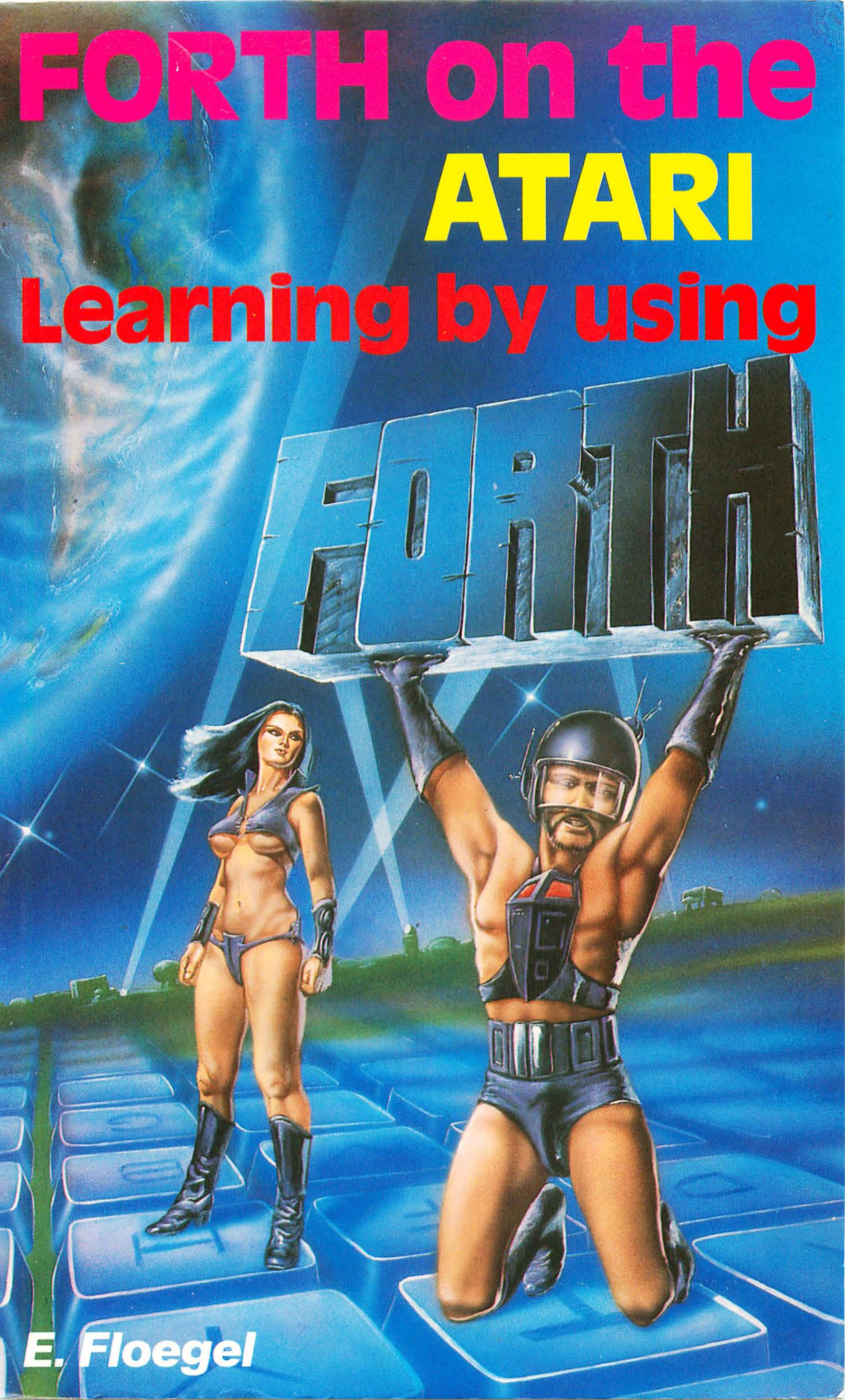
FORTH on the ATARI

Learning by using FORTH



E. Floegel

Example programs are for personal use only. Every reasonable effort has been made to ensure accuracy throughout this book, but neither the author or publisher can assume responsibility for any errors or omissions. No liability is assumed for any direct, or indirect, damages resulting from the use of information contained herein.

First Edition
First Printing
February 1983 in the Federal Republic of Germany

Order-No. 170

ISBN 3-88963-170-3

Publisher:
Ing. W. HOFACKER GmbH, Tegernseerstr. 18, D-8150 Holzkirchen, W.-Germany

US-Distributor:
ELCOMP Publishing, Inc., 53 Redrock Lane, Pomona, CA 91766

TABLE OF CONTENTS

A FORTH-compiler/interpreter is available from the following vendors:

1. ELCOMP PUBLISHING, INC., Pomona, CA 91766
   Phone (714) 623-8314
2. Quality Software, 6660 Resede Blvd. Suite 105, Reseda,
   CA 91335, Phone (213) 344-6599

Books and source code information are available from:

1. FORTH Interest Group
   P.O.Box 1105
   San Carlos, CA 94070

2. Mountain View Press
   P.O.Box 4656
   Mountain View, CA 94040

3. FORTH Inc.
   2309 Pacific Coast Hwy.
   Hermosa Beach, CA 90254

PREFACE

ATARI FORTH PROGRAMMING
Learning by using

FORTH is a new, exciting programming language. It is easy to learn and programs, written in FORTH are very short, compared to other high level languages.

The aim of this book is to show the novice and the experienced programmer how to use this language on the ATARI. The examples are short and use sound and grafics for demonstration. Use these examples to learn FORTH.

Two applications, a mailing list and a serial interface for a printer, are included. These two applications show the wide variety FORTH can be used for.

With FORTH an application can be written and debugged faster than in any other programming language.

I hope this book will add new friends to the community of FORTH programmers.

Thank's to Rick Schwarz, who helped me in proofreading.

E.Floegel

1 What is FORTH?

1. WHAT IS FORTH ?

FORTH is like BASIC, FORTRAN or PASCAL a programming language. It is, however quite different from these languages. It was invented by CHARLES MOORE about ten years ago and its first use was to control telescopes on the Kitt Peak observatory.

Today FORTH is widely used in control applications. One example is the control of movie cameras in the movie BATTLESHIP GALACTICA. Using the concept of virtual memory, FORTH also is used for data base managment.

FORTH can not be compared with other programming languages. While you are learning to program in FORTH it would be better to forget all you have learned about other languages.

What makes FORTH so unique ?

One of the basic elements of FORTH is the stack. Everybody knows what a stack is. Nearly every desk has a stack of paper. Something is laid upon this stack and may be removed later. In terms of a computer this is a LIFO (Last In,First Out) memory. The last thing put onto the stack is removed first.

FORTH uses the stack in two different ways. First there is a parameter stack for data and calculating. Second, there is a stack for words, called the dictionary or vocabulary. In FORTH, words are the elements for programming. Several of these words can be combined to form a new word, performing a certain task. A basic dictionary of predefined words is contained within FORTH. These words are used by the programmer to create new words. This new words are placed on top of the dictionary. Running a program in FORTH means calling a sequence of words.

For example, you have written a FORTH program to control stepper motors. START may be a word for starting the stepper motor. FASTER, SLOWER, may be words for changing the speed while STOP could be the word for stoping it.

Defining your own words makes FORTH very flexible. Words can be rearanged to perform new tasks. All this makes FORTH a dynamic language. On the other hand, all this freedom makes the programmer responsible for the correctness of a program. There are only a few error messages and warnings.

Though FORTH is used sometimes with other operating systems, it could be seen as its own operating sytem. It is an interpreter, calling and executing one word. It is also a compiler, with its capability to compile new words in the dictinary. It uses a text editor, and often an assembler.

In this booklet we will learn FORTH by using this language on an ATARI 400/800. We will use graphics, sound and joysticks.

All programs however, which don't use specific hardware on the ATARI can be easily adapted to other computers.

2 Basic elements of FORTH

2. BASIC ELEMENTS OF FORTH.

Let us first make some remarks concerning the basic elements of FORTH.

2-1 THE STACK.

As mentioned above, the stack is a LIFO memory. In Fig 2-1 a stack is represented in two ways. Fig 2-1a shows the normal representation of a stack. In this book we will use the stack as it is shown in Fig 2-1b.

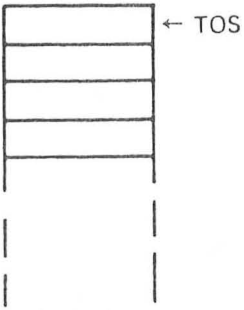


Fig. 2-1a

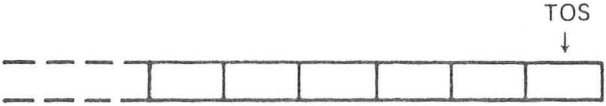


Fig. 2-1b

Fig 2-1 Two representations of the stack.

The top of stack (TOS) is always the rightmost element. One memory cell of the stack is a 16 bit cell.
For documentation and understanding of FORTH words, it is necessary to show how the stack is affected by using a word. We will use the following abbreviations:

a	16-bit address
n	signed 16 bit number
u	unsigned 16 bit number
d	signed 32 bit number
b	8 bit byte
c	7 bit ASCII character
f	boolean number, flag

These abbreviations are used in the following manner:

```
WORD  ( STACK BEFOR - STACK AFTER )
```

Examples:

```
LOOK ( an)
```

The word LOOK, whatever it does, requests an address and a number on top of stack before executing. Both items are removed from stack after execution.

```
FOUND ( -f)
```

FOUND doesn't need any parameter before execution and leaves a boolean number f on the stack after execution.

```
COMP ( aa'c - nf)
```

The word COMP needs two addresses and a character on top of the stack. After execution a number and a boolean flag is left on stack. In this representation, the top of stack is always the rightmost character. To differ between several addresses or numbers, there can be added a hyphen or a number to the character like a a' or n n1 n2.

2-2 WORDS IN FORTH.

A word in FORTH may be any arbitrary string of characters, excluding three special characters. The excluded characters are: the space, backspace, and the return character. The space character is the only delimiter between FORTH words.

The backspace character is used for correcting typing errors, while the return character is used to indicate that the input to the computer is finished.

Examples of words:

```
      FIRST   1TIMES   .NAME   @VALUE
```

THR EE is not recogniced as one word, because there is a space in it. It would represent two words THR and EE.

Some words in the predefined dictionary consist of only one character. The dot . , is for example, the printing statement. The @ sign fetches the content of a 16 bit memory cell. These letters can be used to indicate a special function of a word. In the example above .NAME will print a name while @VALUE will get some value from memory. This naming convention makes it easier to read FORTH programs.

2-3 THE REVERSE POLISH NOTATION.

For calculations, FORTH uses the reverse Polish, or postfix notation. The operator for the calculation is entered here after entering the numbers. If you type in

```
3 6 +
```

first, a three is put on stack, then a six. At this time, the FORTH interpreter recognices the word + . This word takes the two top numbers on stack, adds them together and places the result, 9, on top of the stack. If you now enter the . command a 9 is displayed on the screen. Try it.

The word + is defined in the following manner:

```
+ ( nn1 - n2) n2=n+n1
```

Exercises:

```
3*4-7
(3*5+3)/6-4
5^2-4^2
(3+5)*2
20/(2*5)
```

Using the reverse Polish notation, there is no need to use brackets. Let us look at the following example:

```
(3+5)*(4-2)-7
```

A BASIC interpreter, who has to decipher such an expression starts with the opening bracket, then gets a number, an operator, once more a number and then the closing bracket. At this moment it can do the first calculation. The next step indicates a requirement for multiplication. This can be done after revealing the second expression.

Using RPN, it is easier for a computer program to calculate this expression. In RPN we enter

```
3 5 + 4 2 - * 7 -
```

You see there is no need for brackets. Fig 2-2 shows the evaluation on stack.

STACK					TOS	INPUT
					3	3
				3	5	5
					8	+
				8	4	4
			8	4	2	2
				8	2	–
					16	•
				16	7	7
					9	–

Fig 2-2 Calculating an arithmetic expression using RPN.

To get acquainted with this reverse Polish notation, there are some exercises. Type them into the computer and use the . command to get the results.

Exercises:

```
3*4-7
(3*5+3)/6-4
5^2-4^2
(3+5)*2
20/(2*5)
```

3 Using words

3. USING WORDS

You can find a dictionary of all common words in appendix A. Here we will discuss the most used words for writing programs in FORTH.

3-1 DEFINITION OF NEW WORDS.

The definition of a new word starts with the : sign, followed by a space and the name of the new word. Then all words used by this new word are listed. The definition ends with the ; sign.

: XXX	Begin of a colon definition XXX
;	End of a colon definition.

Until now we have learned only two words. The . word and the + word. We want to combine these two to a new one, called +. (plus print). It adds two numbers and displays the result. The definition is:

```
: +. + . ;
```

Now we can enter 3 6 +. RET and get the result 9 on the screen.

To show that the operators + and - are real FORTH words we change the meaning of these two words by

: + - ;

After hitting return we get a warning :

+ ISN'T UNIQUE OK .

This indicates that the word + has been already defined. The interpreter will use the new defined + . When we now type 3 6 + . RET, we get the result 3.

Let us forget this new + word. By typing

FORGET + RET

this definition and also all definitions made later on are removed from the dictionary.

3-2 CHANGING THE STACK.

The stack can be changed in three ways. It can be enlarged, reduced or rearanged. All this can be done with the following words:

DUP(n - nn) Duplicate top of stack.

DROP (n) Throw away the top of stack.

SWAP(nn'-n'n) Reverse the two top elements.

OVER (n'n-n'nn') Copy of the second element on top.

ROT (n1n2n3-n2n3n1) Rotate the top three elements counter-clock wise.

Fig 3-1 shows how these words affect the stack.

	2	4	5	3	
2	4	5	3	3	DUP
2	4	3	3	5	ROT
2	4	3	5	3	SWAP
	2	4	3	5	DROP
2	4	3	5	3	OVER

Fig 3-1 Changing the stack.

Now let us do some examples and exercises.

Example 1:

The stack contains the numbers 3 2 1 with the 1 on top of the stack. What words must be entered to obtain the sequence 3 2 2 1?

First we enter OVER and get 3 2 1 2. Second we enter SWAP to obtain the result 3 2 2 1.

Example 2:

On the stack we have 3 2 1 with the 1 on top of the stack. Which words must be entered to get the sequence 2 3 3 ? The answer is shown below.

```
              3 2 1
     DROP       3 2
     SWAP       2 3
     DUP      2 3 3
```

Exercise: Start with 3 2 1 on the stack and get

a)	3 1 2
b)	2 1 3
c)	2 3 1
d)	1 2 3
e)	3 2 1 2 1

The word .S (dot s, print stack) shown in Fig 3-2 is very usefull. It is defined in most of the FORTH versions and prints the contents of the stack without destroying it. If your FORTH version doesn't know this word just type it in.

```
: 'S SP@ ;
: DEEP S0 @ 'S - 2 / 1 - ;
: .S CR DEEP 'S 2 - S0 @ 2 -
     DO I @ . -2 +LOOP
     ELSE ." EMPTY" THEN ;
```

Fig 3-2 Non-destructive stack print.

3-3 FUNDAMENTAL OPERATIONS IN ARITHMETIC.

The word for the fundamental operations of arithmetic are defined as follows:

+	(nn1-n2)	n2=n+n1
-	(nn1-n2)	n2=n-n1
*	(nn1-n2)	n2=n*n1
/	(nn1-n2)	n2=n/n1

This arithmetic is done with 16 bit signed fixed numbers. Finding the remainder and the quotient of a definition you can use two more words.

MOD (nn1-n2) n2 is the remainder of n/n1.

/MOD (nn1-n2n3) n2 is the remainder and n3 is the quotient of n/n1.

Example:

We want to calculate the value of the term X^2 + X*Y + Z. The values for X, Y and Z are stored on the stack with Z on top of the stack. Fig 3-3 shows the sequence of words to calculate this value.

	X	Y	Z	
	X	Z	Y	SWAP
	Z	Y	X	ROT
Z	Y	X	X	DUP
Z	X	X	Y	ROT
	Z	X	X + Y	+
		Z	X (X + Y)	*
			ERG	+

Fig 3-3 Calculating the term X^2+X*Y+Z.

We can define a word .VALUE which needs three numbers on the stack and calculates the value of this expression.

```
      : .VALUE ( nnln2)
            SWAP ROT DUP ROT + * + . ;
```

Exercises: Try to get the word sequences for

```
     a)    X^2+X*Y-Z
     b)    x^2+x*Y+Z^2
     c)    X^2-X*Y-Z^2
```

with X Y Z on the stack. Insert numbers and get the results.

3-4 INPUT OUTPUT.

One of the output instructions we have already used was the . word. It definition is:

```
  .     ( n)        Print the top of stack.
```

Other output instructions are:

```
    ."    ( )       Print message. The
                    message ends with " .
```

Example:

```
    ." HELLO " RET HELLO OK
```

```
EMIT   ( c)    Prints ASCII value c.
```

Example:

```
    69 EMIT RET EOK

    CR      ( )     Prints one carriage
                    return.

    SPACE   ( )     Prints one space.

    SPACES ( n)     Prints n spaces.

    .R      ( nnl)  Print n, right-justi-
                    fied in field. Field-
                    with is nl.
```

Example:

First we define a word P as

```
    : P ( n) 4 .R CR ;
```

Now we enter:

```
    CR 1 P 10 P 100 P
```

and get the result

```
       1
      10
     100
```

The following words are for data input:

KEY	(-c)	Reads the keyboard and places the ASCII value on the stack.
?TERMINAL	(-f)	True if the break key is pressed.

But on the ATARI:

```
?TERMINAL ( -n) n=1,2,4 if
                one of the yellow
                keys was pressed.
                n=0 if no key was
                pressed.
                START key   n=1
                SELECT key  n=2
                OPTION key  n=4
```

There are some more input and output words, which will be discussed later on in an example.

3-5 SOME SIMPLE PROGRAMS.

Until now we have learned only a few FORTH words, however they are sufficient to write some simple programs.

3-5a A LANGUAGE TRANSLATOR.

The concept of the dictionary and of defining new words can be used for a language translator. As an example we translate some words into the german language.

```
: I ." ICH " ;
: AM ." BIN " ;
: HERE ." DA " ;
```

If we type

```
    I AM HERE
```

we get the translation

```
     ICH BIN DA
```

This is only a very simple example. But you can use it to remeber special words in a foreign language, chemical formulas, or to make a list of your favourite radio stations and their frequencies. For example:

```
     : KBIG ." 104 FM " ;
     : KIQQ ." 100 FM " ;
```

3-5b WEIGHTWATCHER.

Some peoples want to calculate the amount of calories they had for breakfast or dinner. The input of data should be done in the following way:

```
     BEER  1 GLAS
     COKE  2 GLASSES
     BREAD 1 SLICE    and so on.
```

The program is shown in Fig.3-4. First there is a comment, indicating which units of measurement are used for the different types of foods. Regard this only as an example. Change them as you like. The next screen (we will discuss this expression in the next subchapter), contains the definitions for the units of measurement, while the following screen contains the definitions for the calories.

Fig 3-4 Weightwatcher.

```
SCR # 125

 0   ( AFB WEIGHTWATCHER            ef)
 1   ( UNIT GLAS : BEER, APPLEJUICE
 2          COCA-COLA, CHAMPAGNE
 3
 4     UNIT SLICE : BREAD
 5
 6     UNIT PIECE : CAKE, MUFFINS
 7
 8     UNIT OZ : BEEF, HAM, CHIPS,
 9       NUTS, CHEESE
10     UNIT CUP : RICE, PASTA
11     UNIT TBSP [ TABLESPOON ]
12          BUTTER,
13     UNIT PKG : SEAFOOD,
14
15     )
```

```
SCR # 126

 0   ( AFB WEIGHTWATCHER cntd    ef)
 1   ( MEASUREMENT UNITS )
 2   : GLAS DROP + ;
 3   : GLASSES * + ;
 4   : SLICE DROP + ;
 5   : SLICES * + ;
 6   : PIECE DROP + ;
 7   : PIECES * + ;
 8   : OZ DROP + ;
 9   : OZES * + ;
10   : OF ;
11   : START 0 ;
12   : TBSP DROP + ; : TBSPS * + ;
13   : PKG DROP + ; : PKGS * + ;
14   : CUP DROP + ; : CUPS * + ;
15
```

```
 SCR # 127

  0   ( AFB WEIGHTWATCHER  cntd   ef)
  1   ( CALORIES )
  2   : BEER 255 ; : COCA-COLA 88 ;
  3   : BREAD 100 ;
  4   : BUTTER 100 ; : BROWNIES 224 ;
  5   : RICE 200 ; : GOUDA 108 ;
  6   : MUFFINS 118 ;
  7   : LASAGNE 241 ; : RAVIOLI 210 ;
  8   : BURITO 47 ;    : CNCHIPS 166 ;
  9   : PEANUTS 179 ; : PTCHIPS 156 ;
 10
 11
 12   : AMOUNT DUP . ;
 13
 14
 15

START OK
COCA-COLA 1 GLAS OK
BREAD 2 SLICES OK
MUFFINS 2 PIECES OK
GOUDA 2 OZ.ES OK
BUTTER 1 TBSP OK
AMOUNT 840 OK
CNCHIPS 2 OZES OK
AMOUNT 1172 OK
LASAGNE 2 CUPS OK
AMOUNT 1654 OK
```

Fig 3-4 Weightwatcher.

If you enter the word BEER, the number 255 is put on the stack. When you now enter 2 GLASSES, first the number 2 is put onto the stack, then the both top numbers (255 2) are multiplied and the result is added to the previous amount.

To start this calculation of calories, you have to enter the word START, which puts a 0 on top of the stack. The word AMOUNT duplicates the top of stack and prints it on the screen.

3-6 THE SCREEN.

The writing of a FORTH program can be done in two ways. You can enter the FORTH words direct into the dictionary or use a text editor and write the words into a screen. Fiq. 3-5 shows an example of a screen. FORTH versions running on an ATARI use a screen with 16 lines and 32 characters each.

```
SCR # 148

 0  ( SCRN PRINT  10/14/82       ef)
 1  0 VARIABLE ROW 0 VARIABLE COLN
 2  : ?FIN ( -f)  COLN @ 24 = ;
 3  : 1ROW 40 0 DO ROW @ C@ 32 +
 4    DUP 128 > IF 32 - THEN
 5    OUTCHR   1 ROW +!  LOOP ;
 6
 7  : .SCRN 88 @ ROW ! 0 COLN !
 8  BEGIN 1ROW CRR 1 COLN +! ?FIN
 9  UNTIL ;
10
11
12
13
14
15    -->
```

Fig 3-5 Textscreen.

This uses 512 bytes of memory. Thus, when using a disk drive, you can store 164 screens. In this booklet we don't describe the editor. It has numerous variations in the various FORTH versions. Please refer to the instruction manual of your FORTH.

Once you have written a screen, you can LOAD it. LOAD compiles the words of a

screen into the dictionary. You may also LIST a screen. LIST displays the content of a disk screen on the TV screen.
The definitions are:

```
     LOAD  ( n)     Compile disk screen n
                    into the dictionary.

     LIST  ( n)     List disk screen n on
                    the TV screen.
```

There are some conventions in the writing of a screen. The first line should be a comment, noting the task, the date and the programmer.
For example:

```
( WEIGHTWATCHER 10/16/82        ef)
```

The comment in FORTH starts with the opening bracket and ends with the closing bracket. The (is a FORTH word and must therefore be followed by one space.

```
     (     ( )      Beginning of a comment.
```

You can use the word INDEX to display all first lines of a disk screen on the TV screen.

```
     INDEX   ( nn') Display the first
                    lines from screen n  to
                    screen n'.
```

In writing a screen, every new definition of a word should start in a new line. This makes it easier to read a FORTH program. After writing you save the screen on disk with the word FLUSH.

```
     FLUSH (  )     Saves screen on disk.
```

3-7 CONSTANTS AND VARIABLES.

Most of the data used for calculation is stored on the stack. Sometimes it is necessary to use a constant or a variable. The definitions of a constant and a variable are:

CONSTANT	NAME	(n)	Creates a constant NAME with the value n.
VARIABLE	NAME	(n)	Creates a variable NAME and the initial value n.

There is a difference in calling a constant or a variable. Calling a constant by name, the value of the constant is place on the stack. Calling a variable by name, the address of this variable is placed on the stack. In order to get the value of a variable, you have to fetch it. This is done by the word @. For changing the value of a variable, the word ! (store) is used.

@	(a-n)	Fetches the content stored at address a.
!	(na)	Stores n at address a.

Both instructions, fetch and store use 16 bit numbers. For storing and fetching a single byte, the following words are used.

C@	(a-b)	Fetches a single byte from address a.
C!	(ba)	Stores a single byte at address a.

Try this:

```
0 755 C!  The cursor will disappear.
4 755 C!  The letters are upside down.
3 755 C!  Resets to the normal mode.
```

Later we will use these words to control with the color and sound registers of the ATARI.

You can increment the value of a variable by

```
0 VARIABLE V
 V @ 1 + V !
```

or you can use the word +! (plus-store).

```
+!    ( na)      Add n to the content
                 at address a.
```

Another word ? fetches the content of a variable and prints it.

```
?     ( a)       Prints the content
                 of address a.
```

The FORTH system itself uses several constants and variables. We will use the variable BASE in the following example. The content of this variable determines the number base in which calculations are made. If the value of BASE is 10, all calculations are made in decimal. Changing this value to 16, the calculations are made in hexadecimal. You may use however, any other value for calculating in any other number base system. Two words are defined to set a specific number base.

```
DECIMAL (  )   Set number base decimal.
```

```
     HEX        (  )     Set number base
                         hexadecimal.
```

An example:

We define:

```
     : BIN 2 BASE ! ;
     : TRI 3 BASE ! ;
     : .BIN ( n) BIN . DECIMAL ;
     : .TRI ( n) TRI . DECIMAL ;
```

The word BIN sets the number base to two and the word TRI to three. The trinary system uses only the numbers 0 1 and 2 for representing a number. The words .BIN and .TRI take a decimal value from the stack, converts it to the binary or trinary number, prints it and switches back into the decimal number base. Let us convert some numbers.

```
     120 .BIN 1111000 OK
     140 .TRI 12012 OK
```

The word .BASE fetches the content of BASE, duplicates it and prints it in decimal. The the value is restored in BASE.

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

If you only type BASE @ . you will always get the result 10, regardless in which number base you are.

A constant should stay a constant. But you can change the value of a constant with the ' (tick) word. The ' brings the address of a definition on the stack.

```
' NAME ( -a)    Find the address of
                NAME in vocabulary.
```

Example:

```
10 CONSTANT C OK
C . 10 OK
12 ' C ! OK
C . 12 OK
```

3-8 COMPARISON.

The comparison takes place with the two top numbers on the stack. They are replaced by the result of the comparison. The result is a one if the comparison was true or the result is a zero, if this comparison was not true. This boolean flag is used by the control words, described in the next chapter, to control the flow of a program. The words used for comparison are:

```
 <  ( nn'-f)  f=1, if n less than n'
 >  ( nn'-f)  f=1, if n greater than n'
 =  ( nn'-f)  f=1, if n equal to n'
 0< ( n-f)    f=1, if n is less than zero
 0= ( n-f)    f=1, if n is equal to zero
```

Some examples:

```
2 3 < . 1 OK
3 2 < . 0 OK
3 2 = . 0 OK
-2 0< . 1 OK
```

If you want to use the two top numbers on the stack after a comparison you have to duplicate them with

```
: 2DUP ( nn'-nn'nn') OVER OVER ;
```

4 Control structures

4. CONTROL STRUCTURES.

Until now we have only used words which didn't affect the flow of a program. Now we will take a closer look at the words which do control the flow of a program. First, there is no GOTO statement. Therefore FORTH is like PASCAL, a highly structered programming language.

As branches in a program it uses statements as BEGIN...UNTIL, BEGIN... WHILE. . . REPEAT and IF...ELSE...THEN.

These statements along with the DO...LOOP must be used within a colon definition. They can not be executed directly by the interpreter.

4-1 DO...LOOP .

The DO...LOOP creates a finite number of program loops. The definition is:

DO (nn') Loops from n' to n-1, the increment of the loop is one.

LOOP ( ) Terminates the loop inside the program. Increments the index by one.

The program loop starts at n' and ends at n-1. The increment is always one. The set of words between DO and LOOP is executed from n' to n-1. The comparison between the loop index and the upper limit is performed by the word LOOP. Therefore every loop is executed at least once.

FORTH uses a third stack, wich we haven't mentioned until now. This is the RETURN stack. In a computer with a 6502 CPU the RETURN stack is equal to the stack used by the CPU. During execution of a loop, FORTH places the loop index on top of this stack. With the word I, you can copy the top of the RETURN stack to the parameter stack.

```
     I  ( -n)        Copy of the return
                     stack to the parameter
                     stack.
```

Example:

Let us print the numbers from one to nine on the screen.

```
     : NR 10 1 DO I . LOOP ;
     NR 1 2 3 4 5 6 7 8 9 OK
```

Just to show that a loop is executed at least once, we change the boundaries of the loop.

```
     : NRR 1 10 DO I . LOOP ;

     NRR 10 OK
```

Loops can be nested. In our next example, we print a row of numbers from 0 to 9. Then, in the next line we print a row of numbers from 0 to 8 and so on, until a single 0 is printed. The inner loop is defined as:

```
       : IL ( n) 0 DO I . LOOP ;
```

IL expects the upper limit + 1 on the stack. The outmost loop is defined as:

```
     : OL ( n) 0 DO CR 1 - DUP IL LOOP ;
```

OL expects the number of lines on the stack. To get a printout we call 11 10 OL. The result is shown in Fig 4-1.

```
11 10 OL
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6
0 1 2 3 4 5
0 1 2 3 4
0 1 2 3
0 1 2
0 1
0 OK
```

Fig 4-1 Triangle-matrix

Another example of nested loop is shown in Fig 4-2. A rhombus is printed by the two words RI and RO.

```
SCR # 103

: STAR ( n) 0 DO 42 EMIT LOOP ;
: RI ( nn1n2) DO CR 7 I -
   SPACES   I 2 * 1 + STAR DUP
   +LOOP ;
: RO 1 8 0 RI -1 -1 7 RI CR ;
```

```
RO
       *
      ***
     *****
    *******
   *********
  ***********
 *************
***************
***************
 *************
  ***********
   *********
    *******
     *****
      ***
       *
OK
```

Fig 4-2 Rhombus

Very often it is necessary to increment the loop index by another number instead of one or even to decrement the loop index. Then, the DO. ..+LOOP must be used. The definition of the DO word is the same as above, but the +LOOP expects a number on the stack.

```
+LOOP ( n)     Increments the index
               by n.  If  n  is  less
               zero,   the  index  is
               decremented.
```

Examples:

```
: +NR ( nn') DO I . 2 +LOOP ;

10 0 +NR 0 2 4 6 8 OK

: -NR ( nn') DO I . -2 +LOOP ;

0 10 -NR 10 8 6 4 2 OK
```

4-2 THE RETURN STACK.

In the last chapter we mentioned the RETURN stack. FORTH uses this stack for program loops and for storing addresses.

With care, a programmer can use this stack too. The word >R (to-R) puts the top number of the parameter stack on the RETURN stack and the word R> (R-from) returns it.

>R	(n)	Puts n to the return stack.
R>	(-n)	Retrieve number from return stack.
R	(-n)	Copy the top of the return stack to the parameter stack. Same as I.

Use these words only within a colon definition and outside of loops.

Example:

We define a word 2SWAP, which exchanges the two top numbers on the stack with the third and the fourth number.

```
      : 2SWAP ( nn1n2n3-n2n3nn1)
      >R ROT ROT R> ROT ROT ;
```

The evaluation of this word on the stack is shown in Fig 4-3.

	4	3	2	1	
		4	3	2	〉R
		3	2	4	ROT
		2	4	3	ROT
	2	4	3	1	R〉
	2	3	1	4	ROT
	2	1	4	3	ROT

Fig 4-3 Evaluation of 2SWAP

4-3 IF...ELSE...THEN (ENDIF).

This is the first word to control the flow of a program. It is used in the form IF <words> THEN or IF <wordsl> ELSE <words2> THEN. Some versions of FORTH use ENDIF instead of THEN.

The definitions:

IF (f)
THEN ( )
The words between IF and THEN (ENDIF) are executed if f is non zero.

IF (f)
ELSE ( )
THEN ( )
The words between IF and ELSE are executed if f is non zero. Otherwise the words between ELSE and THEN (ENDIF) are executed.

Example:
We define a word COMP, which compares the two top numbers on the stack. The message EQUAL is printed, if both numbers are equal. If not, the message SMALLER or BIGGER is printed. The definition of the word COMP is shown in Fig 4-4.

```
: 2DUP ( nn'-nn'nn')
  OVER OVER ;
: 2DROP ( nn') DROP DROP ;
: NOTEQUAL ( f) IF ." BIGGER "
  ELSE ." SMALLER " THEN ;
: COMP ( nn') 2DUP = IF
  ." EQUAL " 2DROP ELSE <
  NOTEQUAL THEN ;

2 3 COMP BIGGER OK
3 2 COMP SMALLER OK
2 2 COMP EQUAL OK
```

Fig 4-4 The word COMP.

First we define the word 2DUP, which duplicates the two top numbers on the stack and 2DROP which drops these numbers. The word NOTEQUAL prints the message SMALLER or BIGGER.

The word COMP duplicates the two top numbers and compares them. If they are equal, the message EQUAL is printed and the remaining numbers are removed from the stack. If they are not equal, another comparison is performed and according to this comparison the message SMALLER or BIGGER is printed.

If we didn't include the word 2DROP, the two numbers would be left on the stack. There could be some problems, if this word is executed several times without the 2DROP in a DO. ..LOOP. The stack becomes deeper and deeper. The size of the stack is limited and after a while, the error message STACK FULL is printed and the program is aborted. Try to keep the stack clean.

4-4 BEGIN...UNTIL.

The word UNTIL expects a flag on top of the stack. As long as this flag is zero, all words between BEGIN and UNTIL are repeated. A non zero flag terminates the loop.

The definition:

BEGIN	( )	The words between BEGIN and UNTIL are repeated, until a non-zero flag is encounterd before executing the word UNTIL.
UNTIL	(f)	

Example:

We use the BEGIN. . . UNTIL loop to add together the numbers from 1 to 100. The program is shown in Fig 4-5.

```
0 VARIABLE X
: INCX X @ 1 + DUP X ! ;
: SUM ( n) 0 BEGIN INCX + X @
  100 = UNTIL . ;

      0 SUM 5050 OK
```

Fig 4-5 Adding the numbers from 1 to 100 using BEGIN...UNTIL.

We use the variable X for counting. The word INCX increments the value of X. We don't use the +! , because we must duplicate this value and add it to the previous sum. Before starting the loop, a zero must be on the stack. The loop terminates if the condition X=100 is true. The result is then printed.

Another example for the BEGIN...UNTIL loop is the word .E . It prints an E on the screen until you hit the OPTION key.

```
: .E (  ) BEGIN 69 EMIT ?TERMINAL
          4 = UNTIL ;
```

4-5 BEGIN...WHILE...REPEAT .

This loop is different from the BEGIN... UNTIL loop. The word WHILE expects a flag on the stack. As long as this flag is non-zero, the words between WHILE and REPEAT are executed. An unconditional branch leads back to the BEGIN. If WHILE finds a zero on the stack, the words between WHILE and REPEAT are neglected and the word following REPEAT is executed.

The definitions:

```
BEGIN  (  )    The words between
UNTIL  ( f)    BEGIN WHILE and
REPEAT (  )    REPEAT are executed,
               as long as f is non-
               zero. If f is zero the
               program commences
Examp          after repeat.
```

We use the same example as above, adding the numbers from 1 to 100. The program is shown in Fig 4-6.

```
: TEST ( -f) 1 + DUP 101 < ;

: ADD DUP ROT + SWAP ;

: SUM 0 BEGIN TEST WHILE ADD
  REPEAT DROP . ;
  SUM 5050 OK
```

Fig 4-6 Adding the numbers from 1 to 100 using BEGIN...WHILE...REPEAT.

First we define the word TEST. A one is added to the top number on the stack, duplicated and compared with 101. The next word ADD duplicates the top of stack and rotates the first three elements. The deepest element on the stack is the amount of numbers that have been added until now.

The two top numbers are then added and the result is placed below the number of additions. The word SUM expects a zero on the stack. The word TEST is executed before WHILE. As long as the number of additions is less than 101, the word ADD is executed. If the number is equal to 101, the numbers on the stack are swapped, the result is printed and the remaining number is droped.

4-6 THE CASE STATEMENT.

In earlier versions of FORTH, there was no CASE statement defined. Now most of the FORTH versions have it. The definitions differ slightly from each other. We will describe the CASE statement as it is used in QS FORTH and in POWER FORTH for the ATARI.

The definition is:

```
     CASE:  ( n) WORD0 WORD1 ... WORDK ;
```

CASE: requests a number on the stack. In regard of this number, the corresponding word is executed. If this number is 0, WORD0 is executed or if it is a three WORD3 is executed. CASE: does not proof the limit. If you call the 10th word and there is no word, in most cases the program hangs up. You have to proof the limits by program. An example you find in the next chapter.

5 Sample programs

5. SAMPLE PROGRAMS.

In the following programs we will use the words we have learned up to this point. The most efficient way to learn a programming language is to type programs into the computer and execute them. The ATARI sounds and graphics will be used for the following samples.

5-1 SOME GRAFICS

There are several predefined words in the FORTH versions of the ATARI that use the grafics. In the programs we use SETCOLOR, PLOT and GR.

```
     SETCOLOR  ( nn1n2) Set the color
                    register.
                    n2 color register (  0-
                    4 depending mode ).
                    n1  color  hue number (
                    see Fig 5-1).
                    n  color  luminescance,
                    even  number  between 0
                    and 14. The higher  the
                    number,   the  brighter
                    the display.
```

COLORS	SETCOLOR nl
GRAY	0
LIGHT ORANGE (GOLD)	1
ORANGE	2
RED-ORANGE	3
PINK	4
PURPLE-BLUE	6
BLUE	7
BLUE	8
LIGHT-BLUE	9
TURQUOISE	10
GREEN-BLUE	11
GREEN	12
YELLOW-GREEN	13
ORANGE-GREEN	14
LIGHT ORANGE	15

Fig 5-1 The ATARI hue numbers and colors.

```
PLOT  ( nnln2) Plot a point at x=n
               y=nl  and  color  c=n2.
               The  point  x=0 and y=0
               is   the   upper   left
               corner of the screen.

GR.   ( n)     Sets the grafic mode
```

5-1 LINES.

In the program in Fig 5-2, we define the word START. It opens grafic mode 7 and sets the background color. The next word L>R draws a line from left to right.

```
SCR # 140

( AFB GRAPHICS  10/20         ef)

: START 7 GR. 2 0 0 SETCOLOR ;
: R>L (  )  79 9 DO I 10 2 PLOT
      LOOP ;
: U>D (  )  79 10 DO 78 I 3
  PLOT LOOP ;
: L>R (  )  8 78 DO I 78 2
  PLOT -1 +LOOP ;
: D>U (  )   9 78 DO 9 I 3
  PLOT -1 +LOOP ;
: RECT START R>L U>D L>R D>U ;
```

Fig 5-2 Lines.

The next word U>D draws a line down the screen, starting at the ending point of the previous line. In the same manner the words R>L and D>U are defined. They always start at the end of the former line. The last word RECT combines these words to draw a rectangle.

Here you can see one of the advantages of FORTH. Once you have defined a word, you can use it within other words. With FORTH it is also easy to change a program. You may defin a new word with another sequence of predefined words. Try it in this example. Rearrange the words in such a manner, so that the top line and the bottom line are drawn first and then the two side lines.

In this example, line one of the text screen #140 is left blank. In developing this screen, the words FORGET START were inserted. These words were inserted after the first compilation of the screen. This prevents the stack of the vocabulary from becoming too large, thus recievinig the message word NOT UNIQUE.

In the next program, we will use the joystick to draw lines. Suppose the red button is in the upper left corner, you get the eight values, shown in Fig 5-3 for the eight directions of the joystick by reading memory cell 632.

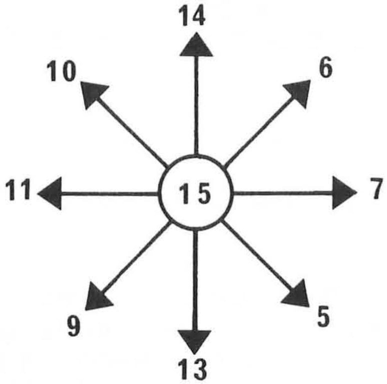


Fig 5-3 Joystick Controller Movement.

Plug in a joystick in game port one, then type

```
632 C@ .   .
```

You will get the values shown in Fig 5-3. The definition of the words is shown in text screen 116. We use two variables X an Y. The word +X increments the value of X by one. The word +Y is defined as

```
: +Y -1 Y +! ;
```

because Y counts positive downward on the screen.

The word STICK expects a value on top of the stack which is equal to the content of game port one. Then it decides which variable has to be incremented or decremented.

```
SCR # 116

( AFB  JOYSTICK 10/18          ef)
0 VARIABLE X 0 VARIABLE Y
: +X 1 X +! ; : -X -1 X +! ;
: +Y -1 Y +! ; : -Y 1 Y +! ;
: STICK ( n)
DUP 14 = IF +Y ELSE
DUP 13 = IF -Y ELSE
DUP 7 = IF +X ELSE
DUP 11 = IF -X ELSE
DUP 6 = IF +X +Y ELSE
DUP 5 = IF +X -Y ELSE
DUP 9 = IF -X -Y ELSE
DUP 10 = IF -X +Y THEN
THEN THEN THEN THEN THEN
THEN THEN ;


SCR # 118

( AFB GRAFICS 10/18            ef)
: START 2 0 0 SETCOLOR 7 GR.
  10 Y ! 10 X ! ;
: NOT ( n-n') 1 XOR ;
: PL ( nn')  X @ Y @ 2 PLOT ;
: NPL ( nn') X @ Y @ 0 PLOT ;
: ?STICK 632 C@ DUP 15 = NOT
   IF     STICK PL THEN DROP ;
: PJOY START PL BEGIN
   ?STICK ?TERMINAL UNTIL
   0 GR. ;
```

Fig 5-4 Controlling the joystick.

The program continues in screen 118. The word START sets the grafic mode, the background colors and the starting values for X and Y. The word NOT is used to change the result of a comparison. If a comparison is fulfilled, a one is on the stack. NOT changes this value to zero. If a zero is on the stack, NOT changes it to one. With this word the Exclusive OR function is used. This function is shown in Fig 5-5 for one bit. FORTH applies this function bit by bit on the two top numbers of the stack.

Bit1	Bit2	XOR
0	0	1
1	0	0
0	1	0
1	1	1

Fig 5-5 Exclusive Or

The other logical functions AND and OR are also implemented.The definitions are:

```
AND  ( nn1-n2) logical AND, bitwise
OR   ( nn1-n2) logical OR, bitwise
XOR  ( nn1-n2) logical XOR, bitwise
```

Now we continue with the program. The word PL plots a point at X and Y in color two. The word ?STICK determines if the joystick is moved in one of the eight directions. Then a point is plotted. The word PJOY combines all these words to plot lines on the TV screen.

The word NPL plots the point in the color of the background. This erases a point on the TV screen. We can insert this word in ?STICK to move a point across the TV screen.

```
: ?STICK 632 C@ DUP 15 = NOT
    IF NPL STICK PL THEN DROP ;
```

5-2 PATTERN.

The program in Fig 5-6 creates a random pattern. It uses a random number generator. RND# expects a number n on the stack. A random number between 0 and n-1 is generated. As a starting value for the random numbers, we use the content of memory location 53770. In this memory location there are created random numbers inside the ATARI.

```
SCR # 117

( AFB RANDOM PATTERN          ef)
( RANDOM GENERATOR )
0 VARIABLE RND 53770 @ RND !
: RANDOM RND @ 31421 * 6972 +
  DUP RND ! ;
: RND# ( n-n') RANDOM U* SWAP
  DROP ;
: RNDP 7 GR. 2 0 0 SETCOLOR
  BEGIN 160 RND# 80 RND#
  16 RND# PLOT
  ?TERMINAL UNTIL ;
```

Fig 5-6 Random Pattern.

The word RNDP sets the grafic mode 7 and the background color. In a loop, terminated by one of the yellow keys, it creates random numbers for the X Y coordinates and the color. The result is displayed on the TV screen.

5-3 SOUND AND COLOR

The word SOUND uses four parameters on the stack.

SOUND (nn1n2n3) n (0-15) is the volume, n1 a distortion, n2 the frequency and n4 the channel.

For creating several sounds and noises, we use the random number generator and a wait loop. The word WAIT requests one number on the stack. The higher the number, the longer the delay. The text screens 81 to 85 in Fig 5-7 contain some sound words, such as the word THUNDER in screen 81. Using random pitch and volume, a sound like a swarm of flies is generated. ENG uses intermittent sound for noise that sounds like an engine.

Fig 5-7 Sound.

```
SCR # 081

( AFB SOUND  10/20             ef)
: WAIT 0 DO LOOP ;
: C ( n-n') 11 RND# 5 + 10 *
  SWAP / ;

: OFF 0 0 0 0 SOUND
      0 0 0 1 SOUND ;

: T 100 5 DO I C 8 I 0 SOUND
  I C 8 I 20 + 1 SOUND
  DUP WAIT 5 RND# +LOOP OFF ;

: THUNDER 300 T ;


-->
```

```
SCR # 082

 0   ( AFB  SOUND   cntd            ef)
 1   ( FLY)
 2   : PI ( n) 7 RND# 250 + ;
 3   : V ( n) 4 RND# 6 + ;
 4   : F V 14 PI 0 SOUND ;
 5   : FLY BEGIN F 500  WAIT
 6     ?TERMINAL UNTIL OFF ;
 7   ( ENGINE )
 8   : ENG BEGIN 10 10 250 0 SOUND
 9     1500 WAIT OFF ?TERMINAL
10     UNTIL ;
11
12
13
14
15   -->
```

```
SCR # 083

 0   ( AFB SOUND  cntd              ef)
 1   : INCR ( n) 10 / 1
 2      DO I  10 60 0 SOUND
 3      LOOP ;
 4   : DECR ( n) 2 * 10 / DUP 1
 5      SWAP DO DUP 10 60 0 SOUND
 6      -1 +LOOP DROP ;
 7   : T 100 0 DO I INCR  2 +LOOP ;
 8
 9   : TT 0 100 DO I DECR -2 +LOOP ;
10
11   : CC 0 100 DO I INCR I DECR
12     -2 +LOOP ;
13
14
15   -->
```

SCR # 084

```
( AFB  SOUND cntd            ef)
15 VARIABLE V1 15 VARIABLE V2
15 VARIABLE V3
: ST 15 V1 ! 15 V2 ! 15 V3 ! ;
: SS V1 @ 8 20 0 SOUND
     V2 @ 8 40 2 SOUND
     V3 @ 8 70 2 SOUND ;

: DEC -1 V1 +! -1 V2 +!
   -1 V3 +! ;
: EXPL ST  BEGIN SS DEC
  1000 WAIT V3 @ 0< UNTIL ;




```

SCR # 085

```
( AFB SOUND  cntd            ef)
: SI 15 0 DO I 10 60 I 2 * -
   0 SOUND 100 WAIT LOOP ;

: SIREN BEGIN SI OFF 50 WAIT
  ?TERMINAL UNTIL ;


: DWN 200 100 DO 8 10 I 0 SOUND
  100 WAIT LOOP ;
: UP  100 200 DO 8 10 I 0 SOUND
  100 WAIT -1 +LOOP ;
: EUSI 10 0 DO UP DWN LOOP
  OFF ;


```

The word CC in screen 83 simulates a dropped coin. These examples are from the book " ATARI BASIC Learning by Using " by Tomas E. Rowley.

Some more sound words are in the next screens. EXPL simulates an explosion by changing the volume and the pitch on three channels. In the last screen we have the word SIREN for simulating an american police siren and the word EUSI for an european police siren.

The best way to create sound effects is the methode of trial an error. Define new words and combine them in different ways.

```
 SCR # 086

 0   ( AFB COLOR   10/22           ef)
 1   : CF 712 C@ 710 C@ 712 C!
 2         709 C@ 710 C! 709 C! ;
 3
 4   : CCF 100 0 DO CF 100 WAIT
 5     LOOP 0 GR. ;
 6
 7   : BG 254 0 DO I 712 C! 500 WAIT
 8     2 +LOOP ;
 9
10   : FG 254 0 DO I 710 C! 500 WAIT
11     2 +LOOP ;
12
 SCR # 087

 0  ( AFB COLOR   cntd            ef)
 1  : DI 16 0 DO I 709 C! 100 WAIT
 2    LOOP ;
 3  : AR 0 14 DO I 709 C! 100 WAIT
 4    -1 +LOOP ;
 5
 6  : CURS ( nnl) 85 ! 84 C! ;
 7  : CLR 125 EMIT ;
 8  : DIS CLR 10 5 CURS
 9    222 710 C! ." HELLO "
10    1000 WAIT DI KEY AR ;
11
12
```

Fig 5-8 Color.

In the text screens 86 and 87 (Fig 5-8), the store and fetch words are used to change the contents of the color registers. The word CF rotates the contents of the color registers. This causes a quick change of foreground and background colors. Use this word in a new definition of THUNDER. BG changes the background color and FG changes the foreground color
In the next screen, the text HELLO disappears and, after pressing a key, it reappears. The word DIS uses two other words CLR and CURS. The word CURS requests two numbers on the stack. N is the horizontal row and nl the vertical column. By sending out an end of file character (155 EMIT) the cursor is placed at the specified position. The word CLR clears the TV screen and positions the cursor in the upper left corner.

5-4 HEXDUMP

Most of the FORTH versions have defined the word DUMP as that which dumps the content of memory locations on the screen. The program in Fig 5-9 (screens 88 and 89) is a similar program. The word DDUMP requests two numbers on the stack. N is the starting address and nl the number of lines, eight bytes each.

```
SCR # 088

( AFB HEXDUMP  10/22          ef)
HEX
: LNE ( n) DUP DUP 8 + SWAP
  DO I C@ 3 .R LOOP ;
: NR CR CR 5 SPACES 8 0
  DO I 3 .R LOOP CR ;
: DOT DROP 2E ;

: ATARI ( n-n) DUP 20 < IF DOT
  ELSE DUP 7D = IF DOT ELSE
  DUP 7E = IF DOT ELSE DUP 90 =
  IF DOT ELSE DUP 9C = IF DOT
  ELSE DUP FB > IF DOT THEN
  THEN THEN THEN THEN THEN ;

  -->
```

```
SCR # 089

( AFB HEXDUMP  cntd           ef)
: ASCII ( a-a) DUP DUP 8 +
  SWAP DO I C@ ATARI EMIT
  2 SPACES LOOP ;




: HDUMP ( n-n') CR DUP . SPACE
  LNE CR 7 SPACES ASCII 8 + ;
: DDUMP ( an) HEX
  NR 0 DO HDUMP LOOP DROP
  DECIMAL ;
  DECIMAL


```

Fig 5-9 Hexdump.

An example of the printout is shown in Fig 5-10.

```
HEX 3008 9 DDUMP

           0  1  2  3  4  5  6  7

3008      86 4B 45 59 4C 49 D4 EA
          f  K  E  Y  L  I  T  j
3010      2F D9  C 60  D F3 2F 7C
          /  Y  .     .  s  /  |
3018       C 60  D E1  8 1C  C 3D
          .     .  a  .  .  .  =
3020       C FA  8 F3 2F 70  D 9C
          .  z  .  s  /  p  .  .
3028       B 9C  B 91  C  9  F 59
          .  .  .  q  .  .  .  Y
3030       8  4  0 22  B FA  8 7A
          .  .  .  "  .  z  .  z
3038       8 E4 FF  F  B 31  0 87
          .  d  .  .  .  1  .  g
3040      43 4F 4E 54 52 4F CC  8
          C  O  N  T  R  O  L  .
3048      30 5F 11 13 20 86 2C 46
          0  _  .  .     f  ,  F   OK
```

Fig 5-10 Hexdump Printout.

The word ATARI changes all unwanted characters (for example the byte 9B which erases the screen) to dots. This word is used within the word ASCII, which prints the ASCII characters. LNE prints one line of hex bytes while NR does the numbering on top of the printout.

Next we look at some mathematical examples.

5-5 LARGEST COMMON DIVISOR.

We use the algorithm of EUCLID to calculate the largest common divisor of two numbers A and B. First the remainder of A/B is determined by A B MOD . If the remainder R is zero, B is the largest common divisor. If R is not zero, A is set to B and B is set to R and the modulo division is repeated until R is zero. The definition of LCD is shown in Fig 5-11.

```
0  ( AFB MATH EXAMPLES            ef)
1
2  : LCD BEGIN SWAP OVER MOD DUP
3        0= UNTIL DROP . ;
4
      27 21 LCD 3 OK
```

Fig 5-11 Largest Common Divisor.

STACK					TOS	INPUT
					27	27
				27	21	21
				27	21	BEGIN
				21	27	SWAP
			21	27	21	OVER
				21	6	MOD
			21	6	6	DUP
			21	6	0	0=
				21	6	UNTIL
				6	21	SWAP
			6	21	6	OVER
				6	3	MOD
			6	3	3	DUP
			6	3	0	0=
				6	3	UNTIL
				3	6	SWAP
			3	6	3	OVER
				3	0	MOD
			3	0	0	DUP
			3	0	1	0=
				3	0	UNTIL
					3	DROP

Fig 5-12 Largest Common Divisor.
Evaluation on the stack.

For a better understanding of this definition, the evaluation on the stack for the given example is shown in Fig 5-12.

5-6 FIBBOCONACCI NUMBERS.

Fibboconacci numbers are a series of numbers. The next element of this series is always the sum of the two predecessors. The series starts with zero and one. The word FIB in Fig 5-13 creates these numbers. This is an example for the BEGIN...WHILE... REPEAT loop. FIB expects on the stack one number which determines the end of the series. If the calculated element is larger than this number, the calculation stops.

```
SCR # 099

( AFB MATH EXAMPLES cndt    ef)

: FIB ( n) CR 0 1 BEGIN DUP
       >R ROT DUP R> > WHILE
       ROT ROT DUP ROT + DUP .
       REPEAT DROP DROP DROP ;


100 FIB
1 2 3 5 8 13 21 34 55 89 144 OK
```

Fig 5-13 FIBBOCONACCI Numbers.

The evaluation on the stack for the first three loops is shown in Fig 5-14.

					20	20
					20	FIB
				20	0	0
			20	0	1	1
			20	0	1	BEGIN
		20	0	1	1	DUP
			20	0	1	〉R
			0	1	20	ROT
		0	1	20	20	DUP
	0	1	20	20	1	R〉
		0	1	20	1	〉
			0	1	20	WHILE
			1	20	0	ROT
			20	0	1	ROT
		20	0	1	1	DUP
		20	1	1	0	ROT
			20	1	1	+
		20	1	1	1	DUP
			20	1	1	.
			20	1	1	REPEAT
		20	1	1	1	DUP
			20	1	1	〉R
			1	1	20	ROT
		1	1	20	20	DUP
	1	1	20	20	1	R〉
		1	1	20	1	〉
			1	1	20	WHILE
			1	20	1	ROT
			20	1	1	ROT
		20	1	1	1	DUP
		20	1	1	1	ROT
			20	1	2	+
		20	1	2	2	DUP
			20	1	2	.
			20	1	2	REPEAT
		20	1	2	2	DUP

Fig 5-14 FIBBOCONACCI Numbers.
Evaluation on the stack.

5-7 PRIME NUMBERS.

In the next example, we calculate the prime numbers between two limits. The algorithm used is very simple. The word PTEST tests within a loop, if a number is divisible by the loop index. The loop starts with two and ends at half of the number. In this program the predefined word LEAVE is used. This word terminates the execution of a loop. The program continues after the next LOOP word.

```
LEAVE  ( )    Terminates a loop
```

If TEST finds a remainder equal to zero, a zero is placed on the stack and the loop is left. The program then continues with the word DUP. With a zero on the stack the word .PRIM is not executed and the top of stack is discarded. To format the output a variable #ROW is used. The program is shown in Fig 5-15.

```
SCR # 100

 ( AFB MATH EXAMPLES cndt    ef)

4 VARIABLE #ROW
: TEST ( n-f) MOD 0= ;
: .PRIM ( n-n) DUP 4 .R #ROW @
  DUP 0= IF CR DROP 4 ELSE
  1 - THEN  #ROW ! ;
: PTEST ( n)  DUP 2 / 2 DO DUP
  I TEST IF 0 LEAVE THEN LOOP
  DUP IF .PRIM ELSE DROP
  THEN DROP ;

: PRIM ( nn') CR 4 #ROW !
    DO I PTEST LOOP CR ;


```

```
200 1 PRIM
   1   3   5   7  11
  13  17  19  23  29
  31  37  41  43  47
  53  59  61  67  71
  73  79  83  89  97
 101 103 107 109 113
 127 131 137 139 149
 151 157 163 167 173
 179 181 191 193 197
 199
OK
```

Fig 5-15 Prime Numbers.

5-8 MORE SOUND AND GRAFICS.

I found some new sounds in the September issue of COMPUTE! magazine. I translated these examples from BASIC to FORTH. These new sound words are shown in Fig 5-16. There is another thunder which you can combine with the old thunder and rain to create stormy weather.

```
SCR # 094

( MORE SOUND 10/26              ef)
: B ( -n) 255 RND# 50 + ;
: X ( -n) 200 RND# ;
: T1 (   ) B 1 DO 15 8 I 0 SOUND
   LOOP ;
: T2 (   ) X 1 DO LOOP ;
: TH (   )  2 0 DO T1 OFF
  T2 OFF    LOOP ;
: STO TH TH THUNDER TH ;
: R1 ( n) 0 0 2 SOUND ;
: RAIN 150 0 DO I 10 / R1
  100 WAIT 2 +LOOP BEGIN
  ?TERMINAL UNTIL ;
```

```
( MORE SOUND   cntd            ef)
: HB 10 1 DO 15 3 12 0 SOUND
1000 WAIT OFF 6000 WAIT LOOP ;

-1 VARIABLE XX
: LO1 40 150 DO I 10 / 0 15 0
  SOUND  XX @  +LOOP ;
: LO  50 1 DO LO1 XX @ -7 > IF
  -1 XX +! THEN LOOP ;
: LS 10 10 40 1 SOUND
       8 10 10 2 SOUND
      10 10 90 3 SOUND ;
: STE 2 0 DO LS 4000 WAIT XSND
     1000 WAIT LOOP ;
: STEAM -1 XX ! LO STE LO
  XSND ;
```

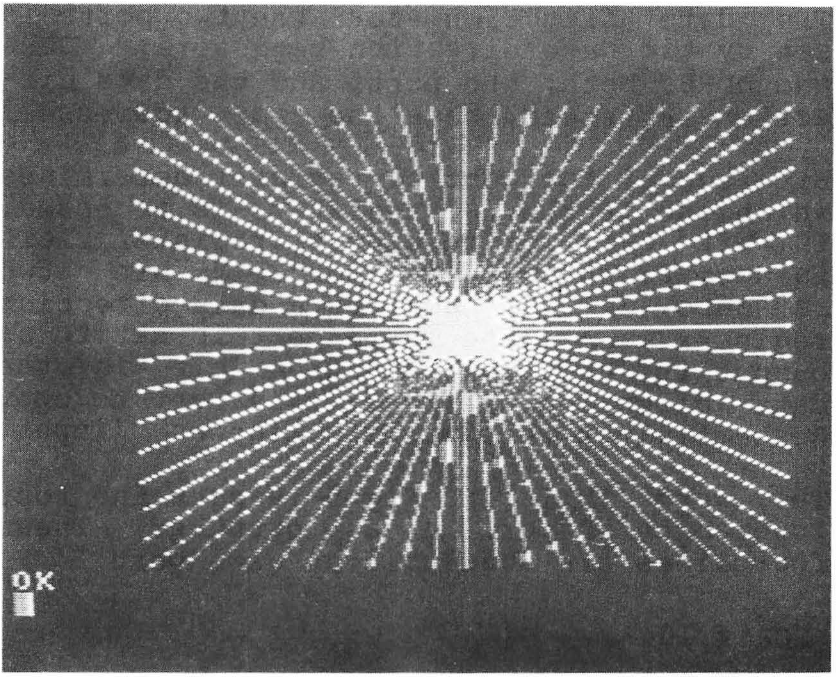
Fig 5-16 More sound.

The program in Fig 5-17 produces the grafic shown in picture 5-18. The lines are drawn in grafic mode eight. Draw this picture as it is and then a second time with the background color. It appears and disappears.

```
( MORE GRAFICS  10/26        ef)
0 VARIABLE X 0 VARIABLE Y
: CG 8 GR. 0 0 2 SETCOLOR ;
: SG 260 X ! 10 Y ! ;
: PG 160 80 1 PLOT ;
: -X -10 X +! ;
: +X  10 X +! ;
: -Y -10 Y +! ;
: +Y  10 Y +! ;
: LG X @ Y @ 1 DRAWTO ;
: LL 20 0 DO PG LG -X LOOP ;
: DD 15 0 DO PG LG +Y LOOP ;
: RR 20 0 DO PG LG +X LOOP ;
: UU 15 0 DO PG LG -Y LOOP ;
: PIC CG SG LL DD RR UU ;

```

Fig 5-17 Rayshaped pattern.



```
OK
```

Fig 5-18 Picture of the rayshaped pattern.

5-9 USING THE GAME PORTS FOR CONTROL APPLICATIONS.

The ATARI uses two 6520 Peripheral Interface Adapters (PIA) for the game ports. These ports can be used to transmit or to recieve data. The 6520 provides two bi-directional ports, A and B; two control registers and four interrupt lines. In our examples we only use the ports and the control registers. The four registers have the following addresses:

```
     PORTA = $D300
     PORTB = $D301
     PACTL = $D302
and  PBCTL = $D303
```

The data direction for the two ports is set by two data direction registers, DDRA and DDRB. Both, the ports and the data direction registers, have the same addresses.

Bit two of the control register determines which one of the two registers is accessed. If bit two of the control register PACTL is one, PORTA acts as a port. If bit two is zero, PORTA acts as a data direction register. A line of a port is set to an output if the corresponding bit in the data direction register is set to one. Respectivly a zero marks a line as input.

The word INIT in Fig 5-19 sets the four lower bits of port A as outputs. These four lines are available at game port 1.

```
SCR # 104

( AFB PORT CONTROL 10/16    ef)
HEX D300 CONSTANT PORTA
    D301 CONSTANT PORTB
    D302 CONSTANT PACTL
    D303 CONSTANT PBCTL
: INIT 30 PACTL C!
       0F PORTA C!
       34 PACTL C!
       00 PORTA C! ;

DECIMAL

: PA! ( b) PORTA C! ;
: WAIT ( n) 0 DO LOOP ;


```

Fig 5-19 Initialisation of the ports.

We use the circuit shown in Fig 5-20 to demonstrate the output of data.

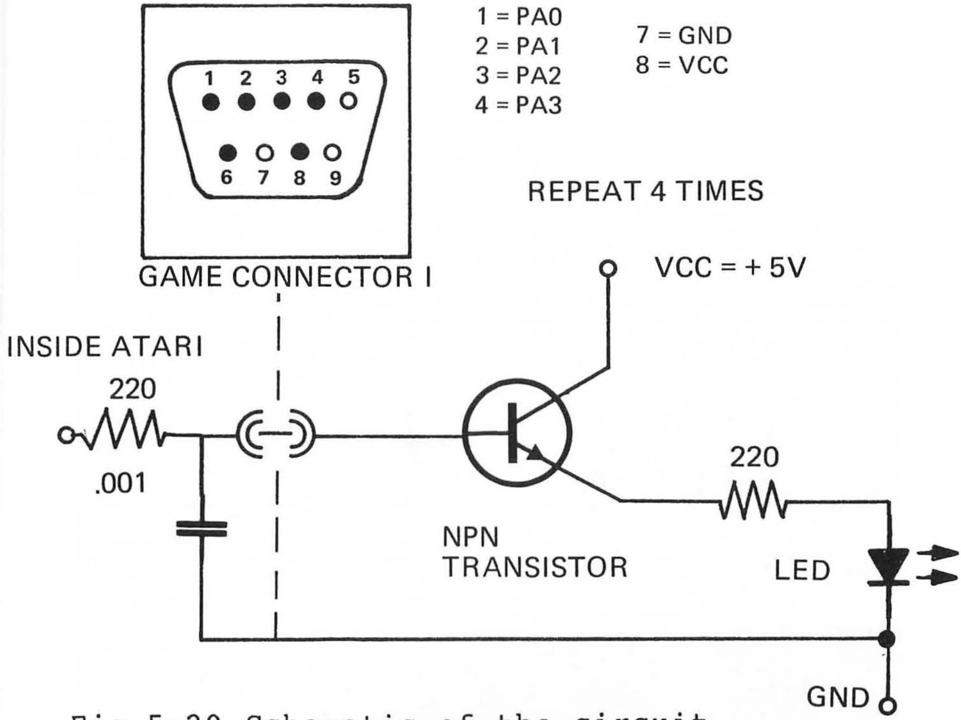


Fig 5-20 Schematic of the circuit.

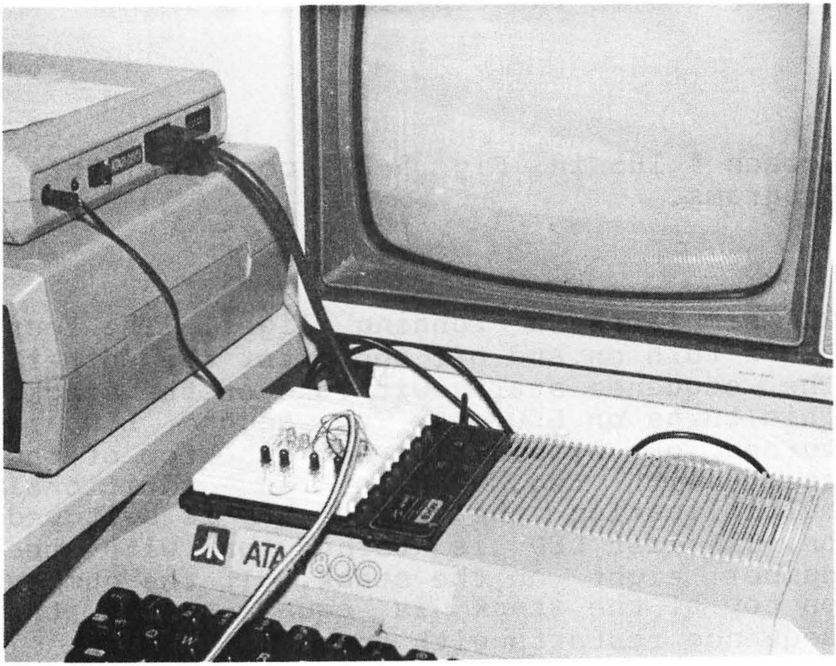


Fig 5-21 The experimenter board.

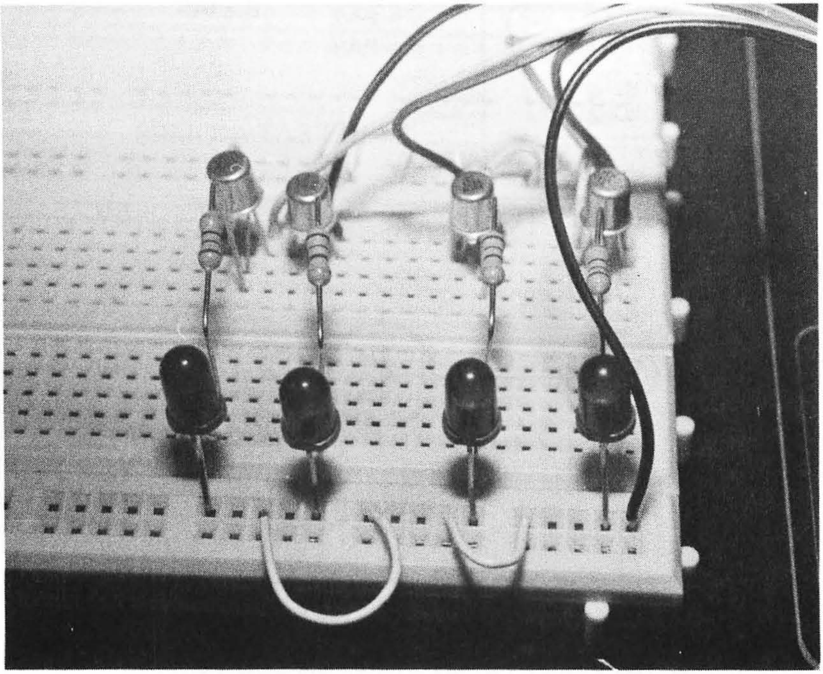


Fig 5-21 The experimenter board.

Screen # 105 in Fig 5-22 contains three programs.

RL simulates a running light. The four LED's turn on and off one after the other. The sequence starts with one on the stack. This turns on LED one. The second LED is turned on by multiplying the top of the stack with two. With an additional multiplication the third LED is switched on. The last LED is turned on with the number eight on the stack. If the number on top of the stack is equal to 16 the sequence restarts with one.

```
SCR # 105

  0  ( AFB PORT CONTROL cntd     ef)
  1  : RL 1 BEGIN DUP PA! 2 * DUP
  2    16 = IF DROP 1 THEN 2000 WAIT
  3    ?TERMINAL UNTIL ;
  4
  5  : LB 0 BEGIN DUP PA! 2 * 1 +
  6    DUP 31 = IF DROP 0 THEN
  7    1000 WAIT ?TERMINAL UNTIL ;
  8
  9  : 0INIT INIT 0 DUP PA! ;
 10  : NEW 1 SWAP DUP 1 = IF DROP
 11    ELSE 1 DO 2 * LOOP THEN ;
 12  : ON ( n) NEW OR DUP PA! ;
 13  : OFF ( n) NEW XOR DUP
 14    PA! ;
 15
```

Fig 5-22 Running light, lightbar and ON OFF

LB simulates a lightbar. The LED's are not turned off until all four LED's are on. The sequence starts with one. By multiplying with two and adding one we get three. This value turns on LED one and two.

An additional multiplication and addition leaves the value seven on top of the stack. This turns on LED one, two and three. After all LED's are turned on a zero on top of the stack turns all off. Both programs use the WAIT loop.

The words defined in screen 105, line 9 thru 14, can be used to turn on or off a particular LED.

```
     1 ON turns on LED one,
     3 ON turns on LED three
and  1 OFF turns off LED one.
```

The word OINIT initialises port A, places a zero on top of the stack and turns off all LED's. The word NEW determines which LED has to be turned on. The state of the four LED's is stored in the four lower bits of a byte on top of the stack. A one markes the corresponding LED as turned on, a zero marks it as turned off. Using the logical OR function the new LED is added. In the same way a LED is turned off using the logical XOR function.

The application in Fig 5-23 simulates a dice. Seven LED's are grouped to form the spots of a dice (see Fig 5-25). The word DINIT initialises port A. Seven lines are used as output, one line as an input. A pushbutton is connected to this line (see Fig 5-24). The word PA@ fetches the content of port A and masks out the seven low order bits. (128 AND). With 128 XOR bit eight is inverted. The number 128 is read if the button is pressed and a zero is read if it is not pressed. The next words 1D thru 6D relate the decimal numbers 1 thru 6 to the six spots of the dice. These words use a binary pattern.

For instance 0001000 turns on the LED in the middle. This is equal to a throw of one. 0D is used as a dummy definition in the CASE: statement CDICE.

```
SCR # 106

( AFB PORT CONTROL cntd     ef)
HEX
: DINIT 30 PACTL C!
        7F PORTA C!
        34 PACTL C!
        00 PORTA C! ;
DECIMAL
: PA@ ( -b) PORTA C@ 128 AND
  128 XOR ;        2 BASE !
: 1D 0001000 PA! ;
: 2D 1000001 PA! ;
: 3D 0101010 PA! ;
: 4D 1010101 PA! ;
: 5D 1011101 PA! ;
: 6D 1110111 PA! ; DECIMAL
: 0D ; -->
```

```
SCR # 107

( AFB PORT CONTROL cntd     ef)
CASE:  CDICE 0D 1D 2D 3D
       4D 5D 6D ;
: BUTTON 1 BEGIN DROP PA@ DUP
    UNTIL 500 WAIT
     BEGIN  WHILE PA@
     REPEAT ;
: DICE 1 BEGIN 1 + DUP 7 =
   IF DROP 1 THEN DUP CDICE
   PA@ UNTIL 500 WAIT 1
   BEGIN WHILE PA@ REPEAT
   DROP ;
: DODI BEGIN BUTTON DICE
  1000 WAIT  ?TERMINAL UNTIL ;
```

Fig 5-23 Simulating a dice.

The word BUTTON waits in the BEGIN...UNTIL loop until the button is pressed. Then a delay loop follows to suppress the bouncing of the key. The BEGIN...WHILE. . . UNTIL loop is left on releasing the button. The word DICE starts the count. Pressing the button stops it and the thrown number is shown on the display.

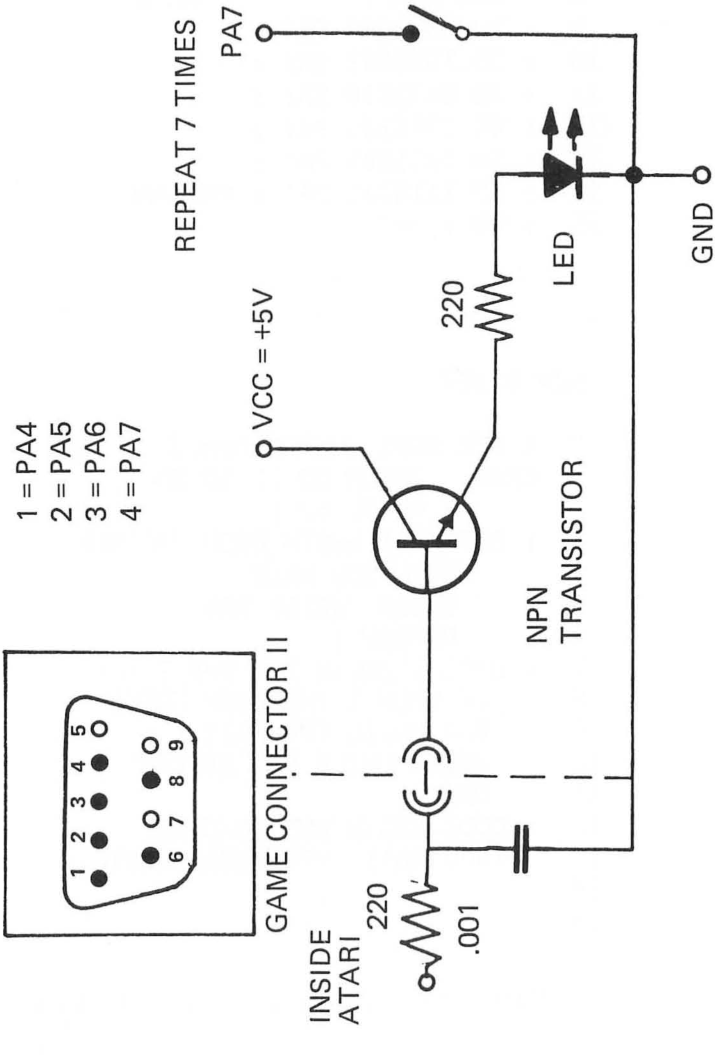


Fig 5-24 Schematic of the dice

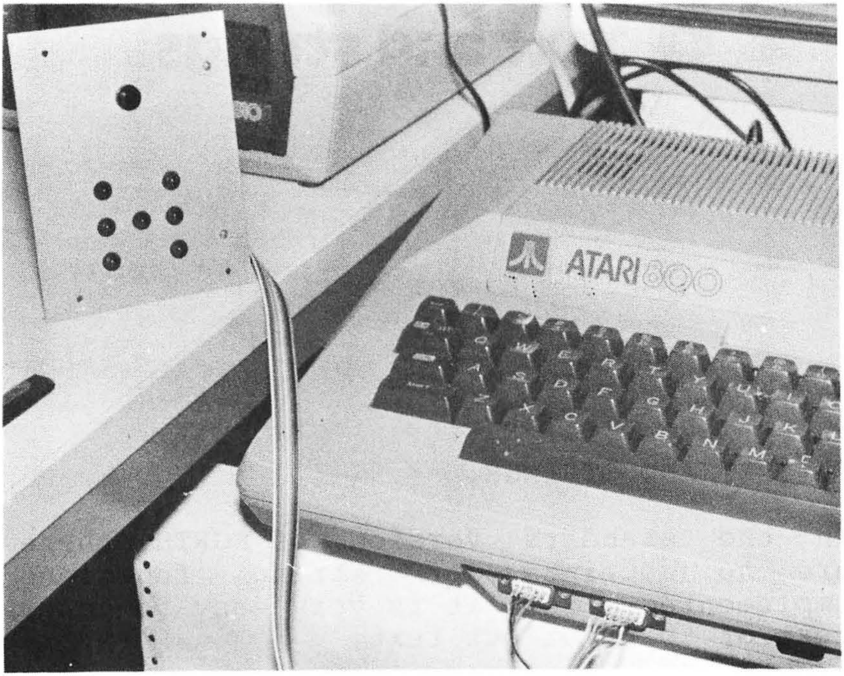


Fig 5-25 The dice.

6 Text and strings

6. TEXT AND STRINGS.

In the standard version of FORTH, there are no strings or string functions implemented. Yet, it is very easy to enter text or to print out text.

6-1 INPUT/OUTPUT OF TEXT.

The word EXPECT expects the input of characters. The definition is:

```
     EXPECT  ( an)  Expects n characters
                    of  text  at address a.
                    Also terminated by  CR.
```

For the input of text you can use a special area in memory, called PAD. The address of PAD is always 68 bytes beyond the top of the vocabulary stack.

```
     PAD  ( -a)     Address of PAD is put
                    on the stack.
```

Example:

```
    PAD 15 EXPECT RET THIS IS FORTH RET
```

The text THIS IS FORTH is stored in PAD. To get the momentary address of PAD type PAD . and try to get the text out onto the screen with DUMP or DDUMP . You can also use the predefined word TYPE.

```
     TYPE  ( an)    Prints n characters of
                    text,     starting   at
                    address a.
```

If we have stored text as in the example above and made no new definitions, which would alter the address of the top of the vocabulary stack, we can get the text from PAD with:

```
      PAD 15 TYPE RET THIS IS FORTH
```

We can make a printout of only a part of the text. We are able to change the starting address and the number of characters.

```
        PAD 2 + 2 TYPE  prints IS.
```

We use a new word, TEXT, to read text into PAD. In some versions TEXT is a predefined word. The definition is:

```
     : TEXT ( c)
       PAD 72 32 FILL WORD HERE
       COUNT PAD SWAP CMOVE ;
```

First, we must describe several new words used in this definition. The word FILL fills memory cells withe the byte b.

```
     FILL    ( anb)   Fill n bytes, starting
                      at address a, with the
                      byte b.
```

The next new word is WORD. There are two different definitions of WORD in FIG FORTH and in FORTH-79.

WORD (c) Read text from the input buffer until the delimiter c is encountered. (FIG FORTH).

WORD (c-an) Read text from the input buffer, until delimiter c is encountered. Leave the address a and the length n of the text on the stack. (FORTH-79)

In our example we use a FIG version of FORTH. Therefore we need two more words HERE and COUNT. HERE leaves the address of the first free memory location of the vocabulary stack on the parameter stack. All text, coming from the keyboard is taken to an input buffer TIB. After hitting RETURN this text is moved to the top of the vocabulary stack.

HERE (a) Leaves the address a of the first free memory location of the vocabulary stack on the parameter stack.

The first byte of the text is the length byte. It gives the length of the text. The word COUNT converts this byte in such a manner that it can be used by TYPE.

COUNT (a-a'n) Converts the length at address a to a' and the length n.

These are the parameters, TYPE needs for printing text. The last word CMOVE moves n bytes from address a to address a'.

```
CMOVE ( aa'n)  Moves n bytes from
               a to a'
```

CMOVE moves the first byte from a to a', the second byte from a+1 to a'+1 and so on. For a correct moving of bytes, the destination address a' must not be in the range of a<a'<a+n.

Example:

```
13 TEXT THIS IS TEXT RET
```

There is no RETURN between the word TEXT and the word THIS. If you now type :

PAD 20 TYPE RET , you get THIS IS TEXT

As a delimiter we used the byte 13. This is the ASCII code for CARRIAGE RETURN. Now let's try this:

```
32 TEXT THIS IS TEXT RET IS ?
```

What happens ? If we type PAD 20 TYPE, we get the printout THIS. We used the ASCII code, 32, of the space character as delimiter. Therefore only the word THIS was moved to PAD. The interpreter finds more text and however tries to interpret it. IS is not a defined word, so we get the error message IS ? .

As an example for text input, we will make the input for a mailing list. The text is stored in PAD. Later we will see how to store this text on the disk.

Screen 91 in Fig 6-1 shows the input of a mailing address. The word FNAME expects 10 characters at the starting address of PAD.

The next word adds ten to the starting address, then it expects an additional ten characters. The other words are defined in the same way.

```
SCR # 091

( AFB TEXT cntd                ef)
: FNAME PAD 10 EXPECT ;
: LNAME PAD 10 + 10 EXPECT ;
: STREET PAD 20 + 15 EXPECT ;
: CITY PAD 35 + 15 EXPECT ;
: STATE PAD 50 + 2 EXPECT ;
: ZIP PAD 52 + 5 EXPECT ;
: INPUT 125 EMIT CR
PAD 58 32 FILL
." FIRST NAME " FNAME   CR
." LAST  NAME " LNAME   CR
."     STREET " STREET  CR
."       CITY " CITY    CR
."      STATE " STATE   CR
."        ZIP " ZIP     CR ;

```

Fig 6-1 Input for a mailing list.

The word INPUT clears the screen and fills the PAD with blanks. Next the message FIRST NAME is displayed on the screen. After this FNAME expects ten characters. In the same way, all the other inputs are made for this mailing list address.

Type in your address and then PAD 58 TYPE. The content of the 58 bytes of PAD are displayed on the screen. All names, if they do not fill all bytes are followed by little hearts. This is the printout of the ASCII zero character. FORTH adds three zeros to the end of the text. On the ATARI screen these zeros are displayed as little hearts. To get rid of this we use a new definition of TYPE, shown in FIG 6-2.

```
: TYPE -DUP IF OVER + SWAP
       DO I C@ 127 AND DUP 0=
          IF DROP ELSE EMIT THEN
       LOOP ELSE DROP ENDIF ;
```

Fig 6-2 Another TYPE.

In the new definition of TYPE, the word -DUP (query dup) is used. This word duplicates the stack only if it non-zero.

The printout of an address is shown in FIG 6-3, screen 92. The word PRINT expects two numbers on the stack. The first number is the starting address, relative to the starting address of PAD. The second number gives the number of characters being printed.

```
SCR # 092

( AFB TEXT OUTPUT                 ef)
: PRINT ( nn') PAD + SWAP
  -TRAILING TYPE ;
: OUT CR 10 0 PRINT SPACE
   10 10 PRINT CR 15 20 PRINT
   CR 15 35 PRINT SPACE
   2 50 PRINT SPACE 5 52 PRINT
   CR ;

```

Fig 6-3 Printout of an address.

For example, 10 0 PRINT prints 10 characters,starting at address zero of PAD.

The word OUT prints the address. An example is shown in Fig 6-4.

```
FIRST NAME EKKEHARD
LAST  NAME FLOEGEL
    STREET 53 REDROCK LANE
      CITY POMONA
     STATE CA
       ZIP 91766
OK
OUT
EKKEHARD FLOEGEL
53 REDROCK LANE
POMONA CA 91766
OK
.SCRN
```

Fig 6-4 Sample printout.

6-2 FORMATTING THE OUTPUT.

FORTH uses several words to format the output. These words require that the number on top of the stack is a double precision, 32 bit, number. A double precision number is entered on the stack, if there is a decimal point in this number.

Examples:

```
123.45  1.2345  1234.
```

These are double precision numbers. There are a few operators used for calculating with these 32-bit numbers.

```
D+  ( dd1-d2)  d2=d1+d

DMINUS ( d-d') d'=-d

D.  ( d)       Prints d.
```

For more words see Appendix A. There is no double precision multiplication or division. These words must be written in machine language.

The words for formatting an output are:

<# (d) Start converting a number into a string.

Convert one digit and add the character to the string.

#S Convert remaining digits.

#> (-an) End of conversion. The address a is the starting address and n is the length of the string.

HOLD (c) Insert the character c in the string.

Example:

We define a word .$ which prints us 10000 for exmple as $100.00 .

```
 : .$ <# # # 46 HOLD #S 36 HOLD #> TYPE ;

          12344. .$ RET $123.44
```

The conversion of this number to a string starts at the end of this number. First the two digits 4 4 are converted. Then a decimal point is inserted, after which the rest of the number is converted. The $ sign is placed in front of this number and the conversion is terminated with #>. This places the address and the length of the string, ready for the word TYPE, on top of the stack. Another example is the printout of a double precision unsigned number:

```
      : UD. <# S# #> TYPE SPACE ;
```

The word NUMBER converts a string to a double precision number.

NUMBER (a-d) Convert a string at a+1 to a double precision number. At address a the length of the string is stored. The variable DPL contains the number of digits right from the decimal point.

We use this word in the program CASH-REGISTER, shown in Fig 6-5.
The word CASH opens the cash register and displays the amount of $0.00 in the upper right corner of the TV screen. The last word in the definition of CASH is the word QUIT. It terminates the interpreter without printing OK. A new amount is added by the word $.

$ 120.00 adds $120 to the previous amount. The result is displayed. $ uses the word NUMBER to convert the string into a double precision number. WORD places the text at HERE, with the first byte as length byte.

```
SCR # 101

( AFB CASHREG 11/12/82       ef)
: CURS ( nnl) 85 ! 84 C! ;
: >S ( d-an) <# # # 46 HOLD #S
   36 HOLD #> ;
: .$ 2DUP    >S DUP 34 SWAP -
   1 SWAP CURS TYPE ;
: CLR 5 2 CURS 20 0 DO 32 EMIT
  LOOP 4 2 CURS ;
: $ ( -d) 13 WORD HERE
  NUMBER D+ .$ CLR QUIT ;

: CASH 125 EMIT 1 2 CURS
   ." CASH:" 0. .$ 4 2 CURS
   ." INPUT:" QUIT ;
```

Fig 6-5 Cash register.

The word >S reconverts the double precision number into a string. This word is used by .$. This word also determines the length of the string and adjusts the printout, that the decimal point is always at the same position on the TV screen.

7 The vocabulary

7. THE VOCABULARY

7-1 DIFFERENT VOCABULARIES.

Every new word is entered into the FORTH vocabulary. But you can create your own vocabulary. This is done with the word VOCABULARY.

VOCABULARY <NAME> Opens vocabulary NAME.

You have to tell the FORTH compiler that all definitions made now, have to be entered in the new vocabulary. This is done with the word DEFINITIONS.

<NAME> DEFINITIONS All new definitions are entered in the vocabulary NAME.

With these two words, two variables are set. The variable CURRENT contains the address of the vocabulary, where the new definitions are entered. The variable CONTEXT contains the address, in which a word is searched for first.

For example:

We are in the FORTH vocabulary and define:

```
        : WHERE ." I AM IN FORTH " ;
```

Now we create a new vocabulary:

```
               VOCABULARY TEST
              TEST DEFINITIONS
```

All new definitions are now entered into the vocabulary TEST. We define the same word WHERE as:

```
         : WHERE ." I AM IN TEST" ;
```

We get the warning WHERE NOT UNIQUE, because we have defined it already in the vocabulary FORTH. If we call now WHERE, we get the message:

```
                I AM IN TEST
```

With the word FORTH, we set the address, stored in CONTEXT to the beginning of the FORTH vocabulary. All words are now searched for first in the FORTH vocabulary. If we now call WHERE, we get the message:

```
                I AM IN FORTH
```

Thats a very powerfull tool. Powerful to the extent of having one word with the same name, defined in two different vocabularies, performing different tasks. We use this to play the piano or an organ with the keyboard of the ATARI.

7-2 PLAY ORGAN OR PIANO WITH THE ATARI.

The numeric keys one to eight of the ATARI keyboard are used to play an organ or piano. They refer to the musical notes of the C scale. The parameters for the frequency of each note are shown in Fig 7-1.

C	60
H	64
A	72
G	81
F	91
E	96
D	108
C	121
H	128
A	144
G	162
F	182
E	193
D	217
C	243

Fig 7-1 Pitch values of the musical notes.

The program is shown in Fig 7-2. The screens 93 and 94 contain the definition of the notes. Each tone is mixed with a tone of half the frequency and a tone which differs slightly from the basic tone.

In the vocabulary PIANO (screen 95, 96) the musical notes are defined as a tone with decreasing volume. In the vocabulary ORGAN (screen 97), they are defined as tones with a constant volume.

SCR # 092

```
( AFB PIANO OR ORGAN 10/22  ef)
: OFF 0 0 0 0 SOUND
      0 0 0 1 SOUND
      0 0 0 2 SOUND ;

: WAIT ( n) 0 DO LOOP ;










-->
```

SCR # 093

```
( AFB TONE TABLE             ef)
15 VARIABLE V
: (C) V @ 10 243 0 SOUND
      V @ 10 240 1 SOUND
      V @ 10 121 2 SOUND ;
: (D) V @ 10 217 0 SOUND
      V @ 10 214 1 SOUND
      V @ 10 108 2 SOUND ;
: (E) V @ 10 193 0 SOUND
      V @ 10 190 1 SOUND
      V @ 10  96 2 SOUND ;
: (F) V @ 10 182 0 SOUND
      V @ 10 179 1 SOUND
      V @ 10  91 2 SOUND ;
-->

```

```
SCR # 094

( AFB TONE TABLE cntd       ef)
: (G) V @ 10 162 0 SOUND
      V @ 10 160 1 SOUND
      V @ 10  81 2 SOUND ;
: (A) V @ 10 144 0 SOUND
      V @ 10 142 1 SOUND
      V @ 10  72 2 SOUND ;
: (H) V @ 10 128 0 SOUND
      V @ 10 126 1 SOUND
      V @ 10  64 2 SOUND ;
: (C1) V @ 10 121 0 SOUND
       V @ 10 119 1 SOUND
       V @ 10  60 2 SOUND ;
: ?TASTE KEY 49 - DUP 0< IF -1
   ELSE DUP 7 > IF -1 THEN
   THEN ; -->
```

```
SCR # 095

( AFB PIANO DEFINITIONS     ef)
VOCABULARY PIANO   IMMEDIATE
PIANO DEFINITIONS
: WAIT 100 0 DO LOOP ;
: C -1 15 DO I V ! (C) WAIT
  -1 +LOOP ;
: D -1 15 DO I V ! (D) WAIT
  -1 +LOOP ;
: E -1 15 DO I V ! (E) WAIT
  -1 +LOOP ;
: F -1 15 DO I V ! (F) WAIT
  -1 +LOOP ;
: G -1 15 DO I V ! (G) WAIT
  -1 +LOOP ;
: A -1 15 DO I V ! (A) WAIT
  -1 +LOOP ; -->
```

SCR # 096

```
( AFB PIANO  cntd           ef)
: H -1 15 DO I V ! (H) WAIT
  -1 +LOOP ;
: C1 -1 15 DO I V ! (C1) WAIT
  -1 +LOOP ;

CASE: TON C D E F G A H C1 ;

: PLAY BEGIN ?TASTE DUP -1 >
       WHILE TON REPEAT
       15 V ! ;


-->
```

SCR # 097

```
( AFB ORGAN DEFINITIONS     ef)
VOCABULARY ORGAN IMMEDIATE
ORGAN DEFINITIONS
15 V !
: C (C) ; : D (D) ; : E (E) ;
: F (F) ; : G (G) ; : A (A) ;
: H (H) ; : C1 (C1) ;


CASE: TON C D E F G A H C1 ;

: PLAY BEGIN ?TASTE DUP -1 >
       WHILE TON REPEAT
       OFF ;
```

Fig 7-2 ATARI plays organ or piano.

The words PIANO PLAY lets you play piano. The words ORGAN PLAY lets you play organ.

The word ?TASTE determines which key was pressed and allows the corresponding note to be picked up by the CASE: statement NOTE.

7-3 THE CONSTRUCTION OF A FORTH WORD.

The word HERE places the address of the first free memory location of the vocabulary stack onto the parameter stack. Type HERE . and you will get a number printed on the screen. The value of this number 'depends of how many definitions you have made. Enter a new definition, such as

```
: ADD + ;
```

and type once more HERE . . The hexdump starting at HERE is shown in Fig 7-3.

```
HERE . 17409

: ADD + ;

HERE . 17421

17409 83 41 44 C4 EB 43 82 26 .ADDkC.&
17417 44 25 B7 24 04          D%7$.
```

Fig 7-3 Hexdump of the word ADD.

HERE AFTER THE DEFINITION	17421			440D
	17419	B7	24	440B
PARAMETER FIELD	17417	44	25	4409
CODEFIELD	17415	82	26	4407
LINK FIELD	17413	EB	43	4405
NAMEFIELD	17411	44	C4	4403
HERE BEFORE THE DEFINITION	17409	83	41	4401

Fig 7-4 Constuction of the word ADD.

The address before the definition of ADD was 17409, the address after the definition is 17421. Fig 7-4 shows the construction of the word ADD. The word starts with the name field. The first byte is the length byte. Here the length of the name is stored in the lower 4 bits. The maximum length of a name is therefore 16 characters.

The next bytes contain the name in ASCII. The highest bit of the last character is set to one. This indicates to the interpreter the name searching routine is at the end of a name.

The next two bytes are the link field. The address stored here points to the previous definition. The search routines uses this address to jump to the next word. The following two bytes are the codefield. This address points to the address where the execution of a word starts.

In our example this is the codefield address of the colon definition. The next two bytes are the parameter field. Stored here are the codefield addresses of the words used in the definition. In our example the first two bytes are the codefield address of the + word and the next two bytes the codefield address of the ; definition. The definition of ADD ends here and the pointer HERE points to the address of the next byte.

We will use the word ADD in a new definition:

```
: #ADD DUP ROT ADD ;
```

FIG 7-5 shows the hexdump and Fig 7-6 the construction of the word #ADD.

```
: #ADD DUP ROT ADD ;

HERE . 17438

17409 83 41 44 C4 EB 43 82 26 .ADDkC.&
17417 44 25 B7 24 84 23 41 44 D%7$.#AD
17425 C4 01 44 82 26 E6 25 C5 D.D.&f%E
17433 28 07 44 B7 24 04       (.D7$.
```

Fig 7-5 Hexdump of the word #ADD.

Notice! In a 6502 CPU system the memory addresses are stored in reverses order. First the low order byte and next the high order byte is stored in two consecutive memory locations.

		17438			441E
	;	17436	B7	24	441C
	ADD	17434	07	44	441A
PARAMETERFIELD	ROT	17432	C5	28	4418
	DUP	17430	E6	25	4416
CODEFIELD	:	17428	82	26	4414
LINKFIELD		17426	01	44	4412
		17425	C4		4411
		17423	41	44	440F
NAMEFIELD		17421	84	23	440D

Fig 7-6 Construction of the word #ADD.

The address 17426 in the linkfield is the namefield address of ADD. In the parameter field you'll find the codefield addresses of DUP, ROT and ADD.

When #ADD is executed, the words DUP and ROT are executed. Next the interpreter jumps to the codefield address of ADD. It executes this word until the semicolon definition, then it jumps back to #ADD. Here the interpreter finds the semicolon definition in #ADD. This terminates the execution of this word. If there are no more words the interpreter prints OK and waits for the next input.

If you know the codefield address of a FORTH word, you can execute it with the word EXECUTE.

```
     EXECUTE ( a)   Execute the word with
                    codefield address a.
```

```
3 5 HEX 2544 DECIMAL EXECUTE . 8 OK
```

7-4 CHANGING THE TOP OF THE DICTIONARY.

We have already used the word HERE, which places on the parameter stack the address of the first free memory location of the vocabulary stack. As pointed out in the last subchapter, the definition of a word starts at HERE and after this, HERE points one byte beyond the end of the word. The definition of HERE is:

```
     : HERE DP @ ;
```

It uses a variable DP. With

```
                 10 DP +!
```

we make a gap in the dictionary. This gap can be used for storing data. The word ALLOT does the same as the line above.

```
     ALLOT    ( n)    Leave a gap of n bytes
                      in the dictionary
```

Attention ! The word ALLOT reserves n bytes in the dictionary. If you want to store n numbers in this gap, you have to reserve place for 2*n bytes.

HERE points to the parameter field. We store the number 1000, for example, at this place with

```
1000 HERE !
```

If we want to store a second number, first we have to add two to HERE, and then we can store it.
We can use the word , (komma) . The definition is:

```
    , ( n)          Places n on top of the
                    dictionary stack. Add
                    two to HERE.
```

Instead of

```
1000 HERE ! 2 DP +!
```

we can write

```
1000 ,          .
```

To read these numbers from the dictionary, we use the word ' (tick).

```
    ' <NAME> ( -a) Places the parameter
                   field address a on the
                   stack.
```

To use this gap in the dictionary, we must name it. We use the word <BUILDS . This is part of a defining word which we discuss in the next subchapter.

<BUILDS <NAME> places the entry NAME into the dictionary.

Fig 7-7 shows an example. We write the word VECTOR into the dictionary, and reserve a place for five numbers. The word !VECTOR stores a number in this area.

It expects two numbers on the stack. The index n and the number itself, which should be stored. If the index is zero, this number is stored in the first place. If the index is four, this number is stored in the fifth place. The word @VECTOR fetches a number out of this array. It expects the index on the stack. Be carefull. If you use an index outside of the specified range the number is stored at this place. This can damage the program and hang up the computer.

```
SCR # 110

( AFB MEMORY RESERVATION    ef)

<BUILDS VECTOR DP @ 10 + DP !

: @VECTOR ( n-n')
  ' VECTOR SWAP 2 * + @ ;

: !VECTOR ( nn')
  ' VECTOR SWAP 2 * + ! ;
```

Fig 7-7 Memory reservation.

Fig 7-8 is another example to store data in an array. We enter the name VECTOR into the dictionary and reserve place for 6 numbers. The word INIT stores a one in the first place. The word +INDEX increases this number by one.

```
SCR # 111

( AFB INDEXED MEM ALLOC.    ef)

<BUILDS VECTOR 12 ALLOT

: INIT (  ) 1 ' VECTOR ! ;

: +INDEX 1 ' VECTOR @ +
   ' VECTOR ! ;

: @VECTOR ( n-n')
  ' VECTOR SWAP 2 * + @ ;

: !VECTOR ( n) ' VECTOR DUP @
  2 * + ! +INDEX ;


```

Fig 7-8 Indexed memory allocation.

The word @VECTOR is the same as in the last example. The word !VECTOR is different. This word takes the parameter field address and duplicates it. Then it takes the number from the first place, multiplies it by two, and adds it to the parameter field address. This gives the new address where the number is stored. The number in the first place is incremented. The words:

```
INIT
1000 !VECTOR
2000 !VECTOR
3000 !VECTOR
```

store the numbers 1000 2000 3000 in three consecutive memory locations. With

```
2 @VECTOR
```

you read the number stored in the second place.

7-5 THE VIRTUAL MEMORY.

In the example above we reserved a place for five numbers. What can we do, if we want to store ten thousand 16 bit numbers? This could only be done with a disk drive.

Now some words, how FORTH handles the disk. This is very easy. If you use a disk, formatted by DOS and no DOS files on it, FORTH recognices this as a memory of 720 blocks with 128 bytes each. The word BLOCK reads one block into the disk buffer.

BLOCK (n-a)	Reads block n if it is not present, in the disk buffer and leaves the address a of the first byte on the stack.

The disk buffer is a region in RAM, reserved for these blocks. If this buffer is full, one block is overwritten. Prior to this, the block was written back to disk if it was marked as updated. The update bit is set by the word UPDATE.

UPDATE ( )	Marks a block as updated.

This word is used in a program when a block is changed and the changings should be saved on disk. Fig 7-9 shows the relation between a block on disk and in memory.

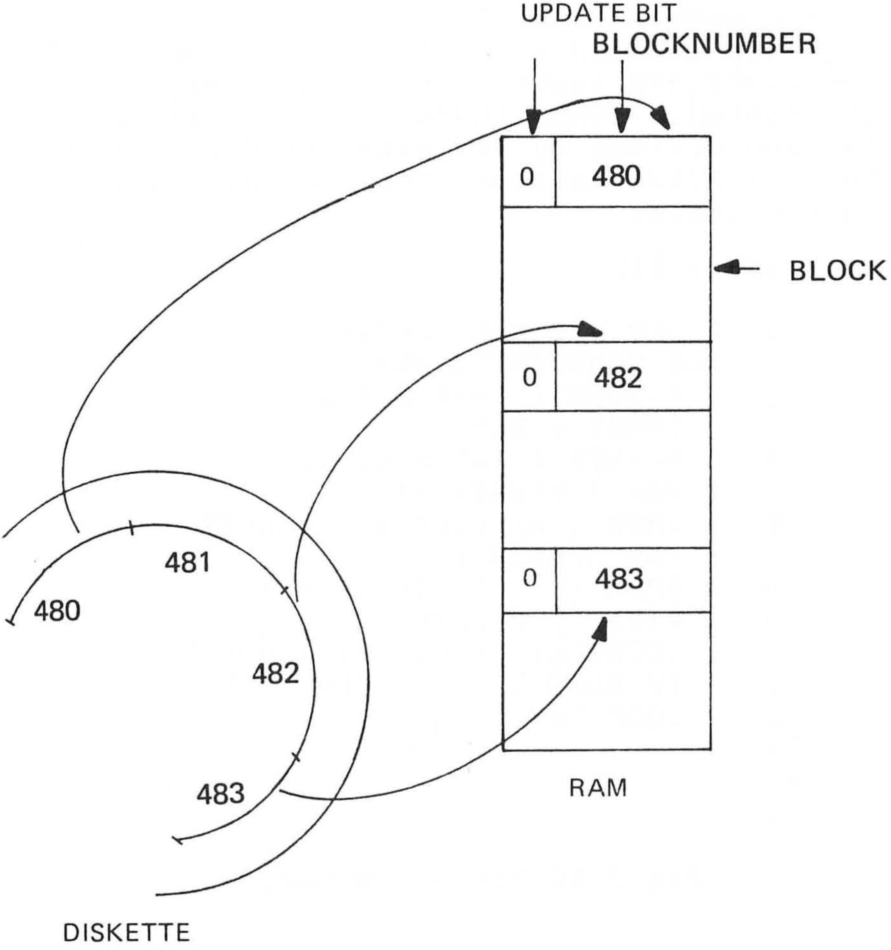


Fig 7-9 Blocks in RAM and on diskette .

The program in Fig 7-10 shows how to use the disk as a virtual memory. To store data on disk we use the same technique as in our last example.

The constant START is the number of the first block on disk we will use for data storage. The blocknumber 480 is equal to screen number 120. The next word #INDEX requests the index of the wanted number on the stack and leaves the address where it is stored. The index n is multiplied by two and divided by 128 with the word /MOD. This leaves the remainder and the quotient on the stack.

```
SCR # 112

( AFB VIRTUAL MEMORY          ef)
480 CONSTANT START
: #INDEX ( n-a) 2 * 128 /MOD
   START + BLOCK + ;
: FIRST# ( -a) 0 #INDEX ;
: +NR 1 FIRST# +! ;
: !MEM ( n) FIRST# @ #INDEX !
   +NR UPDATE ;
: @MEM ( a-n) #INDEX @ ;
: #INIT 1 FIRST# ! ;
: .CONTENT CR FIRST# @ DUP 1 =
   IF ELSE 1 DO I @MEM . CR
   LOOP THEN ;
```

Fig 7-10 Virtual memory.

The constant START is added to the the quotient. This is the block number in which the wanted number is stored. BLOCK brings the address of the first byte on the stack. To this address the remainder is added. The result is the address of the wanted number. FIRST# brings the address of the first byte of block START on the stack. This byte contains the amount of numbers stored on disk. #INIT sets the starting value to one. The words !MEM and @MEM work like !VECTOR and @VECTOR in the last example. The word .CONTENT prints the content of this array. We will use the virtual memory in an application program in the next chapter.

```
#INIT OK
1 !MEM OK
2 !MEM OK
9999 !MEM OK
1234 !MEM OK
FLUSH OK
.CONTENT
1
2
9999
1234
OK
```

Fig 7-11 Example for a virtual memory.

7-6 DEFINITION WORDS

In the last subchapter, we used the word <BUILDS to make an entry in the dictionary. This word is part of the defining word <BUILDS. . .DOES> . In FORTH these defining words are used to create new data structures. Fig 7-12 shows the construction of a defining word.

```
: CONSTANT <BUILDS , DOES> @ ;
```

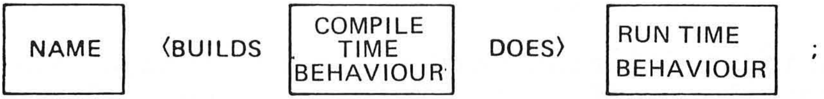


Fig 7-12 Construction of a defining word.

After the word <BUILDS all words are listed which are executed during compiling a word (compile time behaviour). After

DOES> all words are listed, which are executed during running a word (run time behaviour).

Example:
The word CONSTANT is defined as:

```
      : CONSTANT <BUILDS , DOES> @ ;
```

During compile time of CONSTANT a number is stored on top of the dictionary stack (,). During runtime this number is fetched (@) from the stack. As another example we define an array as:

```
:  ARRAY <BUILDS 20 ALLOT DOES> SWAP 2 * + ;
```

We can use the word ARRAY to create new words with the same data structure. They all store ten 16 bit numbers.

```
                   ARRAY VECTOR
```

creates the word VECTOR as a one dimensional array for ten numbers.

```
                  1000 0 VECTOR !
```

stores the number 1000 in the first element of this array and

```
                    0 VECTOR @
```

prints it on the screen.

Using a defining word means to change the compiler of the FORTH system. Other computer languages are sealing the compiler, so that nobody can change it. This does'nt happen in FORTH. You can change the compiler and even can change the compile time behaviour. We will see an example later.

The program in Fig 7-13 is another example for <BUILDS... DOES>. It is a language translator. LIS creates an array for four numbers. 'LIST is a list of addresses. In this list the addresses of the messages . " HELLO" . " GUTEN TAG" and so on are stored.

```
( AFB LANGUAGE TRANSLATOR   ef)
: LIS  <BUILDS 8 ALLOT DOES>
SWAP 2 * + ; LIS 'LIST

: EN ." HELLO " ;
 ' EN 0 'LIST !
: DE ." GUTEN TAG " ;
 ' DE 1 'LIST !
: IT ." BON GIORNO " ;
 ' IT 2 'LIST !
: FR ." BON JOUR " ;
 ' FR 3 'LIST !


  -->

( AFB LANG. TRANSLATOR cntd ef)

0 CONSTANT ENGLISH
1 CONSTANT GERMAN
2 CONSTANT ITALIEN
3 CONSTANT FRENCH

: GREETINGS  'LIST @ 2 -
   EXECUTE ;





  ;S

```

Fig 7-13 Language translator.

The words ENGLISH, GERMAN, ITALIEN and FRENCH are the names of constants. The word GREETINGS fetches the address from 'LIST, subtracts two, to get the codefield address and executes it.

```
GERMAN GREETINGS GUTEN TAG OK
```

8 Applications

8. APPLICATIONS.

8.1 MAILING LIST.

The program in Fig 8-3 is a mailing list. For the input of an address a mask is used. This mask is shown in Fig 8-1 and an example in Fig 8-2.

```
 MAILING LIST



-----------------------------------
-FN              -LN              -
-----------------------------------
-CO                        --------
-----------------------------------
-ST                        --------
-----------------------------------
-CY                      -CT-ZP   -
-----------------------------------
-C1    -C2     -MORE (Y/N)        -
-----------------------------------
 WHAT
```

Fig 8-1 Input mask.

```
 MAILING LIST



-----------------------------------
-EKKEHARD        -FLOEGEL         -
-----------------------------------
-ELCOMP PUBLISHING        --------
-----------------------------------
-53 REDROCK LANE          --------
-----------------------------------
-POMONA                 -CA-91766-
-----------------------------------
-ATARI-99-47  -MORE (Y/N)         -
-----------------------------------
 WHAT
EKKEHARD FLOEGEL
ELCOMP PUBLISHING
53 REDROCK LANE
POMONA CA 91766
```

Fig 8-2 Example for an address input.

The main words of the mailing list are:

NEW

NEW creates a new mailing list. The number FIRST#, stored in the first two bytes in block 100 is set to one. In FIRST# the number of the next entry is stored.

INPUT

INPUT enters the input mode of the mailing list. The mask is placed on the screen and the cursor is placed into the first field. The cursor is moved to the next field by pressing the RETURN key. It is also moved if all places of a field are filled with characters. Typing Y after OK (Y/N) puts the entry into memory. N cancels the entry. For more input type Y after MORE (Y/N) .

.CONTENT

. CONTENT prints the content of the mailing list. Three entries are displayed on the screen at one time. Pressing the space bar displays the next three entries. Any other key cancels the output.

SEARCH <item> <name>

SEARCH searches for a specific entry. SEARCH LN JEFFERSON searches the entry with the last name JEFFERSON. The input SEARCH LN JEF searches all entries in which the last name starts with the characters JEF. The entries are printed three at a time. Pressing the space bar prints the next three entries, until the message END OF LIST appears.

DELETE <item> <name>

DELETE <item> <name> deletes an entry. DELETE LN MILLER searches first for the entry with the last name MILLER. The entry is deleted by Y after the message OK (Y/N). A deleted entry is marked with the character *.

ENTRY

ENTRY replaces a deleted entry with a new one. If there is no deleted entry the message NO DELETED ENTRY, USE INPUT is displayed.

GO

GO erases the TV screen and waits for the input of a main word.

Some words used in the program:

CM sets the background color to a bright yellow. The characters are printed in black on the TV screen.

MASK creates the mask on the TV screen. For printing the words 1R 2R 3R 4R 5R and .- are used.

DESCR <name> creates an entry into the vocabulary with the name <name>. It is used to store the starting address within PAD and the length of a field.

FN	(nn')	First name.
LN	(nn')	Last name.
CO	(nn')	Company.
ST	(nn')	Street.
CY	(nn')	City.
CT	(nn')	State.
ZP	(nn')	Zip code.
C1	(nn')	Codefield 1.
C2	(nn')	Codefield 2.

The two codefields can be used to mark the entries of an address. They are not printed on the screen.

The words in screen four are used to erase an entry within the mask.
(CL) (n) erases n characters starting at the momentary cursor position.
FNC places the cursor at the beginning of the first name field and clears the entry. In the same manner the other words are defined.
The words used in screen #5 are similar to the words defined in chapter six. The virtual memory is expanded to store records with a length of 128 bytes. The same words as in Fig 7-10 are used.

One of the most significant words used in the searching part is (VERGL).

(VERGL) (aa'c-f)

(VERGL) compares two strings starting at address a and a' until a delimiter c is found in the string at address a. If the two strings are equal until the delimiter is found, a one is put on the stack. The zero is left on the stack if they are not equal. In the word VERGL the space character (ASCII 32) is used as delimiter. For large mailing lists (VERGL) should be written in machine language.

(CONTENT) (n)

(CONTENT) prints the entry with the number n on the screen. It uses the variable CNT for counting. After three printouts the screen is cleared and CNT is set to zero.

FOUND moves the content of PAD to PAD + 128, then prints the address found and restors PAD for further searching.

(SEARCH) (nn'0-f)

(SEARCH) searches for an entry between the boundaries n and n'. The zero on top of the stack is replaced by a one if the entry was found. The word (ERASE) is similar.

Used constants and variables:

START contains the number of the first block used for date storage.

RECLEN contains the length of a record.
WO (variable) contains the starting address within PAD for the string comparison.
#NR (variable) contains the number of the entry which is examined.
CNT (variable) is used for counting.

```
SCR # 001

( BUSINESS MASK  10/22        ef)
: CURS ( rocl) 85 ! 84 C!
  155 EMIT ;
: CLR 125 EMIT ; : .- 45 EMIT ;
: CM 222 710 C! 0 709 C! ;
: 1R ( n) 3 CURS 38 3 DO .-
  LOOP ;
: 2R 6 3 CURS .- 6 20 CURS .-
  6 37 CURS .- ;
: 3R ( n) DUP 3 CURS .-  30
  CURS  8 0 DO .- LOOP ;
: 4R 12 3 CURS .- 12 28 CURS .-
  12 31 CURS .- 12 37 CURS .- ;
: 5R 14 3 CURS .- 14 9 CURS .-
  14 17 CURS .- 14 37 CURS .- ;
                            -->
```

```
SCR # 002

( BUSINESS MASK  cntd         ef)
128 CONSTANT RECLEN
: MASK CLR CM 5 1R 2R 7 1R 8 3R
  9 1R 10 3R 11 1R 4R 13 1R
  5R 15 1R ;
( ADDRESS INPUT )
: DESCR 0 VARIABLE -2  ALLOT ;
 DESCR   (FN) 0 , 16 ,
 DESCR   (LN) 16 , 16 ,
 DESCR   (CO) 32 , 26 ,
 DESCR   (ST) 58 , 26 ,
 DESCR   (CY) 84 , 24 ,
 DESCR   (CT) 108 , 2 ,
 DESCR   (ZP) 110 , 5 ,
 DESCR   (NR) 115 , 5 ,
 DESCR   (AM) 120 , 7 , -->
```

```
SCR # 003

( BUSINESS ADDR INPUT cntd  ef)
: 2@ DUP 2 + @ SWAP @ ;
: FN (FN) 2@ ; : LN (LN) 2@ ;
: CO (CO) 2@ ; : ST (ST) 2@ ;
: CY (CY) 2@ ; : CT (CT) 2@ ;
: ZP (ZP) 2@ ;
: C1 (NR) 2@ ; : C2 (AM) 2@ ;
: IN PAD + SWAP EXPECT ;
: (INPUT)  6 4 CURS FN IN
6 21 CURS LN IN 8 4 CURS CO IN
10 4 CURS ST IN 12 4 CURS CY IN
12 29 CURS CT IN
12 32 CURS ZP IN
14 4  CURS C1 IN
14 10 CURS C2 IN ;
                       -->
```

```
SCR # 004

( BUSINESS ADDR INPUT cntd  ef)
: (CL) ( n) 0 DO 32 EMIT LOOP ;
: FNC 6 4 CURS 16 (CL) ;
: LNC 6 21 CURS 16 (CL) ;
: COC 8 4 CURS 26 (CL) ;
: STC 10 4 CURS 26 (CL) ;
: CYC 12 4 CURS 24 (CL) ;
: CTC 12 29 CURS 2 (CL) ;
: ZPC 12 32 CURS 5 (CL) ;
: NRC 14 4 CURS 5 (CL) ;
: AMC 14 10 CURS 7 (CL) ;
: CL FNC LNC COC STC CYC
     CTC ZPC NRC AMC ;
: MC 14 18 CURS  ;
: OK? MC 19 (CL) MC
  ." OK (Y/N) " ;         -->
```

SCR # 005

```
( BUSINESS ADDR INPUT cntd  ef)
: MORE? MC 19 (CL) MC
   ." MORE (Y/N)" ;
: PADC PAD RECLEN 32 FILL ;
( TYPE FOR ATARI              )
: ATYPE -DUP IF OVER + SWAP
      DO I C@ 127 AND DUP 0=
         IF DROP ELSE EMIT THEN
      LOOP ELSE DROP ENDIF ;
: 1S SPACE ; : 5S 5 SPACES ;
: PRINT ( na) PAD + SWAP
 -TRAILING ATYPE ;
: .ADDR CR 5S FN PRINT 1S LN
 PRINT CR 5S CO PRINT CR 5S ST
 PRINT CR 5S CY PRINT 1S CT
 PRINT 1S ZP PRINT ;   -->
```

SCR # 006

```
(     VIRTUAL MEMORY         ef)
100 CONSTANT START
: #INDEX ( n-a) RECLEN * 128
  /MOD START + BLOCK + ;
: FIRST# ( -a) 0 #INDEX ;
: +NR 1 FIRST# +! UPDATE ;
: !MEM PAD  FIRST# @ #INDEX
  RECLEN CMOVE UPDATE +NR ;
: @MEM ( n) #INDEX PAD RECLEN
  CMOVE ;
: NEW 1 FIRST# ! ;

                             -->



```

SCR # 007

```
( BUSINESS ADDR INPUT cntd  ef)
: .MSG 1 4 CURS ." MAILING LIST
 " ;
: .MSG1 16 4 CURS ." WHAT " ;
: GO CLR CM .MSG .MSG1 QUIT ;
: INPUT MASK PADC .MSG
  BEGIN (INPUT) OK? KEY 89 =
  IF  !MEM THEN
  MORE? KEY 89 =
  WHILE CL PADC MC 19 (CL)
  REPEAT .MSG1 FLUSH QUIT ;


 -->


```

SCR # 008

```
 ( BUSINESS SEARCH           ef)
0 VARIABLE WO 1 VARIABLE #NR
 : 2DROP DROP DROP ;
 : ['] [COMPILE] ' ;
 : (VERGL) ( aa'c-f)
   BEGIN ROT DUP C@ >R OVER I =
         R> SWAP DUP IF 0
         ELSE DROP >R ROT DUP C@
         R> = DUP DUP THEN WHILE
         2DROP 1+ >R 1+ R> ROT
   REPEAT >R 2DROP 2DROP R> ;
 : WHAT ['] 2 - EXECUTE SWAP
  DROP DUP WO ! 13 WORD HERE
  COUNT ROT PAD + SWAP CMOVE ;
  -->

```

```
SCR # 009

( BUSINESS SEARCH  cntd      ef)
: VERGL PAD WO @ + #NR @ #INDEX
      WO @ + 32 (VERGL) ;
: NIL CR ." NOT IN LIST "  ;
: EOL CR ." END OF LIST " ;
: .NAME #NR @ @MEM .ADDR ;



-->






```

```
SCR # 010

( BUSINESS OUTPUT              ef)
0 VARIABLE CNT
: 3? ( -f) CNT @ 3 = ;
: (CONTENT) ( n) @MEM .ADDR
  1 CNT +! 3? IF KEY 0 CNT !
  CLR 32 = 1 XOR IF .MSG1 QUIT
  THEN THEN ;
: .CONTENT CLR CM 0 CNT !
   CR FIRST# @ DUP 1 = 1 XOR
  IF 1 DO I (CONTENT)
  CR LOOP THEN .MSG1 QUIT ;




-->
```

```
SCR # 011

( BUSINESS SEARCHING cntd   ef)
: MOVE> PAD PAD 128 + RECLEN
  CMOVE ;
: <MOVE PAD 128 + PAD RECLEN
  CMOVE ;
: FOUND MOVE> #NR @ (CONTENT)
   CR  <MOVE ;
: (SEARCH) DO I #NR ! VERGL
 IF FOUND DROP 1 THEN LOOP ;

: SEARCH 0 CNT ! PADC CLR
  WHAT 0 FIRST# @ 1 (SEARCH)
  IF  EOL ELSE NIL THEN
  .MSG1 QUIT ;
-->
```

```
SCR # 012

( BUSINESS DELETING           ef)
: (ERASE) DO I #NR ! VERGL
  IF DROP 1 LEAVE THEN LOOP ;

: .* #NR @ #INDEX RECLEN
     42 FILL UPDATE ;

: DELETE PADC CLR WHAT 0 FIRST#
   @ 1 (ERASE) IF .NAME OK?
   KEY 89 = IF .* THEN
   ELSE NIL THEN .MSG1 QUIT ;




-->
```

```
SCR # 013

( BUSINESS ENTRY                 ef)
: !ENTRY PAD #NR @ #INDEX
  RECLEN CMOVE UPDATE FLUSH ;

: (ENTRY) CLR MASK BEGIN
  (INPUT) OK? KEY 78 = WHILE
  CL REPEAT !ENTRY ;

: .MSG3 CR ." NO DELETED ENTRY,
USE INPUT " ;
: ENTRY PADC 42 PAD ! 32 PAD
  1 + ! 0 WO ! 0 FIRST# @ 1
  (ERASE) IF (ENTRY) ELSE .MSG3
  THEN .MSG1 QUIT ;

: HELP 14 LIST ;
```

```
SCR # 014

( BUSINESS HELP SCREEN           ef)
( Main words:
NEW starts a new mailing list.
INPUT makes an entry. It is
 placed at the end of the list.
.CONTENT prints the content of
 the list. The printing stops
 after three entries. Use the
 space bar for more. Any other
 key cancels the printing.
SEARCH <item> <name> searches
 for the item with name.
DELETE <item> <name> deletes
the entry with the item name.
ENTRY searches for a deleted
 entry and replaces it.        )
```

Fig 8-3 Mailing list.

8-2 SERIAL OUTPUT VIA GAME PORT THREE.

The game port three is used to transmit data from the ATARI to a printer with a serial input. One half of an RS232 interface is simulated by software. The connections to be made are shown in Fig 8-4.

(FIGURE)

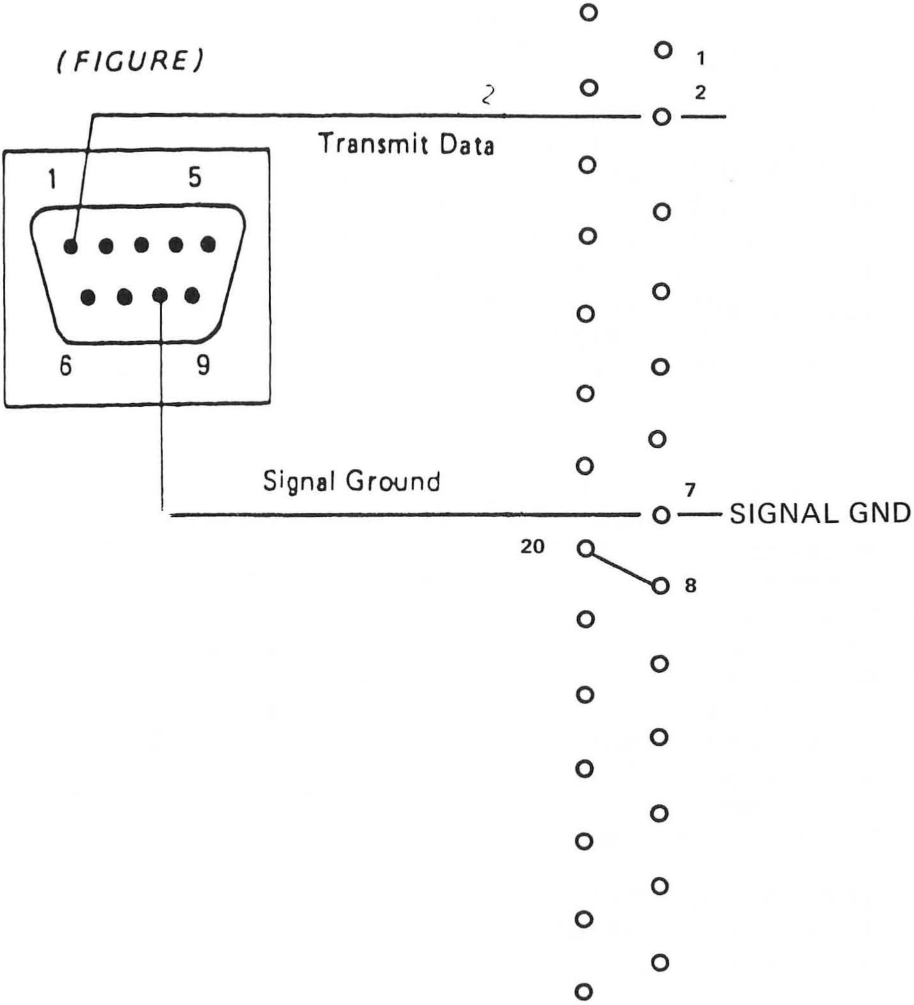


Fig 8-4 Transmit data from the ATARI to a printer or RS232 input.

This application also shows how to use machine code without using an assembler in FORTH. The program is listed in Fig 8-5.

The word CREATE <name> creates an entry into the vocabulary with the name <name>. It writes the parameter field address into the codefield address of this word. The bytes of the machine code are stored in the parameter field by C, . CREATE sets bit 6 of the length byte to one. This makes the name unknown to the search routine of the interpreter. The word SMUDGE sets this bit to zero.
The last instruction of a FORTH word defined in machine code must be a JMP NEXT instead an RTS instruction. With JMP NEXT the FORTH interpreter is called. The address of NEXT is $747 in QS-FORTH and $842 in POWER-FORTH.

The word INIT in text screen #140 initialises the port B. This word could be written with FORTH words. To get the exact timing for the data transfer rate of 300 baud, a subroutine BITWAIT is used. The code for this routine is placed into memory locations $6EB to $6F5. The word OUTCHR takes the byte on the top of the stack and sends it to the printer. During printing all interrupts of the ATARI are disabled. This causes the TV screen to become black and flickering.

The words defined in screen #145 to #149 print the content of a text screen and a TV screen.

.SCREEN (n) Print text screen n.

.SCRN Print TV screen.

```
SCR # 140

( SCREENPRINT RS232 9/29   ef )
CREATE INIT  ( PORT B ) HEX
A9 C, 30 C,          ( LDA #$30   )
8D C, 03 C, D3 C, ( STA $D303 )
A9 C, 01 C,          ( LDA #$01   )
8D C, 01 C, D3 C, ( STA $D301 )
A9 C, 34 C,          ( LDA #$34   )
8D C, 03 C, D3 C, ( STA $D303 )
A9 C, 00 C,          ( LDA #$00   )
8D C, 01 C, D3 C, ( STA $D301 )
4C C, 47 C, 07 C, ( JMP NEXT   )
SMUDGE

DECIMAL   -->


```

```
SCR # 141

( SCREENPRINT RS232 cntd   ef )
HEX  ( BITWAIT )
96A2 6EB !     ( LDX #$96 )
06A0 6ED !     ( LDY #$06 )
88   6EF C!    ( DEY )
FDD0 6F0 !     ( BNE -3    )
CA   6F2 C!    ( DEX )
F8D0 6F3 !     ( BNE -8    )
60   6F5 C!    ( RTS )


DECIMAL

-->


```

SCR # 142

```
( SCREENPRINT RS232 cntd   ef )
HEX
CREATE OUTCHR ( c)
B5 C, 00 C,          ( LDA 00,X   )
E8 C, E8 C,          ( DEC SPTR   )
86 C, B5 C,          ( STX XSAVE  )
49 C, FF C,          ( EOR #$FF   )
8D C, F6 C, 06 C,    ( STA BUFF   )
78 C,                ( SEI        )
A9 C, 00 C,          ( LDA #00    )
8D C, 0E C, D4 C,    ( STA NMIEN  )
8D C, 00 C, D4 C,    ( STA DMACTL)
A9 C, 01 C,          ( LDA #$01   )
8D C, 01 C, D3 C,    ( STA PORTB  )
20 C, EB C, 06 C,    ( JSR WAIT   )
-->
```

SCR # 143

```
( SCREENPRINT RS232 cntd   ef )
A0 C, 08 C,          ( LDY #$08   )
84 C, 1F C,          ( STY COUNT  )
AD C, F6 C, 06 C,    ( LDA BUFF   )
8D C, 01 C, D3 C,    ( STA PORTB  )
6A C,                ( ROR        )
8D C, F6 C, 06 C,    ( STA BUFF   )
20 C, EB C, 06 C,    ( JSR WAIT   )
C6 C, 1F C,          ( DEC COUNT  )
D0 C, EF C,          ( BNE -17    )
A9 C, 00 C,          ( LDA #00    )
8D C, 01 C, D3 C,    ( STA PORTB  )
20 C, EB C, 06 C,    ( JSR WAIT   )
20 C, EB C, 06 C,    ( JSR WAIT   )
A0 C, 01 C,          ( LDY #01    )
-->
```

```
SCR # 144

( SCREENPRINT RS232 cntd   ef )
A9 C, 22 C,          ( LDA #$22   )
8D C, 00 C, D4 C, ( STA DMACTL)
A9 C, FF C,          ( LDA #$FF   )
8D C, 0E C, D4 C, ( STA NMIEN )
58 C,                ( CLI        )
A6 C, B5 C,          ( LDX XSAVE )
4C C, 47 C, 07 C, ( JMP NEXT   )

SMUDGE

DECIMAL

-->


```

```
SCR # 145

( SCREENPRINT                    ef)
3020 VARIABLE ZNR
HEX
0 VARIABLE #SCR -2 ALLOT
53 C, 43 C, 52 C, 20 C,
23 C, 20 C,








DECIMAL -->

```

SCR # 146

```
( SCREENPRINT cntd          ef)
: CRR [ HEX ] 0D OUTCHR
  0A OUTCHR  [ DECIMAL ] ;
: PRINT ( ADR N --> ) 1+ 0
  DO DUP I + C@ OUTCHR LOOP
  DROP ;
: BLANCS 1+ 0 DO [ HEX ] 20
  OUTCHR [ DECIMAL ] LOOP ;
: SCO /MOD 48 + ;
: .ZNR DUP 10 SCO DUP 48 =
   IF DROP 32 THEN OUTCHR
   1 SCO OUTCHR DROP ;
: .TOS DUP 100 SCO OUTCHR
   10 SCO OUTCHR 1 SCO OUTCHR
   DROP ; : .B 10 SCO OUTCHR
   1 SCO OUTCHR DROP ; -->
```

SCR # 147

```
( SCREENPRINT cntd          ef)
DECIMAL INIT
: 1LINE CRR 4 BLANCS ZNR @ .ZNR
  1+ ZNR ! 1 BLANCS ;
: ?CRR 32 MOD 0= IF 1LINE
  THEN ;
: ANF 5 BLANCS #SCR 5 PRINT
  .TOS 0 ZNR ! CRR ;
: ZAUS 128 0 DO DUP BLOCK
  I ?CRR  I + C@ OUTCHR LOOP ;

: .SCREEN ANF 4 * 4 0 DO ZAUS
1+ LOOP CRR DROP ;



-->
```

SCR # 148

```
( SCRN PRINT  10/14/82       ef)
0 VARIABLE ROW 0 VARIABLE COLN
: ?FIN ( -f)  COLN @ 24 = ;
: 1ROW 40 0 DO ROW @ C@ 32 +
  DUP 128 > IF 32 - THEN
  OUTCHR   1 ROW +!  LOOP ;

: .SCRN 88 @ ROW ! 0 COLN !
BEGIN 1ROW CRR 1 COLN +! ?FIN
UNTIL ;






  -->
```

SCR # 149

```
( SCREENPRINT   cndt           ef)
: 6CR 6 0 DO CRR LOOP ;
: 4CR 4 0 DO CRR LOOP ;
: TRI ( n) DUP 3 + SWAP
  DO I .SCREEN 4CR LOOP
  CRR CRR ;
```

Fig 8-5 Serial output via game port three.

9 Appendix

9. APPENDIX

FORTH-GLOSSARY

STACK MANIPULATION

DUP	(n-nn)	Duplicate the top of the stack.
DROP	(n)	Throw away the top element of the stack.
SWAP	(nn'-n'n)	Reverse the two top elements.
OVER	(nn'-nn'n)	Copy second element on top of the stack.
ROT	(nnln2-nln2n)	Rotate the top three elements counterclockwise.
>R	(n)	Move top element to the return stack.
R>	(-n)	Retrieve top element from the return stack.
R	(-n)	Copy top element of return stack to parameter stack

ARITHMETIC AND LOGICAL

+	(nn'-nl)	nl=n+n'
-	(nn'-nl)	nl=n-n'
*	(nn'-nl)	nl=n*n'
/	(nn'-nl)	nl=n/n'
*/	(nln2n3-n)	n=nl*n2/n3 with double precision intermediate.
MOD	(nn'-nl)	nl remainder of n/n'.
/MOD	(nn'-nln2)	nl remainder, n2 qoutient of n/n'.
*/MOD	(nln2n3-nn')	n remainder, n' quotient of nl*n2/n3.
MINUS	(n- -n)	Change sign.
MAX	(nn'-nl)	nl=n if n>n' else nl=n'.
MIN	(nn'-nl)	nl=n if n<n' else nl=n'.

ABS	(n-n')	n' absolute of n.
D+	(dd'-d1)	d1=d+d' double precision addition.
DMINUS	(d-d')	Change sign.
DABS	(d-d')	d' absolute of d.
AND	(nn'-n1)	Logical AND bitwise.
OR	(nn'-n1)	Logical OR bitwise.
XOR	(nn'-n1)	Logical XOR bitwise.

CONTROL STRUCTURES

DO...LOOP		
DO	(nn')	Loops from n' to n-1, loop increment is one.
DO...+LOOP		
DO	(nn')	Loops from n' to n-1.
+LOOP	(n)	Loop increment is n (may be negative).
I	(n)	Put loop index on the stack, same as R.
LEAVE	( )	Terminate loop at next LOOP or +LOOP.
IF <words> THEN (ENDIF)		
IF	(f)	If f is not zero, <words> are executed.
IF <words1> ELSE <words2> THEN (ENDIF)		
IF	(f)	If f is not zero, <words1> are executed, else <words2>.
BEGIN <words> UNTIL (END)		
UNTIL (END)	(f)	<words> are repeated until f is non zero.
BEGIN <words1> WHILE <words2> REPEAT		
WHILE	(f)	If f is zero, program continues after REPEAT, else unconditional branch back from REPEAT to BEGIN.

MEMORY

@	(a-n)	Fetch content from address a and a+1.
C@	(a-b)	Fetch byte from address a.
!	(na)	Store n in address a and a+1.
C!	(ba)	Store byte b in address a.
?	(a)	Print content of address a and a+1.
+!	(na)	Add n to the content of address a and a+1.
CMOVE	(aa'n)	Move n bytes from a to a'. a+n<a'<a.
FILL	(anb)	Store n bytes b into memeory starting at address a.
ERASE	(an)	Store n ASCII 0 into memeory starting at address a.
BLANKS	(an)	Store n ASCII 32 into memory starting at address a.
,	(n)	Store n on top of dictionary. Add two to HERE.
C,	(b)	Store b on top of dictionary. Add one to HERE.
ALLOT	(n)	Leave gap of n bytes on top of dictionary.

COMPARSION

<	(nn'-f)	f=1, if n<n'.
>	(nn'-f)	f=1, if n>n'.
=	(nn'-f)	f=1, if n=n'.
0<	(n-f)	f=1, if n<0.
0=	(n-f)	f=1, if n=0.

NUMBER BASES

DECIMAL	( )	Set decimal base.
HEX	( )	Set hexadecimal base.
BASE	(-a)	Variable, contains number base.

INPUT-OUTPUT

.	(n)	Print n.
." xxx"	( )	Print message xxx. Message ends with " .
.R	(nn')	Print n, rightjustified in field. Fieldwidth is n'.
D.	(d)	Print double precision number d.
D.R	(dn)	Print d rightjustified in field. Fieldwidth is n.
U.	(n)	Print n as unsigned number u.
EMIT	(c)	Print ASCII character c.
TYPE	(an)	Print n bytes starting at address a.
KEY	(-c)	Get character from keyboard.
?TERMINAL	(-f)	f=1, if BREAK key was pressed.
EXPECT	(an)	Expects n bytes at address a.
WORD	(c)	Read characters from the input buffer until delimiter c is found.
COUNT	(a-a'n)	Change length byte at address a to a'=a+1 and length n. Ready for TYPE.

NUMBER FORMATTING

		Start converting a number to a string.
<#	( )	Convert one digit and add it to the string.
#	( )	Convert remaining digits.
#S	( )	End of conversion. Puts starting address a and length n on the stack.
#>	(an)	Ready for TYPE.
		Inserts the ASCII character into the string.
HOLD	(c)	Inserts sign of n into the string.
SIGN	(n)	Convert a string at address a+1 to a double precision number.
NUMBER	(a-d)	The length of the sring must be stored at address a.

DEFINING WORDS

: xxx	( )	Begin of a colon definition.
;	( )	End of a colon definition.
VARIABLE <name>	(n)	Create variable <name> with the initial value n.
<name>	(a)	The address a of the variable is put on the stack.
CONSTANT <name>	(n)	Create a constant <name> with the value n.
<name>	(n)	The value n of the constant is returned.
<BUILDS <wordsl> DOES> <words2>		Used to create a new defining word. <wordsl> are executed during compile time. <words2> are executed during run time.
CREATE <name>	( )	Creates an entry <name> into the vocabulary. The codefield address of <name> is the parameter field address.

VOCABULARIES

CONTEXT	(a)	Returns address of CONTEXT vocabulary. This is searched first.
CURRENT	(a)	Returns address of CURRENT vocabulary. New definitions are put here.
FORTH	( )	Main FORTH vocabulary.
VOCABULARY <name>	( )	Opens new vocabulary. Sets CURRENT to <name>.
DEFINITIONS	( )	Sets CONTEXT to CURRENT.
VLIST	( )	Print all words.
FORGET <name>	( )	Forqet all definitions back and including <name>.
' <name>	(a)	Get parameter field address of <name>.

DISK

LIST	(n)	List content of text screen n.
LOAD	(n)	Compile text screen n into dictionary.
BLOCK	(n-a)	Read block n.
EMPTY-BUFFERS	( )	Erase all disk buffers.
UPDATE	( )	Mark last buffer accessed.
FLUSH	( )	Save all updated disk buffers.
INDEX	(nn')	List all first lines of text screens n to n'.

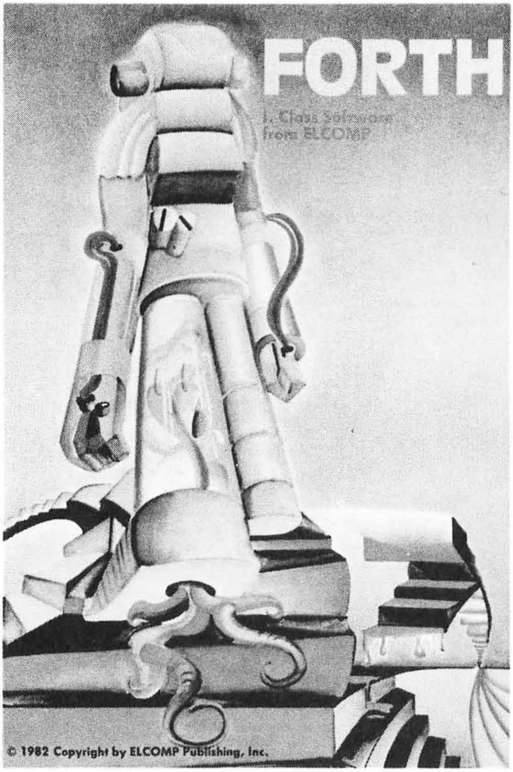
SOME VARIABLES

DP	(-a)	Dictionary pointer. Contains the first free memory location on top of the vocabulary.
HERE	(-n)	Fetches DP.
PAD	(-a)	Scratch buffer PAD. 68 bytes above HERE.

TIB	(-a)	Terminal input buffer.
IN	(-n)	Offset to terminal input buffer.
S0	(-a)	Contains the initial address of the parameter stack.
SP@	(-a)	Fetch content of S0.

BLK	(-a)	Contains current block number.
SCR	(-a)	Contains current screen number.
B/BUF	(-n)	Constant, gives block size in bytes.

FORTH for the ATARI®



Learn-FORTH

A subset of Fig-Forth for the beginner. On disk (32K RAM) or on cassette (16K RAM). Even the ATARI 400 or ATARI 800 /16K RAM owner can program in FORTH.
Order-No. 7053
$19.95

POWER FORTH is an extended Fig-FORTH version, Editor and I/O package included. Utility package includes decompiler, sector copy, Hex-dump (ASCII), ATARI Filehandling, total graphic and sound; joystickprogram and player missile. Extremely powerful. Two game demos (sound and animation) and a mailing list, written in FORTH are included.
Order-No. 7055 **disk** **$39.95**

Floating point package for POWER FORTH with trigonometric functions (0—90°).
Order-No. 7230 **disk** **$29.95**

NOTES

NOTES

NOTES

NOTES

NOTES

ORDER FORM
HOFACKER

ELCOMP PUBLISHING, INC., 53 Redrock Lane, Pomona, CA 91766 (Phone: (714) 623-8314)

Please make checks out to ELCOMP PUBLISHING,INC.

Please bill to my Master Card or Visa account # ....

Name: ....

Card # ....

Address: ....

Expiration Date ....

Master Charge Bank Code ....

City / State / Zip: ....

Signature ....

Qty.	Order No.	Description	Price $	Qty.	Order No.	Description	Price $
.....	29	MICROC. HARDWARE HANDB.	14.95	.....	4889	EXPANDING YOUR VIC	14.95
.....	150	CARE AND FEEDING...	9.95	.....	4894	RUNFILL	9.95
.....	151	8K MICROSOFT BASIC	9.95	.....	4896	MINIASSEMBLER	19.95
.....	152	EXP.HANDB. 6502 AND 6802	9.95	.....	6153	LEARN-FORTH	19.95
.....	153	MICROC. APPLICATION NOTES	9.95	.....	6155	POWER FORTH (APPLE)	39.95
.....	154	COMPLEX SOUND	6.95	.....	7022	ATMONA-1, CASS.	19.95
.....	156	SMALL BUSINESS PROGR.	14.90	.....	7023	ATMONA-1, DISK	24.95
.....	158	SECOND BOOK OF OHIO	7.95	.....	7024	ATMONA-1, CARTRIDGE	59.00
.....	159	THE THIRD BOOK OF OHIO	7.95	.....			
.....	160	THE FOURTH BOOK OF OHIO	9.95	.....	7042	EPROM BURNER FOR ATARI	179.00
.....	161	THE FIFTH BOOK OF OHIO	7.95	.....	7043	EPROM BOARD (CARTRIDGE)	29.95
.....	162	GAMES FOR THE ATARI	7.95	.....	7049	ATMONA-2, CASS.	49.95
.....	164	ATARI LEARNING BY USING	7.95	.....	7050	ATMONA-2, DISK	54.00
.....	166	PROGR. I.6502 MACH.LANG.	19.95	.....	7053	LEARN-FORTH	19.95
.....	169	HOW TO PROGR.IN MACH.L.	9.95	.....	7055	POWER FORTH	39.95
.....	170	FORTH ON THE ATARI	7.95	.....	7098	ATAS	49.95
.....	171	A LOOK INTO THE FUTURE	9.95	.....	7099	ATMAS	89.00
.....	173	PROGRAM DESCRIPTIONS	4.95	.....	7100	PROGRAMS FROM BOOK # 164	29.95
.....	174	ZX-81/TIMEX	9.95	.....	7200	INVOICE WRITING,DISK	39.00
.....	176	TRICKS FOR VICS	9.95	.....	7207	GUNFIGHT FOR ATARI	19.95
.....	202	JANA/1 MONITOR	29.95	.....	7210	WORDPROCESSOR, CASS.	29.95
.....	604	PROTOTYPING CARD	29.00	.....	7211	HOW TO CONNECT...	19.95
.....	605	6522 VIA I/O EXP. CARD	39.00	.....	7212	MAILING LIST, CASS.	19.95
.....	606	SLOT REPEATER	49.00	.....	7213	MAILING LIST, DISK	24.95
.....	607	2716 EPROM PROGRAMMER	49.00	.....	7214	INVENTORY CONTR., CASS.	19.95
.....	608	SOUND WITH THE GI AY3-8912	39.00	.....	7215	INVENTORY CONTR., DISK	24.95
.....	609	8K EPROM CARD (2716)	29.00	.....	7216	WORDPROCESSOR, DISK	34.95
.....	610	12 BIT A/D CONVERTER BOAR	74.00	.....	7217	WORDPROCESSOR, CARTRIDGE	69.00
.....	615	16K RAMROM-BOARD	59.95	.....	7221	PROGRAMS FROM BOOK # 162	29.95
.....	680	THE CUSTOM APPLE BOOK	24.95	.....	7222	KNAUS OGINO	29.95
.....	2398	MAILING LIST FOR ZX-81	19.95	.....	7223	ASTROLOGY PROGRAM	29.95
.....	2399	MACHINE LANG. MONITOR	9.95	.....	7224	EPROM BOARD KIT	14.94
.....	2400	ADAPTER BOARD FOR ZX-81	14.80	.....	7230	FLOATING POINT PACKAGE	299.95
.....	3276	EDITOR/ASS. FOR CBM	39.00	.....	7291	RS232-INTERFACE	19.95
.....	3475	ASSEMBLER FOR CBM	39.95	.....	7292	EPROM BURNER KIT	49.00
.....	4826	GUNFIGHT FOR PET/CBM	9.95	.....	7307	ATCASH	49.95
.....	4844	UNIVERSAL EXP. BOARD	18.95	.....	7309	MOON PHASES	19.95
.....	4848	ADAPTER BOARD	3.95	.....	7310	ATAMEMO	29.95
.....	4870	PROFI WORDPROC. F. VIC	19.95	.....	7312	SUPERMAIL	49.00
.....	4880	TIC TAC VIC	9.95	.....	7313	BUSIPACK 1	98.00
.....	4881	GAMEPACK FOR VIC	14.95	.....	7314	BIORHYTHM FOR ATARI	9.95
.....	4883	MAIL-VIC	14.95	.....	....	....	....

Payment: Check, Money Order, VISA, Mastercharge, Access, Interbank, Eurocheck
Prepaid orders add $3.50 for shipping (USA)
$5.00 handling fee for C.O.D.
ALL ORDERS OUTSIDE USA: Add 15 % shipping.
California residents add 6.5% sales tax.

Superboard and C1P are trademarks of Ohio Scientific Co.
ATARI is a trademark of ATARI Warner Communications
PET, CBM and VIC-20 are trademarks of Commodore Business Machines.
TRS-80 is a trademark of TANDY Radio Shack.
APPLE II is a trademark of APPLE Computer Inc.

BITFIRE

BITFIRE

BITFIRE

BITFIRE

BITFIRE

BITFIRE

BITFIRE