN
LEAVE
THEN
LOOP
SETNO @ SETDEPTH @ MAXDEPTH @ <=
IF
ELSE
SETNO @ SETDEPTH @ MAXDEPTH !
THEN
TRIGOVER C@
128 <
IF LEAVE
ELSE THEN
LOOP
TRIGOVER C@
128 < IF 1
ELSE
40 0 DO
I AVENO @ 0 =
IF
ELSE
127 I AVENO @ *
40 SETNO @ * I + DATASTORE @ -
I AVENO @ /
40 SETNO @ * I + DATASTORE !
THEN
LOOP
SETNO @ 1 + SETNO !
" SETNO = " S. SETNO @ . " SETDEPTH = " S. SETNO @ 1
0 THEN
END
SAMPLE
; Runner
; S. CR
3.14.6.3 Sampling Frequency/Computer Dedication Time

As can be seen in the program, three gate I/O port output signals (AN0, AN1 and STB) are employed to indicate the time taken in certain critical loops. The results are

1) The time consumed between the points at which the computer senses a sampling window, does the sampling and looks for the next window is approximately 38 μs. Therefore this ASSEMBLER algorithm can be used in applications involving sampling frequencies of up to 26 KHz.

2) The processing done after the sampling (transferring and summing data into array, then checking overrun trigger) takes roughly 0.3 s. Thus the computer is effectively idle for 0.25 s or almost one-half the trigger period.
INDEX

A/D + D/A Conversion .................................................. 27, 45-51
ADD ........................................................................... 8, 21
Addition ........................................................................ 31, 32
Address ........................................................................... 26, 27
Annunciator ...................................................................... 53, 54
Arrays ............................................................................. 42-44
Assembler ......................................................................... 55, 56
BEGIN-END ..................................................................... 36, 40, 41
Booting ............................................................................ 7
Buffer .............................................................................. 1
CATALOG, see DOS: CATALOG
CLEAR, see CLR or PAGE
CLR ............................................................................. 8, 23
Comments .......................................................................... 30
Comparison Verbs .......................................................... 44
Conditional Statements ................................................... 36
CONSTANT .................................................................... 24, 25, 27, 28
D/A Conversion, see A/D + D/A Conversion
DEL .................................................................................. 21
DELETE, see DOS: DELETE or DEL
Dictionary ........................................................................ 1, 24, 25, 26
Disk ................................................................................ 1, 7
Division ........................................................................... 31, 32
DLOAD ............................................................................ 3, 4, 13
DO-LOOP ........................................................................ 36-39
DOS: CATALOG .................................................. 11, 22
DOS: DELETE .......................................................... 23
DREAD ............................................................. 3, 4, 12, 17, 22
DWRITE ............................................................. 3, 4, 11, 16, 19, 22
ED ........................................................... 3, 4, 11, 16, 19, 22
Editing ............................................................. 8, 21
Executing Verbs .................................................. 14, 16
Filename .......................................................... 5
FORGET ............................................................. 18, 20, 23
Game I/O Ports .................................................. 51-54
IF-ELSE-THEN .................................................. 36, 41, 42
INS ............................................................. 21
Interrupt ........................................................... 23
L ............................................................. 12, 17, 21, 22
LIST ............................................................. 16, 18, 22 or see L
LOAD ............................................................. 3, 4, 9, 12, 18
Multiplication .................................................. 31, 32
Page ............................................................. 23
Printing ............................................................. 31, 32
Stack ............................................................. 24, 25, 26, 33, 34
String ............................................................. 33
Strobe ............................................................. 54
Subtraction ....................................................... 31, 32
Switches .......................................................... 52, 53
TTL Ports, see Game I/O Ports
Verb ............................................................. 5