Contents

Chapter 1

Programming in FORTH

This chapter briefly reviews the main ideas of FORTH to let the reader understand the program fragments and subroutines that comprise the meat of this book. We make no pretense to complete coverage of standard FORTH programming methods. **Chapter 2 is not a programmers manual!**

Suppose the reader is stimulated to try FORTH - how can he proceed? Several excellent FORTH texts and references are available: *Starting FORTH* [1] and *Thinking FORTH* [2] by Leo Brodie; and *FORTH: a Text and Reference* [3] by M.Kelly and N.Spies. I strongly recommend reading **FTR** or **SF** (or both) before trying to use the ideas from this book on a FORTH system. (Or at least read one concurrently.)

The (commercial) GEnie information network maintains a session devoted to FORTH under the aegis of the Forth Interest Group (fiG).

FIG publishes a journal *Forth Dimensions* whose object is the exchange of programming ideas and clever tricks.

The Association for Computing Machinery (11 West 42nd St., New York, NY 10036) maintains a Special Interest Group on FORTH (SIGForth).

The Institute for Applied FORTH Research (Rochester, NY) publishes the refereed *Journal of FORTH Application and Research*, that serves as a vehicle for more scholarly and theoretical papers dealing with FORTH.

---

[1] L. Brodie, *Starting FORTH*, 2nd ed. (Prentice-Hall, NJ, 1986), referred to hereafter as **SF** .

[2] L. Brodie, *Thinking FORTH* (Prentice-Hall, NJ 1984), referred to hereafter as **TF** .

[3] M. Kelly and N. Spies, FORTH: a Tea and Reference (Prentice-Hall, NJ , 1986), referred to hereafter as **FTR**.

finally, an attempt to codify and standardize FORTH is underway, so by the time this book appears the first draft of an ANS FORTH and extensions may exist.

The structure of FORTH

The "atom" of FORTH is a **word** a previously-defined operation (defined in terms of machine code or other, previously-defined words) whose definition is stored in a series of linked lists called the **dictionary**. The FORTH operating system is an endless loop (outer interpreter) that reads the console and interprets the input stream, consulting the dictionary as necessary. If the stream contains a word[4] in the dictionary the interpreter immediately executes that word.
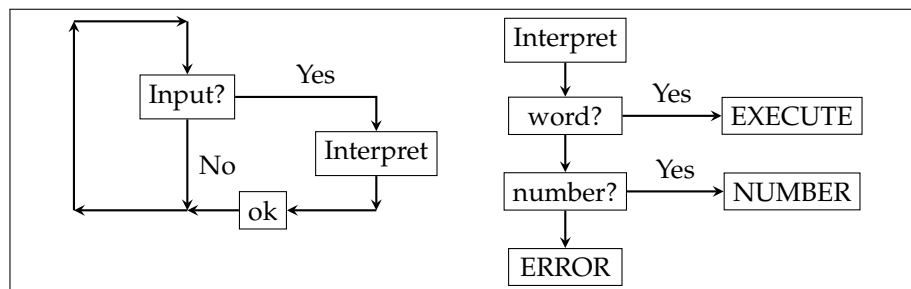
Figure 1.1: *Overview of FORTH outer interpreter*

In general, because FORTH is interpretive as well as compiled, the best way to study something new is in front of a computer running FORTH. Therefore we explain with illustrations, expecting the reader to try them out.

In what follows, anything the user types in will be set in `Helvetica`, such as `DECIMAL` below.

Machine responses appear in ordinary type.

We now give a trivial illustration:

```
DECIMAL <cr> ok
```

Notes:

- **<cr>** means "the user pushes the ENTER or ⇐ button".

- **ok** is what FORTH says in response to an input line, if nothing has gone wrong.

---

[4]Successive words in the input stream are separated from each other by blank spaces, ASCII 20hex, the standard FORTH delimiter.

- `DECIMAL` is an instruction to use base 10 arithmetic. FORTH will use any base on tell it, within reason, but usually only `DECIMAL` and `HEX` (hexadecimal) are predefined.

When the outer interpreter (see Fig. 2.1 on p. 13) encounters text with no dictionary entry, it tries to interpret it as a **NUMBER**.

It places the number in a special memory location called "the top of the stack" (TOS)[5]

```
2 17 +. <cr> 19 ok
```

Notes:

- FORTH interprets 2 and 17 as numbers, and pushes them onto the stack. "**+**" is a word and so is "**.**" so they are **EXECUTE**d.

- + adds 2 to 17 and leaves 19 on the stack.

- The word **.** (called "emit") removes 19 from the stack and displays it on the screen.

We might also have said[6]

```
HEX 0A 14 * . <cr> C8 ok
```

(Do you understand this? Hint: **HEX** stands for "switch to hexadecimal arithmetic")

If the incoming text can neither be located in the dictionary nor interpreted as a number, FORTH issues an error message.

Extending the dictionary

The compiler is one of FORTH's most endearing features. It is elegant, simple, and mostly written in FORTH. Although the technical details of the FORTH compiler are generally more interesting to systems developers than to scientists, its components can often be used to solve programming problems. When this is the case, we necessarily discuss details of the compiler. In this section we discuss how the compiler extends the dictionary. In § 2 § § 8 below we examine the parts of the compiler in greater detail.

FORTH has special words that allow the creation of new dictionary entries, *i.e.*, new words. The most important are "**:**" ("start a new definition") and "**;**" ("end the new definition").

Consider the phrase

---

[5]We will explain about the stack in 2.3.

[6]Since FORTH uses words, when we enter an input line we say the corresponding phrase.

```
: NEW-WORD WORD1 17 WORDZ . . . WORDn ; ok
```

The initial "**:**" is **EXECUTE**d because it is already in the dictionary. Upon execution, "**:**" does the following:

- Creates a new dictionary entry, **NEW-WORD**, and switches from **interpret**- to **compile** mode.

- In compile mode, the interpreter looks up words and – rather than executin them – installs ointers to their code. If the text is a number (**17** above), FORTH builds the literal number into the dictionary space allotted for **NEW-WORD**.

- The action of **NEW-WORD** will be to **EXECUTE** sequentially the previously-defined words **WORD1**, **WORD2**, ...**WORDn**, placing any built-in numbers on the stack as they occur.

- The FORTH compiler **EXECUTE**s the last word "**;**" of the definition, by installing code (to return control to the next outer level of the interpreter[7]) then switching back from compile to interpret mode. Most other languages treat tokens like "**;**" as ags (in the input stream) that *trigger* actions, rather than actions in their own right FORTH lets components execute themselves.

In FORTH *all* subroutines are words that are invoked when they are named. No explicit CALL or GOSUB statement is required.

The above definition of **NEW-WORD** is extremely structured compared with FORTRAN or BASIC. Its definition is just a series of subroutine calls.

We now illustrate how to define and use a new word using the previously defined words "**:**"and "**;**". Enter the phrase (this new word ***+** expects 3 numbers, *a*, *b*, and *c* on the stack)

```
: *+    * + ; ok
```

Notes:

- ***** multiplies b with c, leaving b*c.

- **+** then adds b*c to a, leaving a + b*c behind.

Now we actually try out ***+** :

```
DECIMAL 5 6 7 *+ . 47 ok
```

Notes:

- The period **.** is not a typo, it EMITs the result.

- FORTH's response to **a b c *+ .** is a + b*c **ok**.

---

[7]This level could be either the outer interpreter or a word that invokes **NEW-WORD**.

What if we were to enter ***+** with nothing on the stack? Let's try it and see (**.S** is a word that displays the stack without changing its contents):

```
.S empty stack ok

*+ empty stack ok
```

....

Exercise:

Suppose you entered the input line

```
HEX 5 6 7 *+ . <cr> xxx ok
```

What would you expect the response **xxx** to be?

Answer: **2F**

....

Stacks and reverse Polish notation (RPN)

We now discuss the stack and the "reverse Polish" or "postfix" arithmetic based on it. (Anyone who has used one of the Hewlett-Packard calculators should already be familiar with the basic concepts.)

A Polish mathematician (J .Lukasewcleia) showed that numerical calculations require an irreducible minimum of elementary operations (fetching and storing numbers as well as addition, subtraction, multiplication and division). The minimum is obtained when the calculation is organized by "stack" arithmetic.

Thus virtually all central processors (CPU's) intended for arithmetic operations are designed around stacks. FORTH makes efficient use of CPU's by reflecting this underlying stack architecture in its syntax, rather than translating algebraic-looking program statements ("infix" notation) into RPN-based machine operations as FORTRAN, BASIC, C and Pascal do.

But what *is* a stack? As the name implies, a stack is the machine analog of a pile of cards with numbers written on them. Numbers are always added to, and removed from, the top of the pile. (That is, a stack resembles a job where layoffs follow seniority: last in, first out.) Thus, the FORTH input line

```
DECIMAL 2 5 73 -16 ok
```

followed by the line

```
+ - * . yyy ok
```

leaves the stack in the successive states shown in Table 2-1 below

Cell#	Initial	Ops→	+	-	*	.
0	-16	Result	57	-52	104	...
1	73	→	5	2	...	...
2	5		2	...	...	...
3	2			...	...	...

Table 2-1 *Picture of the stack during operations*

We usually employ zero-based relative numbering in FORTH data structures –stacks, arrays, tables, *etc.*so TOS ("top of stack") is given relative #0, NOS ("next on stack") #1, *etc.*

The operation "**.**" ("emit") displays -104 to the screen, leaving the stack empty. That is, **yyy** above is **-104**.

Manipulating the parameter stack

FORTH system incorporate (at least) two stacks: the **parameter** stack which we now discuss, and the **return stack** which we defer to 2.3.2.

In order to use a stack-based system, we must be able to put numbers on the stack, remove them, and rearrange their order. FORTH includes standard words for this purpose.

Putting numbers on the stack is easy: one simply types the number (or it appears in the definition of a FORTH word).

To remove a number we have the word **DROP** that drops the number from TOS and moves up all the other numbers.

Tb exchange the top 2 numbers we have .

DUP duplicates the TOS into NOS, pushing down all the other numbers.

ROT rotates the top 3 numbers.

Cell#	Initial	Ops→	DROP	SWAP	ROT	DUP
0	-16	Result	73	73	5	-16
1	73	→	5	-16	-16	-16
2	5		2	5	73	73
3	2		...	2	2	5
4	...		...	...	...	2

Table 2-2 *Stack manipulation operators*

These actions are shown on page 19 above in Table 2-2 (we show what each word does to the initial stack).

In addition the words **OVER**, **UNDER**, **PICK** and **ROLL** act as shown in Table 2-3 below (note **PICK** and **ROLL** must be preceded by an integer that says where on the stack an element gets **PICK**ed or **ROLL**ed).

Cell#	Initial	Ops→	OVER	UNDER	4 PICK	4 ROLL
0	-16	Result	73	-16	2	2
1	73	→	-16	73	-16	-16
2	5		73	-16	73	73
3	2		5	5	5	5
4	...		2	2	2	...

Table 2-3 *More stack manipulation operators*

Clearly, **1 PICK** is the same as **DUP**, **2 PICK** is a synonym for **OVER**, **2 ROLL** means **SWAP**, and **3 ROLL** means **ROT**.

As Brodie has noted (**TF**), it is rarely advisable to have aword use a stack so deep that **PICK** or **ROLL** is needed. It is generally better to keep word definitions short, using only a small number of arguments on the stack and consuming them to the extent possible. On the other hand, **ROT** and its opposite, **-ROT**[8], are often useful.

The return stack and Its uses

We have remarked above in 22 that compilation establishes links from the calling word to the previously- defined word being invoked. Part of the linkage mechanism –during actual execution– is the **return stack** (rstack): the address of the next word to be invoked after the currently executing word is placed on the rstack, so that when the current word is done, the system jumps to the next word. Although it might seem logical to call the address on the rstack the **next** address, it is actually called the **return** address for historical reasons.

In addition to serving as a reservoir of return addresses (since words can be nested, the return addresses need a stack to be put on) the rstack is where the limits of a **DO ... LOOP** construct are placed[9]

The user can also store/retrieve to/from the rstack This is an example of using a component for a purpose other than the one it was designed for. Such use is not encouraged by every FORTH text, needless to say, since it introduces the spice of danger. To store to the rstack we say **>R**, and to retrieve we say **R>**. **DUP>R** is a speedup of the phrase **DUP >R**. The words **D>R DR>** , for maving double-length integers, also exist on many systems. The word **R@** copies the top of the rstack to the TOS.

---

[8]defined as **: -ROT ROT ROT ;**
[9]We discuss looping in 2.7 below.

The danger is this: anything put on the rstack during a words execution must be removed before the word terminates. If the `>R` and the `R>` do not balance, then a **wrong next address** will be jumped to and **EXECUTE**d. Since this could be the address of data, and since it is being interpreted as machine instructions, the results will be **always unpredictable**, but seldom amusing.

Why would we want to use the rstack for storage when we have a perfectly good parameter stack to play with? Sometimes it becomes simply impossible to read code that performs complex gymnastics on the parameter stack, even though FORTH permits such gymnastics.

Consider a problem –say, drawing a line on a bit- mapped graphics output device from (x,y) to (x′,y′)– that requires 4 arguments. We have to turn on the appropriate pixels in the memory area representing the display, in the ranges from the origin to the end coordinates of the line. Suppose we want to work with x and y first, but they are 3rd and 4th on the stack. So we have to **ROLL** or **PICK** to get them to TOS where they can be worked with conveniently. We probably need them again, so we use

```
4PICK 4PICK ( -- x y x' y' x y)
```

Now 6 arguments are on the stack! See what I mean? A better way stores temporarily the arguments x and y′, leaving only 2 on the stack. If we need to duplicate them, we can do it with an already existing word, **DDUP**.

Complex stack manipulations can be avoided by defining **VARIABLE**s – named locations– to store numbers. Since FORTH, variables are typically *global* –any word can access them– their use can lead to unfortunate and unexpected interactions among parts of a large program. Variables should be used sparingly.

While FORTH permits us to make variables local to the sub- if routines that use them[10], for many purposes the rstack can advantageously replace local variables:

- The rstack already exists, so it need not be defined anew.

- When the numbers placed on it are removed, the rstack shrinks, thereby reclaiming some memory.

Suppose, in the previous example, we had put x and y on the rstack via the phrase

```
>R >R DDUP  .
```

Then we could duplicate and access x and y with no trouble.

---

[10]See **FTR**, p. 3253 for a description of beheading - a process to make variables local to a small set of subroutines. Another technique is to embed variables within a data structure so they cannot be referenccd inadvertently. Chapters 283-2, 352, 512 and 112 offer examples.

A note of caution: since the rstack is a critical component of the execution mechanism, we mess with it at our peril. If we want to use it, we must clean up when we are done, so it is in the same state as when we found it. A word that places a number on the rstack must get it off again –using **R>** or **RDROP**– before exiting that word[11]. Similarly, since no LOOP uses the rstack also, for each ¿R in such a loop (after DO) there must be a corresponding R ¿ or RDROP (before LOOP is reached). Otherwise the results will be unpredictable and probably will crash the system.

Fetching and storing

Ordinary (16-bit) numbers are *fetc*hed from memory to the stack by "**@**" ("fetch"), and stored by "**!**" ("store"). The word @ expects an address on the stack and replaces that address by its contents using, *e.g.*, the phrase **X @**. The word "!" expects a number (NOS) and an address (T OS) to store it in, and places the number in the memory location referred to by the address, consuming both arguments in the process, as in the phrase **32 X !**

Double length (32-bit) numbers can similarly be *fetc*hed and stored, by **D@** and **DI** . (FORTH systems designed for the newer 32-bit machines sometimes use a 32-bit-wide stack and may not distinguish between single- and double-length integers.)

Positive numbers smaller than 255 can be placed in single bytes of memory using **C@** and **C!**. This is convenient for operations with strings of ASCII text, for example screen, file and keyboard I/O.

In Chapters 3, 4, S and 7 we shall extend the lexicon of @ and ! words to include floating point and complex numbers.

Arlthmetlc operations

The 1979 or 1983 standards, not to mention the forthcoming ANSII standard, require that a conforming FORTH system contain a certain minimum set of predefined words. These consist of arithmetic operators **+ – * / MOD /MOD */** for (usually) 16-bit *signed-integer* (-32767 to +32767) arithmetic, and equivalents for *unsigned* (0 to 65535), double-length and mixed-mode (16- mixed with 32-bit) arithmetic. The list will be found in the glossary accompanying your system, as well as in **SF** and **FTR**.

Comparing and testlng

I n addition to arithmetic, FORTH lets us compare numbers on the stack, using relational operators `>` `<` `=`.These operators work as follows: the phrase

```
2 3 > < cr > ok
```

will leave 0 ("false") on the stack, because 2 (N0S) is not greater than 3 (TOS). Conversely, the phrase

```
23 < <cr> ok
```

will leave -1 ("true") because 2 is less than 3. Relational operators typically consume their arguments and leave a "ag" to show what happened[12]. Those listed so far work with signed 16-bit integers. The operator **U<** tests *unsigned* 16-bit integers (0-65535).

FORTH offers unary relational operators **0=** **0>** and **0<** that determine whether the TOS contains a (signed) 16-bit integer that is 0, positive or negative. Most FORTHs offer equivalent relational operators for use with double-length integers.

The relational words are used for branching and control. The usual form is

```
: MAYBE 0> IF WORDl WORD2 ...
  WORDn THEN ;
```

The word **MAYBE** expects a number on the stack, and executes the words between **IF** and **THEN** if the number on the stack is positive, but not otherwise. If the number initially on the stack were negative or zero, **MAYBE** would do nothing.

An alternate form including **ELSE** allows two mutually exclusive actions:

```
: CHOOSE 0> IF WORD1 . . . WORDn
        ELSE WORDt' . . . WORDn'
        THEN ; (n -- )
```

If the number on the stack is positive, **CHOOSE** executes **WORD1 WORD2... WORD**, whereas if the number is negative or 0, **CHOOSE** executes **WORD1'** ... **WORDn'**.

In either example, **THEN** marks the end of the branch, rather than having its usual logical meaning[13].

---

[12]The original FORTH-79 used +1 for "true", 0 for "false"; many newer system that mostly follow FORTH-79 use -1 for "true". HS/FORTH is one such. Both FORTH-83 and ANSII FORTH require -1 for "true", 0 for "false".

[13]This has led some FORTH gurus to prefer the synonymous word **ENDIF** as clearer than **THEN**.

Looping and structured programming

FORTH contains words for setting up loops that can be definite or indefinite:

```
BEGIN xxx   a g  UNTIL
```

The words represented by **xxx** are executed, leaving the TOS (ag) set to 0 (F) -at which point **UNTIL** leaves the loop - or -1 (T) -at which point **UNTIL** makes the loop repeat from **BEGIN**.

A variant is

```
BEGIN xxx    a g  WHILE yyy REPEAT
```

Here **xxx** is executed, **WHILE** tests the ag and if it is 0 (F) leaves the loop; whereas if flag is -1 (T) **WHILE** executes **yyy** and **REPEAT** then branches back to **BEGIN**. These forms can be used to set up loops that repeat until some external event (pressing a key at the keyboard, *e.g.*) sets the ag to exit the loop. They can also used to make endless loops (like the outer interpreter of FORTH) by forcing flag to be 0 in a definition like

```
: ENDLESS BEGIN xxx 0 UNTIL ;
```

FORTH also implements indexed loops using the words **DO LOOP +LOOP /LOOP**. These appear within definitions, *e.g.*

```
: LOOP-EXAMPLE 100 0 DO xxx LOOP ;
```

The words **xxx** will be executed 100 times as the lower limit, 0, increases in unit steps to 99. To step by -2's, we use the phrase

```
-2 + LOOP
```

to replace **LOOP**, as in

```
: DOWN-BY-2's O 100 DO xxx -2 +LOOP ;
```

The word **/LOOP** 1s a variant of **+LOOP** for working with unsigned limits[14] and increments (to permit the loop index to go up to 65535 in 16-bit systems).

The pearl of FORTH

An unusual construct, **CREATE...DOES>**, has been called "the pearl of FORTH" [15] This is more than poetic license.

---

[14]Signed 16-bit integers run from -32768 to +32767, unsigned from 0 to 65535. See **FTR**.

[15]Michael Ham, "Structured Programming", *Dr. Dobb's Journal of Software Tools*, October, 1986.

CREATE is a component of the compiler that makes a new dictionary entry with a given name (the next name in the input stream) and has no other function.

DOES> assigns a specific run-time action to a newly **CREATE**d word (we shall see this in 28-3 below).

Dummy words

Sometimes we use **CREATE** to make a dummy entry that we can later assign to some action:

```
CREATE DUMMY
CA' * DefinES DUMMY
```

The second line translates as "The code address of ***** defines **DUMMY**". Entry of the above phrase would let **DUMMY** perform the job of ***** just by saying **DUMMY**. That is, FORTH lets us first define a dummy word, and then give it any other words meaning [16].

Here is one use of this power: Suppose we have to define two words that are alike except for some piece in the middle:

```
: *WORD  WORD1 WORD2 *  WORD3 WORD4 ;
: */WORD WORD1 WORD2 */ WORD3 WORD4 ;
```

we could get away with 1 word, together with **DUMMY** fromabove,

```
: *_or_*/WORD
   WORD1 WORD2
   DUMMY
   WORD3 WORD4 ;
```

by saying

```
CA' * DefinES DUMMY *_or_*/WORD
```

or

```
CA' */ DefinES DUMMY *_or_*/WORD
```

This technique, a rudimentary example of vectoring, saves memory and saves programming time by letting us vary something in the middle of a definition *after the definition has been entered in the dictionary*. However, this technique must be used with caution as it is akin to **self-modifying** code[17]

---

[16]This usage is a non-standard construct of HS/FORTH.

[17]Self-modifying machine code is considered a serious "no-no" by modern structured programming standards. Although it is sometimes valuable, few modern cpu's are capable of handling it safely. More often, because cpu's tend to use pipelining and parallelism to achieve speed, a piece of code might be modified in memory, but - having been pre-fetched before modification - actually execute in unmodified form.

A similar procedure lets a subroutine call itself recursively, an enormous help in coding certain algorithms.

Defining "defining" words

The title of this section is neither a typo nor a stutter: **CREATE** finds its most important use in extending the powerful class of FORTH words called "defining" words. The colon compiler "**:**" is such a word, as are **VARIABLE** and **CONSTANT**. The definition of **VARIABLE** is simple

```
: VARIABLE CREATE 2 ALLOT ;
```

Here is how we use it:

```
VARIABLE X <cr> ok
```

The inner workings of VARIABLE are these:

- **CREATE** makes a dictionary entry with the next name in the input stream - in this case, **X**.

- Then the number 2 is placed on the stack, and the word **ALLOT** increments the pointer that represents the current location in the dictionary by 2 bytes.

- This leaves a 2-byte vacancy to store the value of the variable (that is, the next dictionary header begins 2 bytes above the end of the one just defined).

When the outer interpreter loop encounters a new **VARIABLE**'s name in the input stream, that names address is placed on the stack. But this is also the location where the 2 bytes of storage begins. Hence when we type in **X**, the TOS will contain the storage address named **X**.

As noted in 2.4 above, the phrase **X @** (pronounced "X *fetc*h") places the contents of address **X** on the stack, dropping the address in the process. Conversely, to store a value in the named location **X**, we use **!** ("store"): thus

```
4 X ! <cr> ok
X @ . <cr> 4 ok
```

Double-length variables are defined *via* **DVARIABLE**, whose definition is

```
: DVARIABLE CREATE 4 ALLOT ;
```

FORTH has a method for defining words initialized to contain specific values: for example, we might want to define the number 17 to be a word. **CREATE** and "**,**" ("comma") let us do this as follows:

```
17 CREATE SEVENTEEN , <cr> ok
```

Now test it *via*

```
SEVENTEEN @ . <cr> 17 ok
```

Note: The word ”,” (”comma”) puts TOS into the next 2 bytes of the dictionary and increments the dictionary pointer by 2.

A word **C,** (”see-comma”) puts a byte-value into the next byte of the dictionary and increments the pointer by 1 byte.

Run-time vs. compile-time actions

In the preceding example, we were able to initialize the variable **SEVENTEEN** to 17 when we **CREATE**d it, but we still have to fetch it to the stack *via* **SEVENTEEN @** whenever we want it. This is not quite what we had in mind: we would like to find 17 in T0S when we say **SEVENTEEN**. The word **DOES>** gives us precisely the tool to do this.

As noted above, the function of **DOES>** is to specify a run-time action for the ”child” words of a defining word. Consider the defining word **CONSTANT**, defined in high-level[18] FORTH by

```
: CONSTANT CREATE , DOES> @ ;
```

and used as

```
53 CONSTANT PRIME ok
```

Now test it:

```
\textbf{PRIME} . <cr> 53 ok
```

What happened?

- **CREATE** (hidden in **CONSTANT**) made an entry (named **PRIME** , the first word in the input stream following **CONSTANT**). Then ”,” placed the TOS (the number 53) in the next two bytes of the dictionary.

- **DOES>** (inside **CONSTANT**) then appended the actions of all words between it and ”;” (the end of the definition of **CONSTANT**) to the child word(s) defined by **CONSTANT**.

- In this case, the only word between **DOES>** and ; was **@** , so all FORTH constants defined by CONSTANT perform the action of placing their address on the stack (anything made by **CREATE** does this) and fetching the contents of this address.

---

[18]Of course **CONSTANT** is usually a machine-code primitive, for speed.

Klingons

L et us make a more complex example. Suppose we had previously defined a word **BOX (n x y --)** that draws a small square box of n pixels to a side centered at (x, y) on the graphics display. We could use this to indicate the instantaneous location of a moving object – say a Klingon space-ship in a space-war game.

So we define a defining word that creates (not very realistic looking) space ships as squares n pixels on a side:

```
: SPACE-SHIP CREATE , DOES>
    @-ROT (--nxy)      BOX ;
: SIZE ; \ do-nothing word
```

Now, the usage would be (**SIZE** is included merely as a reminder of what 5 means – it has no function other than to make the definition look like an English phrase)

```
SIZE 5 SPACE-SHIP KLINGON <cr> ok
71 35 KLINGON <cr> ok
```

Of course, **SPACE-SHIP** is a poorly constructed defining word because it does not do what it is intended to do. Its child-word **KLINGON** simply draws itself at (x, y).

What we really want is for **KLINGON** to *undraw* itself from its old location, compute its new position according to a set of rules, and then redraw itself at its new position This sequence of operations would require a definition more like

```
:OLD.POS@ (adr--adr n x y) DUP @ OVER
  2+ D@ :
: SPACE-SHIP CREATE , 4 ALLOT DOES>
   OLD.POS@ UNBOX NEW.POSI
   OLD.POS@ BOX DROP ;
```

where the needed specialized operation **UNBOX** would be defined previously along with **BOX**.

Dimensioned data (with Intrinsic unlts)

Here is another example of the power of defining words and of the distinction between compile-time and run-time behaviors.

Physical problems generally work with quantities that have dimensions, usually expressed as mass (M), length (L) and time (T) or products of powers

of these. Sometimes there is more than one system of units in common use to describe the same phenomena.

For example, traffic police reporting accidents in the United States or the United Kingdom might use inches, feet, and yards; whereas Continental police would use the metric system. Rather than write different versions of an accident analysis program it is simpler to write one program and make unit conversions part of the grammar. This is easy in FORTH; impossible in FORTRAN, BASIC, Pascal, or C; and possible, but exceedingly cumbersomein Ada[19].

We simply keep all internal lengths in millimeters, say, and con- vert as follows[20]:

```
: INCHES   254 10 */ ;
: FEET   [ 254 12 * ] LITERAL 10 */ ;
: YARDS [ 254 36 * ] LITERAL 10 */ ;
: CENTIMETERS 10 * ;
: METERS 1000 * ;
```

The usage would be

```
10 FEET . <cr> 3048 ok
```

These are more definitions than necessary, of course, and the technique generates unnecessary code. A more compact approach uses a *defining word*, **UNITS**:

```
: D, SWAP , , ; \ I double-length # in next cells
: UNITS CREATE D, DOES> D@ */ ;
```

Then we could make the table

```
254   10          UNITS INCHES
254   12 * 10     UNITS FEET
254   36 * 10     UNITS YARDS
 10    1          UNITS CENTIMETERS
1000   1          UNITS METERS
\ Usage:
\ 10 FEET . <cr> 3048 ok
\ 3  METERS . <cr> 3000 ok
\ ......
\ \textit{etc}.
```

---

[19]An example (and its justification) of dimensioned data types in Ada is given by Do-While Jones, *Dr. Dobb's Journal*, March 1987. The FORTH solution below is much simpler than the Ada version.

[20]This example is based on 16-bit integer arithmetic. The word */ means "multiply the third number on the stack by NOS, keeping 32 bits of precision, and divide by "TOS". That is, the stack comment for */ is (a b c --a*b/c).

This is an improvement, but FORTH lets us do even better: here is a simple extension that allows conversion back to the input units, for use in output:

```
VARIABLE <AS>                    \ new variable
0 <AS> !                         \ initialize to "F"
: AS -1 <AS> ! ;                 \ set <AS> = "T"
: UNITS CREATE D, DOES>
  D@                             \ get 2 #s
  <AS> @                         \ get current val.
     IF SWAP THEN                \ flip if "true"
 */   0 <AS> ! ;                 \ convert, reset <AS>

BEHEAD' <AS>          \ make it local for security²¹
\ unit definitions remain the same
\ Usage:
\ 10 FEET     . <cr> 3048 ok
\ 3048 AS FEET . <cr> 10 ok
```

Advanced methods of controlllng the compiler

FORTH includes a technique for switching from compile mode to interpret mode while compiling or interpreting. This is done using the words **]** and **[** . (Contrary to intuition, **]** turns the compiler on, **[** turns it off.)

One use of **]** and **[** is to create an "action table" that allows us to choose which of several actions we would like to perform[22].

For example, suppose we have a series of push-buttons numbered 1-6, and a word **WHAT** to read them.

That is, **WHAT** waits for input from a keypad; when button #3 is pushed, *e.g.*, **WHAT** leaves 3 on the stack.

We would like to use the word **BUTTON** in the following way:

```
WHAT BUTTON
```

BUTTON can be defined to choose its action from a table of actions called **BUTTONS** . We define the words as follows:

```
CREATE BUTTONS ] RING-BELL OPEN-DOOR
  ENTER LAUGH CRY SELF-DESTRUCT [
: BUTTON 1- 2* BUTTONS + @ EXECUTE ;
```

---

[21]Headerless words are described in **FTR**, p. 325ff. The word **BEHEAD'** is HS/FORTH's method for making a normal word into a headerless one. See Ch. 5§1§§3 for further details.

[22]Better methods will be described in Chapter 5.

If, as before, I push #3, then the action **ENTER** will be executed. Presumably button #7 is a good one to avoid[23].

How does this work?

- **CREATE BUTTONS** makes a dictionary entry **BUTTONS**.

- **]** turns on the compiler: the previously-defined word-names **RING-BELL**, *etc.* are looked up in the dictionary and compiled into the table (as though we had begun with :), rather than being executed.

- **[** returns to interactive mode (as if it were ;), so that the next colon definition (**BUTTON**) can be processed.

- The table **BUTTONS** now contains the code-field addresses (CFAs) of the desired actions of **BUTTON**.

- **BUTTON** first uses 1- to subtract 1 from the button number left on the stack by **WHAT** (so we can use 0-based numbering into the table – if the first button were # 0, this would be unneeded).

- **2*** then multiplies by 2 to get the offset (from the beginning of **BUTTONS**) of the CFA representing the desired action.

- **BUTTONS +** then adds the base address of **BUTTONS** to get the absolute address where the desired CPA is stored.

- **@** fetches the CFA for **EXECUTE** to execute.

- **EXECUTE** executes the word corresponding to the button pushed. Simple!

You may well ask "Why bother with all this indirection, pointers, pointers to pointers, tables of pointers to tables of pointers, and the like?" Why not just have nested **IF...ELSE...THEN** constructs, as in Pascal?

There are three excellent reasons for using pointers:

- Nested **IF...THEN**'s uickly become cumbersome and difficult to decipher (**TF**). They are also **slow** (see Ch. 11).

- Changing pointers is generally much faster than changing other kinds of data – for example reading in code overlays to accomplish a similar task.

- The unlimited depth of indirection possible in FORTH permits arbitrary levels of abstraction. This makes the computer behave more "intelligently" than might be possible with more restrictive languages.

A similar facility with pointers gives the C language its abstractive power, and is a major factor in its popularity.

---

[23]The safety of an execution table can be increased by making the first (that is, the zeroth) action **WARNING**, and making the first step of **BUTTON** a word **CHECK-DATA** that maps any number not in the range 1-6 into 0. Then a wrong button number causes a **WARNING** to be issued and the system resets.

Strings

B^y now it should be apparent that FORTH can do anything any other language can do. One feature we need in any sort of programming –scientific or otherwise– is the ability to handle alphanumeric strings. We frequently want to print messages to the console, or to put captions on figures, even if we have no interest in major text processing.

While every FORTH system must include words to handle strings(see, e.g., **FTR**Ch. 9) –the very functioning of the outer interpreter, compiler, *etc.*, demands this– there is little unanimity in defining extensions. BASIC has particularly good string-handling features, so HS/FORTH and others provide extensions designed to mimic BASICs string functions.

Typical FORTH strings are Iimite to 255 characters because they contain a count in their first byte[24].The word **COUNT**

```
: COUNT DUP 1+ SWAP C@ ; (adf--nadr+1)
```

expects the address of a counted string, and places the count and the address of the first character of the string on the stack. **TYPE**, a required 79 or 83 word, prints the string to the console.

It is straightforward to employ words that are part of the system (such as **KEY** and **EXPECT**) to define a word like **$"** that takes all characters typed at the keyboard up to a final **"** (close-quote –not a word but a string-terminator), makes a counted string of them, and places the string in a buffer beginning at an address returned by **PAD**[25].

The word **$.** ("string-emit") could then be defined as

```
: $. COUNT TYPE ; (adr --)
```

and would be used with 3" like this:

```
$' The quick brown fox" < cr> ok
S. The quick brown fox ok
```

Since this book in not an attempt to paraphrase **FTR**it is strongly recommended that the details of using the system words to devise a string lexicon be studied there.

One might contemplate modifying the **FTR**lexicon by using a full 16-bit cell for the count. This would permit strings of up to 64k bytes (using unsigned integers[26]), wasting 1 byte of memory per short (255 bytes) string. Although few scientific applications need to manipulate such long strings, the program that generated the index to this book needed to read a page at a time, and thus to handle strings about 3-5 kbytes long.

---

[24]A single byte can represent positive numbers 0-255.
[25]A system variable that returns the current starting address of the "scratchpad".

FORTH programming style

A FORTH program typically looks like this

```
\ Example of FORTH program
:WORD1 ... ;
:WORD2 OTHER-WORDS ;
:WORDS YET-OTHER-WORDS ;
   ...
: LAST-WORD wonon ...WORD3
   WORD2 WORD1 ;
LAST-WORD <cr> \ run program
```

Note: The word means "disregard the rest of this line". It is a convenient
method for commenting code.

In other words, a FORTH program consists of a series of word definitions,
culminating in a definition that invokes the whole shebang. This aspect gives
FORTH programming a somewhat different avor from programming in more
conventional languages.

Brodie notes in **TF** that high-level programming languages are considered
good if they require structured, top-down programming, and *wonderful* if they
impose *information hiding*. Languages such as FORTRAN, BASIC and assembler
that permit direct jumps and do not impose structure, top-down design and
data-hiding are considered *primitive* or bad. To what extent does FORTH follow
the norms of good or *wonderful* programming practice?

Structure

The philosophy of "structured programming" entered the general conscious-
ness in the early 1970s. The idea was to make the logic of program control
flow immediately apparent, thereby aiding to produce correct and maintainable
programs. The language Pascal was invented to impose by fiat the discipline of
structure. To this end, direct jumps (GOTOs) were omitted from the language[27].

FORTH programs are automatically structured because word definitions are
nothing but subroutine calls. The language contains no direct jump statements
(that is, no GOTO's) so it is *impossible* to write "spaghetti" code.

A second aspect of structure that FORTH imposes (or at least encourages)
is *short* definitions. There is little speed penalty incurred in breaking a long
procedure into many small ones, unlike more conventional languages. Each of

---

[27]Ironically, most programmers refuse to get along without spaghetti code, so commercial Pascal's
now include GOTO. Only FORTH among major languages completely eschews both line labels
and GOTOs, making it the most structured language available.

the short words has one entry and one exit point, and does one job. This is the beaux ideal of structured programming!

subsection"Top-down" design

Most authors of "how to program" books recommend designing the entire program from the general to the particular. This is called "top-down" programming, and embodies these steps:

- Make an outline, ow chart, data-ow diagram or whatever, taking a broad overview of the whole problem.

- Break the problem into small pieces (decompose it).

- Then code the individual components.

The natural programming mode in FORTH is "bottom-up" rather than "top-down" – the most general word appears last, whereas the definitions necessarily progress from the primitive to the complex. It is possible – and sometimes vital – to invoke a word before it is defined ("forward referencing"[28]). The dictionary and threaded compiler mechanisms make this nontrivial. The naturalness of bottom-up programming encourages a somewhat different approach from more familiar languages:

- In FORTH, components are specified roughly, and then 3 they are coded they are immediately tested, debugged, redesigned and improved.

- The evolution of the components guides the evolution of the outer levels of the program.

We will observe this evolutionary style in later chapters as we design actual programs.

Information hiding

Information (or data) "hiding" is another doctrine of structured programming. It holds that no subroutine should have access to, or be able to alter (corrupt!) data that it does not absolutely require for its own functioning[29].

Data hiding is used both to prevent unforeseen interactions between pieces of a large program; and to ease designing and debugging a large program. The program is broken into small, manageable chunks ("black boxes") called **modules** or **objects** that communicate by sending messages to each other, but are otherwise mutually impenetrable. Information hiding and modularization

---

[28]The FORmula TRANsIator in Ch. 11.4 uses this method to implement its recursive structure.

[29]Rather like the "cell" system in revolutionary conspiracies, where members of a cell know only each other but not the members of other cells. Mechanisms for receiving and transmitting messages between cells are double-blind. Hence, if an individual is captured or is a spy, he can betray at most his own cell and the damage is limited.

are now considered so important that special languages –Ada, MODULA-Z, C++, and Object Pascal– have been devised with it in mind.

To illustrate the problem information hiding is intended to solve, consider a FORTRAN program that calls a subroutine

```
PROGRAM MAIN
    some lines
CALL SUB1(arg1, ar92, ... , argn, answer)
    some lines
END

SUB1(X1, ... , Xn, Y)
    some lines
    Y = something
RETURN
END
```

There are two ways to pass the arguments from MAIN to SUB1, and FOR-TRAN can use both methods.

- Copy the arguments from where they are stored in MAIN into locations in the address space of SUBl (set aside for them during compilation). If the STATEMENTS change the values X1,...,Xn during execution of SUBl, the original values in the callin program will not be affected (because they are stored elsew ere and were copied during the CALL).

- Let SUBl have the addresses of the arguments where they are stored in MAIN. This method is dangerous because if the arguments are changed during execution of SUBl, they are changed in MAIN and are forever corrupted. If these changes were unintended, they can produce remarkable bugs.

Although copying arguments rather than addresses seems safer, sometimes this is impossible either because the increased memory overhead may be infeasible in problems with large amounts of data, or because the extra overhead of subroutine calls may unacceptably slow execution.

What has this to do with FORTH?

- FORTH uses linked lists of addresses, compiled into a dictionary to which all words have equal right of access.

- Since everything in FORTH is a word –constants, variables, numerical operations, I/O procedures– it might seem impossible to hide information in the sense described above.

- Fortunately, word-names can be erased from the dictionary after their CFAs have been compiled into words that call them. (This erasure is called "beheading".)

- Erasing the names of variables arantees they can be neither accessed nor corrupted by unauthorized words (except through a calamity so drea ful the program crashes).

Documenting and commenting FORTH code

FORTH is sometimes accused of being a "write-only language. In other words, some complain that FORTH is cryptic. I feel this is basically a complaint against poor documentation and unhelpful word names, as Brodie and others have noted.

Unreadability is equally a flaw of poorly written FORTRAN, Pascal, C, *et al*.

FORTH offers a programmer who takes the trouble a goodly array of tools for adequately documenting code.

Parenthesized remarks

The word (–a left parenthesis followed by a space– says "disregard all following text up until the next right parenthesis[30] in the input stream. Thus we can intersperse explanatory remarks within colon definitions. This method was used to comment the Legendre polynomial example program in Ch. 1.

Stack comments

A particular form of parenthesized remark describes the effect of a word on the stack (or on the oating point fstack in Ch. 3). For example, the stack-effect comment (stack comment, for short)

```
( adr - - n)
```

would be appropriate for the word **@** ("fetch"): it says **@** expects to find an address (adr) on the stack, and to leave its contents (n) upon completion.

The corresponding comment for **!** would be

```
( n adr --).
```

An fstack comment is prefaced by a double colon :: as

```
(::x -- f[x]).
```

Note that to replace parentheses within the comment we use brackets [ ] , since parentheses would be misinterpreted. Since the brackets appear to the

---

[30]The right parenthesis,) , is not a word but a **delimiter**.

right of the word ❨, they cannot be (mis-)interpreted as the FORTH words ❩ or
❲.

With some standard conventions for names[31], and standard abbreviations
for different types of numbers, the stack comment may be all the documentation
needed, especially for a short word.

Drop line (\\)

The word \\(back-slash followed by space) has gained favor as a method for
including longer comments. It simply means "drop everything in the input
stream until the next carriage return". Instructions to the user, clarifications or
usage examples are most naturally expressed in an included block of text with
each line set off[32] by \\.

Self-documenting code

By eliminating ungrammatical phrases like `CALL` or `GOSUB`, FORTH presents the
opportunity –*via* telegraphic names[33] for words– to make code almost as self-
documenting and transparent as a simple English or German sentence. Thus,
for example, a robot control program could contain a phrase like

```
2 TIMES LEFT EYE WINK
```

which is clear (although it sounds like a stage direction for Brunhilde to
vamp Siegfried). It would even be possible without much difficulty to define
the words in the program so that the sequence could be made English-like:

```
WINK LEFT EYE 2 TIMES .
```

Safety

Some high level languages perform automatic bounds checking on arrays, or
automatic type checking, thereby lending them a spurious air of reliability.
FORTH has almost no error checking of any sort, especially at run time. Nev-
ertheless FORTH is a remarkably safe language since it fosters fine-grained
decomposition into small, simple subroutines. And each subroutine can be
checked as soon as it is defined. This combination of simplicity and immediacy
can actually produce safer, more predictable code than languages like Ada, that
are ostensibly *designed* for safety.

---

[31]See, *e.g.*, L. Brodie, **TF**, Ch. 5, Appendix E.

[32]For those familiar with assembly language, \\is exactly analogous to ; in assembler. But since ;
is already used to close colon definitions in FORTH, the symbol \\has been used in its place.

[33]The matter of naming brings to mind Mark Twain's remark that the difference between the
almost-right word and the right one is the difference between the lightning-bug and the lightning.

Nonetheless, error checking **–especialiy array bounds-checking–** can be a good idea during debugging. FORTH lets us include checks in an unobtrusive manner, by placing all the safety mechanisms in a word or words that can be "vectored" in or out as desired.

34

---

[34]See **FTR**for a more thorough discussion of vectoring. Brodie, **TF** , sugests a nice construct called **DOER...MAKE** that can be used for graceful vectoring.

Chapter 2

Floating Point Arithmetic

Contents

This chapter discusses various aspects of floating point arithmetic in FORTH. Our approach assumes the central processor (CPU) has a dedicated floating point co-processor (FPU) available to it, such as the 80x87 for the Intel 80x86 family; the built-in FPU on the 80486; the 68881/2 for Motorola 680x0 machines; various add-ins and clones like the Weitek, Cyrix, IIT and AMD chips; or digital signal processing and array-processing co-processor boards.

If no floating point co-processor were available, one could employ co-processor emulation routines. This is the approach taken in commercial software written in FORTH, such as the (unfortunately now-deceased) VP-Planner spreadsheet. Since this text is not a *vade mecum* for writing commercial software, but rather a handbook for using FORTH to solve computational problems in science and engineering, we consider a co-processor essential.

Organization of floating point arithmetic

FORTH was originally invented as a language for controlling machinery. It is still used extensively for this purpose, with the machines in question being as varied as industrial robots, laboratory instruments, the Hubble Space Telescope, special effects motion picture cameras, and other computers. The floating point co-processor in a typical computer is a machine, and any numerical calculation with floating point or complex numbers, e.g. can be organized in terms of loading operands into the coprocessor, and transferring results from it to memory. That is, FORTH can control the FPU through the calculation, as indicated in Fig. 3-1 below:

We assume a stack for floating point numbers separate from the parameter or return stacks. We call this the fstack, and assume i has arbitrary depth. (We denote it by :: in stack comments.)

Whether the fstack should be distinct from the parameter stack is currently a subject of lively debate within the FORTH com- munity. One faction wishes to combine the two. The other faction, including the author and most other FORl'l-l number- cmnchers, believes that to organize a floating point- intensive
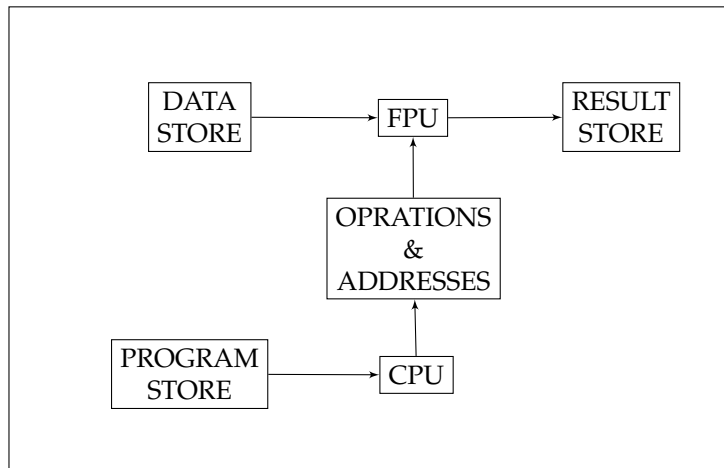
31

Figure 2.1: Fig. 31 Data flow diagram of a CPU controlling an FPU through a calculation.

calculation as data ow through a dedicated coprocessor, the parameter stack must be reserved for addresses, loop indices and ags. The data fed to the coprocessor therefore has to stay elsewhere, i.e. in the data store and the fstack.

The words we shall need fall into the categories of fstack manipulation, special constants, arithmetic, tests and mathematical functions. Their names are nearly standard[1].

Fstack manipulation

The fstack words are[2]

```
F@      ( addr -- :: -- x )
Fl      ( addr -- :: x -- )
FDUP    ( :: x -- xx )
FSWAP   ( :: yx -- xy )
FDROP   ( :: x -- )
FROT    ( :: zyx -- yxz )
```

---

[1]That is, the names in WS FORTH, PIS/FORTH, UniFORTH and PCFORTH are nearly the same and close to the proposed FORTH Vendors' Group (WG) standard - see, e.g., Ray Duncan's and Martin Tracy's article in Dr. Dobb's Journal, September 1984, p. 110. The new ANSII standard does not address floating point, hence it is likely the PVC standard will become pre-eminent by default.

[2]F@ and F! stand for a suite of words for fetching and storing 16-, 32-, 64-, and 80-bit numbers from/to the fstack. In HS/FORTH they have names like I16@ I16! I32@ I32! R32@ R32¡ R64@ R64! R80@ R80! and equivalents with suffices E or L, that transfer data from segments other than the data segment.

```
FOVER  ( :: yx -- yxy )
S->F   ( n-- :: --n)
D->F   ( d-- :: -- d )
F->S   ( :: x-- : -- int[x] )
F->D (::x--:--dint[x])
% (place a FP# from input stream on fstack)
```

To these we sometimes add[3]

```
: FUNDER FSWAP FOVER ;
: FPLUCK FSWAP FDROP ;
Fnx (n = 2 - 6 | defined in code for speed)
FnR (n = 3 - 7 | defined in code for speed)
```

The Intel mathematics co-processors 80x87 (8087/80287/80387) and their clones incorporate a stack of limited depth (in fact 8 deep), the 87stack It is far faster to get a number from the 875tack than from memory. Thus, as Palmer and Morse[4] emphasize, optimizing for speed demands maximum use of the 87fstack to store intermediate results, frequently used constants, etc.

In Ch. 4 @7 we show how to extend the 87fstack into memory. The cost of unlimited fstack-depth is reduced speed when the 87stack spills over to memory.

Special constants

In defining various floating point operations it is convenient to be able to place certain constants on the fstack directly, by invoicing their names. Here are some words that have proven useful:

```
F=0 (::--O)

F=1 (::--1)

F=Pl (2: --pi = 3.14159...)
F=L2(10) (2: - -Iogz10)
F=L2(E) (:2 "log, a)

F= L10(2) (z: - - Iog,2)
F=LN(2) (ii-409.2)
```

---

[3]FnX (exchange ST(n) and ST(0) on fstack) and FnR (roll ST(n) to ST(0) on fstack) are part of HS/FORTH's floating point lexicon.

[4]J.F. Palmer and S.P. Morse, The 8087 Primer (John Wiley and Sons, Inc., New York. 1984). Hereinafter called 8087P. Basically the same principle holds for other coprocessors such as the Weitek 1167 or Transputer 800 that incorporate intrinsic fstacks. The Motorola 68881/2 have registers, hence we must synthesize an fstack in software for these chips.

Arithmetic operations

As noted in Chapters 1 and 2, FORTH arithmetic operators are rds - *dumb* words. FORTH uses a distinct set of operators for each kind (16-bit integer, 32-bit integer, REAL, COMPLEX) of arithmetic. so the compiler has nothing to decide.

Languages like FORTRAN, BASIC and APL overload arithmetic operators - their meanings are context-dependent. This makes it possible to write -say- a FORTRAN expression using REAL@4, REAL@B, COMPLEX@S or COM-FLEX16 literals and variables without worrying about how to fetch. store or convert them. Operator overloading increases the complexity of compilers and limits the speed and efficiency of compilation.

As we shall see in Chapter 5@1, FORTH enables an alternative solution in which "smart" data "know" their own types and "smart" operators "know" what kinds of data are being combined. The slightly reduced execution speed is offset by improved exibility: one canned routine can work with all data types. Even better, adding this kind of "intelligence" makes no extra demands on the FORTH compiler.

The standard, dumb FORTH floating-point arithmetic opera- tions and their actions are

```
F+        ( :: yx -- y+x)
F         ( :: yx -- yx)
FR        ( :: yx -- xy)
F*        ( :: yx -- y*x)
F/        ( :: yx -- y/x)
FRI       ( :: yx -- x/y)
FNEGATE   ( :: x -- -x)
FABS      ( :: x -- |x|)
1/F       ( :: x -- 1/X)
```

To these it is sometimes useful to add words that do not consume all their arguments, such as F*NP (floating multiply, no pop)

```
F'NP (::xy--xy@x).
```

that are faster, more convenient. and less demanding of the 87stack than the phrase FOVER F*.

Example: evaluating a polynomial

Let us now write a little program to calculate something using the floating point lexicon, say a program to evaluate a general polynomial $P_N(x)$. The
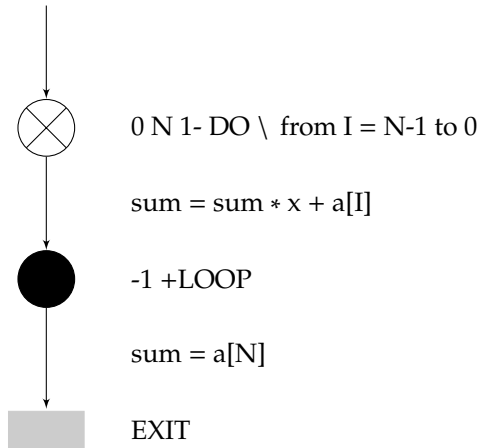
0 N 1- DO \ from I = N-1 to 0

sum = sum * x + a[I]

-1 +LOOP

sum = a[N]

EXIT

Figure 2.2: Fig. 3-2 Pseudo FOR TH flow diagram of polynomial evaluation

formula to evaluate is

$$P_N(x) = a_0 x^0 + a_1 x^1 + ... + a_N x^N = a_0 + x(a_1 + x(... + xa_N))$$

The algorithm can be represented by the (pseudo) FORTH ow diagrams[5], where ☐ indicates the end of the program.

Now we translate Fig. 3-2 above into FORTH [67] :

```
: }POLY ( [a{] [x] -- ) \ evaluate p(x,N)
    FINIT G@              \ x on fstack
    DUP type@ G=0         (z: -- x sum=0)
    LEN@                  ( -- [a{] N )
    SWAP DO               \ begin loop
    DUP I } G@            ( :: -- x sum a[i] )
    G+ GOVER G*           ( :: -- x sum' )
    -1 +LOOP              \ end loop
    GPLUCK               ( :: -- sum )
    0 } G@ G+ ;           ( :: -- p[x,n] )

\ Say: A{ X }POLY
```

---

[5]See, e.g., SF. Lines trace program flow; a branch indicates a decision; at a loop ⊗ marks the beginning and ● the branch back to the beginning, as in BEGIN...REPEAT or DO...LOOP.

[6]We assume arrays have been defied using the intrinsically typed data structures and generic operators of Ch. 5.

[7]The notation [x] means "the address of x".

Note that the function }POLY expects the addresses of its arguments on the stack, consumes them and leaves its result on the fstack. User-defined FORTH functions will in general have an interface of this sort. This will be especially true of the functions built into numerical co-processors.

Actually, such behavior is typical of subroutine linkage in most high level languages, as anyone knows who has written assembler subroutines that can be linked to compiled FORTRAN, C or BASIC. So FORTH really isn't different, only more explicit and efficient.

Optimizing: FORTH vs. FORTRAN

A simple-minded compiler will translate an expression such as

$$\left(sin(x)\right)^2$$

into a form requiring two function calls:

```
Y = SIN(X)*SIN(X) .
```

Obviously this is silly. One of the claims often made for "optimizing" FORTRAN or C compilers is the ability to recognize an expression requiring unnecessary function calls, and to re-express it as. say,

```
TEMP = SIN(X)
v = TEMP * TEMP .
```

A globally optimizing compiler has a more extensive repertoire usually it can recognize static expressions ("invariant code") within a loop and move them outside; and it can find and eliminate code that is never evaluated ("dead" code).

FORTH assumes a good programmer *never* overlooks trivia optimizations like this. Thus nothing in the FORTH incremental compiler or optimizer is inherently capable of recognizing silly code and eliminating it.

Optimization in FORTH takes one of several forms, that can be combined for best results. The simplest is the use of stacks registers to avoid extra memory shuffling. Referring to the preceding bad example, we note that a simple floating-point function f(x) finds its argumentx on the top of the fstack, consume it, and leaves the result in its place. A simple F**2, defined as

```
: F**2 FDUP F* ;
```

will then evaluate $[f(x)]^2$, with no fetch/store penalty from defining a temporary variable.

Some FORTHs can optimize by substituting inline code for jumps and returns to subroutines. In other words, by making the compiled code longer, some advantage in speed can be gained. HS/FORTH offers a recursive-descent optimizer of just this sort that -within its limitations- can optimize as well as good C or FORTRAN compilers. An optimizer-improved word consists of all the code bodies of the words in its definition, jammed end to end and with redundant pushes and pops deleted.

V irtually all FORTH implementations have a built-in assembler

that permits defining a word in machine language. Judiciously machine-coding selected words can dramatically reduce execution time, since careful hand coding offers the ultimate performance the machine is capable of. Some versions of Pascal and C also have this ability; and of course most compiled and linked languages can link to functions and subroutines defined in machine code.

FORTHs advantage over other languages lies in making the process of designing, testing and linking hand- coded components nearly painless.

Another advantage of FORTH over other compiled languages is that one can specify which parts to optimize and which to leave as high-level definitions. This is both faster to compile and much more compact, than optimizing all of the program uniformly. The rationale of partial (sometimes called "peephole") optimization is that most programs spend 90% of their execution time in 10% of the code. This 10% is the only part of the program worth optimizing [8].

Testing floating point numbers

A nalogous to the test words for integer arithmetic, we require the words `F0> F0= F0< F> F= F<`.

Test words leave a ag on the parameter stack depending on the relationship they discover. Moreover, these words consume one or two arguments on the fstack, following the standard FORTH practice. As a simple first example of test words, let us define max and FMIN analogous to MAX and MIN: [9]

```
: XDUP FOVER FOVER ;
: FMAX XDUP F< IF FSWAP THEN FDROP ;
: FMIN XDUP F> IF FSWAP THEN FDROP ;
```

---

[8] An rompk is discussed in Chapter 9, where the innermost loop of 3 nested loops is optimized (even hand-coded), and it is seen from the timings that little is to be gained by optimizing the next outer loop.

[9] XDUP is called `CPDUP` in HS/FORTH's complex arithmetic lexicon.

Mathematical functions - the essential function library

Scientific programming in FORTH requires a suite of exponential, logarithmic and trigonometric functions (included with all FORTRAN systems, most BASICs, Cs, APL LISP, etc.) The minimal function library is

```
FSQRT       ( :: x -- √x )
FLN         ( :: x -- ln[x] )
FLOG        ( :: x -- log₁₀[x] )
F2**        ( :: x -- 2ˣ )
F**         ( :: x y -- yˣ )
FEXP        ( :: x -- eˣ )
FSIN        ( :: x -- sin[x] )
FCOS        ( :: x -- cos[x] )
FTAN        ( :: x -- tan[x] )
DEG->RAD    ( :: x -- x*pi/180 )
RAD->DEG    ( :: x -- x*180/pi )
FATAN       ( :: x -- atan[x] )
FASIN       ( :: x -- asin[x] )
FACOS       ( :: x -- acos[x] )
```

Machine code definitions of the above functions for the 80x87 chip will be given in Chapter 4.

Library extensions

The minimal function library is easily extended. We illustrate below with the **FSGN** function and with hyperbolic and inverse hyperbolic functions. Complex extensions of the function library is deferred to Chapter 6.

The FSGN function

The most useful form of FSGN finds one argument it (from which to take an algebraic sign) on the parameter stack, and the floating, point argument x on the fstack. We may define it using logic, as

```
: FSGN ( n -- :: x -- sgn[n]*abs[x])
    FABS 0< IF FNEGATE THEN;
```

Cosh, Sinh and their inverses

We now code the hyperbolic sine and cosine. The formulae are

$$
\begin{aligned}
sinh(x) &= \frac{1}{2}\left(e^x - e^{-x}\right) \\
cosh(x) &= \frac{1}{2}\left(e^x + e^{-x}\right)
\end{aligned}
$$

and their definitions are

```
: F2/ F=1   ( :: x -- x/2 )
    FNEGATE FSWAP FSCALE FPLUCK ;
: HYPER FEXP FDUP 1/F ; ( ::--e**x e**-x)
:SINH HYPER F- F2/ ;
: COSH HYPER F+ F2/ ;
```

The hyperbolic tangent is then

```
FIND XDUP O= ?(: XDUP FOVER FOVER ;)
              \ conditionally compile XDUP
: TANH HYPER ( :: -- ex e-x)
    XDUP F- F-ROT F+ F/ ;
```

Finally, the inverse hyperbolic sine and cosine can be defined in terms of logarithms:

$$
\begin{aligned}
arcsinh(x) &= \ln\left(x + (x^2 + 1)^{1/2}\right), -\infty < x < \infty \\
arccosh(x) &= \ln\left(x + (x^2 - 1)^{1/2}\right), -\infty < x < \infty
\end{aligned}
$$

The corresponding definitions are [10]

```
FIND F**2 0= ?(: F**2 FDUP F* ; )
:ARCSINH FDUP F**2 F=1 F+
    FSQRT F+ FLN ;
:ARCCOSH FDUP F**2 F=1 F-
    FDUP F0<
    ABORT" x <1 in ARCCOSH"
    FSQRT F+ FLN ;
```

---

[10]0. Note the test (on; 21 inARGCOSH, to prevent an error in FSORT.

Pseudo-random number generators (PRNG's)

The subject of computer-generated (pseudo) random numbers hfilbeen discussed extensively in the literature of computation [11] [12] . We shall confine ourselves here to translating two useful algorithms into FORTH, and discussing tests for pseudo-random number generators (PRNGs).

The first is a method called GGUBS [13] based on the recursion

$$r_{n+1} = 16807 \times r_n \, MOD(2^{31} - 1).$$

Since 32-bit modulo arithmetic is inefficient on a 16-bit processor, the program uses the 80x87 chip, and uses synthetic division to get N MOD (2^{31} - 1). A version that uses the 32-bit registers of the 80386/80486 would not be hard to program.

Two specialized words are needed [14] , that fetch/store 32-bit integers to/from the fstack from/to memory:

```
CODE I32@ <% 9B DB 07 5B 9B %> END-CODE
CODE I32! <% 9B DB 1F 5B 9B %> END-CODE
```

The program data are stored in variables rather than registers so they can be moved directly to the co-processor [15] .

```
DVARIABLE       BIGDIV
21474.83647     BIGDIV D! \2**31-1
DVARIABLE       DIVIS
1277.73         DIVIS D!
DVARIABLE       SEED
VARIABLE        M1 16807 M1 !
VARIABLE        M2 2836  M2 !
```

The high-level FORTH program itself is[16][17]

---

[11]D.E. Knuth, The Art of Computer Programming v.2: Seminumerical Algorithnu, 2nd ed. (Addison-Wesley Publishing Company, Reading, MA, 1981), Ch. 3.

[12]R. Sedgewick, Algorithm (Addison-Wesley Publishing Company, Reading, MA, 1983).

[13]P. Bratley, B.L. Fox and LE. Schrage,A Guide to Simulation (Springer-Verbg, Berlin, 1983).

[14]We anticipate using the FORTH assembler - see Ch. 4 1.11

[15]Of course, it would have been feasible to define, via `CREATE ... DOES >` , data types that place the data on the 87stack, e.g. : `FICONSTANT CREATE , DOES> I16@` ; but such words can't be optimized; to optimize time-critical word(s) in assembler requires VARIABLEs.

[16]The new words are `FINIT` (initialize 80x87 chip). `FTRUNC` (set roundoff mode to truncate oating point numbers toward zero): and `FRNDINT` (round flotaing point number to integer). The 80x87 idiosyncratically possesses several roundoff policies: A policy is selectd by setting bits 10 and 11 (numbering from 0) of the 16-bit control word using `HEX 0C00 OR`. See `8087P` for details.

[17]`BEHEAD"` is one of several HS/FORTH words thet remove dictionary entries and reclaim lost space. space. `BEHEAD'` beheads one word, `BEHEAD"` behads all words, inclusive, in a range: in this case, `BIGDIV, DIVIS. SEED, M1, M2` and `RAND`.

```
: RAND                   ( :: -- seed)
    FINIT SEED DUP 132@
    DIVIS I32@
    XDUP F/
    FTFIUNC FRNDINT
    FUNDER F* FROT FR-
    M1 |16@ F*
    FSWAP M2 I16@
    F* F- FDUP I32! ;
:RANDOM                  ( :: -- random#)
    RAND BIGDIV I32@  ( ::--seed2**31-1)
    FSWAP FDUP F0<    ( -- f :: -- 2**31-1 seed)
    IF FOVER F+ THEN FR/
BEHEAD" BIGDIV RAND
\ make BIGDIV, RAND local
```

To test the algorithm start with the seed 1, and generate 1000 prn's. The result should then be 522329230.

```
: GGUBS.TST 0.1 SEED D!
    1000 0 DO RANDOM LOOP
    SEED D@ D. ;

GGUBS.TST 522329230 0k
```

Testing random number generators

When defining PRNGs it is always important to include a test for randomness. The simplest is called the χ^2 test: use the PRNG to generate N integers[18] in the range [0,n-1] and record the number of occurrences, f_s of each integer $s = 0, 1, \ldots n - 1$. If the PRNG is really random, then the probability that an integer should have any of the n values is $1/n$, hence the expected frequencies are $\langle f_g \rangle \equiv \lambda = \frac{N}{n}$ .The χ^2 statistic for f_s is defined to be

$$\chi^2 = \sum_{s=0}^{\infty} \frac{1}{\lambda}\left(f_s - \lambda\right)^2, \lambda = \frac{N}{n} \tag{2.1}$$

xz should have a value roughly n[19]. GGUBS passes the 12 test: A program to calculate this statistic (with N = 1000 and n = 100) is

---

[18] N is assumed $\gg n$

[19] For those interested in details, the probability a PRNG will produce the integer "s" f times is $p_f = \lambda_f e^{-\lambda}/(f!)$. Thus the e ed value of $(f - \lambda)^2/\lambda$ is 1, and therefore the expected value of χ^2 is n. The variance in the χ^2 statistic should then be $n(2 + \lambda^{-1})$.

```
CREATE FREQS 200 ALLOT OKLW
: IRAND RANDOM 100 S->F
F* FROUND- F->S ;
: INlT-FREOS \initialize freqs array
FREQS 200 0 FILL ;

: GET-FREQS \make frequency table
1000 0 DO IRAND 2*
DUP 199 >ABORT"IRANDTOO LARGE"
FREQS + 1+!
LOOP ;
: STARTCHISQ INIT-FREQS GETFREQS ;
: CHISO 0 \sum on stack
200 0 DO | FREQS + @
10- DUP * +
2 +LOOP ;

: .FREQS \display distribution
2000DO I FREQS + @ ICE
2 +LOOP ;
```

The resul are displayed below in Thble 3-1 0 age 59. The mean of these values is 99.1, and their variance is 210. This agrees remarkably well with the theoretical formula

02=n(2+whenn = 100 and l = 10.

Table 3-1 Values of 1' for causes

STAR@IZG'llSO M80. 1114 olr STARIO-lISO CHSO. 1130 olt STARICHISO GllSO. 540 olt STARIO'IISO CHSO. 1W olt STARICHISQ CHlSO. 1294 olt STARICHISO CHISO . STARIG'lISO CHISQ . 650 OR STARTGslISO M80. 1052 olt STABICHISO G-llSO . g) 0k STARICHISO CHISO . 7m olt STARTCCHISO CHISO . 94 0k STARICHISO CHISO . B42 olt STARIG'IISQ CHlSO. 1072 0k S'TARIG'IISO CHISO . 976 OR STARICHISO CHISQ. 1110 olt STARICHISO CHISO. 1W olt STARIG-IISQ CHISQ. 1183 0k STARICHISO CHISO . 0 0k STARICHISO CHISQ. 360 OR STAR@ECHISO CHISO . 956 OR STABICHISQ CHISQ. 10m 0k STARIOHISO CHISO . me ok

In one application GGUBS was unsatisfactory because it con- tained correlations not revealed by the above tests. This led me to seek another PRNG with perhaps better properties, a longer cycle, etc. I offer it as an alternative, since at the very least it will enable the reader to test his applications with more than one PRNG". Here is the second PRNG:

Chapter 3 - Hosting Point Arithmetic

```
\PFNGHBAWICWANSLDJLLBYFEW
VARIABLEX VARABLEY VARIABLEZ
:RAND-INrI' 1X|1GIDYI le;
:GEN (ab[n]--n'arnodb) \hl-Ievolvoralon

DUP>R @ MT DUP>R

(--nab:R:--[n]b)

@IMOD DROP DUP O<

IF R> + ELSE FIDFDP THEN

DUP R>I ;
\CODE GEN CXPOP. AXPOP.
\ [BX]WORDPTR lMUL
\CXIDN.DXOIW 0MP.
\ DX CX ADD.
\ >>>POSITIVE [BX] DX MOV. END-CODE
(Ext; 171 33259 X GEN )

JGEPOSITIVE.



Sclendc FORTH

:RAMOM FNT FTRLM
171 ms 011 S->F X GEN I160 FR]
172 30311 UL? S->F Y GEN |18@ PW
F+
170mDIPSr>F ZGEN 116@ FFV
F+FRAC:
% \mmmwmwdaw
% \FTRUNCqsecIf-sromdlngbwado
```

The corresponding 12 results are given below in Table 3-2.

Table 3-2 x2 for Wichman-Hill PRNG

START.CHISQ	CHISQ	.	846	START.CHISQ	CHISQ	.	954
START.CHISQ	CHISQ	.	1036	START.CHISQ	CHISQ	.	908
START.CHISQ	CHISQ	.	852	START.CHISQ	CHISQ	.	856
START.CHISQ	CHISQ	.	858	START.CHISQ	CHISQ	.	930
START.CHISQ	CHISQ	.	770	START.CHISQ	CHISQ	.	868
START.CHISQ	CHISQ	.	882	START.CHISQ	CHISQ	.	858
START.CHISQ	CHISQ	.	918	START.CHISQ	CHISQ	.	912
START.CHISQ	CHISQ	.	956	START.CHISQ	CHISQ	.	952
START.CHISQ	CHISQ	.	1202	START.CHISQ CHISQ . 1016			
START.CHISQ	CHISQ	.	1112				
START.CHISQ	CHISQ	.	778				
START.CHISQ	CHISQ	.	1172				

Interestingly, the mean of the 12 statistic for 21 tests is 93.5, perhaps a bit low, but not outrageously so; however, the variance in x is suspiciously small only 135 vs. the expected 210. This may mean the distribution is excessively even!

One very useful test for randomness involves constructing a random walk that is, a sequence of integers generated by the

rule (rn is the n'th PRN)

```
% _ 1mm > 0.5
% @W 'x' +I1itr,., s 0.5
```

(3)

```
% macaw ma-mpmm 81
```

and taliipg the discrete Fourier transform (DFT) of the se- quence . Any serial correlations will show up as periodicities, with periods smaller than N, in the DH" of x, .

552 Random data structures The prngs we have discussed so far produce pms uniformly dis- tributed on the interval [0,1]. What if we want pms that are distributed according to the normal distribution on (-,), or according to some other standard distribution function of mathe- matical statistics?

There are algorithms for generating pms whose distribution function is one of a few standard ones; however, in general one must resort to brute force. We now engage in a brief mathemati digression, before showing how pms with distribution function dp(@) = 0(1 E)d@ can be converted to prns with an arbitrary distribution function dp(x) =f(x)dr.

We suppose there is a function X(E) that converts uniform pm's to pms

distributed according to f(x). But if any of this is to make sense, the inverse function @(x) must also exist, since there is nothing special about one distribution relative to another@s. The

   condition that both functions exist is dig at 0.

```
'I'henz4
th) = 1:45 a (x  no)
(4)
=13dto(x-x(5(x) +5 @(x)))
```

```
which, using standard manipulations, we can evaluate as
```

```
Un'ng, cg, theFast Fourier Transform program from Chapter ssz
ManumbusuamlydimibmufiMOtol.
Emeptineaseolcomputationmfcourse.
llaed@r)isthewcaedDuae6-funaim.8eeanystandardtenondistnbutions
```

```
x)= I@deo[(et(x))%] = "if (5)
The ordinary differential equation 5 has the formal solution
@=r,f@,@dxltx) <6) I
I
```

   that defines the new pm's distributed according to f(x)dr, if E is a pm uniformly distributed on [0,1]. In other words, we have to solve a (usually) transcendental equation to calculate X(E).

   Since most simulation problems demand a lot of pm 5, it is no use solving Eq. 6 in real time. The better solution is to define a large enough table of the Xs, and look them up according to E. In this I case we actually want an integer PRNG uniformly distributed on [0,N-1], where the table has N entries.

```
\ O HARVARD SOFTWORKS 19%, ALL RIGHTS RESERVED.
HEX
IVARA 2VARB OSFFVARMX

:RANDOM AB+ MX OVER U<

IF MX 1+ - THEN

2' MX OVER U<

IF MX - THEN
```

```
BIS A DUP IS 8 ;
:RANDOMIZE (seed1seed2#bits-)

2 MAX 0F MIN \#bitstruncatedtorange216

1 SWAP SAL 1- IS MX

MX MOD IS A

MX MOD IS B

50 DO RANDOM DROP LOOP:
```

h

Fig. 34! PRNG supplied with HS/FORTH

In Fig. 3-3 above we exhibit a PRNG that produces 16-bit intege . uniformly distributed on [0, 2" l] . l h ve no idea of its operandi or its origin, but it passes the 1 tests described above.

We also need a way to invoke user-defined functions: the meth we have found is shown in Fig. 34 below. I

```
ma-MPOHW 83
```

```
VARIAHE <F>

:USE( [COMPILE] ' (FA <F> i
:F(X) <F> BtECUTE ;
BEI'EAD' <F> \rrllto<F>iocl
```

Fig. 34Prorocol for function usage in FORTH

We also require a word analogous to , ("comma") that stores 32-bit floating point numbers in the parameter field of a word:

```
: F32, HERE-L 4 ALLOT R32! ;
```

```
\ store a 32-bit # from the fstack in the first
\ available place in the dictionary
```

With these auxiliary definitions we are in a position to define a word that creates tables of random variates and traverses them in a random order:

```
\ Defining word for tables of pm's distributed according to a given distribution:
\ P(x<X(xi)) = id defhes X(xi).
\ Note: the first entry is autormticaliy 0.
\ fn.name oonvetts xi to X(xi)
\ dist.name is the name of the random N-long table created by )DISTRIBUIION

:SHAKE.UP ( #bls -- ) TIME@ XOR -ROT XOR ROT RANDOMIZE ;

                          \ randomize seeds
: )DISTRIBUTION: DUP LG2 ( --N #bits=ln[N])
SHAKE.UP                  \ Inlhllzepmg
CREATE F=O F32, \rmkefirstemy
(N--) 1 DO \Nentrytabie
IS->F I' S->F F/ \ltiontstack
F00 F32. \evalmteXOtiiand!
LOOP
DOES> RANDOM 4' + R32@ ;
BEI-EAD" A RANDOM \makethesewordslocd

\Usage: USE( Inname N )DISTRIBUTION: dbtname
```

In Fig. 3-5 below we exhibit a frequency plot of 10,000 random variates drawn from a (rather coarse) table of 64 entries, according to the distribution $p(x)dx = xe^{-x}dx$, together with $p(x)$.

Fig. 3-5 Random variates from table of 64 from distribution $p(x) = xe^{-x}$

With slightly more elaboration we can arrange for each table to have a unique prng, thus minimizing correlations between tables. Because the resulting data structure is nearly an "object", it is worthwhile to see how this may be done.

To make the prng unique, all that is necessary is to make the seed VARs unique to a table, and to redefine RANDOMIZE and, RANDOM to know how to get a given tables seeds. We see that RANDOM is invoked only in the runtime code for the structure. This means it has access to the base address, hence the VARs can be replaced with cells in the table, and the seeds planted there.