polyFORTH® Reference Manual

For Use with
Virtual Machines
Implemented on
GreenArrays Chips

Based on the Fifth Edition of the *polyFORTH Reference Manual* by Elizabeth D. Rather, Leo Brodie, and the Technical Staff of FORTH, Inc., this manual has been reformatted for 8.5x11" paper and for distribution by GreenArrays. The only intentional alteration in its content has been the replacement of diagrams that did not translate well from the source materials to PDF format.

This manual describes the model in use by FORTH, Inc. at the time of its writing; this precedes the publication of ANS Forth, which is therefore not mentioned herein.

As is customary with polyFORTH documentation, features and details of each implementation is described in a *Supplement* to this manual. In the case of virtual machine implementations such as those for the GreenArrays chips, optimal use of the hardware demands significant departures from the published standards as well as from the model described in this manual. Read this manual to familiarize yourself with the polyFORTH model and development tools; then, for a complete understanding, read the Supplement for the implementation you will be using.

Your satisfaction is very important to us! Please familiarize yourself with our Customer Support web page at http://www.greenarraychips.com/home/support. This will lead you to the latest software and documentation as well as resources for solving problems and contact information for obtaining help or information in real time.

Terms and Conditions of Use for Free Software

The polyFORTH system and its source code and documentation is made available to you for the express purpose of facilitating your successful use of our chips. By accepting it, you agree that we (GreenArrays, Inc. and FORTH, Inc.) make no warranty of suitability or correctness, no guarantee of future maintenance, and no guarantee of upward compatibility; and you agree that you will use it at your own risk. You may redistribute the system with your own software for our chips added, but all disclaimers and restrictions must be carried forward with each distribution. In addition, so that we may provide appropriate support responses, you must clearly identify which versions of polyFORTH® and arrayFORTH® your distribution is based upon, and include in your redistribution a complete list of any changes you have made to those systems for your application. In addition if you have made any changes to a trademarked product you must label your distribution as "based on", not "as", the trademarked product. It is recommended that you include unchanged versions of the software upon which your work is based so that anyone may confirm your lists of changes and accurately grasp their effects.

Unless specifically noted otherwise in the documentation accompanying any software distributed in our releases, that software may only be used with chips manufactured by GreenArrays.

Otherwise, you may make as many copies of the software as you wish, and you may distribute it without any changes whatsoever to whomever you wish. Any changes you might make to the software, any works you may derive from it, and any software you develop for our chips using it, are entirely your property to do with as you please so long as you respect our trademarks and use them only with our chips as noted above, and so long as you require those to whom you sell or deliver your software to agree with these same terms. For example, if you create an enhanced version of arrayForth and wish to market it under your own name as a different product, that is your right. If on the other hand you have or make chips or FPGAs that run an instruction set like ours, or buy chips similar to ours from someone else, you are prohibited from using our software on your chips without our written permission.

polyForth® was ported to the G144A12 by GreenArrays, Inc. and we have been granted permission by FORTH, Inc. for use of its Registered Trademark as well as for making its model available publicly. You are authorized to use polyFORTH as a development tool for systems and applications that run on our chips. *You are not authorized to use this software with other chips than ours, nor to port it to other computers than the F18A, without a polyFORTH license from FORTH, Inc.*

Do Not Use this software unless you understand and agree with these terms and conditions!

polyFORTH ISD-4

REFERENCE MANUAL

By:

Elizabeth D. Rather
Leo Brodie

and the

Technical Staff of FORTH, Inc.

Fifth Edition

Fifth edition, November 1986
Tenth printing, May 1994

Reformatted for 8.5x11 paper and GreenArrays, Inc. distribution August 2012. No intentional changes were made in its content.

FORTH, Inc.
5959 W Century Blvd Ste 700
Los Angeles, CA 90045
(800) 55-FORTH or (310) 999-6784
http://www.forth.com

Table of Contents

1.0 INTRODUCTION

The *polyFORTH ISD-4 Reference Manual* is designed to provide a reference source for the most common features of the polyFORTH integrated software development system. We assume at least an elementary knowledge of Forth, of a level consistent with having studied *Starting FORTH* and attended a polyFORTH programming course, or the equivalent. If you are new to Forth, we encourage you to begin learning by reading *Starting FORTH* carefully, working the problems at the end of each chapter.

This *Reference Manual* is primarily intended to describe how a programmer can use polyFORTH to solve problems. This is a rather different goal from explaining how polyFORTH works, but it is a practical necessity for the new user of a polyFORTH system. This manual is also organized to serve experienced programmers who need to check some point quickly.

We also highly recommend that you spend some time actually reading the polyFORTH source supplied with your system, about 40 pages of program source with extensive embedded documentation. polyFORTH was designed to be highly readable, and the source listing offers many examples of good usage and programming practice.

We cannot cover "all" polyFORTH commands: polyFORTH is an extensible system and no two implementations need or use identical components. What we can do is provide a detailed exposition of the most valuable and most commonly used features and facilities of the system from which your application begins.

FORTH, Inc. supports polyFORTH for a growing number of mini and micro-computers. Since hardware is unique for each computer, it is not feasible for this document to cover every feature of every computer supported. This *Reference Manual* presents features common to all polyFORTH systems. In discussing features that are hardware-specific, particularly dictionary structure, high-level object format, data base management, and peripheral drivers, an idealized model of a polyFORTH system is used. Refer to the CPU-specific supplement that accompanies each system for specifics of that system.

1.1 FORTH LANGUAGE FEATURES

Many of the following topics are treated in a tutorial form in *Starting FORTH*. This section highlights special considerations arising from the polyFORTH implementation. More detailed technical discussions of subjects covered here will be found in later sections of this manual, especially Section 2.0.

1.1.1 Dictionary

The dictionary contains all the executable routines (or "words") which make up a polyFORTH system. System routines are entries predefined in the dictionary that become available when the system is booted. Electives are definitions that are optionally compiled to the dictionary just after booting. User-defined words are entries that you add subsequently. System and elective definitions are available to all users in a multi-user configuration, whereas user-defined words are available only to the user. Otherwise there are no differences in size, speed, or structure. You may make user words available to other users simply by loading them with the other electives.

The dictionary is the fundamental mechanism by which polyFORTH allocates memory, and by which Forth performs "symbol-table" operations. Because the dictionary serves so many purposes, you should be sure you understand how to use it. You may wish to review this material in *Starting FORTH*.

The dictionary is a linked list of variable-length entries of Forth words and their definitions. It grows toward high memory; each entry points to the one that logically precedes it (see Fig. 1.1). The next available cell at the end of the dictionary is pointed to by the variable **H**. This address will be put on the stack by the word **HERE**.

The dictionary is searched by sequentially matching names in source text against names compiled in the dictionary. To speed dictionary searches, the dictionary is organized in eight linked chains of approximately equal length. A word will be found in one of these chains depending upon a "hash value" computed from its name and the "vocabulary" in which the word is located (see the reference below on vocabularies, and *Starting FORTH*, p. 219*). The search follows the selected chain until a match is found or the end of the chain is reached. Since the latest definition will be found first, this organization permits a word to be redefined, a technique that is frequently useful.

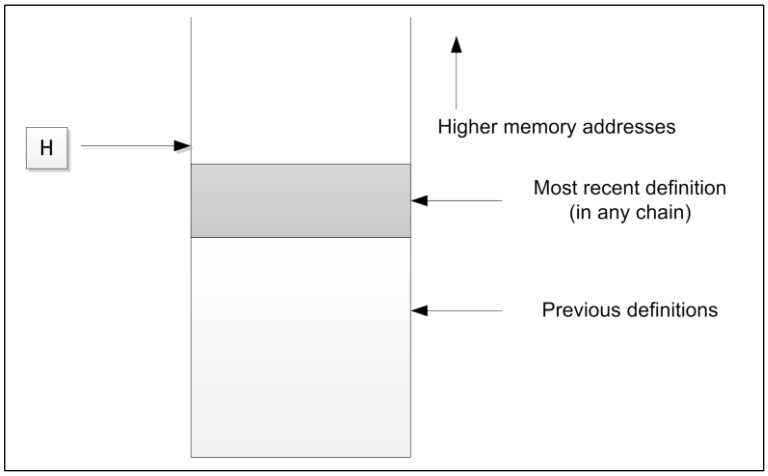


Fig. 1.1

The "top" of a user's dictionary. **H** points to the next available byte; there is a copy of **H** for each user in a multi-user system.

A "vocabulary" is a subset of the dictionary containing words for some special purpose. There are three vocabularies present in all systems and these are available to all users on a re-entrant basis:

Word	Description
FORTH	Contains all "system" commands plus the shared re-entrant portion of an application.
EDITOR	Contains commands used to edit polyFORTH text.
ASSEMBLER	Contains commands used only in code definitions (op-codes, etc.).

In addition, each user in a multi-terminal system may have his own private dictionary, which may contain words added to the standard vocabularies. Each user has a private variable called **CONTEXT** which tells the text interpreter the order in which to search the vocabularies. All private dictionaries are linked to the public dictionary, which means that a search will start in the user's dictionary and thread back through his private definitions before searching through the shared dictionary. This has the effect of protecting private portions of the dictionary from mistaken or unauthorized use.

* In the second edition; p. 242 in the first edition.

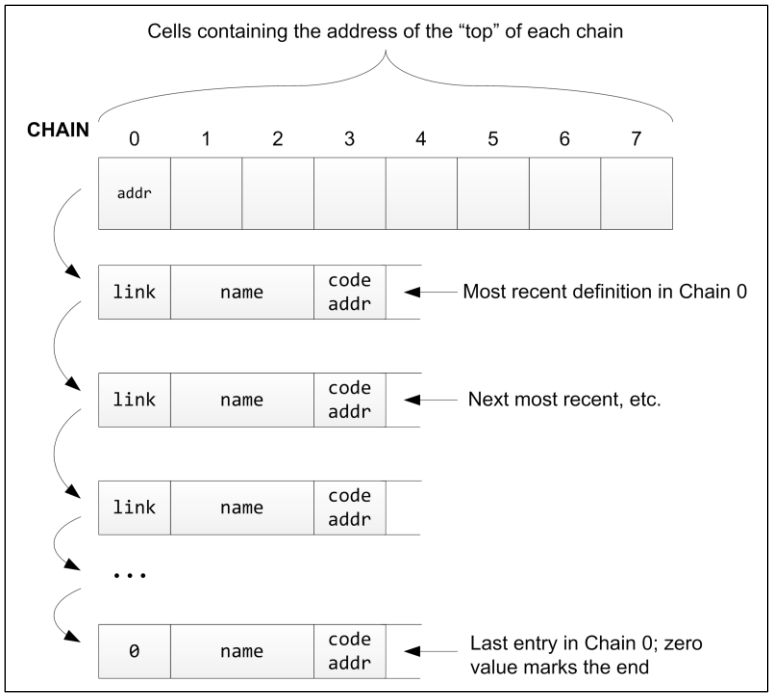


Fig. 1.2

Diagram of the logical structure of the polyFORTH dictionary. Logically consecutive definitions don't have to be in contiguous memory.

The essential structure of all dictionary entries is the same for all words and is diagrammed in Fig. 1.2. The first cell, called the link, contains the location of the preceding entry. This speeds up searches, which start at the "recent" end of the dictionary and work backwards to the "older" end. By this process, the most recent definition of a word is always found. In a developed application, where the user is dealing with the highest level of the program, this process optimizes search time.

The second cell of a dictionary entry contains the count of characters in the name field, followed by the first of three or more characters of the name. The name may be longer, depending on what the value of the user variable **WIDTH** was when the definition was compiled, and how long the name is. **WIDTH** sets the maximum number of characters which will be saved when a name is compiled. On processors that require addresses to be on even-byte addresses, **WIDTH** should be set to an odd value. On most machines, **WIDTH** works with any value from 3 to 31.

WIDTH is a variable, two of whose bytes are used separately. The lowest addressed byte holds a character count up to 31, to be used as the maximum character count of the next definition. The next byte of **WIDTH** stores a default value (also up to 31), to which the lowest addressed byte is set after each definition. Most of the polyFORTH nucleus is compiled using three-character names, so **WIDTH** in the system as furnished is set to the hex value 0303. If you want to change **WIDTH** to compile a 31-character name, you may do it two ways. Using:

```
31 WIDTH C!
```

changes the lowest addressed byte only. After the next definition, **WIDTH** will again be set to 3. You may get the same offset by using the **~** ("tilde") command. Using **~** preceding a definition sets **WIDTH** to 31, specifying that the next definition only will be compiled to its full width. For example:

```
: CONTEST   ... FORTH words ... ;

~ : CONTEXT   ... FORTH words ... ;
```

```
: DABBLE   ... FORTH words ... ;
```

In this example, three definitions will be created. The entries for **DABBLE** and **CONTEST** will each have names with three letters and a count. The entry for **CONTEXT** will contain all seven letters. The words **CONTEST** and **CONTEXT** will not interfere with each other. If their order in the dictionary were reversed, **CONTEXT** would never be found by a search because **CONTEST** would be found and executed by each search for **CONTEXT**. As a result, it is important that such name conflicts be resolved by always using ~ with the most recent of the conflicting pair.

To make a "permanent" change of **WIDTH** to, let's say, 31 characters, use:

```
HEX  1F1F WIDTH !  DECIMAL
```
(hex 1F is 31 decimal)

This changes both bytes of **WIDTH** to 31.* Remember: **WIDTH C!** stores a byte into the lowest addressed byte of **WIDTH**, while **WIDTH 1+ C!** stores a byte into the highest addressed byte of **WIDTH**, and **WIDTH !** changes both lowest and highest addressed bytes. *Never* use **31 WIDTH !**, which puts 0 in the default byte of **WIDTH**, rendering all subsequent definitions invisible (because **WIDTH** will be thereafter set to 0).

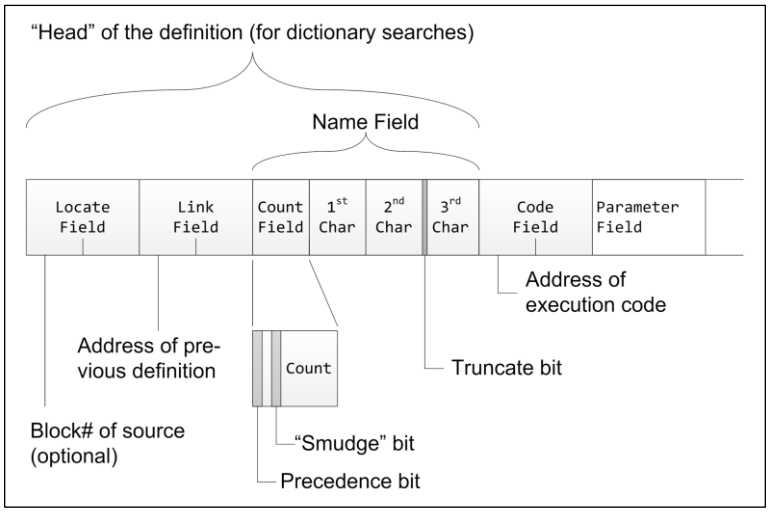


Fig. 1.3

Structure of a dictionary entry. The actual byte order and position of bits varies somewhat with each CPU; see the *polyFORTH ISD-4 CPU Supplement* for the correct order for your system.

The use of three characters and a count gives you far more flexibility than a simple limit on the number of characters, and saves a substantial amount of space in comparison with compiling all names to full width, but it does require uniqueness in the length and first three characters. Any characters other than space, backspace and carriage-return can be used as part of a name field.

Although the name field is arranged in each implementation to optimize each machine's dictionary search, the general model in Fig. 1.3 is followed whenever possible. In this example, since the longest allowed name field has 31 characters, and needs only five bits to express a count, the byte containing the count in a dictionary entry has its three most significant bits available for other purposes. The most significant bit, bit seven, is used as the

* On 32-bit sytems such as the M68000, **WIDTH** is a 16-bit cell. Therefore, to change both bytes at once you would use the phrase:

```
HEX   1F1F WIDTH W!   DECIMAL
```

precedence bit. When the precedence bit is set, the word is executed by the compiler at compile time. The precedence bit is set by the word **IMMEDIATE**, and is used for a few special words, such as compiler directives. The precedence bit is set to zero for most words. Bit six is reserved for future use by FORTH, Inc. Bit five is the "smudge" bit. When this bit is set, the word is invisible to a dictionary search. The smudge bit is set by the compiler when starting to compile at the beginning of a colon definition, to prevent unintentional recursive references. It is reset by the semicolon that ends the definition. This is accomplished by the word **SMUDGE**, which toggles the smudge bit of the most recent definition, setting it if it is reset and vice-versa.

In another common arrangement the most significant bit of the cell containing the count and first character remains the precedence bit, but bit seven of the cell is the smudge bit. Consult your *CPU Supplement* for your machine's arrangement. The first cell after the name field contains a pointer to the code to be executed for the definition. This pointer is called a "code field" and points to machine code whose behavior specifies the type of word.

For a **CONSTANT**, the code field address refers to code that puts the value of the constant (which is in the second cell after the name field) on the stack:

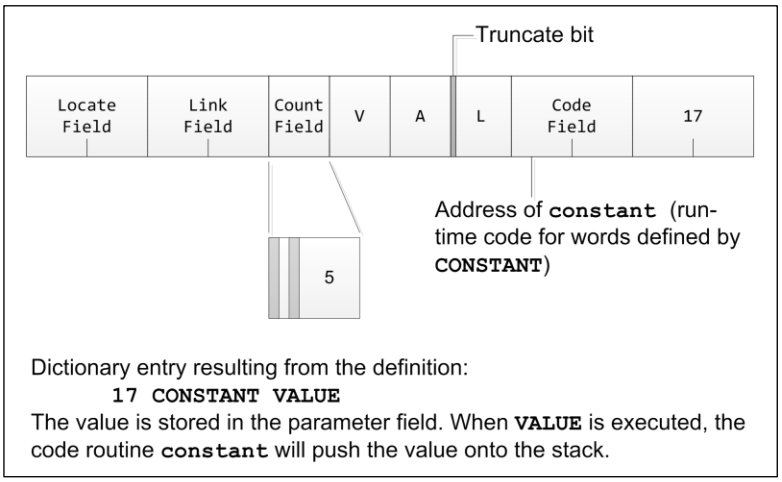


Fig. 1.4

For a **VARIABLE**, the code field address refers to code that puts the address of the *value* (stored just after the code field address) on the stack:

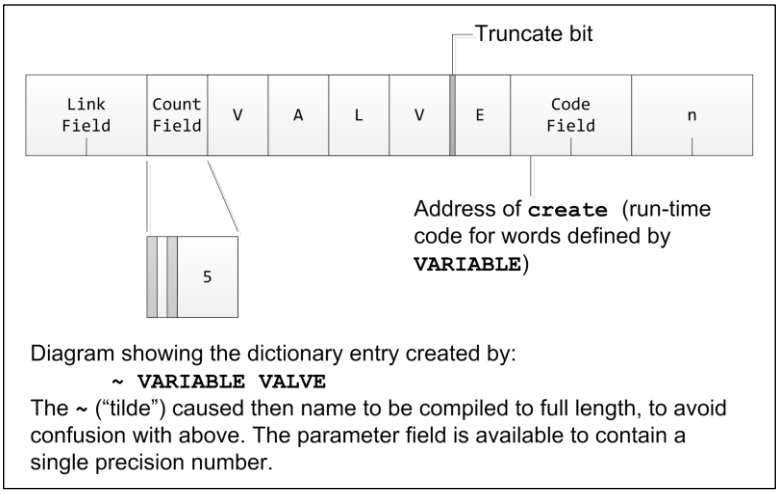


Fig. 1.5

For a defined word, the code field address points to a portion of the address interpreter that will begin following a string of addresses starting just after the code field cell and will continue until it finds the **EXIT** compiled by **;** which terminates that definition. A diagram of a **:** definition is shown in Fig. 1.6.

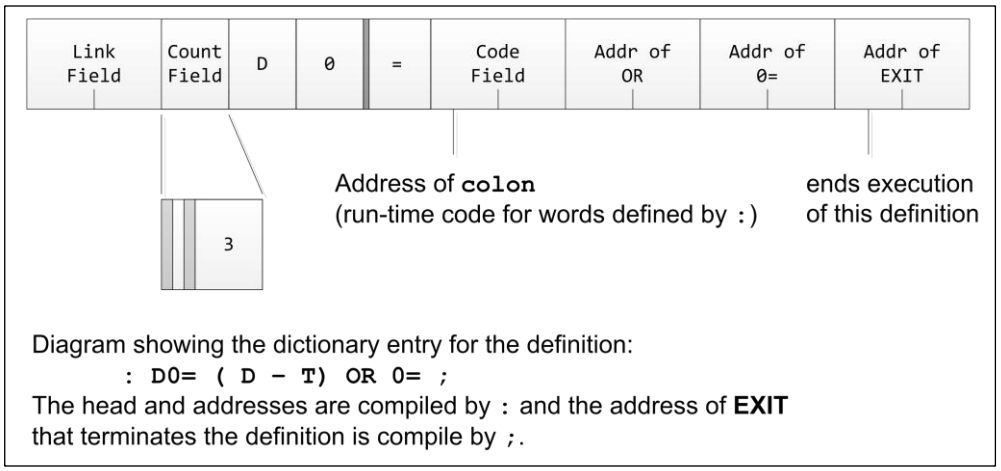


Fig. 1.6

For assembler coded words, the pointer is to the next cell itself, which contains the first instruction of the code that is executed directly:

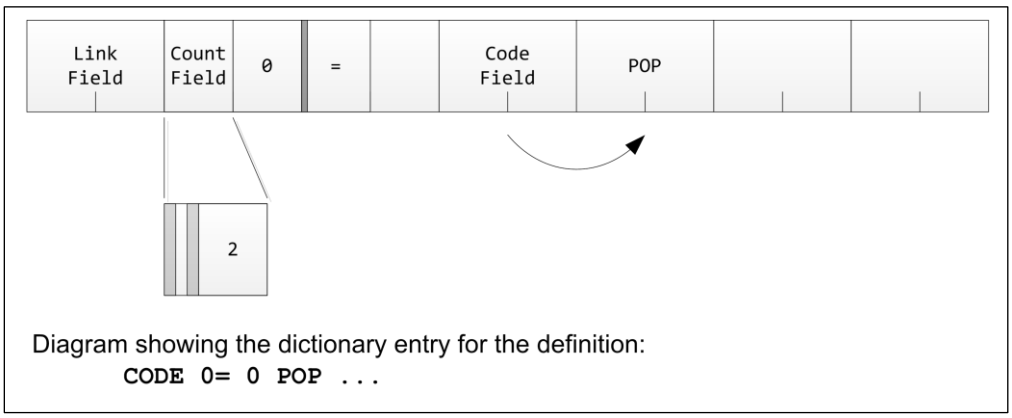


Fig. 1.7

A problem which can occur is finding the code field address from a given link field address, despite a variable-length, possibly truncated name field. The predefined word **CFA** translates a link field address to a code field address, so that you do not need to write this code yourself.

The cells after the code field address are called the parameter field, which is of variable length. Words such as **CONSTANT**s and **VARIABLE**s keep their values in the first parameter field cell, as noted above. Other definitions may keep several values. In the latter cases, the length of the parameter field is either determined by the type of definition or kept in one of the early cells in the field.

The polyFORTH words concerned with dictionary management are **H** and **HERE**:

Command	Function
H	(A variable.) Contains the location of the head (next available cell) of the user's private dictionary.
H 2+	(**H 4+** for 32-bit machines.) Contains the address of the beginning of the user's private dictionary.
HERE	Pushes the contents of **H** onto the stack.
ALLOT	Increments **H** by a number of bytes given on the stack.

REFERENCES

CODE Definitions, Section 6.1
Code Field Addresses, Section 2.7.4
Creating Dictionary Entries, Section 2.7.1
Vocabularies, Section 3.4

1.1.2 Parameter Stack

Each multiprogrammed task has a parameter stack in the upper part of its partition, as shown in Fig. 1.8. The purpose of the parameter stack is to contain numeric operands for polyFORTH operators. polyFORTH operators expect their parameters on this stack and leave their results there. The size of a parameter stack is indefinite; its legal domain extends from its start downward to the top of the user's dictionary, less the space required for **PAD** and working storage. These amounts vary, but typically approximately 256 bytes must be reserved for working storage. The parameter stack rarely grows beyond 10-20 entries in a well-written application.

When numbers are pushed onto or popped off of the stack, the remaining numbers are not moved. Instead, a pointer is adjusted to indicate the last used (bottom) cell in a static memory array. On most computers, the stack pointer is kept in a register.

It is important that the stack extend toward low memory for two reasons:

1. So that a positive address relative to the stack pointer will locate numbers on the stack, and

2. So that double-cell items are placed on the stack in the same relative position that they have in memory. Because the stack grows toward low memory, a "push" operation involves decrementing the stack pointer, while a "pop" involves incrementing it.

A number encountered in text by the text interpreter will be converted to binary and pushed onto the stack. polyFORTH "nouns" (data items such as **VARIABLE**s and **CONSTANT**s) are defined to push their addresses or values onto the stack. Thus, the stack provides a medium of communication not only between routines but between a person and the computer. You may, for example, place a number or address on the stack and then type a word which acts on this to produce a desired result. Typing:

```
12 2400 * 45 / .
```

pushes the number 12 on the stack, pushes 2400 over it (see Fig. 1.9), replaces both numbers by their product (*****), pushes 45 on the stack, divides the product by 45 (**/**) and types the quotient (**.**). All numbers put on the stack are removed, leaving the stack as it was before typing 12.

The basic dictionary provides some words for simple manipulation of single and double-length operands on the stack: **SWAP**, **DUP**, **DROP**, **2SWAP**, etc. (covered in detail in Section 2.1).

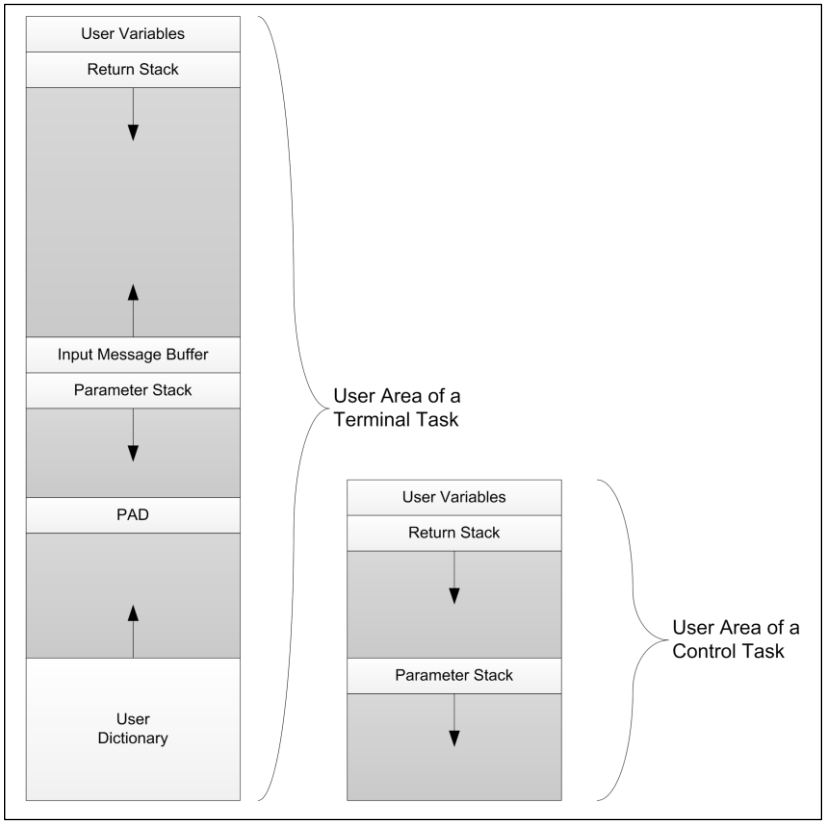


Fig. 1.8

Relative location of the parameter stack in both a terminal task and a background task.

The use of the push-down stack simplifies the internal structure of polyFORTH and produces naturally re-entrant routines. Passing parameters via the stack means that fewer variables must be named, reducing the amount of memory required for named variables (as well as the associated programmer's housekeeping).

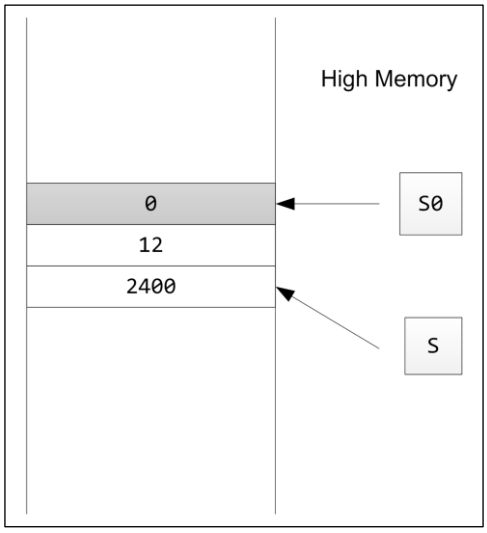


Fig. 1.9

The parameter stack in memory. **S0** points to the "empty stack" location (which is constant for each user), and **S** (in a register) points to the top item. A zero is kept in the stack underflow position for "fail-safe" protection.

A pointer to the top (latest entry) of the user's stack is maintained by the system. Its actual value may be obtained at any time by use of the word **'S** (pronounced "tick-S").

The cell immediately above the "bottom" parameter stack location always contains a zero. Just above this is a space which, in most systems, is used for the terminal input buffer. At the top of this region lies the return stack. This arrangement helps make the system resistant to minor stack underflows. Severe underflows, however, can cause a system crash.

polyFORTH checks for stack underflow only after attempting to execute a word from the input stream. Underflows that occur during execution will not be detected at that time.

The result of a detected stack underflow is the message:

```
Stack empty
```

followed by a system abort.

REFERENCES

Forth Re-entrancy, Section 4.1
Stack Manipulation, Section 2.1
System Abort Routines, Section 2.4.7

1.1.3 Return Stack

Each multiprogrammed task has its own return stack, located above its parameter stack in memory (see Fig. 1.8). The return stack serves the following purposes:

1. It holds return addresses for nested definitions.
2. It holds loop parameters.
3. It saves temporary data, especially file and record pointers in the Data Base Support.
4. It saves interpreter pointers when loading source text blocks.

Since the return stack has multiple uses, care must be exercised when using it to avoid conflicts.

There are no commands for direct manipulation of the return stack, except for those moving parameters to and from the top of the stack.

The maximum size of the return stack for each task is specified at the time that the task is defined, and remains fixed during operation; a typical size is 64 cells (136 cells on 32-bit machines).

REFERENCES

Loading, Section 3.3.1
Loop Parameters, Section 2.4.4
Parameter Stack, Section 1.1.2
Return Stack Use in Data Base Support, Section 8.1.3
Transfers Between Stacks, Section 2.1.3

1.1.4 Text Interpreter

The text interpreter serves two critical functions:

1. It executes the commands that users type.

2. It executes commands in source blocks stored on disk.

The terminal is the default text source. The terminal input interrupt handler will accept up to 80 characters into a text buffer called the "Terminal Input Buffer," or **TIB**. Either the acquisition of the eightieth character or an ASCII RETURN character ($0D_H$) will cause the terminal's task to be activated, under the control of the interpreter which will process the text in the buffer. For example, typing:

```
100 LOAD
```

however, directs the interpreter to the 1024-byte string that is stored in Block 100. This string is brought into memory from mass storage automatically. In order to let one block load another, the current block number (the user variable **BLK**, set to zero for terminal input) and the text interpreter character pointer (the user variable **>IN**, pronounced "to-in") are saved on the return stack. So, in the middle of a **LOAD** of one block, that block may direct **LOAD**ing a different block, and return to complete the first block.

When the text interpreter executes a defining word (such as **CONSTANT**, **VARIABLE**, or **:**) the result is that a definition is compiled into the dictionary.

Because the text interpreter is a colon definition, it is interpreted by the address interpreter. The text interpreter is defined by a phrase equivalent to:

```
: INTERPRET   BEGIN  -' IF  NUMBER  ELSE
   DROP  EXECUTE  DEPTH 0<
   ABORT" Stack empty"  THEN AGAIN ;
```

where:

Word	Function
BEGIN	Marks the beginning of an infinite loop.
-'	Extracts a word from the input string using the delimiter 32 ("blank" in ASCII code). It then searches the dictionary, returning a true value on the stack if the interpreter does not find a match for the word, false otherwise.
IF	Continues on a true condition, otherwise skips to the word after **ELSE**.
NUMBER	Attempts to convert the word to a binary number; if the conversion fails (due to illegal characters), it aborts with an error message.
ELSE DROP	Eliminates an unused address left by **-'.**
EXECUTE	Executes the address of the word found in the dictionary.
DEPTH 0<	Returns a true value if there is a stack underflow condition, false otherwise

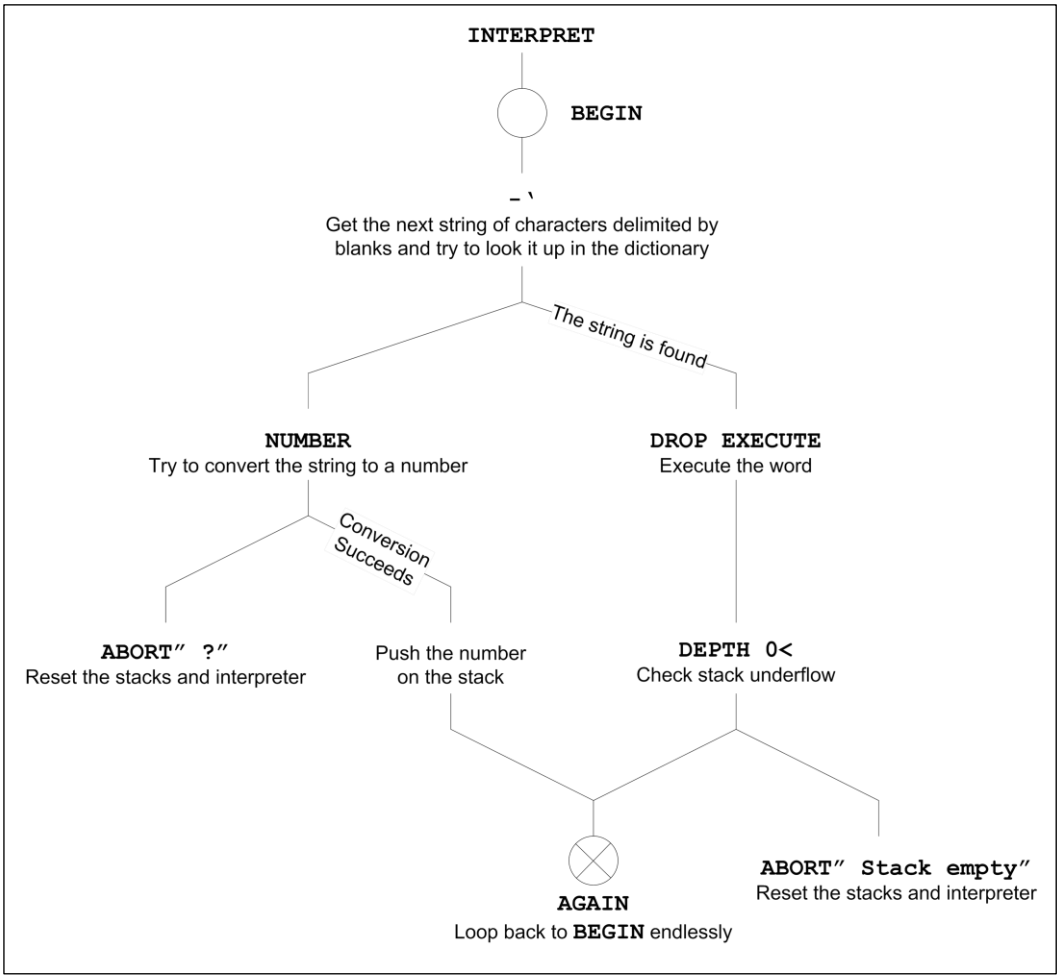


Fig. 1.10

Diagram of **INTERPRET**, the text interpreter.

Word	Function
ABORT" Stack empty"	Aborts with the error message "Stack empty" if there was a stack underflow.
THEN	Marks the place where both branches merge and continue.
AGAIN	Returns to **BEGIN** to repeat the procedure.

This process is diagrammed in Figure 1.10.

REFERENCES

Disk Blocks, Section 1.2.2
System Abort Routines, Section 2.4.7
Text Interpreter Words, Section 2.6
The **LOAD**ing Process, Section 3.3.1
Using **EXIT** to Leave a Load Block, Section 2.4.6

1.1.5 Numeric Input

The word **NUMBER** is used by the text interpreter to convert strings of ASCII numerals and punctuation into binary integers that are pushed onto the stack. polyFORTH interprets any number containing an imbedded punctuation character (see below) as a double-precision integer.* Single-precision numbers are recognized by their lack of special punctuation. Conversions operate on character strings of the following format:

[-] dddd [punctuation] dddd ... delimiter

where "dddd" is one or more valid digits according to the current base or radix in effect for the user. The user variable **BASE** is always used as the radix. All numeric strings must be ended by a blank or a carriage return. If another character is encountered (*i.e.*, a character which is neither a digit according to the base nor punctuation or a blank), an abort will occur. There must be no spaces within the number, since a space is a delimiter.

The leading minus sign, if present, must immediately precede the first digit or punctuation character.

Any of the following punctuation characters may appear in a number:

Word	Description
,	(comma)
.	(period)
+	(plus)
–	(hyphen, may appear anywhere other than to the immediate left of the most significant digit)
/	(slash)
:	(colon)

All punctuation characters are functionally equivalent. A punctuation character causes the digits that follow to be counted. This count may be used later by certain of the conversion words. The punctuation character performs no other function than to set a flag that indicates its presence. Multiple punctuation characters may be contained in a single number; the following two character strings would convert to the same number:

1234.56

1,234.56

Punctuation characters do not affect the resulting number.

The user variable **PTR** is used during the number conversion process to track punctuation. **PTR** is initialized to a large negative value, and is incremented every time a digit is processed. Whenever a punctuation character is detected, it is set to zero. Thus the value of **PTR** immediately following a number conversion contains potentially useful information:

- If it is negative, the number was unpunctuated and is single-precision.

* On 8-bit and 16-bit computers, a "single-precision integer" is 16 bits wide, and "double-precision" implies 32 bits. On 32-bit CPUs such as the 68000, widths are 32 and 64 bits, respectively.

- Zero or a positive non-zero value indicates the presence of a double-precision number, and gives the number of digits to the right of the right-most punctuation character.

This information may be used to scale a number with a variable number of decimal places. Since **PTR** doesn't care (or, indeed know) what punctuation character was used, it works equally well with American decimal points and European commas to start the fractional part of a number.

The working copy of the number during the conversion process is 32 bits on the stack. If the number is single-precision (negative **PTR**), the high-order part of the working number (normally zero) is saved in a location given by the phrase:

```
'NUMBER 2+          (4+ on 32-bit CPUs)
```

This information may be recovered to force the number to double precision. This is useful when dealing with naturally unpunctuated large numbers such as five-digit zip codes.

Most large numbers are punctuated, however, because humans remember them better that way. Some numbers such as:

229-48-0332
8/03/40
372-8493
3,124,896
8:45:06

will automatically convert appropriately.

REFERENCES

Use of the Text Interpreter for Number Input, Section 2.6.2

1.1.6 Address Interpreter

The address interpreter executes previously compiled high-level definitions, which compile to a string of absolute addresses of definitions to be executed in turn. When a standard Forth colon definition is invoked, the run-time code for a colon definition (called **colon**) sets the interpreter pointer **I** to the parameter field of the definition and executes **NEXT**.* **NEXT** is the most fundamental routine of the address interpreter, which branches to the next routine (address) to be executed as indicated by **I**. **NEXT** performs the following functions:

1. Moves the current value of **I** into **W**, so that **W** points at the code field address of the word now being executed.

2. Increments **I** to point to the next address.

3. Performs an indirect jump to the address in **W**.

4. Increments **W** to point to the parameter field of the word being executed.

The efficiency with which these operations can be performed is a measure of a machine's overhead to run high-level polyFORTH. On most computers **NEXT** can be reduced to one or two instructions. If the word being

* The interpreter pointer **I** should not be confused with the **I** which fetches a copy of the top of the return stack.

executed is a code word, the machine code is executed directly. If it is another **:** definition, however, the run-time code for **:** is executed. The word **colon** performs the following functions:

1. Pushes the current value of **I** onto the return stack (which has a top-of-stack pointer named **R**).
2. Moves **W** (the parameter field address) into **I**.
3. Executes **NEXT**.

At the end of a **:** definition, the run-time code for **;** (the word **EXIT**) is executed. **EXIT** performs the following functions:

1. Pops the return stack into **I**.
2. Executes **NEXT**.

The time required to execute the run-time code for the **: ... ;** pair is the overhead to nest high-level definitions.

REFERENCES

:, Section 2.7.4
Compiling Words and Literals, Section 2.8
EXIT, Section 2.4.6
I, Section 2.7.4, 6.4

1.2 polyFORTH OPERATING SYSTEM FEATURES

polyFORTH is based on a multitasking, multi-user operating system. Many polyFORTH implementations run in a fully stand-alone mode, in which polyFORTH provides all drivers for the hardware attached to the system. Some versions of polyFORTH, however, run in a "co-resident" mode with a host operating system. In the latter case, the drivers that supply I/O services for peripherals such as disk and terminals do so by means of issuing calls to the host system. Although co-resident systems may be somewhat slower than the standalone versions, they do offer full file compatibility with the host OS, and usually somewhat more flexibility in hardware configuration. Co-resident versions of polyFORTH usually offer all the system-level features of the native systems (including multi-user support on otherwise single-user systems such as MS-DOS), plus added commands for interacting with the host OS; the latter are documented in the *CPU Supplement* that accompanies this *Reference Manual*.

1.2.1 Typical Memory Organization

A diagram of memory use in a typical application is shown in Fig. 1.11. The area in low memory contains the precompiled portion of the program. The system electives are kept in source form to facilitate changes and additions. Since re-compiling electives and application into memory takes only a few seconds, this costs little. Re-compiling is only necessary during the boot procedure or after a system crash.

Each terminal task has a partition that contains its stacks, private (or "user") variable area, **PAD** (for text strings) and dictionary. A selected vocabulary may be compiled into this partition to do some particular kind of processing which is a subset of the application but which is not available to other users. The background task has a much smaller area, with only enough space for its stacks; there is no terminal associated with it. The routines that the background task executes are located in the shared area or the dictionary of one of the terminal tasks.

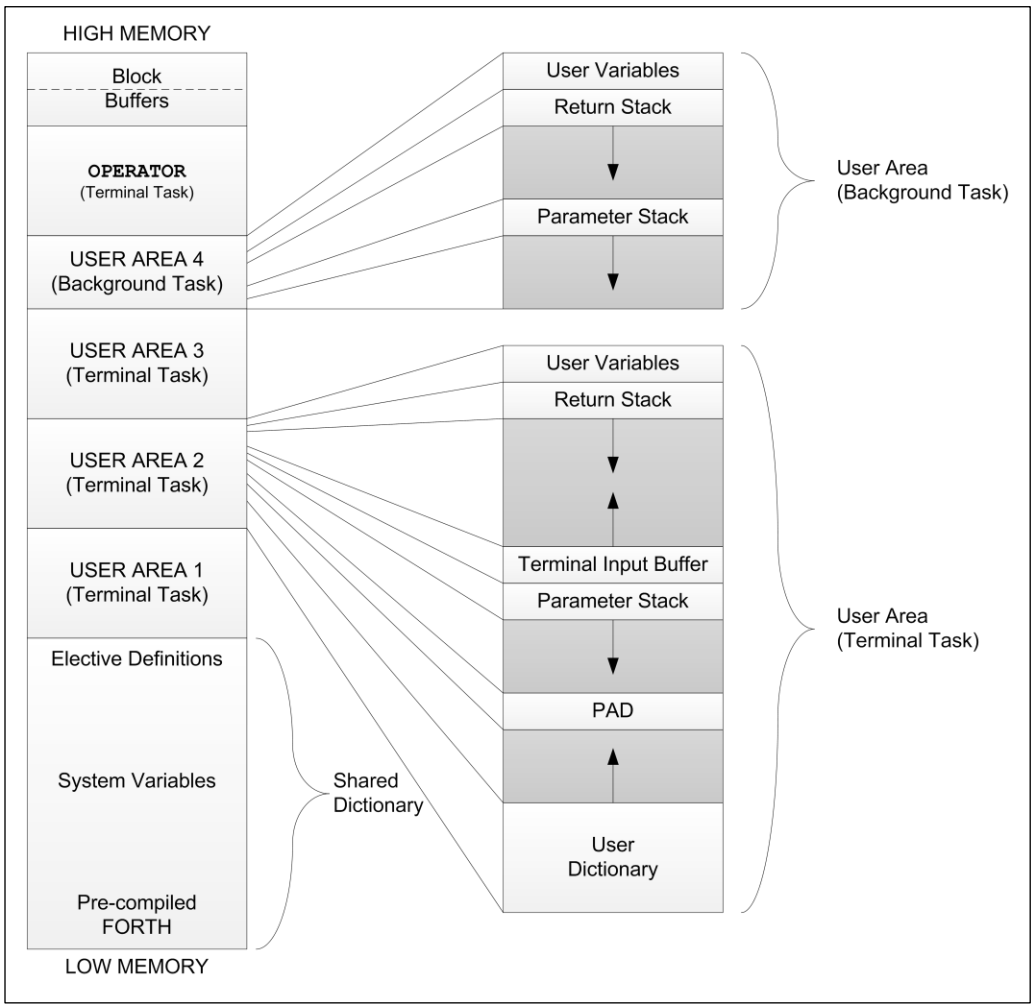


Fig. 1.11

Typical memory organization in a polyFORTH system.

1.2.2 Disk Block I/O

Disk I/O is handled by polyFORTH in standard blocks of 1024 bytes. This fixed block size applies both to polyFORTH source program text and to data used by polyFORTH programs. It is used because it allows I/O on different media with different physical sector or record sizes to be handled by one standard block handler. polyFORTH applications may put several data records into one block or combine several blocks to form one data record.

Standard polyFORTH systems have at least two 1024-byte block buffers in memory. This number may be increased (for example, in an application with many tasks). For each buffer, there is a "descriptor table" that contains the block number of the block currently residing in that buffer. The sign bit of the block number will be set if the block has been changed since it was read into the buffer.

A block is requested by the instruction:

```
n BLOCK
```

where *n* is the logical block number. The block handler will check the block buffers to see whether the requested block is already in memory; if not, it will fetch the block from disk.

When a block is to be read from disk, it will necessarily over-write a block that is currently in a buffer. polyFORTH optimizes disk performance by attempting to minimize the number of physical disk reads and writes that must be performed. This is done by maintaining the list of buffers in an order reflecting how recently the block currently in each buffer was referenced.

- When searching for a block, the most recently used buffer is checked first.
- When selecting a buffer to be over-written, the least recently used buffer will be selected.

This method of buffer management is called an LRU (for **L**east **R**ecently **U**sed) algorithm (discussed in detail in Section 3.2.1).

If the block buffer to be over-written has been updated, the updated block will be automatically written to mass storage before the requested block is read. Finally, **BLOCK** will push the address of the first cell of the requested block onto the stack.

There are no explicit disk input or output commands; **BLOCK** will always return the address of the block buffer in memory, having performed actual I/O only if necessary. To the programmer, all data is always manipulated in memory.

The command:

```
UPDATE
```

marks the most recently referenced buffer as having been updated. **UPDATE** is included in those operators in polyFORTH's text editor and Data Base Support option that change fields.

The disk buffer pool is shared among multiprogrammed tasks. This is an advantage because it tends to reduce disk access for "popular" blocks and avoids the problem of multiple copies of a block. In applications where block accesses need to be restricted, you may easily add the appropriate protections.

Please note that native (stand alone) versions of polyFORTH do not require any sort of directory in memory or on disk since block numbers are a direct function of disk address (the exact relationship is designed to suit the particular disk involved). Applications that involve management of complicated data-file structures sometimes do use a disk directory; this is a feature of the application, however, rather than of polyFORTH.

The only requirement for fitting data records into this scheme is that data record numbers be a fixed function of the block number; then you can define a word that will use **BLOCK** to fetch the block(s) that contain records requested by a specified record number. The accessing words in the Data Base Support use **BLOCK**.

Another related technique is useful for keeping very large arrays on disk. Sometimes data-record applications are written using a virtual byte array. An example of a handler for a virtual array of bytes stored beginning at Block 250 would be:

```
: VIRTUAL ( v - a)   1024 /MOD
   250 + BLOCK + ;

: V@ ( v - n)   VIRTUAL @ ;

: V! ( n v)   VIRTUAL !  UPDATE ;
```

In the example above, **VIRTUAL** converts an address for the virtual array into an address that points into a block buffer in memory. The phrase 1024 **/MOD 250 +** produces a block number on top of the stack, with an offset

into the block just underneath. The word **BLOCK** uses the block number on top of the stack and leaves the address of the first byte of the block buffer containing a 1024-byte segment of the byte array. The **+** adds the block buffer address to the offset to produce an absolute address into the correct block buffer. The operation of **V@** is straightforward, but note the **UPDATE** in **V!**. Without the word **UPDATE**, the changed block buffer data would never be written out to disk when the block buffer is reused.

Error checking is not performed in the low-level definition of **BLOCK**. Status bits are preserved, however, in order that higher-level routines may check for errors and take appropriate action. The reason that error checking is not done at the lowest level is that it is easiest to take the appropriate action at the application level. The action that is appropriate in a data base application, for example, will necessarily differ from the action appropriate in a disk diagnostic. You should check your system listing for details of your particular implementation. On most systems, the following information is available:

Variable	Content
PREV	Address of the descriptor table for the most recently referenced buffer.
DISK	Address of the **STATUS** cell of the task currently accessing disk, if any. Zero if disk is available.
DISK 2+	(**4+** for 32-bit machines.) Status data from the most recent operation using the disk controller.
'BLOCK	Address of the routine to be executed by **BLOCK**.
'BUFFER	Address of the routine to be executed by **BUFFER**.
'BUFFER 2+	(**4+** for 32-bit machines). The block number of the block just written.

The buffer descriptor pointed to by **PREV** contains the address of the buffer, as well as the block number of the block in the buffer. It also contains the address of the descriptor for the next most recently used buffer, and so on for as many buffers as are currently configured in the system.

REFERENCES

Data Base Support, Section 8.0
DISKING Utility, Section 5.3
The Disk Driver, Section 3.2

1.2.3 Multitasking

polyFORTH makes it easy to control multiple tasks, either asynchronous background tasks or independent terminal service tasks. Terminal tasks have text input/output facilities. Background tasks have some other function. A small but crucial set of commands is dedicated to the multitasking facility.

The polyFORTH multitasker consists of one relatively small code routine called **PAUSE**. **PAUSE** contains the idle loop of the system. When the system is "quiet," it is running around the **PAUSE** loop looking for a task ready to be awakened.

The limits on the number of tasks in the system are usually defined by memory size. Since polyFORTH definitions are naturally re-entrant, tasks rarely require much memory. The polyFORTH multitasking facilities do not limit the number of tasks. The tasks are linked in a circular chain (Fig. 1.12) in which a new task may be inserted. The multiprogrammer follows the chain. Each task has a "user variable area" which contains a pointer to the user area of the next task to be examined. This "round-robin" algorithm is extremely fast in execution; since the actual process of changing tasks is also simple and fast, the result is maximized service to all tasks. Should the constraints of a particular application require a more elaborate queuing scheme, it may be substituted directly for the standard definition of **PAUSE**.

The single task present after booting is named **OPERATOR**. This task services the terminal that is used to type additional loading commands. Since **OPERATOR** is the first terminal task up after the boot, it is the logical task to begin system loading and other operations. However, aside from being first and usually having more memory, **OPERATOR** does not differ from other terminal tasks.

A task is referred to by name. Invoking a task's name puts on the stack the address of a pointer to the first cell of its user area. This first cell contains the task's status. When a task is inactive the first cell typically contains a jump to the succeeding task, whose address follows in the next cell. Thus, in a quiet system the processor is simply jumping from one task to the next. When a task needs to be activated, a subroutine call to the "awaken" section of **PAUSE** is substituted for the jump.

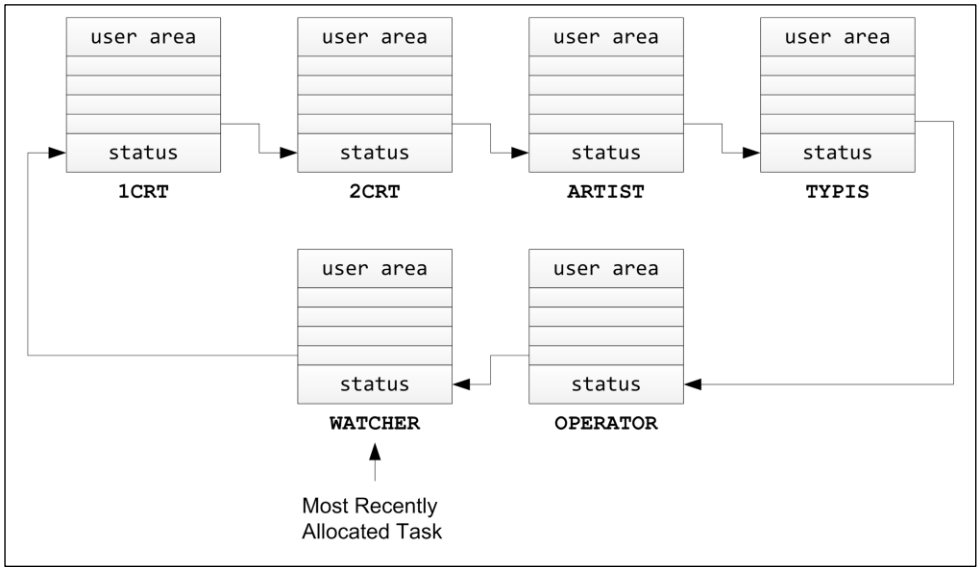


Fig. 1.12

Diagram showing several tasks linked in a polyFORTH "round robin."

When a task relinquishes control of the CPU, the following actions are performed to inactivate it:

1. The task's system pointers **I** and **R** are pushed on the task's parameter stack.

2. The task's current parameter stack pointer is saved in the task's user area.

3. The task's status is set to the appropriate instruction to jump to the multiprogrammer code that will activate the task.

There are three ways to enter the **PAUSE** loop in polyFORTH:

1. The word **PAUSE** gives each task the possibility of a turn, and then resumes execution.

2. The word **STOP** puts a task to sleep until something outside that task, such as an interrupt routine or another task, stores the "wake" instruction into the task's status cell.

3. The assembler code ending **WAIT** is used to re-enter the pause loop from assembly code, and like **STOP**, does not automatically re-awaken the task containing the **WAIT**.

The last step of most I/O routines is to enter the multitasking loop via **WAIT** to wait for the device interrupt routine to complete its data transfer (or other operation). By performing multitasking while waiting, service to

other tasks is maximized during I/O operations. Because most I/O operations enter the round-robin, polyFORTH's multitasker is said to be "I/O driven." Note that a task which performs no I/O and does not execute **PAUSE** will retain control of the CPU.

This method has several advantages. Since a change of tasks can only happen between Forth words, there is no need to save working registers from high-level programs. As a result, changing tasks takes only a few CPU instructions on any system. Secondarily, the programmer has full control and knowledge of when the CPU will and will not be relinquished, which can substantially simplify some coding situations.

On the other hand, the programmer does have the responsibility for putting **PAUSE** in any CPU-intensive routine that may run more than a few milliseconds without performing I/O. In general, such situations are rare; timing studies have shown that on a native 8088 polyFORTH in normal operation a task needs to wait over 1 ms for service only 1% of the time.

polyFORTH normally runs with interrupts enabled. Interrupt vectors branch directly to the code which services the interrupting device, without any system intervention or overhead. The interrupt code is responsible for saving and restoring any registers it needs.

Interrupt code (actual assembler code) is responsible for performing any time-critical actions needed, such as reading a value from an analog device and storing it in a temporary location. The interrupt routine must also notify the task responsible for the device. Notification may take many forms, ranging from incrementing a counter to "awakening" the task by storing **WAKE** in the task's **STATUS** cell. **WAKE** is a constant containing the code that will cause the task to become active the next time it is polled in the round-robin; on most systems this will be in less than 1 millisecond. Many interrupt handlers do nothing else.

Any processing which is not time-critical can be done by a task running a routine written in high-level polyFORTH. In effect, the time-critical aspect of servicing an interrupt is *decoupled* from the more logically complex aspects of dealing with the consequences of the event. Thus, it is *guaranteed* that interrupts will be serviced promptly, without having to wait for task scheduling, and yet as a programmer you have the convenience of using high-level Forth executed by the responding task for the main logic of the application.

The process of activating a task is the converse of the process of de-activating it:

1. The stack pointer is set to the value in the task's user area.

2. The task's values for **R** and **I** are popped off the task's parameter stack and set.

The task will then execute the next Forth word (pointed to by **I**). This is the text interpreter if the task was awakened by a terminal message.

REFERENCES

Multitasking, Section 4.0
polyFORTH Re-entrancy, Section 4.1
User Variables, Section 4.6

1.3 THE polyFORTH ASSEMBLER

All polyFORTH systems contain an assembler for the CPU on which the system runs. Although it offers most of the same capabilities of other assemblers, its integration into the polyFORTH environment means it will not be fully compatible with assemblers supplied by the computer's manufacturer.

A polyFORTH assembler produces *exactly the same code* as a conventional assembler (which means it runs at full machine speed), but it does it somewhat differently. The differences are in *notation* and *procedure*, and are described below. The differences occur for two reasons:

1. To improve transportability of polyFORTH applications between processors by making assembler notation as similar as possible without impairing the programmer's ability to access and control the processor fully, and

2. To yield a compact assembler which can be resident at all times to facilitate interactive programming and debugging.

1.3.1 Notational Differences

1.3.1.1 INSTRUCTION MNEMONICS

Most mnemonics specifying instructions are the same as the manufacturer's. Occasionally there are differences where the manufacturer uses a prefix or suffix on the mnemonic to describe something we specify as a parameter (*e.g.*, polyFORTH uses **`MOV B`** whereas DEC uses **`MOVB`** as a separate op-code) or to differentiate instructions which are really different (*e.g.*, Intel uses **`MOV`** for both memory/register operations and segment register operations, whereas polyFORTH has different instruction names when segment registers are involved because the internal instruction format is quite different).

1.3.1.2 ADDRESSING MODES

In all computing, there are only a few specific addressing modes (register direct, register relative, memory indirect, etc.). Notation specifying these has been standardized across all polyFORTHs, to make it easier for programmers working with several different CPUs. Naturally, this means the notation differs from the manufacturer's notation; however all modes supported by the processor are implemented in the polyFORTH assembler.

1.3.1.3 INSTRUCTION FORMAT

Most assemblers encourage a "four-column" format, with one instruction per line, allowing space for labels, op-codes, addressing operands, and remarks. In polyFORTH, the op-code itself is a Forth command which assembles the instruction according to operands passed on the stack giving the addressing information. This leads to a format in which the addressing mode specifiers precede the op-code.

1.3.1.4 LABELS, BRANCHES, AND STRUCTURES

polyFORTH assemblers support structured programming in the same way that high-level Forth does. Arbitrary branching to labeled locations is discouraged; on the other hand, structure such as **`BEGIN ... UNTIL`** and **`IF ... ELSE ... THEN`** are available in the assembler (implemented as macros that assemble appropriate conditional and unconditional branches).

1.3.2 Procedural Differences

1.3.2.1 RESIDENT ASSEMBLER

The polyFORTH assembler is resident at all times. This means that a programmer can assemble code at any time, either from a source block or by typing it in directly from the terminal. Regardless of where the code comes from, the assembled version will be the same.

1.3.2.2 IMMEDIATELY EXECUTABLE CODE

In conventional programming, assemblers leave the code in a file, which must be integrated with code in files from high-level language compilers (if any) by a linker before the resultant program can be loaded into memory for testing. The resident polyFORTH assembler assembles the code directly into memory in executable form, thus avoiding this whole cumbersome procedure.

1.3.2.3 RELATIONSHIP TO OTHER ROUTINES

The polyFORTH assembler is used to write short, named routines that function just like routines written in high-level Forth; that is, when the name of the routine is invoked, it will be executed. Like other Forth routines, code routines normally expect their arguments on the stack and leave their results there. Within a code definition, one may refer to defined constants (to get a value), variables (to get an address) or other defined data types. Code routines may be called from high-level definitions just as other Forth words are, but cannot themselves call high-level definitions.

1.3.2.4 REGISTER USAGE

polyFORTH (like other Forths) runs on a "virtual computer." For optimum performance, some of its "registers" are kept in hardware registers, which are permanently assigned. The *CPU Supplement* for each polyFORTH system documents the register assignments for that CPU. On each CPU some registers are always designated as "scratch" (meaning they can always be used within a code routine without saving or restoring); those containing polyFORTH pointers must be saved and restored, if needed. polyFORTH system registers are given names which make references to them in code easy and readable. Since most polyFORTH code routines can do what they need using the designated scratch registers, there is less need to save and restore registers than in conventional programming.

REFERENCES

Principles of polyFORTH Assemblers, Section 6.4
The *CPU Supplement* for Your polyFORTH System

1.4 SYSTEM CONFIGURATION AND ELECTIVES

When you first boot a polyFORTH system, it clears the screen and displays a message identifying the product version and date, then greets the user by saying, "**hi**." At this time, the *nucleus* has been loaded. This pre-compiled nucleus contains:

1. Most Forth primitives (stack manipulation, single-precision arithmetic, input and output number conversion, most string operations, etc.).

2. Disk and terminal drivers.

3. The multitasker, although at this time only one task (**OPERATOR**) is defined.

4. The address and text interpreters.

5. The Forth compiler and assembler.

The capabilities included in the nucleus are the minimum set required for meaningful programming. Nonetheless, you can do quite a great deal.

Your response to polyFORTH's greeting is to type **HI**. This command loads Block 9, a "load block" which, in turn, loads some additional capabilities which we refer to as *system electives*. These are routines which you are more likely to modify in the course of your work, or to selectively include or omit. They also include the configuration of the system, specifying such things as the additional tasks defined, type of system printer (serial or parallel), and even the number of disk buffers.

We strongly recommend that you read through Block 9 on your system, and make note of the various capabilities being loaded and specified.

1.4.1 Task Definition

Specific details of task definition and control are discussed in Section 4.0 of this manual and the *CPU Supplement* for your system. This section covers the organization of your system with respect to allocation of task user areas.

All tasks should be defined as part of the **9 LOAD** process, so that their definitions will reside in the shared dictionary of the system. This is important because once a task has been linked into the round-robin loop it is difficult to remove without breaking the chain, which would halt the system. Although the tasks must be defined and initialized at this time, the definitions that they will be asked to execute need not be defined until later.

There are three types of tasks: printer tasks, serial tasks supporting terminals and other serial devices, and background tasks performing application-related functions. Printers are controlled by **TERMINAL** tasks, as are the serial lines. Printer tasks differ, however, in that they usually have only a limited provision for keyboard input.

TERMINAL tasks further differ from one another in that CRTs and printers have special control codes that control such functions as "clear screen" ("form feed" on printers), carriage return or new-line, tabbing, highlighting, etc. polyFORTH provides for such differences by *vectoring* the commands that perform these functions. Specific routines are provided to specify these functions for many of the most popular printers and terminals; these are called "personality blocks," because they reflect the individual characteristics of the devices in question.

The terminal "personalities" supplied with each polyFORTH system are named in Block 10, and the "personalities" for printers are listed in Block 10's shadow block. Provision was made in the first 60 blocks of the system for two personality blocks, for a primary terminal and printer. If the pair configured in the system as shipped is inappropriate, you should replace them with an appropriate pair from the set provided. If your terminal and/or printer aren't included, you can develop your own by selecting a given personality block for a terminal or printer that is similar to yours and modifying it according to the documentation for your device.

The personality block for a printer should be loaded immediately before the block that defines the printer task. The terminal personality blocks are designed to be loaded by the user at the terminal in question. That is, *after* the terminal task has been **PROMPT**ed, the user at that terminal would type (for example):

```
ANSI
```

to load the ANSI-standard set of terminal control functions. Alternatively, if one or more of the additional terminals configured by Block 9 is *always* of the same type, the personality can be pre-specified by loading its block just prior to the block that defines the terminals. If you are doing this, you need to modify the block by removing the **EMPTY** at the top of the block, and placing the command **EXIT** before the phrases at the end of the block that stores the addresses of the functions in the vectored words **'TAB**, etc.

```
 0 ( Electives)   DECIMAL
 1 EDITOR
 2 ( Aids)  18 19 THRU   ( Buffers)  44 LOAD   60 HELPS
 3 ( 32-BIT)  21 25 THRU   ( Traps)  29 LOAD
 4 ( Date/time)  36 37 THRU  40 4* THRU  55 56 THRU  DATE TIME
 5 ( Extensions)  27 28 THRU
 6 ( Disk Support)  43 46 THRU  57 5* THRU
 7 ( Utilities)  10 LOAD
 8 ( Editor Support)  51 53 THRU   ( F-83)  48 LOAD
 9 ( Fast Display)  54 LOAD
10 : SYSTEM   [ 60 ] +U DATED  ."  Time " TIME ;
11 ( Tasks)  30 31 THRU   ( Serial)  282 LOAD
12 ( Printer) 35 LOAD 34 LOAD  ( VDTs) 168 LOAD 291 LOAD
13 : HELP   SYSTEM ;
14 GILD  ' ?CREATE 'CREATE !  JOE PROMPT
15 EXIT  polyFORTH II ISD-4 pF86/IBM-B  Disk 1  10 July 1986
```

Fig. 1.13

Typical system electives load block, showing order of defining tasks.

Fig. 1.13 shows a typical load block of system electives. To summarize the load sequence for terminals and printers, Block 9 should be:

1. Task definition words (normally Blocks 30-31).
2. Hardware drivers for serial port, as needed.
3. Printer personality block.
4. Printer task definition.
5. Terminal personality block (optional).
6. Terminal task definition(s).
7. The command **GILD**, which defines the "top" of the shared dictionary.
8. **PROMPT** commands for any terminal tasks that have been defined.

The **PROMPT** commands, if any, must come after the **GILD** command in order to attach the terminal tasks' dictionaries to the full shared dictionary of the system.

REFERENCES

Background Tasks, Sections 4.3, 4.4, 4.5
Defining Terminal Tasks, Section 4.8 and the *CPU Supplement*
GILD, Section 3.4.4
Multitasking, Section 4.0
Printer Tasks, Section 4.11

1.4.2 System Feature Selection

There are many options that are selected during the loading of the system electives. These include the creation of tasks (discussed in Section 1.4.1), the selective loading of software facilities as needed by your applications, and the installation of features that are normally resident but not always active. The latter two categories are discussed in this section.

Block 9 of your system as shipped loads all the features documented in this manual, except for utilities which are normally handled as overlays. Many routines are included for each system that aren't part of standard polyFORTH, and are appropriate for some applications but not others; these are not loaded in the product as shipped, but you may wish to include them. You may wish to omit routines that you don't expect to use, in order to allow more dictionary space for your application. You may very likely want to add routines developed to support your application (ranging from custom hardware drivers to special math functions) in order to have them always available in the shared dictionary.

The following items (including the section in this manual where a fuller discussion of the feature may be found) are common configuration issues:

Feature	Section	Discussion
LOCATE	1.5.1	This feature compiles the source block from which each definition of any type was compiled, so that you may interactively display the block. The cost is two bytes per definition. If you are short of dictionary space you may wish to do without this convenience. It is installed in the **EDITOR** load block (12) by the phrase: `' <CREATE> 'CREATE !` Omit this phrase (or disable it by putting it in parentheses) if you don't want to use this feature.
32-bit Math	2.2	Many applications don't need all of these operators (especially control applications and those using the 8087 for computation).
Calendars	3.5	There are two calendars: the month-day form, used when most dates are current, and the m/d/y calendar which is useful when dates may be in any year (*e.g.,* birthdates). Select the calendar of your choice by loading Block 39 or 40, plus Block 41. The internal forms are the same for both.
?CREATE	2.6.1	This version of **CREATE** checks for name conflicts, and issues a warning message if a word being defined may conflict with a previous name. Such conflicts may be resolved by using **~** ("tilde"). It is enabled by the phrase, `' ?CREATE 'CREATE !` in Block 9. If you don't wish to get these messages, omit the phrase. If you think you have an inadvertent name conflict in your system electives, you may wish to move this phrase earlier in the block. It may be used any time after Block 18 is loaded. If you are loading application features, you may choose to load them before or after this phrase, depending on where you wish the check to be performed.

As an application progresses, you will probably include more and more application functions in the electives as they are fully tested, leaving routines that are still under development to be loaded in a terminal partition as overlays. Your application should have a main load block, which loads features included with polyFORTH that are special to your application (such as graphics), plus your application drivers and other routines. This block may be loaded at the end of Block 9 (*before* the **GILD** command) in order to include these routines in the shared dictionary.

1.5 DOCUMENTATION AND SOURCE MANAGEMENT FACILITIES

In polyFORTH, as in all other languages, the primary responsibility for producing readable code lies with the programmer. polyFORTH does, however, support the programmer's efforts to produce easily managed code by providing a number of aids to internal documentation. In addition to these, we also recommend each Forth programming group adopt editorial and naming standards and conventions which all share. Although "readability" is fundamentally an aesthetic and rather personal value, adopting a set of standards that all members of a group adhere to will aid significantly in the ability of the group to share code and support one another.

1.5.1 Internal Documentation

polyFORTH comes with a comprehensive listing utility, capable of printing indexes, program text, double-sided program text, and program text with associated shadow blocks. These listing programs are most useful when you follow certain simple conventions in writing your application program (see the section on disk and block layout referenced below).

For each block of program source, there is a corresponding block, called its "shadow," which contains comments and descriptions of the words defined in the source. Shadow blocks are a documentation facility intended to help you document applications and use polyFORTH. Shadow blocks reside on disk at a fixed offset from the source code blocks which are being documented. The offset is a constant called **SHADOWS**. **SHADOWS** is used by the word **Q** (for "Query"), which alternately displays the block of source code and the shadow block containing comments for the source code, adjusting the **EDITOR**'s block number **SCR** accordingly.

Shadow blocks are edited the same way as source code, and form a powerful commenting facility. polyFORTH source code is often compact, so corresponding shadow block space is at a premium. Shadow blocks are therefore intended to document intimate code details which are necessary to use or to rewrite important words. If space is a problem, documentation of usage should take precedence over theory of operation.

Comments embedded in polyFORTH source are enclosed in parentheses. For example:

```
( This is a comment)
```

The word **(** must have a space after it, so that it can be recognized and executed. polyFORTH comments are most often used to give a picture of a word's stack arguments and results; for example, a high-level definition of the polyFORTH word **=** is:

```
: = ( n n - t)    - NOT ;
```

The dash in the comment separates a word's arguments from its results. Certain letters have specific meanings by convention. The most common are:

Word	Description
n	A single-precision signed integer.
u	A single-precision unsigned integer.
t	A single-precision Boolean value (zero is false and non-zero is true).
a	A single-precision address.
d	A double-precision signed integer. Thus, in the example above, the word **=** expects two single-precision integers and returns a truth flag.

Words which have separate interpretive and run-time behaviors should have comments for both sections:

```
: CONSTANT ( n)   CREATE ,  DOES>
   ( a - n)   @ ;
```

Electives and resident applications are generally loaded immediately after the system boot, by the command **HI**. If the "programmer aid" block is loaded and the appropriate version of **CREATE** (called **<CREATE>**) is installed, the source text of any word already compiled from disk may be displayed by typing:

```
LOCATE word
```

If the word is not in the dictionary, the response will be:

```
LOCATE word     word ?
```

If a word was defined from the keyboard or if the word exists, but was defined before the electives were loaded, the response will be:

```
LOCATE word     can't
```

If the word exists and was defined during or after the electives load, the block containing the definition of the word will be displayed. **SCR** is set by **LOCATE**, so that the block's shadow block can be seen by typing **Q** followed by a carriage return.

REFERENCES

Behavior of **=**, Section 2.2.2
Disk and Block Layout and Design, Section 5.2.5
Listing Utility, Section 5.2
Stack Notation Conventions, Section 2.1
Vectored **CREATE** (**<CREATE>**), Section 3.1

1.5.2 Source Management

The polyFORTH system source is usually followed on disk by additional blocks for applications. On most systems, particularly those with hard disks available, there is a region of 600 blocks or more for source, followed by a region of the same size for the shadow blocks documenting this source. Systems with dual floppy disks usually keep the source on the disk in Drive 0 and the shadow blocks on the disk in Drive 1. Hard disks are usually organized into "volumes" of a convenient size so that you may easily keep a backup copy of your source and shadow blocks on the hard disk, as well as archival copies on floppy disks or other removable media.

polyFORTH includes several utilities to assist in managing the blocks of source for an application.

The **DISKING** utility provides simple commands for copying ranges of blocks from one disk to another or one place to another on a hard disk. polyFORTH organizes hard disks into logical volumes of a convenient size for most applications. Commands are included to copy entire volumes as well as shorter ranges of blocks. A command is also provided to copy a range of source blocks along with its associated range of shadow blocks.

The **AUDIT** utility includes facilities for comparing ranges of blocks, displaying non-matching blocks on the screen or printer with the differences highlighted. This facility is especially useful when several programmers are working on the same application on different computers.

Several useful source management aids are also resident in the system. The command:

```
n QX
```

(for **Q**uick Inde**X**) displays the first part of Line 0 of a 60-block region of disk starting with block *n*. The related commands **NX** and **BX** display the next and previous 60-block regions, respectively. This is useful when you are looking for certain blocks, or for an appropriate block to put a new feature.

The editor contains the command **S**, which **S**earches forward over a range of blocks for a specified text string. This is useful for finding words that cannot be **LOCATE**d, or finding instances of a word or phrase.

REFERENCES

The **AUDIT** Utility, Section 5.5
The **DISKING** Utility, Section 5.3
S Command, Section 5.1.7

2.0 BASIC FORTH VOCABULARY

This section defines the major elements of the polyFORTH implementation of the Forth language. These words are grouped into categories and documented in the following sections.

2.1 STACK OPERATIONS

Stack operators work on data that are present on the parameter and return stacks. The words defined in this section use the stack as the major source and destination for their operands. Many other Forth words also result in modification of the stack. Other commands are described in the sections of this manual that deal with their primary functions. Besides the stack operators discussed in this manual, stack manipulation words that relate to assembly language are covered in Section 6.0 and your *CPU Supplement*.

In this section, several notational conventions have been adopted for clarity. The item on the top of the stack is referred to as "S"; the item immediately below the top of the stack is referred to as "S+1," etc. A double-length item on the top of the stack is referred to as "S+0,1"; a double-length item below it would be "S+2,3," etc. This notation reflects usage on all polyFORTH systems in which stacks grow toward low memory.

Operations that use the stack usually require that a certain number of items be present on the stack and then leave another number of items on the stack as results. Most operations remove their operands, leaving only the results. To help see the effect of the operation on the number and type of items on the stack, each word in this section has a notation under the stack column. This same stack notation is used in your system listing to document the stack arguments of system words.

The specific notation of the stack items follows these conventions:

Word	Description
a	A cell-wide byte address.
b	A byte, stored as the least significant 8 bits of a stack entry. The remaining bits of the stack entry are zero in results or ignored in arguments.
c	An ASCII character stored as a byte (see above) with the parity bit reset to zero.
n	A signed single-precision 2's complement number. On 16-bit machines the range is from -32768 through +32767. On 32-bit machines, the range is from -2,147,483,648 though +2,147, 483,647. (Note that Forth arithmetic rarely checks for integer overflow.)
u	A single-precision unsigned number, with a range from 0 to 65536 on 16-bit machines, or from 0 through 4,294,967,295 on 32-bit machines.
d	A double precision, signed, 2's complement integer, with a range from -2,147,483,648 to 2,147,483,647; stored as two stack entries (in most systems the least significant cell is the cell below the most significant cell). On 32-bit machines, the double-precision range is from -9,223,372,036,854, 775,808 through 9,223,372,036,854,775,807.
ud	A double-precision, unsigned integer, with a range from 0 to 4,294,967,296. On 32-bit machines, the range is from 0 through 18,446,744,073,709, 551,615.

t	A single-precision Boolean truth flag (zero means false, non-zero means true).

If you type several numbers on a line, the right-most will end up on top of the stack. As a result, we show multiple stack arguments with the top element to the right.

These are examples of special cases which will be explained when used:

Word	Description
l c	Screen position, in lines and columns, respectively.
s d n	Source, destination and count.
y x	A 2-vector for graphics, etc.
f l	First and last limits, inclusive.

Word	Description
l h	Low and high limits (exclusive high limit).
r	Register (for assembler words).

Where there are several arguments of the same type, and clarity demands that they be distinguished, a ' (pronounced "prime") will be used. In extraordinary instances, numeric suffixes may be used.

Please remember that these locations are relative to the top of the stack and do not affect any stack items below the lowest stack item referenced by the operation. For example, the notation (n n - n) means that the operation uses the top two stack items and leaves a one-item result. Therefore, if the stack initially contained three items, execution would result in a stack of two items, with the bottom item unchanged and the top item derived as a result of the operation.

2.1.1 Parameter Stack Manipulation Operations

This category of stack operations contains words which manipulate the contents of the parameter stack without performing arithmetic, logical, or memory reference operations.

Word	Stack	Function
OVER	(n n' - n n' n)	Duplicates S+1 on the top of the stack (S).
SWAP	(n n' - n' n)	Swaps the top two items on the stack. S is placed in S+1 and S+1 is placed in S.
DUP	(n - n n)	Duplicates the top entry on the stack.
DROP	(n -)	Removes the top entry from the stack.
'S	(- a)	Fetches the address of the top of the stack, placing this address on the top. Normal usage is to add to this value an offset such that the resulting address is that many items down the stack. For example, to push the address of the third item below onto the top of the stack, the following code would be used:

```
'S 6 +
```

Word	Stack	Function
'S (cont.)		('S 12 + on 32-bit processors.) Note that although the actual definition is machine code, **OVER** is equivalent to: **'S 2+ @** (**'S 4+ @** on 32-bit systems.) Pronounced "tick-s."
2DROP	(d -)	Removes the top double-length item (2 cells) from the stack.
2SWAP	(d d' - d' d)	Swaps the top two double-length items.
2DUP	(d - d d)	Duplicates the top double-length item.
2OVER	(d d' - d d' d)	Pushes the double-length item (S+2,3) onto the top of the stack (S+0,1).
ROT	(n1 n2 n3 - n2 n3 n1)	Rotates the top three items on the stack.
?DUP	(n - n , n n)	Conditionally duplicates the top item on the stack if its value is non-zero. Equivalent to the following: **DUP IF DUP THEN**
DEPTH	(- n)	Tells how many items are on the stack. Note that **DEPTH** will return 2 for each double-precision integer on the stack.

The following words are included for compatibility with the FORTH-83 standard. Good Forth programming rarely accesses a parameter stack to a depth of more than 4 cells. Therefore, the following words are available in the FORTH-83 compatibility block, rather than being coded in the nucleus.

Word	Stack	Function
PICK	(n - n')	Copies the nth stack entry onto the stack. S is entry number zero, S+2 is entry number two. The phrase **1 PICK** is equivalent to the word **OVER**.

Word	Stack	Function
ROLL	(n)	Moves the nth stack entry to the top of the stack, moving down all the stack entries in between. **ROLL** numbers the stack in the same way as **PICK**.

REFERENCES

FORTH-83 Standard Compatibility, Section 2.9

2.1.2 Memory Stack Operations

This category of operations allows you to reference memory by using addresses that are on the stack.

Word	Stack	Function
C@	(a - b)	Replaces S with the byte addressed by S. The byte fetched is stored in the low-order byte of S with the remaining bits cleared to zero. Pronounced "c-fetch."
C!	(b a -)	Stores the low-order byte of S+1 at the address in S, removing both from the stack. Pronounced "c-store."

C+!	(b a -)	Adds the low-order byte of S+1 to the byte addressed by S, removing both from the stack. Pronounced "c-plus store."
@	(a - n)	Replaces S with the item addressed by S. Pronounced "fetch."
!	(n a -)	Stores S+1 at the item addressed by S, removing both from the stack. Pronounced "store."
+!	(n a -)	Adds S+1 to the contents of the item addressed by S and stores the result in the item addressed by S, removing both from the stack. Pronounced "plus-store."
2@	(a - d)	Pushes the double-length item that is addressed by S onto the top of the stack (S+0,1). Pronounced "two-fetch."

Word	**Stack**	**Function**
2!	(d a -)	Stores S+1,2 into the double-length item that is addressed by S, removing three cells from the stack. Pronounced "two-store."

The following words are in 32-bit systems only:

Word	**Stack**	**Function**
H@	(a - n)	Replaces S with the 16-bit (half-word) content of S, sign-extended. Pronounced "h-fetch."
H!	(n a -)	Stores the low-order half-word of S+1 into the 16-bit location addressed by S. Pronounced "h-store."
U@	(a - u)	Unsigned version of **H@**. Pronounced "u-fetch."
H+!	(n a -)	Adds the low-order signed half-word of S+1 to the 16-bit contents of S. Pronounced "h-plus-store."

2.1.3 Return Stack Manipulation Operations

The "return" stack is so named because it is used by the address interpreter to keep track of where Forth words will return when they have finished execution. When a high-level Forth word invokes a lower-level Forth word, the address of the next cell of the high-level word is pushed onto the return stack by the address interpreter.

The return stack is a convenient place to keep frequently used values (because of the words **I**, **I'** and **J**), but it should be cleared before the word reaches an **EXIT** (**EXIT** is compiled by **;**) or else the address interpreter will return to the "address" on the return stack. This behavior is occasionally useful. For example:

```
: VECTOR   >R ;
```

will act like the word **EXECUTE**, but will only execute **:** definitions. **VECTOR** works by pushing a word's parameter field address onto the return stack. Therefore, when **EXIT** pops the return stack into the address interpretive pointer **I** the address interpreter will begin to execute the word whose address was on the stack for **VECTOR**.

This section documents those operations which involve both the return stack and the parameter stack.

Word	**Stack**	**Function**
>R	(n -)	Removes the item on the top of the parameter stack and puts it onto the return stack. Pronounced "to-r."

R>	(- n)	Removes the item on the top of the return stack and puts it onto the parameter stack. Pronounced "r-from."
2>R	(n n -)	Pops the top two cells from the parameter stack and pushes them onto the return stack. This word is the run-time code for **DO**. Pronounced "two-to-r."
2R>	(- n n)	Pops the top two cells from the return stack and pushes them onto the parameter stack. **2R>** is the inverse of **2>R**. Pronounced "two-r-from."
I	(- n)	Duplicates the item on top of the return stack on the parameter stack. Normally used to obtain a copy of the index of a **DO ... LOOP** structure.
R@	(- n)	Identical to **I**; defined for compatibility with FORTH-83 standard. For maximum compatibility, **I** should be used only within a **DO** structure to obtain its index, and **R@** for general return stack access. Pronounced "r-fetch."
I'	(- n)	Pushes onto the parameter stack a copy of the second return stack item. Used to access either the limit inside a loop or **I** from a definition invoked inside a loop. Pronounced "i-prime."
J	(- n)	Pushes onto the parameter stack a copy of the third return stack item. Used to access the index of the next outer loop from an inner loop.

REFERENCES

Counting **LOOP**s (**DO**), Section 2.4.4
EXECUTE, Section 2.4.8.1

2.1.4 Conveniences

Word	Stack	Function
DUMP	(a n -)	Types out a region of cells in memory, given the starting address and length: `addr count` **DUMP** Output is formatted with the address on the left and up to eight values on a line; the output conversion radix is the current value of **BASE**. Two cells are removed from the stack.
.S		Displays the contents of the parameter stack using the current base. Stack contents remain unchanged.
?	(a -)	Fetches from the address in S, and displays the result according to the current conversion radix. Equivalent to the phrase: `@ .`
'	(- a)	Searches the dictionary for the word that follows tick in the input stream; leaves the address of the word's parameter field on the stack. Aborts if the word cannot be found. Pronounced "tick."

2.2 ARITHMETIC AND LOGICAL OPERATIONS

polyFORTH offers a comprehensive set of commands for performing arithmetic and logical functions. polyFORTH's arithmetic is optimized for integer arithmetic, since most processors lack hardware floating-point capability and software floating-point is too slow for most real-time applications. polyFORTH supplies words to perform fast, precise scaled-integer and fixed-point fraction computations.

Programmers who are new to Forth should review *Starting FORTH*, Chapters 2, 3, 5, and 7, which provide a comprehensive discussion of Forth's handling of numbers and arithmetic.

2.2.1 Arithmetic and Logical Operators

The basic principles of Forth arithmetic are covered in *Starting FORTH*, Chapter 2. More advanced concepts are discussed in Chapter 5.

In order to achieve maximum performance, each version of polyFORTH implements most arithmetic primitives to use the internal behavior of hardware multiply/divide instructions on that particular processor. Therefore, to find out at the bit level what these primitives do, you should consult the manufacturer's hardware description.

The following general guidelines may help you use these operators:

1. The order of arguments to order-dependent operators (*e.g.*, **–** and **/**) is such that if the operator were moved to an infix position it would algebraically describe the result. Some examples:

Forth	**Algebraic**
a b -	a - b
a b /	a / b
a b c */	a * b / c

2. All arithmetic words containing **MOD** (**MOD**, **/MOD**, ***/MOD**, etc.) are unsigned; others are normally signed. The exception to this rule is that **M*/** on most systems require a positive divisor.

These operators perform arithmetic and logical functions on numbers that are on the stack. In general, the operands are removed (popped) from the stack and the results are left on the stack.

Single-Precision Operations

Word	**Stack**	**Function**
+	(n n - n)	Adds S (popped) to S+1; leaves the result as S.
–	(n n' - n)	Subtracts S (popped) from S+1; leaves the result as S.
1+	(n - n)	Adds one to the value in S.
1–	(n - n)	Decrements the value in S by one.
2+	(n - n)	Adds two to the value in S.*

* 32-bit systems also include **4+**, **4-**, and **4***. These words are most valuable for converting cells to bytes, and incrementing and decrementing by cell widths.

Word	Stack	Function
2-	(n - n)	Subtracts two from the value in S.*
2*	(n - n)	Multiplies S by two (arithmetic left shift).*
2/	(n - n)	Divides S by two (arithmetic right shift).
*****	(n n - n)	Multiplies S+1 by S (popped); leaves the result as S.
/	(n n - n)	Divides S+1 by S (popped); leaves the quotient as S.
MOD	(n n - n)	Divides S+1 by S (popped); leaves the remainder as S. Operands are unsigned.
/MOD	(n n - r q)	Divides S+1 by S; leaves the quotient as S and the remainder as S+1. Operands are unsigned.
***/**	(n n n - n)	Multiplies S+2 by S+1; divides the result by S; leaves the quotient as S (uses a double-precision intermediate result).
***/MOD**	(n n n - r q)	Multiplies S+2 by S+1; divides the result by S; leaves the quotient as S and the remainder as S+1 (gives a double-precision intermediate result). Operands are unsigned.

Double-Precision Operations

Word	Stack	Function
D+	(d d - d)	Adds the top two double-precision stack values.
D-	(d d - d)	Subtracts the top double-precision stack value (S+0,1) from the next double-precision value (S+2,3); leaves a double-precision result as S+0,1.

Mixed-Precision Operations

These arguments have been determined by experience to be the most generally useful.

Word	Stack	Function
M+	(d n - d)	Adds a double-precision value in S+1,2 to the single-precision value in S; leaves a double-precision result.
M-	(d n - d)	Subtracts a single-precision value in S from a double-precision value in S+1,2.
M*	(n n - d)	Multiplies two single-precision values (S and S+1) to form a double-precision value (as S+0,1). An alternate "quick and dirty" definition of **M*** may use **M*/** when stack arguments of the form (d n - d) are needed.
M/	(d n - n)	Divides a double-precision value in S+1,2 by the single-precision value in S; leaves a single-precision result as S; does not perform an overflow check.
U/MOD	(ud u - r q)	Divides S+1,2 by S, leaving a remainder as S+1 and a quotient as S. This operation is called **U/MOD** because it assumes that the arguments are unsigned, and it produces unsigned results.
M*/	(d n' n - d)	Multiplies S+2,3 (d) by S+1 (n'), giving a triple-precision result, and then divides the triple-precision result by S (n), leaving a double-precision result. S may be unsigned; see the stack arguments in your system listing.

T*	(d n - t)	Multiplies S+1,2 by S, yielding a triple-precision result (48 bits on 16-bit machines and 96 bits on 32-bit machines) as S+0,1,2. Used in **M*/**.
T/	(t n - d)	Divides a triple-precision number in S+1,2,3 by S, yielding a double-precision result. S may be unsigned. See the stack arguments in your system listing. Used in **M*/**.

2.2.2 Logical and Relational Operations

As in the case of arithmetic operations, polyFORTH's implementation of logical and relational operations optimizes speed and simplicity. This does imply some limitation on generality in 16-bit systems, although this limitation rarely is an issue in real-time applications.

In order to fully understand the issues, we can represent the entire set of 16-bit integers in three ways, as shown in Fig. 2.1.

A relational which treats a given 16-bit integer as a point on the full signed number line (a) is needed for true arithmetic or algebraic numbers in which the application has carefully determined that there will never be over-flow or underflow.

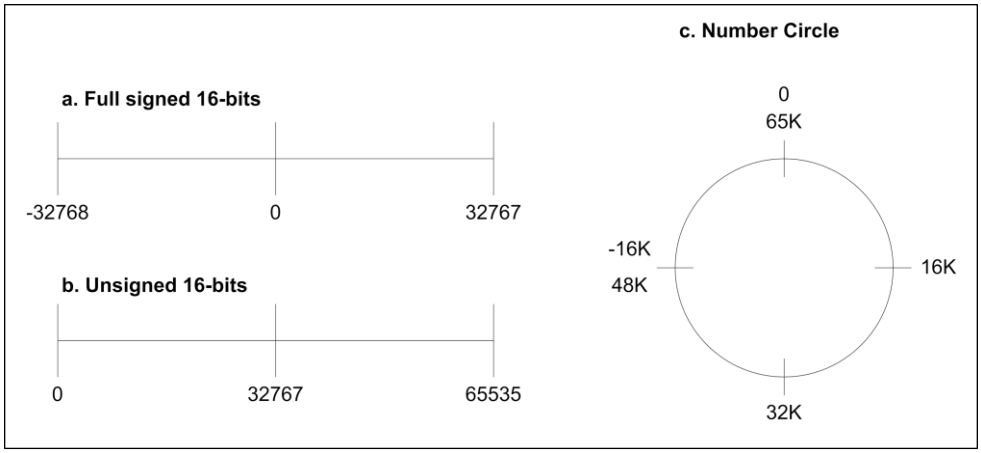


Fig. 2.1

Three ways of representing 16-bit binary numbers

For example, this type of relational is needed to test whether -20,000 is less than +20,000.

A relational which treats all values as unsigned (b) is also needed, primarily to test locations of given addresses. The number line, or rather "number circle" shown in (c) probably needs more explanation than the other two. Relational operators which treat numbers in this way have the advantage of being able to act like signed tests around zero and like unsigned tests around 32K. Thus the relation between two numbers is totally independent of their absolute position on the number circle.

It may seem that the third case is the rarest and least useful. In fact, though, it turns out that it handles the vast majority of signed comparisons in real applications, and better still, it is much faster in execution and easier to implement than a relational that assumes number line (a).

Consider three points on the number circle, as shown in Fig. 2.2. It doesn't matter where these points lie in relation to true zero on the circle. Since numbers increase in a clockwise direction, point A is considered to be greater than point X. Point B is considered to be less than point X. Notice that the maximum range of comparisons in the circle is 32K. Further away than that, a value will appear to lie in the opposite semicircle and will produce the opposite result.

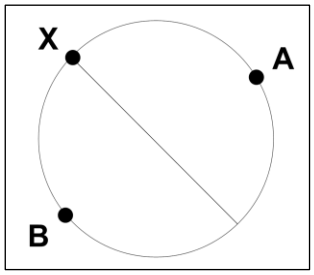


Fig. 2.2

A circular representation of the range of 16-bit numbers.

Thus a relational which uses this number circle is limited to a 32K range for any single test. Since most comparisons use an extremely small fraction of the total 65K range of the circle, this limitation is generally safe.

The number circle relational is easy to implement because it can be defined in terms of subtract, *e.g.*,

```
: <   - 0< ;
```

When A is subtracted from X, the arc of the difference is greater than 180 degrees. That is to say, the result of subtraction will appear to be negative in sign. When B is subtracted from X, the arc of the difference is less than 180 degrees; this difference will appear positive. In the first case, the 0< test produces "true," in the second case "false."

16 bit versions of polyFORTH ISD-4 use the fully signed model (option "a" in Fig. 2.1) to implement most relationals, as well as **MAX** and **MIN**. 32-bit versions of polyFORTH use the circular model. The operator **U<** is provided for unsigned comparisons, particularly for memory addresses that can extend over the full 0-65K range.

Single-Precision Logical Operations

Word	Stack	Function
NEGATE	(n - n)	Negates the top value (S is set to -S). The phrase **NEGATE 1-** is equivalent to logical negation (1's complement).
MAX	(n n - n)	Compares S+1 and S (popped); leaves the larger as S.
MIN	(n n - n)	Compares S+1 and S (popped); leaves the smaller as S.
ABS	(n - n)	Replaces the top stack value with its absolute value (S is set to **ABS S**).
AND	(n n - n)	Places the logical **AND** of S+1 and S as S.
OR	(n n - n)	Places the logical **OR** of S+1 and S as S.
XOR	(n n - n)	Places the exclusive **OR** of S+1 and S as S. The phrase **-1 XOR -1** is equivalent to logical negation.

Double-Precision Logical Operations

Word	Stack	Function
DNEGATE	(d - d)	Changes the sign of a double-precision stack value.
DABS	(d - d)	Takes the absolute value of a double-precision stack value.

DMIN	(d d - d)	Returns the smaller of two double-precision stack values.
DMAX	(d d - d)	Returns the larger of two double-precision stack values.

Comparison and Testing Operations

These operations leave on the stack a number that is based upon a test of the contents of one or more items on the top of the stack. This number may be interpreted as a true/false value; zero equals false and *any* non-zero value equals true. The words below, which perform explicit tests, return -1 for 'true.'* Comparison and testing operations generally precede an **IF**, **WHILE**, or **UNTIL**.

In general, the test always replaces the item(s) tested with the results of the test.

Word	Stack	Function
0=	(n - t)	Tests S for a value of zero. Pronounced "zero-equals."
0<	(n - t)	Tests S for a value less than zero. Pronounced "zero-less-than."
NOT	(t - t)	Equivalent to 0=; tests S for a value of zero. Used for program clarity to reverse the results of a previous test. For example, the following code would test for a value greater than or equal to zero: **0< NOT**
=	(n n - t)	Tests S and S+1 for equality.
<	(n n -)	Tests for S+1 less than S. Pronounced "less-than."
>	(n n - t)	Tests for S+1 greater than S. Pronounced "greater-than."
D0=	(d - t)	Tests the double-precision item in S+0,1 for zero. Pronounced "d-zero-equals."
D<	(d d' - t)	Returns a one if S+0,1 is less than S+4,5, a zero otherwise. Pronounced "d-less-than."

The following are written in high-level and are available when the FORTH-83 compatibility block is loaded:

Word	Stack	Function
0>	(n - t)	Returns a one if S is greater than zero. Pronounced "zero-greater-than."
D=	(d d - t)	Returns a one if the double-precision number in S+0,1 is equal to the double-precision number in S+4,5. Pronounced "d-equals."

You may also use **-** (minus) or **D-** as a "not-equal" test, since it will return a non-zero difference if the two single or double-precision numbers are unequal.

REFERENCES

Conditionals , Section 2.3.5
MAX and **MIN**, Section 2.1.2
Post-testing Loops, Section 2.4.2

* For users of previous versions of polyFORTH that used 1 as the standard 'true' value, compatibility definitions may be obtained from FORTH, Inc.

Pre-testing Loops, Section 2.4.3
String Comparisons, Section 2.3.5
FORTH-83 Standard Compatibility, Section 2.9

2.3 CHARACTER AND STRING OPERATIONS

polyFORTH contains many words that are used to reference single characters (bytes) or character strings. Characters may be grouped together and thought of as a string; this group is then operated on as a single variable. Character strings are supported by the words that are documented in this section.

A standard working area is used to hold most character strings for processing; this area is referred to as **PAD**.

In addition to the words described in this section, several other words are used to reference character data in different environments, *e.g.*, Data Base Support. Such words are described in the appropriate sections of this manual.

REFERENCES

Data Base Support, Section 8.0

2.3.1 The PAD—Scratch Storage for Strings

PAD is an area of storage of indefinite size that is used to hold strings for intermediate processing. Each terminal task contains a **PAD** area. The word **PAD** places the address of the first byte in **PAD** on the top of the stack.

PAD is physically located at a fixed displacement from the current top of the dictionary (**H**). The actual displacement is a system function, but will be at least 34 bytes. The largest displacement is the maximum length used for pictured output number conversion. The number conversion area grows downwards from **PAD** (see Fig. 2.3).

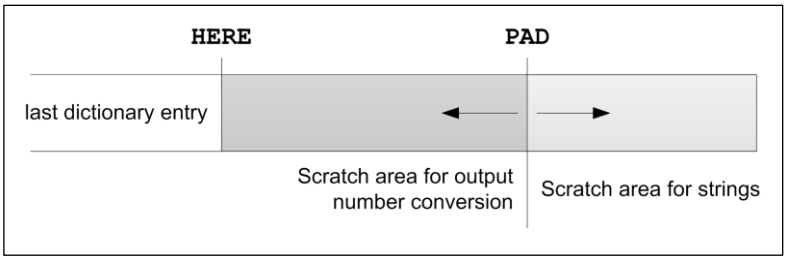


Fig. 2.3

Portion of user's dictionary showing **PAD**.

Because **PAD** is located relative to the dictionary pointer **H**, the location of **PAD** changes whenever **H** changes. Common operations that affect **H** include adding definitions; adding data or data areas using **,**, **C,**, or **ALLOT**; or discarding definitions using **FORGET** or **EMPTY**. This means that information left in **PAD** before one of these operations will not be addressable following the definition (and may in fact be overwritten by a new definition).

Although **PAD** does not have a fixed maximum size, a minimum size of 250 bytes is maintained; an error message ("Dictionary Full") is issued if the dictionary threatens to grow too close to the top of the user area. The exact current size may be computed by the phrase:

```
'S PAD -
```

REFERENCES

, and **C,**, Sections 2.7.6.1, 2.8.2
ALLOT, Section 2.8.1
EMPTY, Section 3.3.4.1
FORGET, Section 3.3.4.2
Pictured Number Conversion, Section 2.5.2

2.3.2 Single-Character Reference Words

The words **C@** and **C!** are used to reference single characters in the same way that **@** and **!** are used to reference cells.

C@ expects an address on the top of the stack. This address is replaced with the contents of the addressed byte. This byte will be placed in Bits 0-7 of the cell on the top of the stack, with the higher order bits set to zero (**C@** does not "sign extend").

C! expects an address on the top of the stack and a character in Bits 0-7 of the cell underneath the byte pointer. The high-order bits of this lower cell are ignored. The character is stored in the addressed byte; the address and character cells are removed from the stack.

For example, the following phrase would fetch the first character in **PAD** to the top of the stack:

```
PAD C@
```

REFERENCES

C@, **C!**, Section 2.1.2

2.3.3 String Defining Words

Short messages of a few characters in length may be added to the dictionary by using their ASCII codes with the defining word **MSG** (pronounced "message"). A word defined by **MSG** will, when invoked, type out its characters on the terminal. For this reason, a word defined by **MSG** may only be executed by a **TERMINAL** task.

The principal use of **MSG** is for sending control information to terminals and other ASCII devices. Some examples are **CR** (which sends a carriage-return, line feed, and sometimes nulls for timing to a terminal); **PAGE** (which sends a "clear screen" or "form feed"); and **DRAW** (an example of an application command which sends graphics commands to a serial graphics terminal).

The basic form of a **MSG** is as follows:

```
HEX MSG NAME nn C, cc C, cc C, ...
```

The first byte of the string (represented by *nn* above) gives its length (number of bytes).* The second byte represents the first letter of the string. Each *cc* above represents the ASCII equivalent of one character of the string.

On processors that will not tolerate odd byte addresses (such as the PDP-11 or 68000) odd-length strings should be filled out with a zero at the end; the count may be even or odd, however, and the exact specified number of characters will be typed.

* On DEC (PDP-11), Intel, and Zilog computers the first byte is in the low-order position of each cell.

REFERENCES

Example of **MSG**, Sections 2.7.6.3, 3.7.3
TERMINAL Tasks, Sections 4.8, 4.9, 4.10
Terminal Output, Section 3.7.3
Use of **,** and **C,** to Compile Values, Section 2.8.2

2.3.4 String Management Operations

polyFORTH contains several words that are used to reference strings, compare strings, and move strings between different locations. In addition to these words, other words are used to input or output character strings.

Most words that operate on a single character string expect the length of that string to be on the top of the stack, and its address beneath. Many words that operate on two separate character strings expect three items on the top of the stack, in the format shown in Fig. 2.4. The exceptions are **-TEXT** and the byte string operators in the Data Base Support system.

Word	Stack	Function
ERASE	(a n -)	Erases a region of memory (sets it to zeros), given its starting address and length: **addr count ERASE** Two cells are removed from the stack.

Word	Stack	Function
BLANK	(a n -)	Sets a region of memory to blanks (hex 20); S+1 and S are an address and length, as in **ERASE**. Two cells are removed from the stack.
FILL	(a n b -)	Fills a region of memory with the least significant byte of S. S+2 and S+1 are an address and length, as in **ERASE**. Three cells are removed from the stack.
MOVE	(s d n -)	Moves the number of bytes in S from a source starting at S+2 to the destination starting at S+1. **MOVE** is always the fastest way to move data, but on cell addressed machines, s and d may be restricted to even addresses. Transfers bytes beginning at low memory and ending at higher memory. Three cells are removed from the stack.

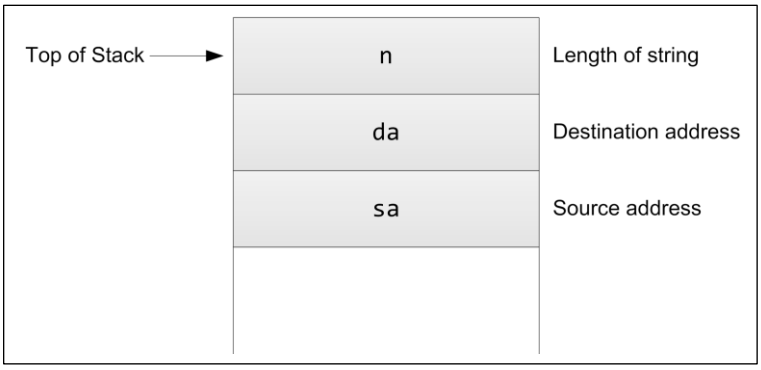


Fig. 2.4

Format of arguments for most two-string operators. One length applies to both strings. The above format is used instead of two separate character counts.

Word	Stack	Function
CMOVE	(s d n -)	Moves the number of bytes in S from a source starting at S+2 to the destination starting at S+1. Arguments and behavior are as **MOVE**, but *s* and *d* may have odd values. Three cells are removed from the stack.

Word	Stack	Function
<CMOVE	(s d n -)	**<CMOVE** has arguments and behavior as **CMOVE,** but starts transferring at high memory and works toward low memory. **<CMOVE** is good for transferring from a data field to an overlapping data field in higher memory. Three cells are removed from the stack. Pronounced "back c-move."

Field names which are character strings contained in files may be used to provide operands for string reference operations.

REFERENCES

-TEXT, Section 2.3.5
Character String I/O, Section 2.3.6
Fields, Data Base Support, Section 8.0

2.3.5 Comparing Character Strings

Character-string comparisons operate on two separate character strings; this allows the two to be compared by use of the ASCII collating sequence. The following words are provided:

Word	Stack	Function
-TEXT	(a n a' - n)	Compares two strings of the same length. The string beginning at the address in S is subtracted from the string beginning at the address in S+2. S+1 is the number of characters to compare. Returns -1 if the "S+2" string is less than the "S" string, 0 if they are equal, and 1 if the S+2 string is greater than the S string. "Greater than" and "less than" are determined on a cell-by-cell basis. Odd-length strings may not be used with **-TEXT** because the strings are compared *cell-by-cell*.

Word	Stack	Function
-MATCH	(a1 n1 a2 n2 - a1 n1 t)	Searches for a match for the string a2 n2 in the string a1 n1 (which is presumed to be longer). If a match is found, returns 0 'false' with the address of the first non-matching character and the length of the remainder the string. If no match is found, returns a1 n1 and 'true.' Pronounced "not-match."

For example, you could compare a string whose address is returned by **NAME** with **PAD**, testing:

```
PAD length NAME -TEXT
```

-MATCH is generally used to find a short string in a longer string. It is used by the polyFORTH editor. The use of '**-**' in the names of **-TEXT** and **-MATCH** is intended to approximate the Boolean symbol ¬, as a reminder that both words return 'true' if the strings do *not* match.

REFERENCES

B@ and **B!**, Data Base Support, Section 8.0
PAD, Section 2.3.1

2.3.6 Character String Input and Output

Character strings may be input or output by a **TERMINAL** task through words defined in this section. Such input or output is only possible where the terminal or pseudo-terminal device is capable of supporting the required operation.

2.3.6.1 CHARACTER STRING INPUT

EXPECT awaits a character string from the terminal or other serial device, given the maximum number of characters and the address of the place where they are to be stored. Input will be terminated by receipt of the specified number of characters or a 'return' ($0D_H$), whichever comes first. For example,

```
PAD 10 EXPECT
```

will await ten characters and place them at **PAD**.

During input, these user variables are used:

Word	Description
CTR	Contains the number of characters yet to come, as a negative value (*e.g.*, -80) which is incremented for each character received.
PTR	Contains the address into which the next character will be placed.
SPAN	Contains the number of characters input.
#TIB	Maximum number of characters to be processed by **WORD**.

Incoming characters are checked for "return" ($0D_H$) which terminates input, and backspace (08) or DEL ($7F_H$) which causes **CTR** and **PTR** to be "backed up" one and a backspace (or equivalent) to be sent to the terminal. All characters except these will be "echoed" to the terminal.

EXPECT should not be executed if there is no terminal or serial device capable of providing input for the task.

No indication is provided at the terminal that the system is awaiting input as a result of an **EXPECT** request. The programmer should indicate this fact through some output message issued prior to the **EXPECT** request.

In some situations it may be desirable to avoid both the editing of the input string and the echoing. For this reason, an alternative word is available, called **STRAIGHT**, which is used exactly like **EXPECT** but does not perform any editing or echoing.

The command **KEY** awaits one character and leaves it on the stack. It uses **STRAIGHT**, and does not edit or echo.

The conventional place to put incoming strings is the input message buffer, which starts at the bottom of the user's stack (whose address may be found by the word **TIB**) and extends upward in memory toward the return stack. At least 80 bytes are available. The command **QUERY** will **EXPECT** 80 bytes into the input message buffer and perform the necessary housekeeping to use words from the text interpreter to process the text. **QUERY** is used in **QUIT**.

REFERENCES

Character String Output, Section 2.3.6.4
Number Conversion, Section 2.6.2
Terminal Input, Section 3.7.1
TEXT, Section 2.3.6.3

2.3.6.2 SCANNING CHARACTERS TO A DELIMITER

WORD is the main work-horse of polyFORTH's text interpreter. It fetches characters from the input stream (terminal input buffer or a disk block) starting at the offset given by the user variable **>IN**, to a specified delimiter. **WORD** is used by **TEXT**, and functions in the same manner as **TEXT** except for the following differences:

The input characters are placed in storage starting two bytes (four bytes on 32-bit systems) beyond the current dictionary head (**HERE 2+** or **4+**). **H** is not modified. The first byte of the resulting string contains the actual length of the string, up to the occurrence of the delimiter. The area where the characters are placed is not initialized, although there will be one trailing blank (not included in the length at **HERE 2+**) inserted by **WORD** following the string. **WORD** returns on the stack the address of the string. This is a convenience for the words that conventionally follow it, such as **NUMBER**.

The space between **H** and **PAD** is used by other polyFORTH functions, such as output number conversion. As a result, when you use **WORD** to pick up a string from the input stream you should finish working with it or move it to another area (such as **PAD**) promptly to avoid confusion. **TEXT** is generally preferable, unless you need the length of the string.

As an example of **WORD**'s use, consider the following simple example (**COUNT** is a standard word whose definition is shown here for convenience):

```
: COUNT ( a - a' n)   DUP 1+  SWAP C@ ;
: TEST   32 WORD  COUNT TYPE ;
```

TEST would be used:

TEST ABC (carriage return) **ABC ok**

REFERENCES

Fetching Input Characters to **PAD**, Section 2.3.6.3
NUMBER, Section 2.6.2.1
Text Interpreter, Section 1.1.4
TYPE, Section 2.3.6.4

2.3.6.3 FETCHING INPUT CHARACTERS TO PAD

TEXT scans the current input stream until a delimiter is encountered. These characters are placed in **PAD**, which **TEXT** has previously set to blanks for a length of at least 72 bytes.

TEXT expects the delimiter character in the low-order byte of the top of the stack. **TEXT** bypasses any occurrences of this character until a non-delimiter character is encountered. That character is then placed in the first byte in **PAD**. Succeeding characters are then moved into the following positions of **PAD** until a delimiter character is encountered or the length of the string (given by **#TIB**) has expired, which terminates the operation.

TEXT takes its input from the current input source. This is normally the terminal input buffer. During the time that text is being interpreted through a **LOAD** operation, however, the current input source is the selected block being loaded.

TEXT uses **WORD** to input the string, which is then moved to **PAD**.

REFERENCES

PAD, Section 2.3.1
Terminal Input, Section 3.7.1
The **LOAD** Operation, Section 3.3.1
WORD, Section 2.3.6.2

2.3.6.4 CHARACTER STRING OUTPUT

TYPE outputs a character string to the terminal or other serial device (such as a printer). The character string is output exactly as it appears in storage, with parity bits added by the hardware if required by the terminal in use.

The length of the string, in bytes, must be on the top of the stack with the address of the first byte of the string on the stack beneath it.

For example, you could use the following phrase to print thirty-two characters from **PAD** on the terminal:

```
PAD 32 TYPE
```

During output, user variables **CTR** and **PTR** contain the count and address, respectively, of characters yet to be sent.

The command **EMIT** will type the ASCII of a single character from the stack. Thus,

```
65 EMIT
```

will type an “A.”

REFERENCES

PAD, Section 2.3.1
Terminal Output, Section 3.7.3

2.3.6.5 COMPILING MESSAGES

There are two cases in which it is desirable to have text messages compiled in programs: to issue error messages, and to communicate information during normal operation. The following words provide compiled strings:

Word	Description

"	Compiles the string following, terminated by another **"**. At run time, the address and length of the string will be pushed on the stack. For example:
	`: "TEMP" ( n)  68 > IF`
	`  " WARM "  ELSE  " COOL "`
	`  THEN TYPE ;`
	This will display the message "**WARM**" if the temperature in the stack is greater than 68, and "**COOL**" otherwise.

." string"	Compiles a string which will be typed-out when the word that contains it is executed. For example: `: Greeting  ." Hi there" ;`

t ABORT" string"	Compiles a string which will be typed out as an error message if the value on the stack (*t*) is true when the phrase is executed. For example:
	`: ?TEMP ( n)  95 >`
	`  ABORT" TOO WARM!" ;`

In both cases the quote mark serves as the delimiter that marks the end of the string. Both words use the compiling word **STRING**.

Note that each of these words has functions to be performed both at compile time and at execute time. At compile time the address of the execute-time function is compiled, along with the string. At execute time the behavior differs. For **"** the address and length of the string must be pushed on the stack. For **ABORT"** the test must be performed. For both **."** and **ABORT"** the string must be typed out.

These words must appear only inside a definition.

If **ABORT"** finds its argument to be true (non-zero), it will echo the command most recently interpreted, issue the message, clear both stacks, and return control to the operator.

Examples:

```
: HI   ." HI THERE" ;

: CHECK ( n - n)   1000 OVER <
   ABORT" TOO BIG" ;
```

REFERENCES

Error Handling, Section 2.4.7
Report Titles and Headings, Section 8.8.2
STRING, Section 2.8.7

2.4 PROGRAM STRUCTURES

Forth contains a set of words that may be used to establish program loops and alter the normal sequential execution of words. Such words are described in this section. Please note that similar words which also alter the normal flow of execution are defined in the **ASSEMBLER** vocabulary for use in **CODE** definitions.

Logic control words must be used within a definition, because they cannot operate properly when typed from a keyboard. Loops generally must be opened and closed within the same definition. Loops may be nested to any depth.

Some of the words in this section are called "compiler directives." When the compiler sees most words, it compiles cell-wide addresses pointing to the words' code field addresses. When the compiler sees a *compiler directive*, it executes it immediately rather than compiling it. Forth is extensible, so you may define your own compiler directives. Exact techniques are in the section referenced below.

REFERENCES

Compiler Directives, Section 2.8.8

2.4.1 Infinite Loops

The simplest looping method available in Forth is the **BEGIN ... AGAIN** loop. The **BEGIN ... AGAIN** loop repeats the code between the **BEGIN** and **AGAIN** endlessly. **BEGIN ... AGAIN** loops are used for control activities which are not expected to stop. Examples of such applications include process control loops and computer-sequenced machinery. **BEGIN ... AGAIN** is also used in **QUIT**, the highest level word of an interactive polyFORTH system.

An example of a high-level program to control an industrial process might be:

```
: REACTION   CONTROLS CLEAR
   BEGIN  DATA ERROR CORRECT  AGAIN ;
```

This process control loop clears the controls, then enters an infinite loop which continuously collects data, calculates an error quantity, and applies a correction function. Usually such a program is run asynchronously by a background task, and the operator stops it with a word built from task control words.

Note that the word **BEGIN** does not compile anything; it simply pushes the address of the next available dictionary location on the stack at compile time. **BEGIN** thus marks a location for a later compiler directives' compile time operations.

REFERENCES

Task Control, Sections 4.5, 4.10

2.4.2 Post-Testing Indefinite Loops

BEGIN and **UNTIL** allow the user to set up a loop which may be executed repetitively in a manner similar to **BEGIN ... AGAIN** loops, except that a test is performed before the loop repeats.

The form of a **BEGIN ... AGAIN** construction is:

BEGIN {words to be executed repeatedly} {test words} **UNTIL**

When execution reaches the word **UNTIL**, the value on the top of the stack is examined and removed from the stack. If this value is false (zero), then execution returns to the word that follows **BEGIN**; if the value is true (non-zero), execution continues with the word that follows **UNTIL**.

BEGIN loops can be nested. However, a loop of any type must be nested entirely within any outer loop.

BEGIN ... AGAIN may only be used within a single definition; it may not be executed by direct entry from a terminal.

The **ASSEMBLER** vocabulary also contains words named **BEGIN** and **UNTIL**; these words perform similar, but not exactly equivalent, functions.

REFERENCES

BEGIN ... UNTIL for the Assembler Vocabulary, Section 6.8
BEGIN ... AGAIN, Section 2.4.1
Logic Operations, Section 2.2.2

2.4.3 Pre-testing Indefinite Loops

Pre-testing indefinite loops are similar to **BEGIN ... UNTIL** loops except that the test to leave the loop is performed before the end of the loop code. The syntax of the Forth pre-testing loop is:

BEGIN {executed every iteration} {test } **WHILE**
{not executed on the last iteration} **REPEAT**

WHILE removes the number at the top of the stack and tests it, then leaves the loop if the value is false (zero), skipping the words between **WHILE** and **REPEAT**. If the value on the stack is true (non-zero) then **WHILE** continues to the next word in the loop. When the CPU reaches **REPEAT**, the CPU will go back to the words immediately following **BEGIN** and repeat the loop.

For an example, consider a word that counts fruit in a mechanical sorter:

```
: GOOD ( - n)   0  BEGIN  FRUIT ?GOOD
   WHILE  1+ FETCH  REPEAT ;
```

As long as the machine sees good fruit in the test cell the loop continues, and the machine considers the next fruit. When the test fails, the fruit remains in the test cell, to be evaluated by some process other than the word **?GOOD**.

In situations when both are equally convenient, the **BEGIN ... UNTIL** loop is faster and requires fewer bytes and is thus preferable to the **BEGIN ... WHILE ... REPEAT** loop.

REFERENCES

BEGIN ... UNTIL Loops, Section 2.4.2
Logic Operations, Section 2.2.2

2.4.4 Counting (Finite) Loops

Forth provides words to allow looping within a specified index range in a manner similar to FORTRAN DO-loops.

The possible forms of a counting loop are as follows:

```
      limit index DO {words to repeat} LOOP
or    limit index DO {words to repeat} value +LOOP
or    limit index DO {words to repeat} value /LOOP
```

The words associated with counting loops are:

Word	Function
DO	Establishes the loop parameters. This word expects the initial loop index value on the top of the stack with the limit value beneath it. These values are removed from the stack and stored on the return stack when **DO** is executed.

Word	Function
LOOP	Causes the index value to be incremented by one and compared with the limit value. If the index value is equal to or greater than the limit value, the loop is terminated and execution resumes with the next word. If the index value is less than the return value, control returns to the word that follows the **DO** that opened the loop.
+LOOP	Resembles **LOOP** but increments the index by the specified signed value. On 16-bit machines, **+LOOP** will handle any 32K range (*e.g.,* -16384 through +16383, 0 through 32767, -32768 through 0, etc.). On 32-bit machines, **+LOOP** will handle any 64K range (e.g., -32768 through 32767, 0 through 65535, and 65536 through 0).
/LOOP	Resembles **+LOOP** but uses an unsigned value, and can thus support a range from 0 through 65535 (131070 on 32-bit machines). Where incrementing values are known to be positive (which is most of the time), **/LOOP** may be significantly faster than **+LOOP**. Pronounced "ramp-loop."
I	Pushes a copy of the top item on the return stack (the current value of the index) onto the top of the parameter stack. This word may only be used for this purpose within the definition that opened the loop, not in definitions that the loop invokes, because nested colon definitions cause a return address to be put on the stack on top of the loop index.
I'	Pushes a copy of the second item on the return stack onto the parameter stack. **I'** is often used in a word called inside a loop to obtain access to **I**, or within the loop to reference the limit.
J	Pushes a copy of the third return stack item onto the parameter stack. When two **DO ... LOOP**s are nested, this is the value of the outer index from inside the inner loop.
LEAVE	Sets the limit equal to the current value of the index, forcing the loop to terminate on the next execution of **LOOP**, **+LOOP** or **/LOOP**.

To illustrate the use of loops, the word **SUM** is defined in such a way that it sums the values of the integers 1 to 100 and leaves the result on the stack:

```
: SUM   0  101 1 DO  I +  LOOP ;
```

Note that the limit value is specified as 101, not 100, because the loop terminates when the limit value is reached or exceeded after the index value is incremented. Loops may be nested to any depth, limited only by the capacity of the return stack. At each point in a nested loop, the word **I** refers to the index of the innermost active loop.

+LOOP allows descending index values to be used. When an index value is descending, however, the loop is terminated when the limit is passed (not merely reached). When the index value is ascending (*i.e.*, the increment value specified for **+LOOP** is positive), the loop terminates when the index value is reached, as for **LOOP**.

To illustrate the use of **+LOOP** with descending index values, the following definition is equivalent to the first definition of **SUM**:

```
: SUM   0  1 100 DO  I +  -1 +LOOP ;
```

Here the initial value of the index is 100 and the final value is 1.

Since the loop parameters are checked at the end of the loop, any loop will always be executed once, regardless of the initial values of the parameters. Also, loop parameters are kept on the return stack and they are not affected by loop structures other than **DO ... LOOP**.

2.4.5 Conditionals

These words allow conditional execution of words within a single definition. They may only appear within a definition and may not be used in interpretive text or in text executed by direct entry from a terminal.

The general form of usage of these words is:

{test words} **IF** {true-clause} **ELSE** {false-clause} **THEN**
or {test words} **IF** {true-clause} **THEN**

When **IF** is executed, the item on the top of the stack is removed from the stack and examined. If the value is true (non-zero), execution continues with the words that follow **IF** (*i.e.*, the true-clause). If the value of the word is false (zero), execution resumes with the words that follow **ELSE** (*i.e.*, the false-clause) or with the words that follow **THEN** if **ELSE** is not present.

Execution of the true-clause terminates with the word **ELSE**, if present. Execution resumes with the word that follows **THEN**.

Both the true-clause and the false-clause may be any group of defined Forth words. Either clause may contain **DO ... LOOP**s, **BEGIN ... UNTIL** loops, and/or other **IF ... ELSE ... THEN** structures, so long as the entire structure is contained within the clause. In other words, a **DO** or **BEGIN** loop may be used in an **IF** clause or an **ELSE** clause so long as the terminator (whether **LOOP**, **UNTIL**, or **REPEAT**) appears within the same clause. Similarly, one **IF** structure may be nested inside another structure of any kind so long as the **THEN** that terminates the structure appears within the same clause.

An "out of range" error message means too much code is between the beginning and end of a control structure. It can also mean that one part of the control structure is missing.

REFERENCES

Logic Operations, Section 2.2.2

2.4.6 EXIT

EXIT allows the address interpreter to leave a definition at any point. **EXIT** is also the word compiled by **;** at the end of every **:** definition. **EXIT** pops a value from the top of the return stack into **I** (the address interpreter's pointer to the next word to interpret), and then performs a **NEXT**, to begin execution of the next word in the definition which called the word containing **EXIT**. The return stack must be clear of any loop parameters or temporarily stored data before an **EXIT** can be performed.

A trivial example of **EXIT** is:

```
: TEST ( n)   1 .  IF EXIT THEN  2 . ;
0 TEST 1 2
1 TEST 1
```

Frequently, words containing **EXIT** will have different stack results, depending on whether the word **EXIT**s or not. The standard stack notation for such a situation is:

(input-arguments - **EXIT**-case, normal-case)

EXIT is the only Forth word which permits unstructured programs (program modules with multiple exit points). It is very bad form to use **EXIT** more than once in a word, because clarity and readability often suffer. If you believe you need to use **EXIT** twice in a word, try factoring that word into several words.

Another property of **EXIT** is the way it behaves during the text interpretation of disk blocks. When a disk block is **LOAD**ed, the text of the disk block is interpreted by the text interpreter defined in the word **INTERPRET**. **INTERPRET** is an endless loop which exits either by a **' EXIT JMP** in **WORD**, or by executing a word which removes the proper depth of entries from the return stack, and then executes **EXIT**. When the **INTERPRET** in **LOAD** encounters an **EXIT**, the **EXIT** pops the address interpreter's return stack, and performs a **NEXT** which executes the word after **INTERPRET** in **LOAD**. Thus interpretation of a disk block can be stopped by placing an **EXIT** after the last word to be interpreted.

REFERENCES

Address Interpreter, Section 1.1.6
LOAD, Sections 3.3.1, 3.3.2
Text Interpreter, Section 1.1.4

2.4.7 Abort Routines

There are three routines available for handling abort conditions for terminal tasks, plus two that may be used by any task:

Word	Function
ABORT	Unconditionally terminates execution and returns to the task's idle behavior. No message is issued. May be executed by any task.
t ABORT" text"	If *t* is true (non-zero), types the specified text at the user's terminal, clears both stacks, and returns to the task's idle behavior. (No further words will be executed.) Must be used inside a definition; may be executed only by a **TERMINAL** task. The definition of **ABORT"** concludes with the word **ABORT**.

Word	Function
QUIT	Terminates execution of the current word (and all words above it). All words are removed from the return stack and the parameter stack is cleared. No indication is given to the terminal that a **QUIT** has occurred. Execution will not resume until a new word (or words) is entered from the terminal. **QUIT** is the default idle behavior for **TERMINAL** tasks.
STOP	Terminates execution, leaving the task inactive. The task may only be awakened by an interrupt routine or another task.
NOD	An infinite loop containing **STOP**. This is the default idle behavior of background tasks. If an interrupt awakens the task it will simply go inactive again.

Since a background task by definition has no terminal on which to issue messages or to which control may be passed, abort behavior for such tasks must be handled differently, normally by setting a flag and calling **STOP**.

REFERENCES

Background Tasks, Sections 4.3, 4.4, 4.5
Terminal Tasks, Sections 4.8, 4.9, 4.10

2.4.8 Vectored Execution

Although normal Forth usage (as well as good programming practice) emphasizes the "structured programming" modes of sequential, iterative, and conditional execution, it is sometimes desirable to direct Forth to execute a specific function in response to some external stimulus. This technique may be used, for example, by a report that searches a data base, selecting records according to one of a number of specified criteria; by a bank of push-buttons, each of which is attached to a particular Forth word; or by a routine that computes the address of a function to be executed.

2.4.8.1 USING EXECUTE FOR VECTORED EXECUTION

The word **EXECUTE** expects on the stack the address of the parameter field of a definition. **EXECUTE** will execute the routine by jumping to its code address.

For example:

```
VARIABLE NUMERAL
: T1   1 . ;
: T2   2 . ;
: ONE   ['] T1  NUMERAL ! ;
: TWO   ['] T2  NUMERAL ! ;
: N   NUMERAL @ EXECUTE ;
```

If the user types:

```
ONE N
```

the computer will type 1.

Typing:

```
TWO N
```

will produce 2.

The phrase **@ EXECUTE** is so common that a special word **@EXECUTE** is defined to save space and CPU time. The behavior of **@EXECUTE** is exactly the same as the phrase **@ EXECUTE**, with the addition of a check on the contents of the address supplied: If the address to be executed is zero, **@EXECUTE** will simply return to the calling definition without performing any operation. This means that such execution vectors may not require special initialization.

The stack effect of all members of a set of words to be **EXECUTE**d in a particular context must be the same. That is, they must all require or leave the same number of items on the stack. The behavior of **@EXECUTE** in ignoring zero addresses is appropriate only in cases where there are no stack arguments.

REFERENCES

['], Section 2.8.6

2.4.8.2 USING ASSIGN FOR VARIABLE FUNCTIONS

ASSIGN provides a convenient means of storing a code address into RAM.

ASSIGN must be used inside a **:** definition. When executed, **ASSIGN** expects an address (usually that of a **VARIABLE**) to be on the stack. **ASSIGN** will store into that address the address of the cell that immediately follows **ASSIGN** in the current definition. Execution of the definition containing **ASSIGN** then terminates, so that the words following **ASSIGN** can be executed by a subsequent use of the phrase:

```
variable-name @EXECUTE
```

Thus in the structure:

```
VARIABLE  A
: B   A ASSIGN  X  Y  Z ;
```

executing **B** will set the address of the cell that follows the address of **ASSIGN** in the definition of **B** (*i.e.*, the address of **X**) in **A** but will not execute **X**, **Y**, and **Z**. Thereafter, the phrases:

```
A @ EXECUTE   or   A @EXECUTE
```

will result in the execution of **X**, **Y**, and **Z**.

Using this technique, the example of vectored execution may be simplified to:

```
VARIABLE NUMERAL
: ONE NUMERAL ASSIGN  1 . ;
: TWO NUMERAL ASSIGN 2 . ;
: N NUMERAL @EXECUTE ;
```

The use of these words would be the same as in the previous example.

An actual example of the use of **ASSIGN** (in the standard polyFORTH graphics package) specifies plotting modes:

```
VARIABLE  MARK

: LINES   MARK ASSIGN
        ( code to connect the last data point to this one) ;
```

```
: POINTS   MARK ASSIGN
        ( code to draw a + for this data point) ;

: HISTOGRAM   MARK ASSIGN
        ( code to draw a histogram segment from the last data point        to this one) ;

: PLOT   N 0 DO  I PTS @  I MARK
   @EXECUTE LOOP ;
```

All three "markings" routines expect the **Y** and **X** values for the current point, where the phrase **I PTS @** supplies the **Y** value and **I** gives the **X** value. **N** is the number of points.

The user can now type:

```
      LINES PLOT
or    POINTS PLOT
or    HISTOGRAM PLOT
```

in a natural manner. The chosen function will remain set until changed.

2.4.8.3 CREATING VECTORED EXECUTION TABLES

Most uses of **EXECUTE** and **@EXECUTE** are for implementing a variable function as described in the previous section. The ability to generate and manage a table of execution addresses is also extremely useful for such purposes as managing a function-button pad, function menu on a graphics tablet, etc. This section will outline a simple button response application which may serve as a model for similar situations.

Let us assume that the word **BUTTON** has been defined to wait until a button is pressed and then to return the button number (0-15) of the button (the actual definition of **BUTTON** would depend on the computer and interface). Now consider the following:*

```
VARIABLE BUTTONS   30 ALLOT
: IGNORE ;
' IGNORE BUTTONS !  BUTTONS DUP 2+ 30 MOVE
```

The above lines create a table with one cell for each button and also initialize all positions to contain the address of an empty definition (which effectively "ignores" an undefined button).

Now we will define a special defining word that will not only create an ordinary **:** definition but also store the address of its parameter field into a specified cell of **BUTTONS**:**

```
: :B ( n)   :   LAST @ @ CFA 2+
   SWAP 2* BUTTONS + ! ;
```

Now we can create definitions which are attached to certain buttons by using **:B** with the button number as a parameter. Each such definition will have a name to allow it to be tested independently of the button pad. For example,

---

* For 32-bit machines, use **60 ALLOT**, **4+**, and **56 MOVE**.

** For 32-bit machines, use **CFA 4+** and **4***.

```
0 :B ESCAPE   1 ABORT" ?" ;
```

defines Button 0 to be an “escape” button.

All that remains is to define a routine to monitor the button pad and handle responses:

```
: MONITOR   BEGIN BUTTON BUTTONS
   + @EXECUTE  AGAIN ;
```

Typing **MONITOR** will place the terminal task in an infinite loop that responds to buttons. Button 0 will cause an abort and return control to the terminal.

In practice, **MONITOR** may very likely be executed by a background task. In this case you must use **STOP** or **NOD** for halting rather than **ABORT"** (which requires a terminal).

REFERENCES

ABORT", Section 2.4.7
BACKGROUND Tasks, Sections 4.0, 4.3-4.5
STOP and **NOD**, Sections 2.4.7, 4.2

2.5 NUMERIC OUTPUT WORDS

Numeric output words allow the output of numeric quantities in ASCII. This output is generally directed to the terminal.

Numeric output words are divided into two categories: normal output words and conversion output words. The latter allow the “picturing” of ASCII text, in a manner that resembles COBOL picturing.

All numeric output words produce ASCII text, which is the ASCII number expressed in the current radix contained in **BASE**. The **BASE** is controlled either through the appropriate radix word (*e.g.*, **OCTAL**, **DECIMAL**, or **HEX**) or by directly setting the current value of **BASE**. For example, **BASE** might be set to binary by:

```
2 BASE !
```

REFERENCES

Numbers, Section 1.1.5

2.5.1 Standard Numeric Output Words

Several standard words allow printing of single or double-precision signed numbers. Each prints an output string that consists of the following characters:

1. If the number is negative, a leading minus sign (hyphen).

2. The absolute value of the number, with leading zeroes suppressed. The number zero results in a single zero in the output.

3. A trailing blank.

The following table lists the standard numeric output words.

Word	Stack	Function
.	(n)	Prints a signed single-precision integer followed by one space. Pronounced "dot."
?	(a)	Prints the contents of the address on the stack (**?** is equivalent to the phrase **@ .**).

Word	Stack	Function
U.	(u)	Prints an unsigned single-precision integer followed by one space. Pronounced "u-dot."
U.R	(u n)	Prints the unsigned single-precision integer *u* with leading spaces to fill a field of width n, right-justified. This word expects an integer on the top of the stack to specify the length of the output field. The width of the printed string that would be output by . is used to determine the number of leading blanks printed. No trailing blanks are printed. If the magnitude of the number to be printed prevents printing within the number of spaces specified, additional output characters result.
D.		Prints a signed double-precision integer.
D.R		Prints a signed double-precision integer in a specified field-width, as for **U.R**.

2.5.2 Pictured Number Conversion

polyFORTH contains a series of words that allow numeric quantities to be output through use of a pictured format control. These words allow specification of field sizes, editing characters, etc.

In Forth, the description of these words starts with the low-order portion of the field and continues to the high-order portion. Although this is the reverse of the method apparently used in other languages, it is the actual conversion process in all languages. **BASE** is a user variable containing the current conversion radix.

These words are used to convert numbers on the stack into ASCII character strings which have been edited according to the picture specifications. These strings are built in an area in memory which immediately follows the end of the dictionary (the address left by **HERE**). This area is large enough to accommodate at least 32 characters of output (64 characters on 32-bit machines). Following the end of picture conversion, the address of the beginning of the string and the count of the number of characters in it are passed to the user. At this point, the converted string can be printed at the terminal with **TYPE** or used in some other way.

All of the standard numeric output words use the same region in the user's partition (the area below **PAD**). As a result, these words may not be executed while a pictured output conversion is in process (*e.g.*, during debugging). Furthermore, the user may not make new definitions during the pictured conversion process, since this would move the area in which the string is being generated.

REFERENCES

Numbers in Forth, *Starting FORTH*, Chapter 7
PAD, Section 2.3.1
Report Generator, Data Base Support, Section 8.0
Standard Numeric Output, Section 2.5.1

2.5.2.1 USING PICTURED NUMERIC OUTPUT WORDS

These words allow control over the conversion of binary numbers into digits. This section only describes pictured words which result in digit output; the following sections describe output of non-numeric punctuation such as periods and commas. Throughout the number conversion process the number being operated on remains on the stack, where it is repeatedly divided by **BASE** as digits are converted. The number is finally discarded by **#>** at the end of the process.

Word	Stack	Function
<#	(ud - ud) or (n ud -n ud)	Initializes pictured output of an unsigned double-precision integer. If the output is to be signed, a signed value must lie immediately beneath this integer to control whether or not a minus sign will be introduced into the output string by **SIGN** (below).
#	(ud - ud)	Adds a digit to the low-order portion of the resulting string. Must be used after **<#** and before **#>**. The first digit added is the lowest-order digit (units), the next digit is the tens digit, etc. Each time **#** is used, a digit is generated.
#S	(ud - ud)	Converts digits repetitively until all significant digits in the source item have been converted, at which point conversion is completed. Must be used after **<#** and before **#>**. **#S** always results in at least one output character, even if the number to be converted is a zero.

Word	Stack	Function
SIGN	(n ud - ud)	Inserts a minus sign at the current position in the string being converted if the signed value in the third stack position is negative. This signed value is a single-precision number; if the high-order bit is set, a minus sign will be introduced into the output as the left-most non-blank character. The magnitude of the signed value is irrelevant. In order for the sign to appear at the left of the number (the usual place) **SIGN** must be called after all digits have been converted.
#>	(ud - a n)	Completes the conversion process after all digits have been converted. This word discards the (presumably) exhausted double-precision number, and pushes onto the stack the address of the output string, with the count of bytes in this string above it.

To aid in understanding the use of these words, consider a definition of the standard Forth word **.** ("dot"):

```
: . ( n)   DUP ABS 0  <# #S SIGN #>
   TYPE SPACE ;
```

DUP ABS puts two numbers on the stack; the absolute value of the number on top of the number itself, which is now useful only for its sign. **0** adds a cell on top of the stack, so that the **0** cell and the **ABS** cell form a double-precision integer to be used by the **<# ... #>** routines.

If you want to print a signed double-precision integer with the low-order three digits always appearing, regardless of the value, you could use the following definition:

```
: NNN ( d)   SWAP OVER DABS  <# # # #S
   SIGN  #>  TYPE SPACE . ;
```

The **SWAP OVER DABS** phrase establishes the signed value beneath the absolute value of the number to be printed for the word **SIGN**. The sequence **# #** converts the low-order two digits, regardless of value. The word **#S** converts the remaining digits and always results in at least one character of output, even if the value is zero.

From the time when the initialization word **<#** executes until the terminating word **#>** executes, the number being converted remains on the stack. It is possible to use the stack for intermediate results during pictured processing but any item placed on the stack must be removed before any subsequent picture editing or fill characters may be processed.

2.5.2.2 USING PICTURED FILL CHARACTERS

In addition to pictured numeric output, it is possible to introduce fill characters (or punctuation) into the output string through the use of **HOLD** and **'.'**. When one of these words is used, the appropriate character is entered into the output string at the current position.

Arbitrary fill characters may be inserted in a string being formatted by using the word **HOLD**. **HOLD** requires as a parameter the numeric value of the ASCII character to be inserted. Thus,

```
2F HOLD        (value given in hex)
```

inserts the character / into the output string.

'.' produces a decimal point at the current position in the pictured numeric output. To illustrate, the word **.$** will print double-precision integers as signed amounts with two decimal places:

```
: .$ ( d)   SWAP OVER DABS  <# # # '.'
   #S SIGN #> TYPE  SPACE ;
```

If fill characters are likely to be used in several definitions, you may wish to add commands similar to **'.'**. The following format is used for such a definition:

```
: 'name'   char-value HOLD ;
```

where **char-value** is the ASCII value of the character in the current radix and **'name'** is the name of the word to be defined. **HOLD** is defined in such a way that executing **'name'** during pictured editing causes the indicated fill character to be introduced into the output.

2.5.2.3 PROCESSING SPECIAL CHARACTERS

The normal pictured output capabilities described in the preceding two sections are generally sufficient to handle most output requirements. Certain special cases, however, such as the introduction of commas in a number or the floating of a character (*e.g.*, **$**), require special processing. In order to perform certain of these operations, it is necessary to refer to the unconverted portion of a number being printed.

This unconverted portion is a number that is equivalent to the original number divided by 10 (or the current radix) for each numeric digit already generated. For example, if the initial number is 123, the intermediate number is 12 (following the conversion of the first digit) and 1 (following conversion of the second digit).

The value of this number may be tested and logical decisions may be made based upon its value. To illustrate, consider the following definitions. The word **D.ENG** prints a double-precision integer in engineering format:

```
 0 ( Number Formats)   DECIMAL
 1 : ','   44 HOLD ;
 2 : (D.ENG ( d)   SWAP OVER DABS  <#  BEGIN
 3    ',' #  2DUP D0= NOT WHILE   2 0 DO
 4    #  2DUP D0= IF  LEAVE  THEN   LOOP   REPEAT
 5   SIGN #> ;
 6 : D.ENG ( d)   (D.ENG)  TYPE  SPACE ;
 7
 8
 9
10
11
```

```
12
13
14
15
```

Using techniques similar to those set forth above, you can do almost any kind of numeric output editing in Forth.

REFERENCES

Report Generator, Data Base Support, Section 8.8

2.6 TEXT INTERPRETER WORDS

The text interpreter in Forth is used both for terminal interaction and for processing text in disk blocks (either in direct execution or compilation). The purpose of this section is to discuss ways in which the programmer may use the text interpreter in application routines.

REFERENCES

Text Interpreter, Section 1.1.4

2.6.1 Dictionary Searches

For the dictionary to be useful, it must be possible to look up words and their definitions in it. Forth provides several words, all of which perform a search and then return information about the word. These searches are used in the text interpreter and colon compiler without modification. Note that since polyFORTH has variable length names, words are available which find addresses that point both before and after the name.

The following are dictionary search words:

Word	Stack	Function
'	(- a)	Gets the next word from the input stream, and attempts to look up the word in the dictionary. If the word is found, then **'** returns the word's parameter field address, otherwise **'** aborts. This word calls **-'**. Pronounced "tick."
[']		Must be used in a colon definition. **[']** finds the next word in the text and compiles the word's parameter field address as a literal. If the next word is not in the dictionary, **[']** aborts. **[']** calls **-'** and **LITERAL. [']** is an **IMMEDIATE** word (executed rather than compiled by the colon compiler, see the references section). Pronounced "bracket-tick."
-'	(- a a' 0 , a t)	Gets the next word from the input stream and attempts to look up the word in the dictionary. If the look-up succeeds, **-'** returns the parameter field address (a), the dictionary link address (a') and a zero. If the word is not found, **-'** returns the address of the string (**HERE 2+** or **HERE 4+** on 32-bit machines) and a number guaranteed to be non-zero (*i.e.,* 'true'). This word is called by the text interpreter and colon compiler. Pronounced "dash-tick."

Word	Stack	Function
'HEAD	(- a)	Gets the next word from the input stream, and attempts to look up the word in the dictionary. If the word is found, then **'HEAD** returns the address of the word's first byte, otherwise aborts. This word calls **-'**. Pronounced "tick-head."

Tick performs a dictionary search for the word that immediately follows it in the current input stream.

The phrase:

```
' name
```

when typed at a terminal or executed interpretively in a source block, pushes onto the stack the address of the parameter field of *name* if *name* can be found in the dictionary. If *name* cannot be found, an abort will occur with the error message:

```
name ?
```

The most common uses of **'** for dictionary searches are:

1. To find out whether a word has been defined.
2. To find the location of a word (for example, to **DUMP** its contents).
3. To obtain the address of a **CONSTANT** (for example, in order to change it).

Note that **'** is not immediate. That is, if you wish to *compile* a reference to an address, as in item 3 above, you must use **['']**. The words **'** and **'HEAD** always take their operands from the current input stream at the time they are executed. This is an important characteristic because it makes possible the use of these words in such words as **FORGET** (which uses **'HEAD** to find the location of the word to be "forgotten").

The order in which the various vocabularies are searched is specified by **CONTEXT**. The content of **CONTEXT** is best examined as a four-digit hexadecimal number whose high-order digit is the index of the first vocabulary to be searched, whose next digit gives the next vocabulary, etc. An index of zero terminates the search. Thus, if **CONTEXT** is 1500, the search order is **FORTH**, then **EDITOR**. A value of 3150 would specify **ASSEMBLER**, then **FORTH**, then **EDITOR**. In some CPUs, the order is reversed. For example, 0513 on the Z80 would specify **ASSEMBLER**, **FORTH**, then **EDITOR**.

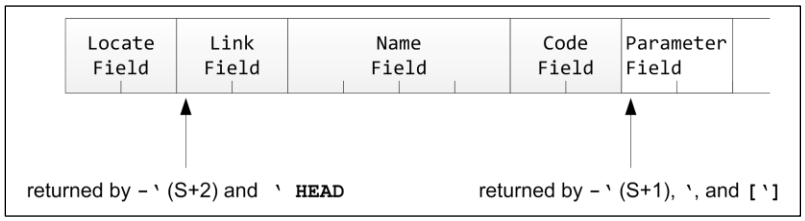


Fig. 2.5

Dictionary entry, showing the addresses returned from common dictionary search words.

Please note that **'** and **'HEAD** make use of a lower level dictionary search word: **-'** (pronounced "dash-tick"). The word **-'** searches for the next word in the input stream, delimited by blanks. If the word is found, **-'** returns the parameter field address, the link field address and a zero (or false) on top. If not found, **-'** returns two numbers. The number on top is guaranteed to be non-zero and the other is the address of the string which was delimited by blanks (**HERE 2+**, or **4+** on 32-bit machines). The definition of **'**, using **-'** is:

```
: ' ( - a)   -'  ABORT" ?"  DROP ;
```

The top of the stack after **-'** is used as a Boolean truth value by **ABORT"**. If the search succeeds, the **DROP** removes the link address, leaving the desired word's parameter field address on the stack. If the search fails,

ABORT" resets the user environment and prints the text placed just above **HERE** by **WORD** (which happens to be the word that **-'** couldn't find) followed by an error message.

REFERENCES

['], Section 2.8.6
ABORT", Section 2.4.7
IMMEDIATE Words, Section 2.8.8
Vocabularies, Section 3.4
WORD, Section 2.3.6.2

2.6.2 Input Number Conversion

Wherever possible an application should be designed to take advantage of polyFORTH's naturally interactive capability. Thus, a hypothetical word **SCANS** whose function is to perform some user-specified number of scans (an application function) should expect only its parameter on the stack. Then to perform 100 scans, the user could type:

```
100 SCANS
```

Such a usage is natural and convenient for the operator and requires no special programming to handle the input parameter.

There are occasions in which normal Forth syntax is inadequate. Some examples include:

1. Parsing a text string that comes from a source other than a terminal, such as magnetic tape.
2. Entry of numbers that must be in double-precision but are not punctuated (*i.e.*, zip codes).
3. Entry of numbers that must follow rather than precede the command.

polyFORTH provides several words to enable the user to handle input numbers in a variety of circumstances. This section describes these methods.

2.6.2.1 NUMBER CONVERSION USING THE TEXT INTERPRETER

The word **NUMBER** is the standard input number conversion routine used by the text interpreter and colon compiler. It is available to users and performs number conversions explicitly from ASCII to binary. **NUMBER** uses the value in the user variable **BASE** to determine which radix should be used when converting numeric strings to binary.

Normally **NUMBER** immediately follows **WORD**. It expects on the stack the address of the string that is to be converted, with a count in the first byte of the string and one trailing blank following the counted bytes. **NUMBER** will attempt to convert that string to binary and, if successful, will leave the result on the stack. Its rules for behavior in the conversion are those described in the earlier section on numbers, depending upon the presence of the extended-precision math option. If the conversion fails due to illegal characters, an abort will occur, with an error message echoing the string followed by "**?**."

Thus if the number to be converted is coming from the normal input stream, the complete sequence would be:

```
QUERY 32 WORD NUMBER
```

This would leave the converted binary value on the stack.

Note that **QUERY** inputs to the input message buffer, and automatically takes care of the housekeeping necessary to prepare the input message buffer for the interpreter.

NUMBER may be used to convert a string from another location (*e.g.*, a string which has not been fetched by use of **WORD**). If the location of the string does not contain the count in the first byte, you may simply simulate its presence by subtracting one byte from the starting address of the string. **NUMBER** does not actually make use of the count; it only adds one byte to the address before beginning. Thus, for a string whose actual location is given by a word named **BUF**, the sequence would be:

```
BUF 1- NUMBER
```

The requirement that a blank follow the string is more absolute. If it is not feasible to guarantee this, you may prefer to move the string to **PAD**, as shown below, or use **CONVERT** which will not abort out of the application program as **NUMBER** does when it sees a non-numeric character.

REFERENCES

CONVERT, Section 2.6.2.2
Numbers, Section 1.1.5
QUERY, Sections 2.3.6.1, 3.7.1
WORD, Section 2.3.6.2
Use of **NUMBER** in **INTERPRET**, Section 1.1.4

2.6.2.2 DIRECT CONVERSION OF STRINGS

CONVERT is useful because it stops when it encounters any non-numeric digit, rather than aborting as **NUMBER** does. For this reason, **CONVERT** is often used when a number is input by a program directly, without using the text interpreter.

CONVERT expects a double-precision integer and byte address, and leaves a double-precision integer and address. The initial address into **CONVERT** must point to the byte preceding the first digit of the string of numerals. This byte is ignored. The initial double-precision number is usually set to zero.

After **CONVERT** stops, the address on top of the stack is the address of the first non-numeric character **CONVERT** encountered. The double-precision integer will contain data from all the digits that have been converted thus far.

An example of the use of **CONVERT** is:

```
: INPUT ( - n)    PAD 8 BLANK  PAD 1+ 5
   EXPECT  0. PAD CONVERT  2DROP ;
```

This definition initializes a region of **PAD** to blanks, and awaits up to five digits which will be stored there. **0.** provides an initial double-precision value, and **PAD** provides the address for **CONVERT**. The **2DROP** discards the address and high-order part of the numbers.

INPUT will not convert input strings with a leading minus sign, because a minus is not a digit. If negative input is necessary, the above definition can be extended to check the character upon which conversion stopped to see if it is a minus sign, and if it is, start **CONVERT** again, and negate the result. For an example of the system use of **CONVERT** see the definition of **NUMBER** in the double-precision input block.

CONVERT returns the address of the string's next byte so that **CONVERT** may be called in a loop. **NUMBER** calls **CONVERT** in just this way. An application similar to **NUMBER**'s is the parsing of a packet of data received over a communications line or in a tape record, in which numeric fields are separated by an arbitrary delimiter such as

//. To skip over such items, or to skip fields that are not of interest, the appropriate count of bytes may simply be added to the address, which is carried on the stack.

In some cases numbers may be in fields of known length but not separated by any delimiter. In these cases the best solution may be to use **CMOVE** to move groups of digits to **PAD**, where they may be converted easily by **NUMBER**.

REFERENCES

CMOVE, Section 2.3.4
EXPECT, Sections 2.3.6.1, 3.7.1
NUMBER, Section 2.6.2.1
PAD, Section 2.3.1

2.7 DEFINING WORDS

polyFORTH provides a basic set of words that are used to define objects of various kinds. As with other features of polyFORTH, the set of such commands may be expanded. Here we will present those which are standard in all polyFORTH systems, exclusive of the defining words that are part of the data base support option, the assembler defining words (see your *CPU Supplement*), and **MSG**.

REFERENCES

MSG, Sections 2.3.3, 2.7.6.3, 3.7.3

2.7.1 Creating a Dictionary Entry

A word is defined when an entry is created in the dictionary. **CREATE** is the Forth word which creates a dictionary entry. A dictionary entry as constructed by **CREATE** is shown in Fig. 2.6. **CREATE** is used by **:**, **CODE**, **VARIABLE**, **CONSTANT**, and other defining words. **CREATE** behaves as follows:

1. The width between the top of the dictionary and the top of the parameter stack is checked to see if at least 250 bytes remain. If not, there will be an abort with the error message, "Dictionary full." At the same time **H** is adjusted to an even cell address on machines such as the PDP/LSI-11 and 68K that don't tolerate odd-byte addresses.

2. **WORD** fetches the next word of the input string to the top of the dictionary, with a cell left empty for the dictionary link. The first byte of the copied string contains its length.

3. The empty cell is filled with a pointer to the previous entry in the proper dictionary chain for the new word.

4. The user variable **LAST** is set to point to the head of the chain containing the new word.

5. Space is allotted for the new word's name, depending upon the value of **WIDTH**. (Truncation of the name may take place at this point).

6. The value of **WIDTH** is reset to the default width in **WIDTH 1+**.

7. The code field address of the new word is set to point to the run time code of **CREATE**, which will push the address of the parameter field onto the stack when the word just defined is executed.

CREATE names a location in memory. Other defining words which use **CREATE** may reset the new word's code field address by using the words **;CODE** or **DOES>** to define different run-time behavior.

CREATE is often used to mark the beginning of an array. The space for the rest of the array is reserved by incrementing **H** with **ALLOT**, as in this example:

```
CREATE DATA   200 ALLOT
```

The example reserves a total of 100 cells for an array named **DATA** (50 cells on 32-bit processors). When **DATA** is used in a colon definition, the address of the first byte of **DATA** will be pushed on the stack by the run-time behavior of **CREATE**. The array is not initialized. If you wish to set all the elements of the array to zero, you may use **ERASE** as in the following example:

```
DATA  200 ERASE
```

This usage is not appropriate for applications that are to be target compiled for ROM, because in such an environment **CREATE** returns a ROM address. **VARIABLE** should be used in such applications, and the count for **ALLOT** reduced by two to allow for the two bytes **ALLOT**ed by **VARIABLE**. On 32-bit machines, the count for **ALLOT** should be reduced by four.

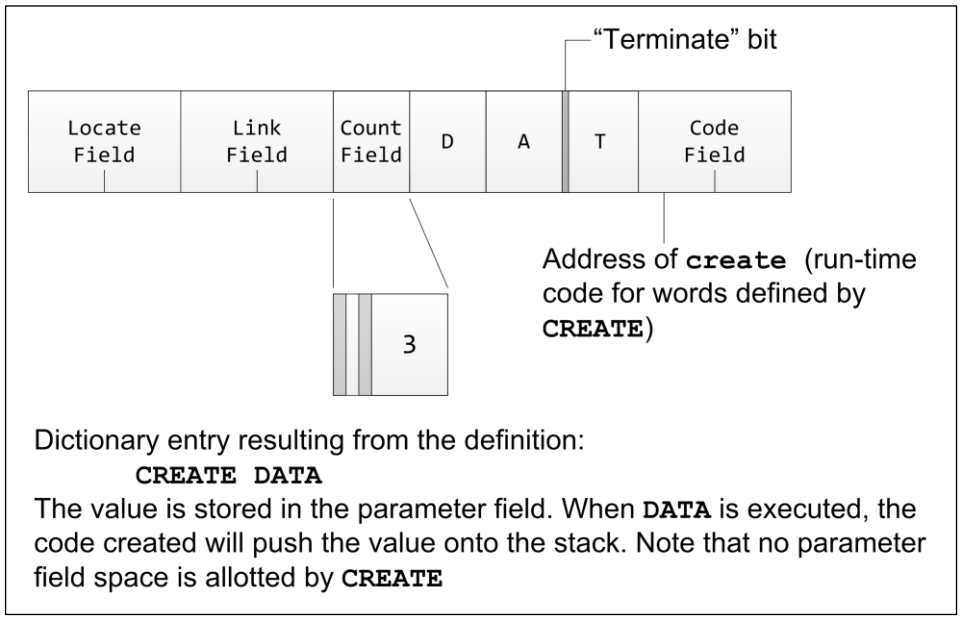


Fig. 2.6

CREATE is the fundamental builder of dictionary entries. It is used by **:**, **CONSTANT**, **VARIABLE**, and all other Forth defining words. Since it is a vectored routine you may modify or enhance the behavior of the Forth compiler in constructing dictionary entries. Your system provides three levels of enhancement:

Word	Source	Description
(CREATE)	nucleus	Primitive level; constructs a dictionary entry as described in Section 2.7.1, whose code address points to code that pushes the parameter field address on the stack. Does not allot any parameter field space.
<CREATE>	12	Compiles the information used by **LOCATE** (source block number) in a cell, then calls **(CREATE)**. Thus the **LOCATE** field precedes the head of the definition.
?CREATE	21	Searches the dictionary for the word to be compiled and issues a warning message if the word was previously defined, then executes **<CREATE>**.

In order to "enable" one of these versions of **CREATE**, you would store its address in the execution vector **'CREATE**. Thus, the phrase:

```
' <CREATE> 'CREATE !
```

in Block 9 enables the version that compiles the **LOCATE** cell. Since this version changes the size of the head of a definition, a constant called **B/H** is provided which contains the number of bytes added to the head size. If **B/H** is 2, in other words, **'HEAD** will return an address two bytes before the link field of all words, even nucleus words that don't have a **LOCATE** field. The word **-'** (Section 3.3.2) always returns the link field address, however.

?CREATE may be enabled in Block 9 by enclosing **EXIT** within parentheses, *i.e.*, **(EXIT)**. Since it doesn't change the actual dictionary entry in any way, it doesn't require any change in **B/H**.

REFERENCES

:, Section 2.7.4
CODE, Sections 2.7.5, 6.0
CONSTANT, Section 2.7.3
ERASE, **BLANK**, **FILL**, Section 2.3.4
Host Defining Words, Section 7.5
Target Defining Words, Section 7.8
VARIABLE, Section 2.7.2
Vectored Routines, Section 3.1
WIDTH, and the word **~**, Section 1.1.1

2.7.2 Variables

A **VARIABLE** is a named memory location whose value may be either fetched onto the stack or stored into, with equal ease. It may also be treated as the beginning of an array.

The form of the definition of a **VARIABLE** is:

```
VARIABLE NAME
```

This constructs a definition whose name is **NAME**, with two bytes allotted for a value (four bytes on 32-bit systems). A single-precision value may be stored into the parameter field of the definition. For example:

```
6 NAME !
```

will store 6 in the parameter field of **NAME**.

When a **VARIABLE** is referenced by name, the address of its parameter field is pushed onto the stack. This address may be used with **@** or **!** to fetch or store a value, respectively.

The word **2VARIABLE** defines a variable whose parameter field is two cells long. Such a variable may contain one double-precision number or a pair of single-precision numbers (such as x,y coordinates). **2VARIABLE** differs from **VARIABLE** only in the number of bytes allotted. The operators **2@** and **2!** are used with this format.

Similarly, **CVARIABLE** defines a variable one byte long. This is only available on machines that tolerate odd-byte addresses, such as the 8086/88. The operators **C@** and **C!** are used with this format.

Since **VARIABLE** defines a four-byte item on 32-bit systems, the word **HVARIABLE** is available on these systems to define two-byte variables. The operators **U@**, **H@**, and **H!** are available on 32-bit systems to access **HVARIABLES**.

In summary, to get the value of a **VARIABLE** on the stack, you invoke its name and fetch instruction. For example, you could type:

```
      varname @
 or   varname 2@
```

To store into a variable, you invoke the name and a store instruction. For example:

```
      value varname !
 or   value1 value2 varname 2!
```

In a read-only memory environment, **VARIABLE** is re-defined to allot space in read/write memory rather than its own parameter field; in this case the read/write memory address assigned is compiled into the parameter field. The run time behavior of a variable in read-only memory is the same as a constant.

REFERENCES

@, **!**, **2@**, and **2!**, Section 2.1.2
ALLOT, Section 2.8.1

2.7.3 Constants

The purpose of a **CONSTANT** is to provide a name for a value which is referenced often but changed seldom or never. There are two constructs:

Word	Description
CONSTANT	Defines a single-precision constant.
2CONSTANT	On systems using the 32-bit electives, or on 32-bit machines, defines a double-precision constant whose value may be either a double-precision number or a pair of single-precision numbers.

The procedure for defining constants is to declare:

```
value CONSTANT name
```

For example, you may define:

```
1000 CONSTANT LIMIT
0 5000 2CONSTANT LIMITS
3.141593 2CONSTANT PI
```

When a **CONSTANT** is referenced by name, its value (not its address) is pushed onto the stack. Similarly, when a **2CONSTANT** is referenced, its double-precision value is pushed onto the stack. In the case of **2CONSTANT** (used for two values as in **LIMITS**, above), the values are placed on the stack in the order specified (*e.g.*, 5000 on top, 0 below). In the case of a double-precision number, the high-order part of the number is on top of the stack.

In order to change a **CONSTANT** you must first obtain its address. This is done by using **'** when in interpretive mode or **[']** when inside a colon definition, in either case followed by the name of the **CONSTANT**. The command **!** will store in to the address of a **CONSTANT** thus obtained and 2! into the fetched address of a **2CONSTANT**.

For example, you might type this at your terminal:

```
100 CONSTANT COUNT   500 ' COUNT !
```

The first phrase creates a **CONSTANT** named **COUNT** whose value is 100. The second phrase changes the value to 500.

CONSTANTs in read-only memory applications cannot be changed, as the value is kept in read-only memory. Thus you should avoid storing into constants if you are developing read-only memory applications.

2.7.4 Colon Definitions

The defining word **:** has already been discussed by implication in Section 1.2 and numerous examples have appeared in other sections. The purpose of this passage is to describe the use and behavior of this important defining word.

The basic form of a **:** definition is:

```
: name   content ;
```

Here the **:** constructs a dictionary entry for the word **name**. **content** represents a list of previously defined words that will be executed in sequence whenever **name** is invoked. The **;** terminates the definition.

Each of the words **:** and **;** has two types of behavior: one type at compile time and another at execution time.

At compile time, **:** constructs the dictionary entry using **CREATE** and begins compiling. In addition, **:** resets the search vocabulary in **CONTEXT** to the default in **CURRENT**, and sets the smudge bit so that the word will not normally compile a reference to itself. The word **RECURSE** is used where the definition of a word must call itself.

The code field address of the new definition called **name** is the parameter field address of the run-time code shared by all **:** definitions. This code starts the address interpreter executing the words whose addresses form the body of the definition, by pushing the current value of the interpreter pointer **I** onto the return stack and setting **I** to the address of the parameter field of the word, which is obtained from the system pointer **W**.

The **;** ends compilation and compiles the address of the code field address of the run-time code for **;** (the word **EXIT**). This code pops the address on top of the return stack into the interpreter pointer **I**. The effect is to return to the calling environment.

Most of the words that make up the content of the definition are not executed during compilation; instead their absolute addresses are compiled in the form given in Fig. 1.6. The exception to this procedure are the words which are compiler directives or literals. These generally have both compile time and run-time behaviors, just as **:** and **;** do.

Note: Every colon definition requires a minimum of three components: a colon, a name, and a semicolon. Such a minimum definition will execute properly, but will do no work. It does have useful purposes; for example as described in **FORGET** to mark a location in the dictionary as the beginning of an overlay area.

REFERENCES

Address Interpreter, Section 1.1.6
Compiler Directives, Section 2.8.8
EXIT, Section 2.4.6

Overlays, Section 3.3.4
Program Structures, Section 2.4

2.7.5 Code Definitions

The form of a **CODE** definition is:

CODE name {assembler instructions} {code-ending}

The word **CODE** performs the following functions at assembly time:

1. Constructs a standard dictionary entry for **name** using **CREATE**.
2. Sets the code field address of the word to point to the word's parameter field.
3. Selects the **ASSEMBLER** vocabulary.

With **CODE** there is nothing analogous to a compiler; assembler words are executed directly, with the effect of assembling machine instructions in the parameter field of the word being defined. When high-level Forth words are encountered, they are executed directly as well—thus, words such as **SWAP** and **DUP** manipulate the stack during assembly. Macros can be defined as **:** definitions containing assembler words. Macros must be defined in the **ASSEMBLER** vocabulary.

The basic principles of polyFORTH assemblers are covered in Section 6.0. Assembler mnemonics, addressing modes, and conventions are covered in the *CPU Supplement* for each CPU on which polyFORTH is implemented.

The most common code ending is **NEXT**, which is a return to the address interpreter. All other code endings also go to **NEXT** after performing additional functions. Both the form and exact list of code endings must be taken from the *CPU Supplement* for your CPU; a typical list might include these:

Code Ending	Action
NEXT	Returns to the address interpreter.
WAIT	Forces the current task to enter the multitasking loop pending an interrupt.
' PAUSE 2+ JMP	Awakens the task and enters the multitasking loop. The task regains control on its next turn. Commonly used in polling cycles. The **2+** may be smaller or greater depending on the CPU.
' PAUSE JMP	Same as above, but **I** is first decremented by a cell, so that on return from **PAUSE**, the current code definition will be re-executed, from the beginning. This is useful for polling.
' EXIT JMP	Leaves the high-level word which called the code word containing this phrase.

REFERENCES

Assembler Code Endings, Section 6.2
NEXT, Section 1.1.6
WAIT, Section 4.2

2.7.6 Custom Defining Words

One of the most powerful capabilities in Forth is the ability to define words that are themselves defining words. Thus the programmer may create either new data types with characteristics peculiar to the application or new generic types of words.

In creating a custom defining word, the programmer must specify two separate behaviors:

1. The compile-time behavior of the defining word (the creation of the dictionary entry, compiling of parameters, etc.).

2. The run-time behavior (what action members of the new class of words perform when invoked).

In the cases discussed in the next two sections, compile-time behavior is described in high-level Forth. Several methods are available for specifying run-time behavior; these also will be covered below.

2.7.6.1 BASIC PRINCIPLES OF DEFINING WORDS

There are two ways to create new defining words in Forth. They differ in that, in the one case (using **DOES>**) the run-time behavior is described in high-level Forth, and in the other (using **;CODE**), the run-time behavior is described in assembler code. The basic principles are the same and will be covered in this section.

The Forth term "defining word" means a word which will create a new dictionary entry when executed. All words defined by the same defining word share a common characteristic compile-time and run-time behavior. For example, **VARIABLE** is a defining word; all words defined by **VARIABLE** share two common characteristics:

1. Each has one cell allotted in which a value may be stored. These bytes may (in a resident system) be initialized to zero.

2. When executed, each of these words will push onto the stack the address of this one-cell reserved area.

On the other hand, all words defined by **CONSTANT** have two different characteristic behaviors:

1. Each has compiled with it a single-precision value which was on the stack when **CONSTANT** was executed.

2. When a word defined by **CONSTANT** is executed, it fetches its value and pushes the value on the stack.

In each of these examples, Behavior 1 relates to the physical construction of the word, which is determined when the word is compiled: space allotted, values compiled, etc. Behavior 2 describes what all defined words of a certain type do when they are executed. All defining words must have some compile-time behavior (1) and some run-time behavior (2). The general form of the definition of defining word is:

: {name} {compile-time behavior} {transition word}
{run-time behavior} **;**

The "transition word" marks the end of the specification of compile-time behavior and begins the specification of the run-time behavior. There are two "transition words":

Word	Description
;CODE	Begins run-time behavior described in code.

DOES>	Begins run-time behavior described in high-level Forth.

The exact behavior of these words is discussed in the following sections. The description of compile-time behavior is the same regardless of which transition word is used. In fact, if you change the transition word and run-time behavior from **DOES>** plus high-level to **;CODE** plus equivalent code, no change to the compile-time behavior is necessary.

The compile-time portion of a defining word must contain a previously defined defining word to create the dictionary entry. If one or more parameters are to be compiled or if space for variable data is to be allocated, it is convenient to use a defining word which takes care of that for you. The most common defining words are:

Word	Function
CREATE	Used when there are no compile-time parameters.
CONSTANT	Used when there is to be one compile-time parameter.
2CONSTANT	Used when there are to be two compile-time parameters.
VARIABLE	Used when one cell of storage is to be allocated.
2VARIABLE	Used when two cells of storage are to be allocated.
CVARIABLE	Used when one byte of storage area is to be allotted.

Every defining word must allow space for some piece of data or code belonging to each member of the new class of words. For example, when a variable is defined, space is allotted for its parameter field. If more space is required than that allotted by a standard word (**VARIABLE** or **2VARIABLE**, for instance), the standard form is to use **CREATE** followed by **ALLOT**. If the application is to be target compiled for ROM, the combination of **VARIABLE** or **CVARIABLE** and **ALLOT** must be used.

When used, a defining word may be followed by any number of words such as **,** (which compiles a single-precision value) or **C,** (which compiles an 8-bit value).

REFERENCES

, and **C,**, Section 2.8.2
;CODE, Section 2.7.6.2
ALLOT, Section 2.8.1
CONSTANT, Section 2.7.3
CREATE, Section 2.7.1
Defining Words, *Starting FORTH*, Chapter 11
DOES>, Section 2.7.6.3
Target Compiling Defining Words, Sections 7.5, 7.8
VARIABLE, Section 2.7.2

2.7.6.2 DEFINING CODE DEFINING WORDS

The use of **;CODE** allows the user to specify the run-time behavior of a new class of words in assembler code. Every word in the new class of words will share the same piece of code: the code that defines the word's run-time behavior.

The form of a **;CODE** definition is as follows:

```
: name      {compile-time words}  ;CODE
    {run-time code} code-ending
```

Here name is the word that will be used to create new definitions of this class. The code address of all such words will be the address of the code that follows **;CODE**.

An example of a defining word using **;CODE** on an 8086 might be:

```
: VARIABLE   CREATE  2 ALLOT
   ;CODE  W INC  W INC  W PUSH  NEXT
```

The part of **VARIABLE** which creates a new dictionary entry is the phrase:

```
CREATE  2 ALLOT
```

CREATE compiles the head of a new dictionary entry, and then **2 ALLOT** makes space for the data in the parameter field. **;CODE** stops the colon compilation of **VARIABLE**, then compiles the address of a high-level word which will reset the code-field address of the word defined by **CREATE**, so that the code-field address points at the byte assembled by the first **W INC**.

The run-time code for defining words (which is shared between all words of the same type) must find the unique data in each word's parameter field. The first cell of the new word's parameter field immediately follows its code field. That is, the code field address precedes and is adjacent to the parameter field (see Fig. 2.7). For example, when the address interpreter executes a **VARIABLE**, some code in **NEXT** sets **W** to point at the code field address of that particular **VARIABLE** (not **VARIABLE**s in general, but a particular **VARIABLE**). The code field address of the **VARIABLE** points to the first byte of the shared run-time code for all **VARIABLE**s. The shared code uses the value in **W**, which points to the byte following the code field address of a particular **VARIABLE**, to find that particular **VARIABLE**'s parameter field.

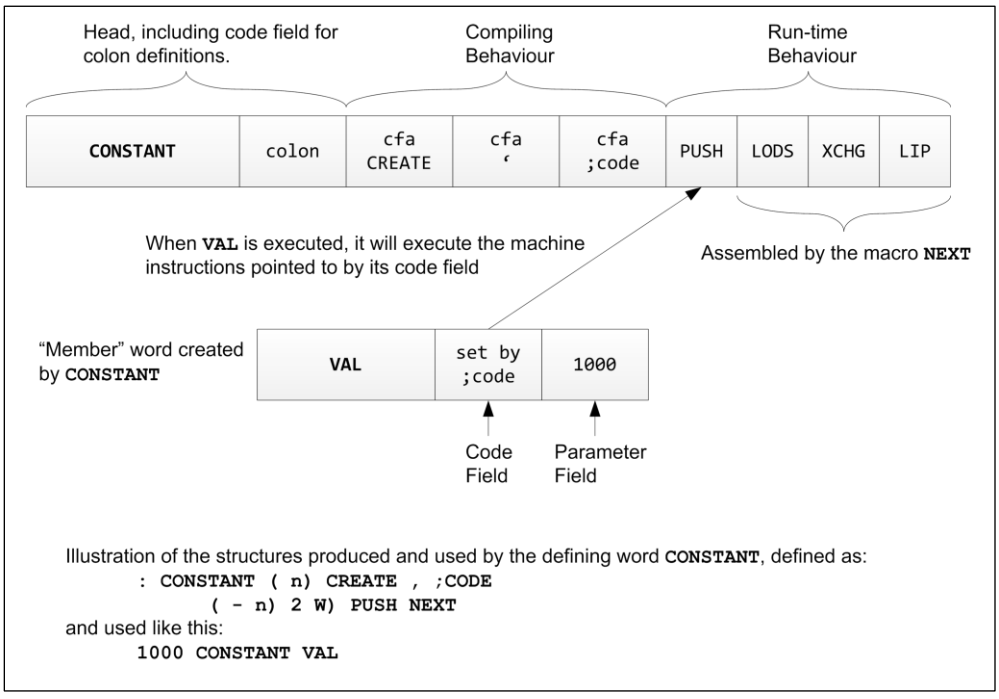


Fig. 2.7

If you wish to examine your system's definition of **VARIABLE**, please note that the run-time code is defined in the nucleus, near the beginning of the system source, while the compile-time definition is back in the defining words.

Examine several system defining words before writing your own using **;CODE**. The compile-time behavior of **;CODE** is to compile a word called **;code** which is the last word to execute in the defining word's compile-time behavior. The word **;code** stores the address of the cell following **;code**'s cell (in the defining word) into the most recent definition's code field address. The most recent definition will always be the one created by the defining word containing **;code**. Finally **;CODE**'s compile time behavior turns off the **:** compiler and enters the assembler vocabulary.

REFERENCES

Address Interpreter, Section 1.1.6
ALLOT, Section 2.8.1
Assembler Conventions, Section 6.4
CONSTANT, Section 2.7.3
CREATE, Section 2.7.1
DOES>, Section 2.7.6.3

2.7.6.3 DEFINING HIGH-LEVEL DEFINING WORDS

New defining words whose run-time behavior is specified in high-level polyFORTH may be created by using a technique similar to that used for **;CODE**. For these definitions, the word **DOES>** terminates the compile-time portion of the definition and introduces the run-time portion. The form of a **DOES>** definition is:

: name {compile-time words} **DOES>** {run-time words} **;**

As with **;CODE** above, name is a word which defines a new definition of this class. In this case, however, the run-time behavior of this class of words is described at high level.

At run time, the address of the parameter field of the word is pushed onto the stack before the run-time words are entered. This provides easy access to the parameter field.

The standard example of a **DOES>** definition is **MSG**, which types short character sequences:

```
: MSG   CREATE DOES> COUNT TYPE ;
```

Here is an example of how **MSG** is used:

```
HEX  MSG (CR)   2 C, 0D C, 0A C, DECIMAL
```

The defined word **(CR)** shares the **DOES>** definition with all other **MSG** words, but puts out its own unique character string, using the shared definition.

The values that make up the string are stored in the parameter field of a word that is defined by **MSG**. At execution time the address of the string is on the stack to serve as the parameter for **COUNT**, which returns the length of the string and its byte address as the arguments for **TYPE**. A good example of a **DOES>** word with more elaborate behavior occurs in the data base support system; if you have this option, refer to the definition of **FILE**. Other examples of **DOES>** words may be found in the high-level definition of your assembler.

DOES> works in the following way:

1. When the **:** compiler sees a **DOES>**, the **DOES>** is executed. The compile-time behavior of **DOES>** is to compile the address of a word called **;code** (see Fig. 2.8). The word **;code** resets the code field address of the latest definition (the one constructed by the defining word containing **DOES>**) to point to the cell following the compiled address of **;code**.

2. After the address of **;code**, **DOES>** compiles a subroutine call to the run-time code for **DOES>**. The compiler then proceeds to finish compiling addresses in the new defining word. The use of a subroutine call in the defining word is hardware dependent. However, all implementations of **DOES>** compile something in the defining word which will allow the run-time code for **DOES>** to find the defining word's high level code without losing the defined word's parameter field address. When the new defining word is executed, its last step is to change the code field address of the entry it creates to point to the jump to subroutine created by **DOES>** in the defining word.

3. When one of the defined words created by the new defining word is executed, the address interpreter jumps to the subroutine call in the defined word's defining word. Then the subroutine call saves the address of the cell following itself in some CPU-dependent way and jumps to the run-time code for **DOES>**. The run-time code for **DOES>** uses the address from the subroutine linkage to find the high-level run-time code of the defining word. The run-time code for **DOES>** also pushes the unchanged value of **W** onto the parameter stack to form the address of the defined word's parameter field.

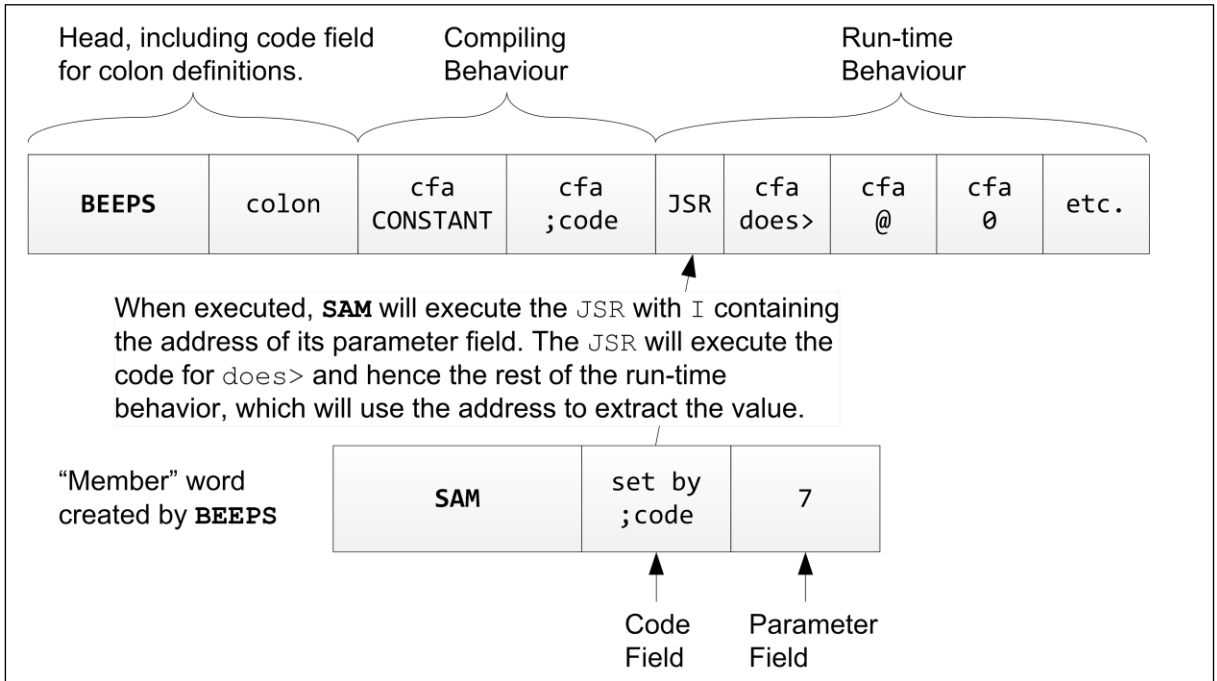


Illustration of the structures produced and used by the defining word **BEEPS**, defined as:

```
: BEEPS ( n) CONSTANT DOES> @ 0 DO 7 EMIT LOOP ;
```

BEEPS is used to define "members" of its class as follows:

```
1000 CONSTANT VAL
```

Use of **SAM** will emit server beeps. Arrows show path of execution when a defined word is executed. Note that JSR following code represents the machine code which is equivalent to a subroutine call to does>.

Fig. 2.8

REFERENCES

, and **C,**, Section 2.8.2
;CODE, Section 2.7.6.2
CONSTANT, Section 2.7.3
CREATE, Section 2.7.1
MSG, Sections 2.3.3, 3.7.3
TYPE, Sections 2.3.6.4, 3.7.3

2.8 COMPILING WORDS AND LITERALS

A compiling word stores addresses or values into the dictionary and allots space for definitions and data.

A literal is a number that is compiled directly into a definition or in some other unnamed form. Covered in this section are several Forth words for compiling literals, including **LITERAL** and **[']**.

2.8.1 ALLOTing Space in the Dictionary

The resident version of **ALLOT** reserves a specified number of bytes in the dictionary by adding to the dictionary pointer in **H**. The dictionary grows from low memory toward the "top" of the downward growing parameter stack. **ALLOT** ensures that at least 250 bytes remain between **HERE** (the next empty dictionary byte) and **'S**. (The actual number of bytes assured by **ALLOT** is system-specific, but is intended to be sufficient to support all editor operations in the remaining space.) If not enough space remains, **ALLOT** aborts the compilation by issuing the error message "Dictionary Full." If the minimum number of bytes are available, **ALLOT** adds the argument on the stack to the address of the next free dictionary byte in the user variable **H**. This addition prevents other compiling words from compiling into the portion of memory reserved by the **ALLOT**.

An example of **ALLOT**'s use to create a 200-byte array is:

```
CREATE ARRAY 200 ALLOT
```

The Target Compiler's version of **ALLOT** differs from the resident version in that it allots space in the target system's RAM, rather than the target dictionary (which is presumed to be in ROM).

REFERENCES

ALLOTting Target RAM, Section 7.3.1
CREATE, Arrays, Section 2.7.1

2.8.2 Use of , and C, to Compile Values

The word **,** (pronounced "comma") stores the item at the top of the stack into the next available dictionary location (given by **H**) and increments **H** by two bytes (four bytes on 32-bit machines).

The most common use of **,** is to set values into a table whose starting address is defined by using **CREATE**. This word creates a definition that behaves in a manner identical to **VARIABLE**, in that when the new word is executed, its address is returned. **CREATE** differs from **VARIABLE** only in that it does not allot any space. Consider this example:

```
CREATE TENS  1 , 10 , 100 , 1000 , 10000 ,
```

This establishes a table, the starting address of which is given by **TENS**, which contains powers of ten from zero through four. Indexing this table by a power of ten will give the appropriate value. A possible use might be:

```
: 10X ( n n - n)   2*  TENS + @ * ;
```

Given a single-precision number on the stack with a power of ten on top, **10X** will multiply the number by the appropriate power of ten to yield the product.

If the table will be referred to in only one instance, you may avoid the memory overhead of a **CREATE** definition by using **HERE** to provide the location. For example, a one-usage table of twos would look like:

```
HERE   2 ,  4 ,  8 ,  16 ,  32 ,  64 ,
: 10X   ( n n - n)   2*  LITERAL + @ * ;
```

The word **HERE** leaves on the stack the current dictionary pointer where the storage of data values starts. The word **LITERAL** in a definition picks up a number from the stack (the address in this example) and compiles it. When this version of **10X** is executed, the address is pushed on the stack in the same way as before.

When a single byte of data is sufficient, **C,** performs for bytes the same function that **,** performs for cells. On processors that do not tolerate odd byte addresses (*e.g.*, LSI-11), uses of **C,** must be for strings of even byte length. Note that **C,**'s name implies compiling a character-sized value in the dictionary.

REFERENCES

CODE, Sections 2.7.5, 6.1
CONSTANT, Section 2.7.3
CREATE, Section 2.7.1
LITERAL, Section 2.8.5
MSG, Section 2.3.3

2.8.3 The polyFORTH Compiler:] and [

The polyFORTH compiler uses **ALLOT**, **,** ("comma"), and occasionally **C,** to lay down addresses in the dictionary for the address interpreter to interpret. The compiler finds the addresses by using **-'** ("dash-tick") repetitively on the input stream. The address compiled for each word is the address of the word's code field address.

The compiler must handle two special cases besides address compilation. The first case occurs when numbers are included in a high-level definition. The compiler handles numbers much like the standard Forth text interpreter. When a dictionary search fails, the compiler attempts to convert the ASCII string using **NUMBER**. When conversion succeeds, the address of a special word called "cell" is compiled, followed by the number in binary. At run time, "cell" will push the binary value onto the stack. On machines that tolerate odd addresses, when the number is less than 256 the compiler saves space by compiling a special word called byte (instead of cell) followed by the least significant eight bits of the number.

When the numeric conversion fails, the conversion word aborts, printing the text not understood followed by a question mark.

The second special case occurs when a word needs to be executed at compile time by the compiler. These words are called "compiler directives." The words **IF**, **DO**, and **UNTIL** are examples of compiler directives. After the word is found in the dictionary, the compiler checks the precedence bit in the header of the dictionary entry. If the precedence bit is set, the word is executed, rather than compiled by the compiler. If the precedence bit is reset, the address of the code field address of the word is compiled. The precedence bit of a word may be set by placing the word **IMMEDIATE** after the word's definition.

The most common use of **[** and **]** is to leave compile-mode temporarily to perform some operation at compile-time. For example, suppose you wish to refer to an ASCII code in hex, in a definition containing numbers most naturally thought of in decimal:

```
: GAP ( n)    10 0 DO   [ HEX ] 0A
   EMIT  LOOP ;
```

Since the words that control **BASE** aren't **IMMEDIATE**, it is necessary to leave compile-mode to execute **HEX** to permit compilation of the hex code following. The Forth word which ends compilation is **[** (pronounced "left-bracket"). The word **[** is an **IMMEDIATE** word which performs an **EXIT** to leave the compiler and resume interpretation. This word is used in the definition of **;**.

Sometimes when Forth high-level code is necessary, but a dictionary header is not, (as in some power-up code) the word **]** is used rather than **:**. Similarly, where high-level polyFORTH is necessary but no address for **EXIT** need be compiled on the end of the definition, (as when compiling endless loops) **[** is used instead of **;**. In polyFORTH the phrases **[** and
; RECOVER are equivalent, except that many people find **; RECOVER** more readable.

Consider, for example, the response to a "restart" key on an Intel 8086:

```
HERE ] ." Break"  CR ABORT [
ASSEMBLER BEGIN   SWAP # I MOV   STI
   NEXT   0A INTERRUPT
```

The **HERE** pushes the address of the next available byte in the dictionary onto the stack. **]** enters compile mode, and compiles the **."** message followed by the references to **CR** and **ABORT**. Note that **ABORT** will abort the operation of the terminal task which initiated the interrupt (which, on the IBM-PC from which this example was taken must be **OPERATOR** due to the hardware configuration) and return control to the keyboard. Immediately following the address of **ABORT** is the assembler **MOV** instruction, followed by the rest of the code through **NEXT**. The **BEGIN** pushed the address of the **MOV** on the stack; this address and **0A** (the interrupt vector) are the arguments to **INTERRUPT**, which stores the address in the interrupt vector.

When the user presses the "Break" key, the interrupt causes a branch through the vector to the **MOV** instruction, which will set Forth's interpreter pointer I to the beginning of the high-level phrase starting with **."**. The **NEXT** at the end of the code will start execution of the high-level phrase, terminating with the **ABORT**. Since the phrase is only entered in this way (never called from another high-level word, for example) there is no need to begin it with : name and since it terminates in **ABORT** there is no need for an **EXIT** (compiled by **;**) at the end.

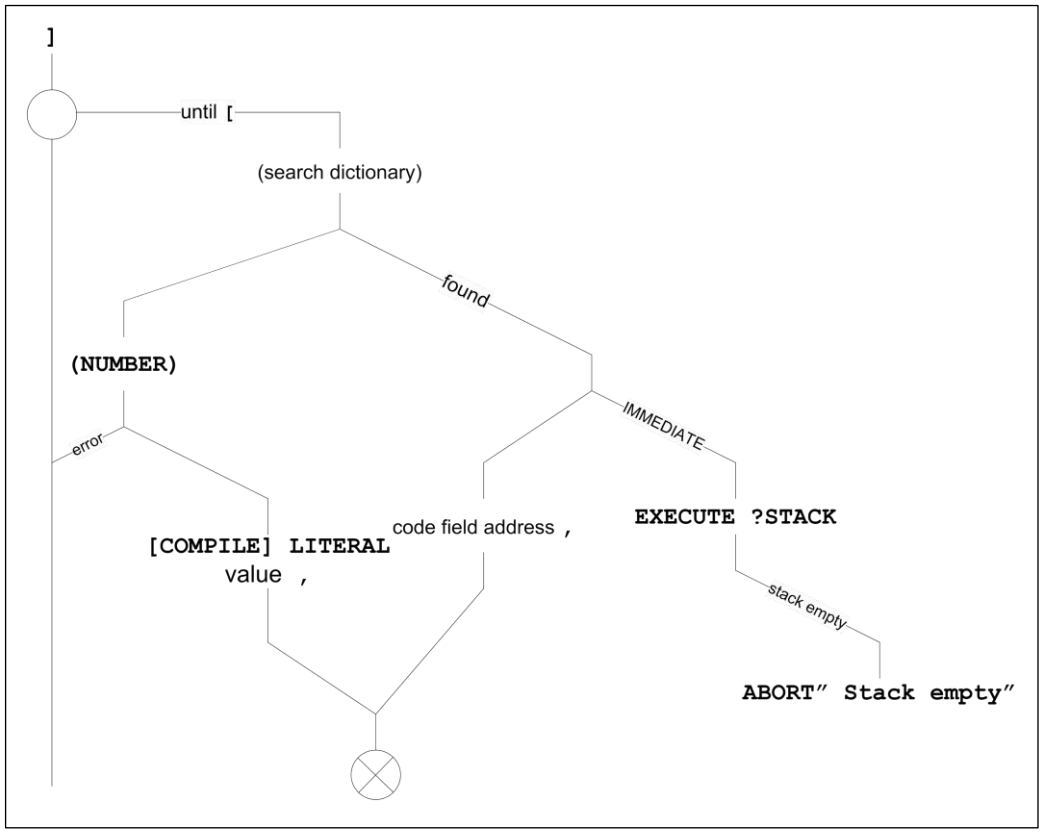


Fig. 2.9

The Forth word which begins compilation is **]** (pronounced "right-bracket"). This word contains the definition of the compiler, and is used in the definition of **:**.

Please note that it is rarely the case in a multitasking environment that you can know which task is controlling the CPU at the time an interrupt occurs. The technique used in this example is therefore appropriate only in a narrow range of applications.

REFERENCES

ABORT, Section 2.4.7
Address Interpreter, Section 1.1.6
Colon Definitions, Section 2.7.4
Compiler Directives, Section 2.8.8
Dictionary Searches, Section 2.6.1
Input Number Conversion, Section 2.5.2
Interrupts, Section 6.11

2.8.4 Use of Literals in : Definitions

When the polyFORTH compiler encounters a number in a **:** definition the number is converted to binary and compiled as a literal. The compiled form of a literal in a **:** definition occupies two cells (one cell on 32-bit machines). The first contains the address of a routine which, when executed, will push the second cell onto the stack. When Forth is compiling a definition and a number is encountered, this form is automatically compiled. There are other ways in which a literal in a definition may be generated (which will be discussed in the following section), but this is the most common situation.

On computers that tolerate odd byte addresses, a literal less than 256 will be compiled as a byte. A byte literal occupies three bytes (the address of the routine plus the actual byte). The compiler automatically determines which form to use.

Since a literal occupies two cells on 16-bit machines, a number that will be used frequently (more than six times) should be defined as a **CONSTANT** to save space. There is not much difference between the time required to execute a **CONSTANT** and a literal. The space saving is even more significant in the case of 32-bit literals versus the use of **2CONSTANT**.

REFERENCES

Explicit Literals, Section 2.8.5
Literal Addresses, Section 2.8.6

2.8.5 Explicit Literals

The word **LITERAL** compiles into a definition the number that was placed on the stack at compile time. When the definition is executed, that number will be pushed onto the stack. The compiled result of **LITERAL** is identical to that of a literal number, described in the previous section. **LITERAL** is useful for compiling a reference to an address or computed number that may be computed at compile time.

Consider the table **TENS** in the example in Section 2.8.2. If **TENS** is to be used only by the definition **10***, a more compact form would be:

```
HERE  1 ,  10 ,  100 ,  1000 ,  10000 ,
: 10* ( n n - n)   2*  LITERAL + @ * ;
```

HERE leaves a value on the stack which **LITERAL** compiles into the definition of **10***.

Both the usage and result of **10*** are identical to those of **10X** in Section 2.8.3. Here, however, instead of using **CREATE** to provide a definition which will return the address of the beginning of the table, **HERE** is used to supply it at compile time; **LITERAL** compiles this address as a literal. The net saving is six bytes of memory.

Another technique is to compile the results of complex calculations that only need to be performed once. As a trivial example, disk status information is stored in the location **DISK 2+**. A word to retrieve that information might be:

```
: STATE   [ DISK 2+ ]  LITERAL  @ ;
```

The **[** stops **:** compilation, and **]** restarts **:** compilation. During this hiatus, the words **DISK 2+** are executed, leaving on the stack the address of the status word, which, after compilation is resumed, is compiled into the definition by **LITERAL**. Time savings can be large if the calculations are in an inner loop. (Note: On 32-bit machines, the words **DISK 2+** would be **DISK 4+**.)

REFERENCES

[and **]**, Section 2.8.3
Compilation of Literals, Section 2.8.4

2.8.6 Use of ['] to Compile Literal Addresses

The word **[']** is used inside a definition to compile as a literal the address of the parameter field of the word that follows it at compile time. The most common use of **[']** is to obtain the address of either a **CONSTANT** or a **2CONSTANT** (on systems that have the 32-bit option or on 32-bit machines). Consider the following example:

```
0 500 2CONSTANT LIMITS
: RANGE   ['] LIMITS 2! ;
```

Given these definitions, the phrase:

```
0 2000 RANGE
```

resets the values of **LIMITS** to 0 and 2000. The address of the beginning of the double-precision parameter field for **LIMITS** is compiled as a literal in **RANGE** and pushed onto the stack when **RANGE** is executed to provide the address for **2!**.

2.8.7 Compiling Strings

polyFORTH provides two methods for compiling strings. The most generally useful word is **"** (pronounced "quote"). This word is used only in colon definitions. It compiles a string, which follows in the dictionary and is terminated with another quotation mark. When the word containing this string is executed, the address of the beginning of the string is pushed on the stack. The string is compiled with its count in the first byte (as is typical with strings in polyFORTH). Therefore, it will frequently be useful to use the word **COUNT** to fetch the length and address of the actual first byte of the string.

For example, lets consider a word that compares a data string whose address and length are given to a specific string, and returns 'true' if it matches:

```
: ?NO ( a n - t)   " no" 1+ -TEXT ;
```

When **?NO** is executed, **"** will push the address of the compiled string on the stack. It is incremented to the beginning of the actual text, leaving the appropriate arguments for **-TEXT** (which performs the comparison).*

Here is a word which will search for the compiled string in a longer string whose address and count are given:

```
: ?DUCK ( a n - t)   " duck"
   COUNT -MATCH  SWAP DROP ;
```

The phrase **SWAP DROP** discards the address of the first non-matching character in the first string, which is returned by **-MATCH**.

In either of the examples above you may need to allow the test string to contain an arbitrary mixture of upper and lower case characters. If so, you should either set or clear the appropriate bit in each byte of the test string to standardize on all upper or all lower case before making your comparison.

Often you will find it more convenient to specify another run-time behavior than returning just the address. You may always want the results of **COUNT**, for example, or perform some direct action such as typing the string. Or,

* Remember that **-TEXT** compares *cell by cell.* This is fine in this case, since "no" is of even length. If your string is of odd length, you must add a space to make it even length and ensure that your test string is also blank-filled if necessary. This is usually easy to do; for example, if the test string is input from the keyboard, **TEXT** will move it to a blank-filled **PAD**.

perhaps you want a terminating character other than a quotation mark. The word **STRING** will compile a string with a specified delimiter from the input stream to the dictionary. It has no run-time behavior. **STRING** is used by **"** as well as by **."**, **ABORT"** and other similar words.

STRING expects the ASCII code for the delimiter on the stack. It will compile the string terminated by that delimiter with the length of the string in the first byte. An example of **STRING**'s use is the word:

```
: MESSAGE"   >IN @  CREATE  >IN !
   34 STRING  DOES> COUNT TYPE ;
```

MESSAGE" creates messages which are named by their first word, for example:

```
MESSAGE" HELLO THIS IS FORTH! " ok
```

Executing **HELLO** will cause the computer to print:

```
HELLO THIS IS FORTH! ok
```

The way **MESSAGE** works is as follows. The phrase:

```
>IN @ CREATE  >IN !
```

Creates a definition whose name is the first word (delimited by spaces) in the string following **MESSAGE"**. It saves and restores the text interpreter pointer **>IN** so that the first word will be incorporated in the string as well as being the name of the definition. The phrase **34 STRING** then compiles the string to a terminating **"** (22_H). The resulting definition is shown in Fig. 2.10 (Section 2.8.9). The code field address of the new definition points to the code for the run-time behavior provided by **DOES>** in the definition of **MESSAGE"**. When the word **HELLO** is executed, the 21-byte string in its parameter field will be typed out. Note that this method will handle strings up to 127 bytes long.

REFERENCES

." and **ABORT"**, Section 2.3.6.5
Defining Words, Section 2.7
String Comparisons, Section 2.3.5

2.8.8 Compiler Directives

A "compiler directive" in Forth is a word that is executed at compile time, *i.e.*, during a **:** compilation. Many such words exist: **DO**; **LOOP** and **+LOOP**; **BEGIN** and **UNTIL**; **IF**, **ELSE** and **THEN**; literals; and others. It is rare that a user needs to add compiler directives; it is not difficult, but requires mastery of the use of **IMMEDIATE** and the use of **COMPILE**.

COMPILE is frequently used to lay down (in the dictionary) the address of the run-time behavior of a compiler directive. When **COMPILE** is executed by the address interpreter, **COMPILE** compiles the cell immediately following **COMPILE**'s address cell, and then adds two (or four) to the interpretive pointer so that the compiled address will not be executed. The word **IMMEDIATE** is used just after a definition. Its effect is to make the word just defined a compiler directive by setting its precedence bit (usually the high-order bit in the count field). This bit signals the compiler that the word is to be executed at compile time (when all non-immediate words are being compiled).

Consider the definition of **BEGIN**:

```
: BEGIN   HERE ;  IMMEDIATE
```

The actual behavior of **BEGIN** is identical to **HERE**. The crucial difference is that when **HERE** appears in a definition, its address is compiled; it will push the value of **H** onto the stack when the word that contains **HERE** is executed. **BEGIN**, on the other hand, compiles nothing; it pushes **H** onto the stack at compile time, to serve as the address that **UNTIL** needs to compile a conditional return to that location.

Any kind of word may be **IMMEDIATE**. Sometimes it is useful to make a **VOCABULARY IMMEDIATE**, so that it may be used to select words inside a definition.

An example of the use of **IMMEDIATE** vocabulary words is the use of **FORTH** and **HOST** in the following definition from Block 51 of the target compiler:

```
: ;CODE   FORTH COMPILE  HOST ;code
   R> DROP  ASSEMBLER  SMUDGE ; IMMEDIATE
```

Note that **ASSEMBLER** is not an **IMMEDIATE** vocabulary word, and that the **IMMEDIATE** following **;CODE**'s definition will make **;CODE** an **IMMEDIATE** word.

The definition of **IF** shown in Fig. 2.10 illustrates the definition of a compiler directive which uses **COMPILE** to lay down a run-time behavior into a **:** definition. Note that the run-time definition is a separate word.

REFERENCES

Colon Definitions, Section 2.7.4
COMPILE, Section 2.8.9
DO, **LOOP**, Program Structure Words, Sections 2.4, 6.8
Literals, Section 2.8.5
The polyFORTH Compiler, Section 2.8.3
Use of **BEGIN**, Section 2.4.1
Vocabularies, Section 3.4

2.8.9 COMPILE and [COMPILE]

Some compiler directives have only compile-time behavior (such as **BEGIN**). Other directives need to perform some actions at compile time and other actions at runtime. For example, at compile time **DO** must mark the position to which **LOOP**, **/LOOP** or **+LOOP** will return; at run-time it must push the index and limit for the loop onto the return stack.

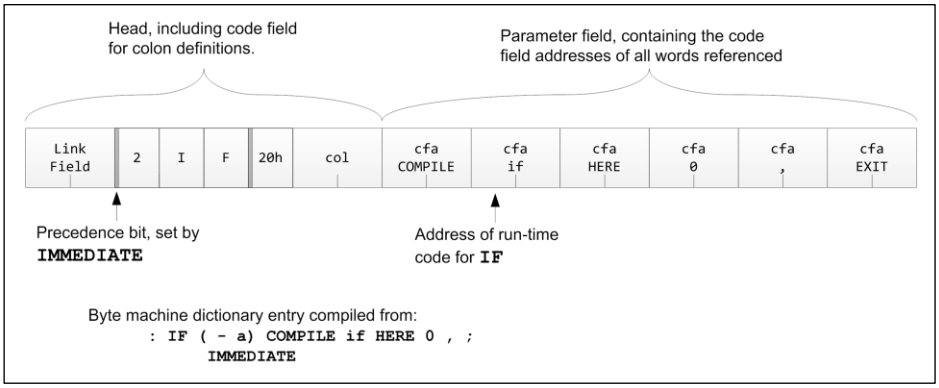


Fig. 2.10

The way these functions are managed is to define (usually with **CODE**) the run-time activity as a separate word and then to have the compile-time definition, which is **IMMEDIATE**, compile the address of the run-time code (in addition to its other activities). Fig. 2.10 shows a compiled byte-machine dictionary entry.

The word used to compile the address of the run-time code is **COMPILE**. This word is very similar to **[']**, except that whereas **[']** compiles as a literal the parameter field address of the word that follows (so that the address will be pushed onto the stack at run time), **COMPILE** lays down a pointer to the code field address, so that the word can be executed by the address interpreter.

A very similar word, **[COMPILE]**, is available to compile the addresses of words which are **IMMEDIATE** (such as compiler directives.)

When executed, this definition does the following:

1. The phrase **COMPILE if** compiles the address of the previously defined run-time code **if**.

2. **HERE** pushes an address on the stack. This address will be used subsequently by **THEN** or **ELSE** to set the address for the conditional branch run-time word **if**.

3. Finally a zero is compiled to leave space for the conditional branch address.

REFERENCES

['], Section 2.8.6
[and **]**, Section 2.8.3
Compiler Directives, Section 2.8.8

2.9 FORTH-83 STANDARD COMPATIBILITY

Efforts to standardize Forth have been under way since the late 1970s. An organization called the Forth Standards Team (or FST) published a widely recognized standard in 1980, called FORTH-79. In 1983 this was extensively revised, and the result, called FORTH-83, is now the most widely accepted standard.

FORTH, Inc. has been a participant in these standardization efforts since the beginning. As shipped, polyFORTH ISD-4 is extremely close to the FORTH-83 standard, however, there are several significant incompatibilities:

1. polyFORTH adheres to a consistent usage wherein all addresses of Forth words passed on the stack are parameter field addresses. In FORTH-83, some functions return and use parameter field addresses, while others (**'**, **[']**, and **EXECUTE**) return and use *code addresses*.

2. Finite loop structures and related words behave differently in FORTH-83 in order to provide a full 65K range in **LOOP** and **+LOOP** on 16-bit machines. Specifically, inside the loop the return stack parameters are modified, so that **I** and **J** references *compute* the index values (rather than simply copy them). There is no equivalent of **I'**. Moreover, **LEAVE** exits the loop at once (rather than adjusting loop parameters so that the loop will terminate on this cycle). The FORTH-83 versions of these words compile full branch addresses instead of byte offsets, costing an extra byte but preventing problems for long definitions. Also, **DO** compiles the loop exit address for use by **LEAVE**. In summary, the FORTH-83 versions are slightly slower and longer, but more general.

3. In FORTH-83, integer division is *floored*, meaning that the division operators round, rather than truncate, when there is a remainder. This provides a continuous function through zero, at some cost in performance. The actual performance difference depends upon the CPU.

3.0 SYSTEM FUNCTIONS

This section describes words used to load, organize, and manage polyFORTH applications, as well as the standard system devices (disk, terminal, and clock).

3.1 VECTORED ROUTINES

Since there are several system functions users often wish to change (because of changing hardware or application requirements) without recompiling the nucleus, pF/x provides "execution vectors" containing the addresses of current versions of these functions. There are two groups of vectored routines: those controlling system-wide functions, and those controlling terminal-dependent functions (*i.e.*, those whose behavior differs between different kinds of CRT or between keyboard/display and printer). For each vectored function, there are at least three Forth words: the function itself (which performs a **@EXECUTE** on the vector), the vector itself, and at least one routine to be executed.

The following table summarizes the vectored routines controlled on a system-wide basis:

Function	Vector	Primitive	Description
BLOCK	**'BLOCK**	**(BLOCK)**	Returns the address of a specified block. See references for details.
BUFFER	**'BUFFER**	**(BUFFER)**	Returns the address of an available buffer, identified as containing a specified block. See the references.
CREATE	**'CREATE**	**(CREATE)**	Creates a dictionary entry. See the references for additional versions.
Function	**Vector**	**Primitive**	**Description**
NUMBER	**'NUMBER**	**(NUMBER)**	Converts a string at a given address to binary on the stack. See the references.

The following table summarizes the routines vectored through user variables for differing task-specific functions. **'IDLE** is used by all tasks; the remainder control CRTs and printers and are relevant only for **TERMINAL** tasks. See the section references below for further details.

Function	Vector	Primitive	Description
ABORT	**'IDLE**	**QUIT**	"Idle" behavior for a task—the routine it executes following an abort.
TYPE	**'TYPE**	**(TYPE)**	Types a string, given address and count.
EXPECT	**'EXPECT**	**(EXPECT)**	Accepts a string, given address and count.
CR	**'CR**	**(CR)**	Performs a new-line function.
PAGE	**'PAGE**	**(PAGE)**	Clears the screen or issues a form-feed on the printer.
TAB	**'TAB**	**(TAB)**	Positions a terminal's cursor.
MARK	**'MARK**	**(MARK)**	Types highlighted text.
CLEAN	**'CLEAN**	**(CLEAN)**	Clears to end of line.

TYPE and **EXPECT** require some further discussion. Both words expect an address and count on the stack, and both store these parameters in the user variables **PTR** and **CTR** respectively. **EXPECT** negates its count so **CTR** starts negative and will count up to zero. **EXPECT** also sets the user variable **SPAN** to zero where it will be incremented, to leave a record of how many characters were received.

You may write custom versions of vectored routines—for example, to handle a special kind of disk. To substitute a user-written routine in a vector, follow the examples given for **CREATE** in the section referenced below. In addition to those additional versions of **CREATE**, the system electives include error-handling versions of **BLOCK** and **BUFFER** (Block 48).

REFERENCES

BLOCK, Sections 3.2.2, 3.2.6, 3.2.8
BUFFER, Sections 3.2.3, 3.2.6, 3.2.8
CREATE, Section 2.7.1
EXPECT, Sections 3.7.1, 3.7.2
NUMBER, Section 2.6.2
Support of Special Terminal Functions, Section 3.7.5
TYPE, Sections 3.7.3, 3.7.4
User Variables, Section 4.6
Vectored Execution, Section 2.4.8

3.2 THE DISK DRIVER

This section discusses the words used to access and manage disk blocks and block buffers in Forth.

3.2.1 Overview of polyFORTH Disk Access

The polyFORTH disk access scheme is intended to be simple and to require a minimum of effort to use. The polyFORTH disk driver makes data on disk directly accessible to other Forth words by placing disk data into a buffer in memory, returning on the stack the address of the buffer containing the requested data. Forth routines thus access disk data using the same techniques as with other memory accesses. Since disk data always appears to be in memory, this disk access scheme is a form of "virtual memory" for program source and data storage. Basic principals of disk usage in Forth are discussed in *Starting FORTH*, Chapter 10. If you are a newcomer to Forth, you should review this material before proceeding further.

Another consideration in the design of the polyFORTH disk driver is to make disk access as fast as possible. For this reason, data is read from disk to memory or written from memory to disk only when necessary, because disk operations are very slow compared to memory operations.

The disk is partitioned into 1024-byte data areas called "blocks." This standard unit has proven over many years to be a useful increment of mass storage. As a unit of source text, for example, it contains an amount of source which can be comfortably displayed on a CRT screen; as the basis for a data base system, it contains a useful number of records of typical sizes.

Each block is addressed by a block number. On native polyFORTH systems the block number is a fixed function of the block's physical position on the disk. Absolute addressing of the disk both speeds the driver's execution and simplifies the knowledge a programmer must have to use the disk. Absolute addressing also eliminates most of the need for disk directories and indexes.

A program ensures a block is in memory in a "block-buffer" by executing the word **BLOCK**. **BLOCK** uses a block number from the stack and returns the address of the first byte of that block in memory. For example:

```
9 BLOCK U.
```

will return an address such as:

```
46844 ok
```

where 46844 is the address of the first byte of the buffer containing Block 9.* If a block is already in memory, **BLOCK** will not re-read it from disk. Although **BLOCK** uses a disk read to get data if it is not already in memory, **BLOCK** is not merely a read command.

If **BLOCK** must read a requested block from disk, it uses **BUFFER** to select a buffer to put it in. **BUFFER** frees a block buffer, writing the block buffer's previous data to disk if the data is marked as having been changed since the block was read into memory.

BUFFER takes a block number on the stack and returns the address of the first byte of an available block buffer, to which this block number has been assigned. For example:

```
127 BUFFER U.
```

will get a block buffer, assign the block number **127** to the buffer, and then type out the address of the buffer's first byte:

```
36084 ok
```

BUFFER will always select the Least Recently Used buffer. Use of this "LRU" buffer management algorithm in polyFORTH can substantially improve throughput in a disk-intensive multi-user application.

* The command **U.** was used to type out the number as an unsigned integer because block buffers are usually in high memory. Otherwise, this address would appear as a negative number.

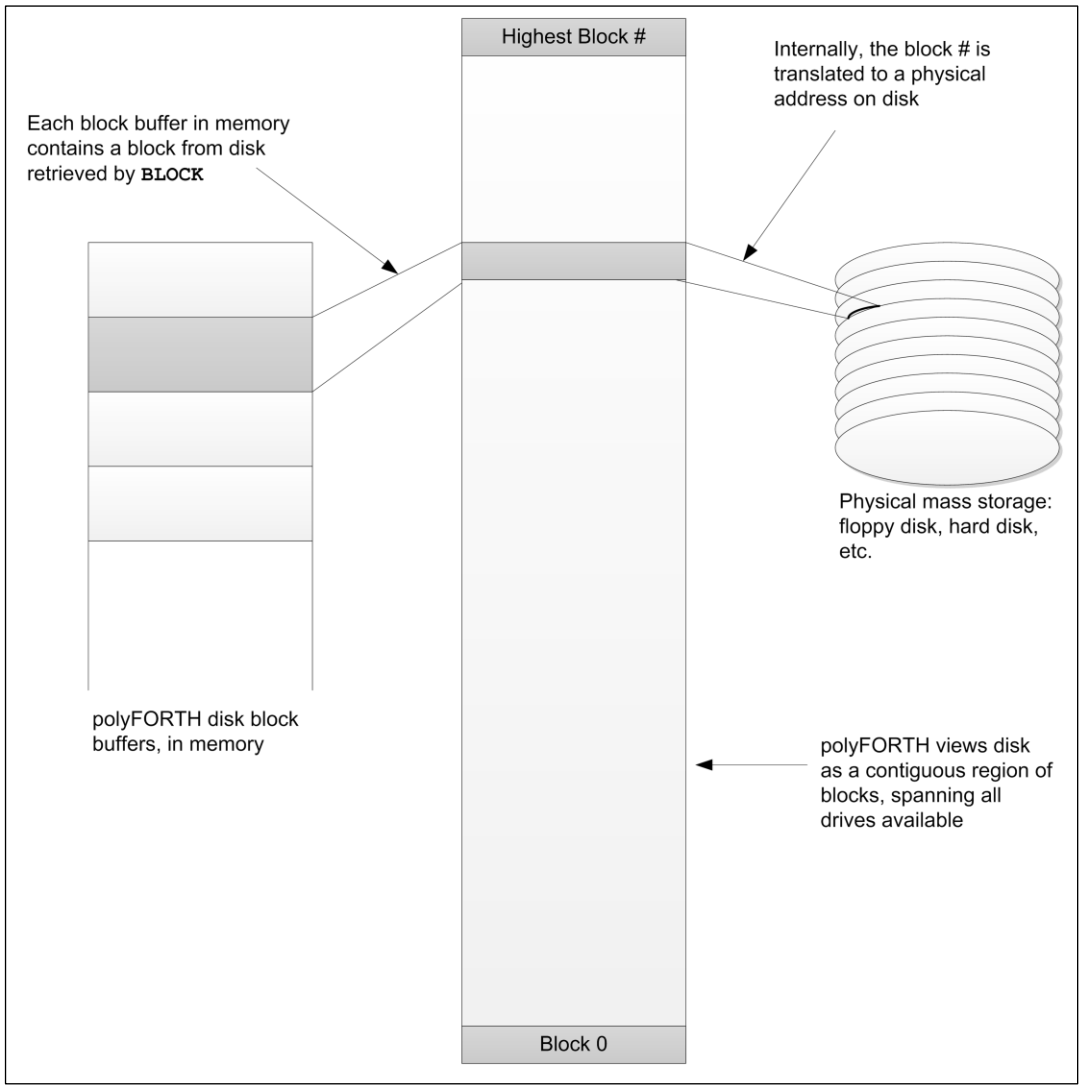


Fig. 3.1

Although **BUFFER** may write a block if necessary, it will *not* read data from disk. When **BUFFER** is called by **BLOCK** to assign a buffer, **BLOCK** will follow the selection of a buffer by actually reading the requested block from disk into the buffer.

The following example displays an array of 100 16-bit signed integers starting at the beginning of Block 1000, five numbers per line:

```
: SHOW   100 0 DO  I 5 MOD 0= IF CR
   THEN 1000 BLOCK I 2* + ?  LOOP ;
```

The phrase **I 2* +** converts the loop counter from 16-bit cells to bytes, as internal addresses are always byte addresses, and adds the resulting byte offset to the address of the block buffer returned by **BLOCK**. The word **?** fetches and types out the cell at that address.

BUFFER may be used directly in situations where no data from the disk need be saved. Examples include initializing a region of disk to a default value such as zero or a high-speed data acquisition routine writing incoming values directly to disk from a memory array 1024 bytes at a time.

You may determine the number of block buffers in your system by typing:

```
NB ?                    4 ok
```

As shipped, polyFORTH returns 4 (8 on 32-bit systems), the default number of block buffers. The number of buffers may be changed easily. Applications with several users using disk heavily may run slightly faster with more buffers. Your *CPU Supplement* will give details on changing the size of the buffer pool.

The command **UPDATE** marks the data in a buffer as having been changed, so that it will be written to disk when the buffer must be used for another block. **UPDATE** works on the most recently referenced buffer, so it must be used immediately after the operation that modifies the data.*

The following example uses **BUFFER** to clear a range of blocks to zero:

```
: ZEROS ( f l)   1+ SWAP DO  I BUFFER
   1024 ERASE  UPDATE  LOOP ;
```

To take another example, assume that an application has defined **A/D** to read a value from an **A/D** converter. To record up to 512 samples in Block 700, you could use:

```
: SAMPLES ( n)   512 MIN 0 DO  A/D
   700 BLOCK I 2* + ! UPDATE  LOOP ;
```

In this example, the phrase **512 MIN** "clips" the specified number of samples at 512 (the next example will show how to record data in a range of blocks). As in the example of **SHOW** above, the phrase **I 2*** converts the loop counter (in samples) to a byte offset added to the address of the start of the block, returned by **BLOCK**. **BUFFER** cannot be used in this case, as we must preserve previous samples written in the block.

Because **BLOCK** maps disk into memory, "virtual memory" applications are simple. The first step of a virtual memory application is to write a word transforming application addresses to physical addresses. For a virtual byte array, such a definition is:

```
: VIRTUAL ( v - a)   1024 /MOD  250 +
   BLOCK + ;
```

The 1024 is the number of bytes per disk block and 250 is the block number where the virtual array starts. See the reference below (virtual arrays) for a detailed explanation of this technique.

Fetch and store operations for this virtual memory scheme are defined as:

```
: V@ ( v - n)   VIRTUAL C@ ;

: V! ( n v)   VIRTUAL C!  UPDATE ;
```

BLOCK does not normally perform any error checking or retries at the primitive level, because an appropriate error response is fundamentally application-dependent. Some applications processing critical data in non-real-time (*e.g.*, accounting applications) should attempt retries,* and if these fail, stop with an error message

* In polyFORTH's text editor, the commands changing a block being edited perform **UPDATE** automatically. Similarly, the **DISKING** utility, Target Compiler, and other standard utilities serve as good examples of how **UPDATE** should be used in applications.

* Many "intelligent" disk controllers perform retries automatically. On these, as well as on versions of polyFORTH running as a "guest" under another operating system, there is nothing to be gained by attempting retries from within polyFORTH.

identifying bad data. Other applications running continuously at a constant sampling rate (*e.g.*, data loggers) cannot afford to wait, and should simply log errors.

The low-level routines save error data obtained from the disk controller in the cell at **DISK 2+** (on 32-bit machines at **DISK 4+**) so you may add appropriate error handling code in your application. By making error response completely user-programmable, polyFORTH makes it possible to ensure an appropriate response in any application environment. A standard error handling facility is provided in Block 48. This may serve as an example for application use. This facility retries failed disk operations a fixed number of times before aborting with an error message.

Native versions of polyFORTH do not put source blocks in named files. The block structure suffices for polyFORTH source and for simple disk-based applications. A data base support option supporting named files consisting of ranges of blocks is included with polyFORTH. This provides programming tools to define and maintain more complex data bases on disk.

Often the hardware provides several disk drives with removable media. In this situation, it is often convenient for a user to work as though the disk's first block was block number zero, even though the disk is on Drive 2, for example. By convention, therefore, when a block number is processed by **BLOCK** or **BUFFER**, an offset is added to generate the physical block number. The offset is set by the word **PART**, which uses a disk drive number (disks are numbered starting with Drive 0, as in most hardware). For example, the phrase:

```
1 PART
```

sets the user's value for the offset so when the user types **9 LIST**, the block listed will be from Drive 1.

Certain variables and arrays exist in memory to allow the system to manage disk access. These are available to be used by the programmer. Variable names for 32-bit systems are given in parentheses.

Word Description

OFFSET A user variable containing the physical block number of a user's first block number. Set by **PART** and **UNIT**.

NB A system variable containing the system's number of block buffers. Typical values in **NB** range from two to eight. Multi-user systems tend to run faster with more block buffers, if there are more users than block buffers.

Word Description

NB 2+ (**NB 4+** on 32-bit systems.) Contains the address of the first byte of the block buffer lowest in memory. In polyFORTH, the traditional place for block buffers is in high memory, above application memory and all user partitions.

DISK A facility variable used to ensure only one task at a time may have access to the block buffers. **DISK** is set by **GET** and **RELEASE** in **BLOCK** and **BUFFER**. **DISK** is also used by the disk interrupt routine to find the user area of the task using the disk.

DISK 2+ (**DISK 4+** on 32-bit systems.) Contains hardware status information, and is often only a copy of the disk controller's status register(s) after the last read or write operation. On many systems, **DISK 2+** will be zero if no errors occurred. Some disk controllers return several bytes of status information. Check your *CPU Supplement* for the handling of your disk.

'BUFFER 2+ (**'BUFFER 4+** on 32-bit systems.) Contains the absolute block number most recently written to disk, set by **BUFFER**.

PREV Contains the address of the descriptor of the buffer most recently used by **BLOCK** or **BUFFER**. This value is used by **UPDATE** to mark the buffer descriptor to indicate a block buffer will need to be written to disk, as well as to start the search for a block in memory or a buffer to be used.

REFERENCES

?UPDATED, Section 3.2.3
32-Bit Block Number Conventions, Section 3.2.7
Data Base Support, Section 8.0
Facility Variables, Section 4.7
User Variables, Section 4.6
Virtual Arrays, Section 1.2.2

3.2.2 Using BLOCK for Disk Access

BLOCK is the basic disk access word in Forth. **BLOCK** moves data between disk and memory, forming a "window" in memory, which shows the data on disk. While **BLOCK** contains a read operation, **BLOCK** only reads from disk when the data is not available in memory. Blocks are only written to disk when an **UPDATE**d block buffer must be overwritten to free a block buffer.

BLOCK expects a block number on the stack, and returns on the stack the address of the first byte of the data in memory. For example:

```
65  250 BLOCK 3 + C! UPDATE
```

stores an "A" (decimal 65) in the fourth byte of Block 250. **UPDATE** marks the block buffer containing Block 250 "updated" so that it will be written to disk before the buffer is re-used. If the system goes down before the buffer is re-used, the data will not be saved to disk. The word **FLUSH** writes all updated buffers to disk. If no changes are made to the block buffer (when fetching data, for example) the **UPDATE** should be omitted.

The following is a detailed discussion of the mechanics of **BLOCK**. Please refer to the Dijkstra diagram in Fig. 3.2 and to your system listing.

The word **BLOCK** performs all high-level functions and calls the hardware level routine **(BLOCK)**. **BLOCK** adds **OFFSET** to the block number on the stack, gains exclusive access to the block buffers by the phrase **DISK GET**, and then vectors execution through the variable **'BLOCK** to the hardware-dependent routine **(BLOCK)**.

The hardware-dependent portion of **BLOCK** is vectored through **'BLOCK** so a new version for a different disk controller can be inserted while polyFORTH is running, by a phrase such as:

```
' <BLOCK> 'BLOCK !
```

where **<BLOCK>** is the name of the new version of **(BLOCK)**.

(BLOCK) first uses the word **?ABSENT** to determine whether the block is already available among the buffers. If the block is already in memory, a superfluous disk read should not be performed. Therefore, **?ABSENT** scans **PREV**, checking the block numbers of all blocks in memory. If the block is already in memory, **?ABSENT** takes the place of **(BLOCK)**. If the block is in memory, **?ABSENT** uses the block number on the stack, leaves the buffer address on the stack, marks the buffer as the most recently accessed (by moving the number of the block buffer into **PREV**), and then exits from **(BLOCK)** back into **BLOCK**, which then releases **DISK**. If the block is not present, **?ABSENT** is invisible: it does not affect the stack, does not mark the buffer, and returns normally to **(BLOCK)**.

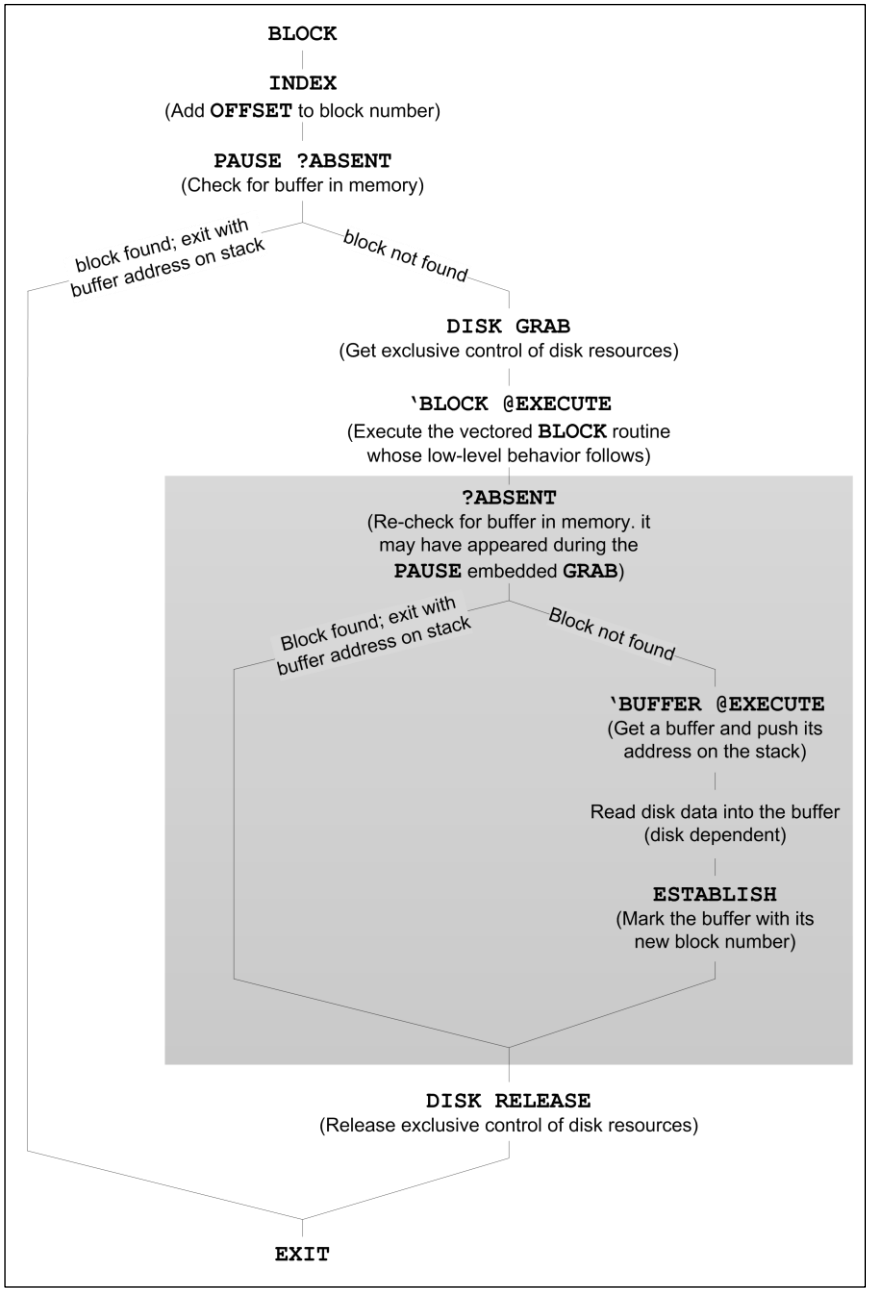


Fig. 3.2

Dijkstra diagram of **BLOCK**. The shaded area describes the behavior of **(BLOCK)**, containing the hardware-dependent functions. Custom definitions used with **'BLOCK** should call **(BLOCK)** or its equivalent.

The next step of **(BLOCK)** is to vector execution through **'BUFFER** to **(BUFFER)**. As with **(BLOCK)**, **(BUFFER)** is hardware-dependent.

(BUFFER) finds an old block buffer, frees it (writing the data to disk if necessary), and leaves the buffer address of the freed block buffer on the stack. See the next section for a detailed discussion of **(BUFFER)**.

The next step of **(BLOCK)** is totally hardware-dependent: it reads the desired block whose block number is on top of the stack into the buffer whose address is just below the top of the stack. Note that this step is only performed when **BLOCK** already knows, via **?ABSENT**, that the data must be read from disk, and is not already in memory.

Usually the disk addressing calculation word is shared by this read operation and the write operation in **(BUFFER)**. The block number and buffer addresses are left on the stack by the address calculation word.

The final step of **(BLOCK)** is the word **ESTABLISH**. **ESTABLISH** makes the oldest buffer (at the end of the buffer chain) into the newest buffer (by storing the address of the buffer descriptor into **PREV**), and marks the new buffer with the block number of the second stack entry. **ESTABLISH** uses a block number and buffer address, and leaves only the buffer address. **(BLOCK)** then returns to **BLOCK**, which gives up control of the disk system by the phrase **DISK RELEASE**.

REFERENCES

(BUFFER), Section 3.2.3
Adding a Disk Driver, Section 3.2.8
Virtual Arrays, Sections 1.2.6, 3.2.1

3.2.3 Using BUFFER to Select a Block Buffer

The word **BUFFER** is called by **BLOCK** to obtain a buffer when a block must be read from disk. It may also be called in any application needing "write-only" access to a block buffer. Examples of the latter include initializing an entire block to some value (*e.g.*, zeroes or spaces) or copying a block's worth of data from a memory array. **BUFFER** should not be used when writing individual bytes or cells into a buffer when there is input or output (and hence multitask access) between writes.

BUFFER expects a block number on the stack and returns the address of a buffer marked as containing that block. The contents of the buffer are undefined. **BUFFER** will write the selected buffer's data to disk if necessary to free the buffer for this use.

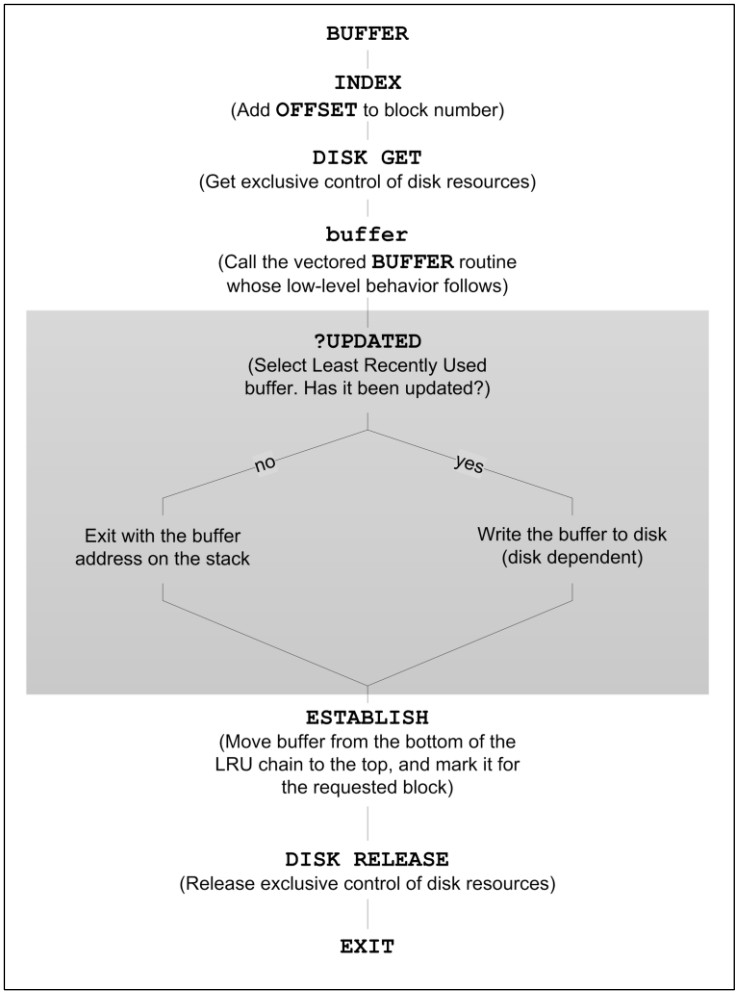


Fig. 3.3

Dijkstra diagram of **BUFFER**. The shaded area describes the behavior of **(BUFFER)**, containing the hardware-dependent functions. Custom definitions used with **'BUFFER** should call **(BUFFER)** or its equivalent.

BUFFER is a shell routine vectoring execution through the system variable **'BUFFER** to the hardware-dependent routine **(BUFFER)**. This vectoring enables new disk drivers to be installed while polyFORTH is running. The hardware-dependent function performed in **(BUFFER)** is a disk write; disk reads are performed by a similarly vectored hardware-dependent portion of **BLOCK**. This vectoring may also be used to modify other aspects of disk write behavior, such as adding an error checking function like the one in Block 48.

If the new version of **(BUFFER)** is called **<BUFFER>**, the installation of **<BUFFER>** would be achieved by the phrase:

```
' <BUFFER>  'BUFFER  !
```

The logical flow within **BUFFER** is diagrammed in Fig. 3.3. The principal routines used by **BUFFER** are:

Word	Stack	Function
?UPDATED	(- a) or (- a d)	Selects the Least Recently Used buffer, by an algorithm given below, and checks its "update bit" to see whether it has been marked as changed. Returns the address of the selected buffer, now marked as "empty," and the block number in case a write is to be performed. **?UPDATED** must be followed by hardware-dependent words to perform the write. If a write is not required **?UPDATED** will exit directly from the calling definition, which is normally **(BUFFER)**.

ESTABLISH (d a - a) Moves the least recently used buffer, selected by **?UPDATED** to the top of the LRU chain (*i.e.*, making it the most recently referenced buffer), and marks it as containing the requested block.

Block numbers used internally by these words are 32 bits long, to allow for disk controllers handling more than 32,768 blocks; see the references below for a more detailed discussion of 32-bit block numbers.

The method used for selecting a block buffer is called the "LRU Buffer Manager" because it selects the Least Recently Used buffer to be over-written. This attempts to minimize the number of physical disk accesses performed, assuming the least recently referenced block is the least likely to be needed again soon by any of the currently running tasks. Use of this algorithm can significantly improve performance in a disk-intensive application. A general statement of this algorithm is as follows:

1. **BLOCK** always checks whether a block is in memory before requesting a buffer preparatory to performing a read operation. The search for the desired block always starts with the *most* recently referenced buffer, assuming it is the most likely target, and proceeds through the buffers looking at successively older buffers.

2. **BUFFER** will always select the *least* recently referenced buffer, assuming that it is the one of least interest.

The implementation of this algorithm maintains the necessary information about each buffer in a table whose starting address is returned by the word **PREV**. The structure of this table is represented in Fig. 3.4 (details of the organization of this table are system-dependent; refer to your listing and *CPU Supplement*).

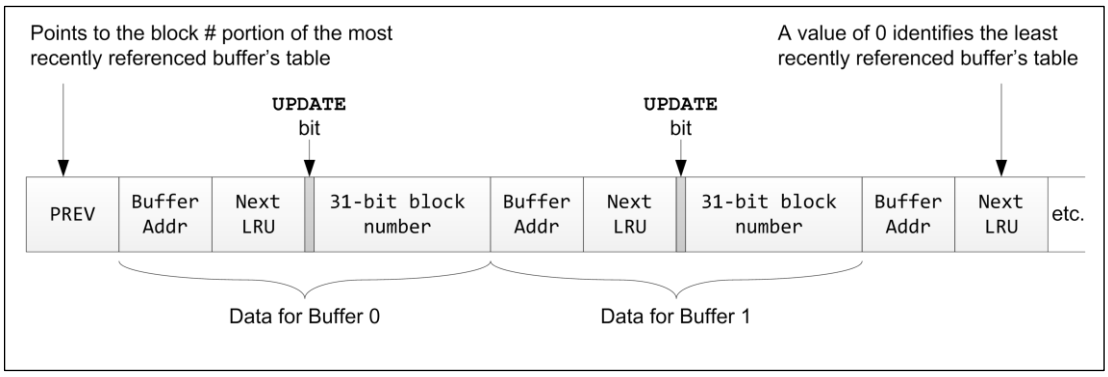


Fig. 3.4

Diagram showing details of the first portion of a **PREV** table in a typical polyFORTH implementation. There are as many sets of buffer description data (8 bytes per buffer in this example) as there are buffers. The exact structure of this table, including the order of items within a buffer descriptor, is system-dependent.

PREV contains the address of the most recently referenced buffer's data. Within that buffer's data packet is the address of the next most recently referenced buffer's data. The least recently referenced buffer is identified by a zero in this location. Any reference to a buffer through **BLOCK** or **BUFFER** will cause that buffer to be moved to the top of the chain.

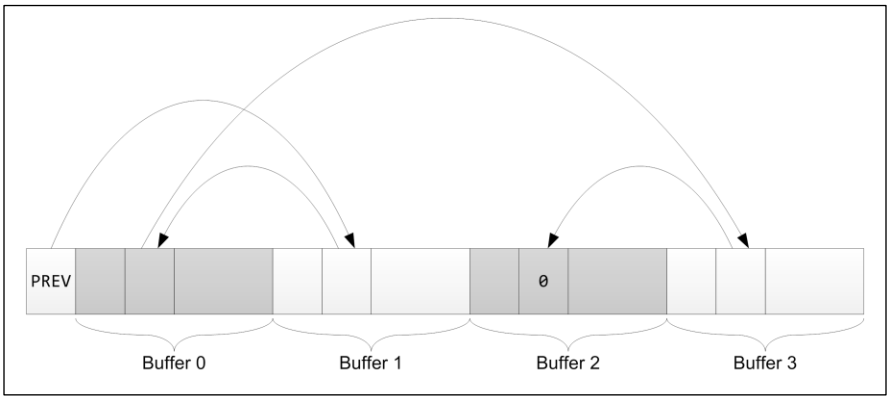


Fig. 3.5

Example showing the use of the LRU buffer pointers in a typical implementation with four buffers. In this case a search for a block in memory will start with Buffer 1, the most recently referenced buffer (whose descriptor is pointed to by **PREV**), and proceed through 0, 3, and 2. **BUFFER** would select Buffer 2, the least recently referenced buffer.

REFERENCES

32-Bit Block Number Conventions, Section 3.2.7

3.2.4 Marking Buffers Updated with UPDATE

When the contents of a block buffer have been changed, the change must be followed by the word **UPDATE** to ensure the change will be recorded on disk. **UPDATE** does not write data to disk. **UPDATE** marks a block buffer. When the word **BUFFER** frees a marked block buffer for re-use, it must write the marked block buffer's contents to disk. **UPDATE** requires no arguments because **UPDATE** marks whatever block buffer was most recently requested by either **BLOCK** or **BUFFER**. This buffer is the one referenced in **PREV**. The following example illustrates the use of **UPDATE** in a virtual storage array (the array begins in Block 250). Note that **UPDATE** is used only in the "store" operation.

```
: VIRTUAL ( v - a)   1024 /MOD  250 +
   BLOCK + ;

: V! ( n v)  VIRTUAL C!  UPDATE ;

: V@ ( v- n)   VIRTUAL C@ ;
```

In the example above, **VIRTUAL** converts an address for the virtual array into an address pointing into a block buffer in memory. The phrase:

```
1024 /MOD 250 +
```

produces a block number on top of the stack, with an offset into the block just underneath. The word **BLOCK** uses the block number on top of the stack and leaves the address of the first byte of the block buffer containing a 1024-byte segment of the byte array. The plus adds the block buffer address to the offset to produce an absolute address into the correct block buffer. The operation of **V@** is straightforward, but note the **UPDATE** in **V!**. Without the word **UPDATE**, the changed block buffer data would never be written out to disk when the block buffer is reused.

UPDATE sets the sign bit of the block number in the buffer descriptor whose address is in **PREV**. The sign bit is the update bit of the corresponding block buffer. Because the data in the buffer has changed, if the buffer must be freed, the data must be written to disk to prevent data loss.

No words which enter the multitasking loop should be allowed between **BLOCK** and **UPDATE** or between **BUFFER** and **UPDATE**. If this is allowed, another task may use **BLOCK** or **BUFFER** to change which block buffer is the most recently used block buffer. Similarly, a block buffer address should not be used after entering the multitasking loop because another task may have changed the buffer's contents. **BLOCK** and **BUFFER** enter the multitasking loop only before they access the disk, so **UPDATE** can be performed before another task gains control of the disk. Words entering the multitasking loop are: words containing **PAUSE** or **STOP**, almost all input/output words, and any assembly code word ending in **WAIT**. If the CPU enters the multitasking loop before doing an **UPDATE**, some other task may overwrite the data in the buffer, or reset which buffer is most recent by changing the buffer pointer in **PREV**.

For example, if the word **V!** from the virtual array was redefined for debugging as:

```
: V! ( n v)   VIRTUAL  DUP CR ?  !  UPDATE ;
```

the results would be unpredictable in a multitasking system because the words **?** (equivalent to a **@ .**) and **CR** both use **TYPE**. **TYPE** is an output word that enters the multitasking loop.

A possible way to correct such an error is to place the phrase **DISK GET** immediately after the occurrence of **BLOCK** or **BUFFER**, and **DISK RELEASE** immediately after the occurrence of **UPDATE**.

For example, to correct the "debugging" version of **V!** given above:

```
: V! ( n v)   VIRTUAL  DISK GET  DUP CR ?
   !  UPDATE  DISK RELEASE ;
```

REFERENCES

GET and **RELEASE**, Section 4.7

3.2.5 Other Buffer Management Words

Several other words are available to aid in buffer management. These are:

Word Stack Function

IDENTIFY (n) Marks the most recently used buffer with the block number on top of the stack, after adding the contents of **OFFSET**. **IDENTIFY** is used in **COPY.**

FLUSH Ensures all updated buffers are written to disk by freeing all of the buffers, using **BUFFER** in a **DO LOOP**.

SAVE-BUFFERS Writes all **UPDATE**d buffers to disk, leaving them still in the buffers with their **UPDATE** flags cleared.

EMPTY-BUFFERS Erases all of the block buffers to disk without saving them. **EMPTY-BUFFERS** works by resetting all the update bits in **PREV** and performs a **FLUSH** to free all the buffers.

REFERENCES

BUFFER, Section 3.2.3
COPY, Section 5.1.8
FORTH-83 Standard Compatibility, Section 2.9
OFFSET, Section 3.2.1

3.2.6 Disk Error Checking

Sometimes the disk hardware generates errors. The appropriate response to a disk error depends upon the application. A disk diagnostic, for example, may only want to count errors. An application where background tasks (which cannot issue error messages) use disk may set a flag to be detected by a terminal task. Two predefined forms of error checking exist in polyFORTH: The word **SWEEP** (which resides in the **DISKING** utility

and checks for read errors), and the disk error handling block. See references below for further discussion of **SWEEP**.

The disk error handling block has facilities for finding and correcting hardware errors. The redefinitions available in this block of the hardware routines for **BLOCK** and **BUFFER** check the error flags in **DISK 2+** (**DISK 4+** on 32-bit processors) and abort with a "Read error" or "Write error." Loading the disk error handling block installs the new definitions. If you wish to use this feature, the block must be loaded by Block 9.

The error handling routines use **(BLOCK)** and **(BUFFER)**, and are device-independent to a degree; however, any changes to the buffer management code will have to be reflected in the disk retry code.

Word	Function
/BLOCK/ and **/BUFFER/**	Behaves like **(BLOCK)** and **(BUFFER)** except that **/BLOCK/** and **/BUFFER/** abort with an error message if too many retries occur. **/BLOCK/** and **/BUFFER/** are designed to abort safely, so **/BLOCK/** does not leave the block in memory, and **/BUFFER/** does not free a buffer, but does remove the update bit. A background task can't execute **ABORT"** to issue an error message. If background tasks are accessing disk, **/BLOCK/** and **/BUFFER/** must be changed to return the buffer address without aborting and/or record the error occurrence so, if desired, a terminal task might issue an error message.

Many disk controllers don't report the sorts of "soft" errors retries may correct, and others do retries themselves. For controllers reporting soft errors, polyFORTH's low-level routines automatically perform retries. As a result, these routines will only detect un-recoverable errors.

For diagnostic purposes you may find it convenient to temporarily define and install a version of **(BLOCK)** that displays disk controller status data from **DISK 2+** (**DISK 4+** on 32-bit machines).

```
: [BLOCK] ( n - a)    (BLOCK)  DISK 2+ @
   BASE @  SWAP HEX U.  BASE ! ;
   ' [BLOCK]  'BLOCK !
```

REFERENCES

32-bit Block Number Conventions, Section 3.2.7
BACKGROUND Tasks, Sections 4.2, 4.3, 4.4
DISKING Utility, Section 5.3
Disk Diagnostics, Section 5.3.4
SWEEP, Section 5.3.4

3.2.7 32-Bit Block Number Conventions*

Some disk systems have capacities in excess of the range 15-bit block numbers can handle (up to 32,767). An IBM-AT, for example, can support a full 30 Mb hard disk plus a 1.2 Mb floppy, plus a 320 Kb floppy using 15-bit block numbers, but could not add a second 1.2 Mb floppy or a RAM disk.

polyFORTH's standard words **BLOCK**, **BUFFER**, **LIST**, and **LOAD** all handle single precision numbers.** However, the polyFORTH disk routines facilitate the handling of double-precision block numbers by designing the entire

* This section applies only to 8-bit and 16-bit CPUs. On 32-bit CPUs all block numbers are 32 bits, allowing for a total capacity of 2 Gbytes.

block and buffer management scheme around 32-bit block numbers (the high order bit is the **UPDATE** bit) and defining the single-precision words "on top of" double-precision versions. The following are defined:

Word	Stack	Description
2BLOCK	(d - a)	Takes a 31-bit block number and returns the address of the buffer containing that block.
2BUFFER	(d - a)	Takes a 31-bit block number and returns the address of a buffer identified by that block number.
2OFFSET	(- a)	Returns the address of the **USER** variable containing the 31-bit offset used to "bias" block numbers.

Word	Stack	Description
2IDENTIFY	(d)	Marks the most recently used buffer with a 31-bit block number, after adding the contents of **2OFFSET**.

There are no 31-bit versions of **LIST** and **LOAD**. Source blocks are restricted to a 32 Mb range of logical block numbers. However, since **2OFFSET** is added before the actual disk access, the actual range may be larger.

Merely plugging in a large disk drive, however, doesn't automatically provide access to its entire contents. At a minimum, tables in the disk driver specifying the number of heads, tracks, and sectors must be adjusted. If the polyFORTH nucleus doesn't contain a driver for the particular controller being used, new driver routines may be required. Since these issues are hardware dependent, consult your *CPU Supplement* and program listing for configuration information.

REFERENCES

?ABSENT and **ESTABLISH**, Section 3.2.2
?UPDATED, Section 3.2.3
Adding a Disk Driver, Section 3.2.8
Disk Retry Block, Section 3.2.6
Error Handling, Section 2.4.7
IDENTIFY, **FLUSH** and **EMPTY-BUFFERS**, Section 3.2.5
Structure of **PREV**, Section 3.2.1
UPDATE, Section 3.2.4

3.2.8 Adding A Disk Driver

Disk controllers vary considerably. Nevertheless, certain principles remain constant. The guiding principle behind disk driver design is to minimize both the number of disk accesses and the time of each access. This section is intended to show how the basic arrangement of a polyFORTH disk driver helps these ends, as well as certain low-level interfacing conventions and a technique for using more than one disk controller in a polyFORTH system.

3.2.8.1 THE BEHAVIOR OF HARDWARE-DEPENDENT CODE

The words **(BLOCK)** and **(BUFFER)** are hardware-dependent words whose addresses must reside in **'BLOCK** and **'BUFFER** for use by the higher-level words **2BLOCK** and **2BUFFER** (and, hence, **BLOCK** and **BUFFER**). **(BLOCK)** ensures that the data from the requested disk block is in a block buffer in memory. **(BLOCK)** only reads the data from disk if the data is not already in memory. When **(BLOCK)** needs an empty buffer to move disk data to, it calls **(BUFFER)** to obtain the address of a free buffer. **(BUFFER)** determines whether a buffer has

** The actual range is ± 32767. A single precision block number may be negative, for accessing blocks in the previous partition.

been changed since it was read from disk (it checks a flag set by the word **UPDATE**). If the data has been changed, **(BUFFER)** writes the data back to disk. In either case **(BUFFER)** returns the address of a free block buffer.

Wherever possible, **BLOCK** and **BUFFER** should share words. Fairly often, the addressing word mapping block numbers into physical track and sectors can be shared. Sharing words simplifies the driver and minimizes the amount of code to be debugged.

In the following sample skeleton definitions of **(BLOCK)** and **(BUFFER)**, **d** represents a 31-bit block number and **a** represents a block buffer address. Note that the read section in **(BLOCK)** and the write section in **(BUFFER)** use and leave similar arguments so code can be shared more easily between the read and write operations. **?ABSENT** exits **(BLOCK)** if a block is already in memory. **?UPDATED** exits **(BUFFER)** if the next buffer to be freed does not need to be written to disk.

```
: (BLOCK) ( n - a)   ?ABSENT
   'BUFFER @EXECUTE 'S 2+ 2@ ( - d a d)
   read-words  ( - d a)  ESTABLISH ;

: (BUFFER) ( d - a)   ?UPDATED
   ( - a d)  write-words  ( - a) ;
```

REFERENCES

32-bit Block Number Conventions, Section 3.2.7
Vectored System Routines, Section 3.1

3.2.8.2 SERVICING DISK INTERRUPTS

The interrupt routine for a native polyFORTH disk driver is usually very simple. The interrupt routine performs the most time-critical operations required (*e.g.*, acknowledging the interrupt and reading status bits if this cannot be postponed) and then stores **WAKE** into the status of the task using the disk to notify the task that the operation is complete. The interrupt routine finds the task by using the task address in the facility variable **DISK**. The value of **DISK** is set when the phrase **DISK GET** is executed in **BLOCK** or **BUFFER** (see reference below, Facility Variables).

Usually a disk operation is accomplished by giving a controller a sequence of commands. Disk controllers usually interrupt as they complete a command (or a sequence if the controller is "intelligent"). The segmentation of the machine code words for a command sequence is dependent on where the code must pause to await an interrupt. For a controller needing separate commands to move to a track, find a sector and transfer data, the following phrase might form the "read-words" of **(BLOCK)**:

```
ADDRESS  TRACK SECTOR READ
```

The controller would interrupt after **TRACK**, **SECTOR**, and **READ** to indicate it had completed each of those operations. All three of the operations are assembly coded words, each of which sends a command sequence to the controller and then enter the multitasker's round robin (via the code ending **WAIT**) for an indefinite time. Each time the task is awakened by the interrupt routine, the next word is executed. Waiting for interrupts in the multitasker allows other tasks fairly frequent opportunities to get work done. Interference between tasks trying to use the disk simultaneously is eliminated by the use of the word **GET** in the high-level definitions of **BLOCK** and **BUFFER**.

REFERENCES

GET, Facility Variables, Section 4.7
WAIT, Sections 4.2, 6.2
WAKE, Section 4.2

3.2.8.3 INTERLEAVING THE DISK'S DATA FORMAT FOR SPEED

If the data on the disk is ordered properly, a disk driver may run four (or more) times as fast as with a sequentially ordered disk. Depending on how the disk controller works, two forms of data interleaving are possible. If the controller can only transfer sectors which are a fraction of a block, then the sectors of each block may be interleaved with the sectors of other blocks. If the controller can transfer pieces of data the size of a block, then blocks themselves may be interleaved to speed sequential block access. On some disk drives it may be worthwhile to perform both types of optimization (usually it is not).

Ideal timing permits the driver code to finish its data transfer and housekeeping just in time to enable the disk interrupt before the next piece of data becomes available. Calculation of a good data spacing consists of choosing the smallest average distance between data segments (sectors or blocks) still resulting in a physical access time greater than the amount of time required by the data transfer. Usually the best pattern is determined by formatting a disk in several ways and then benchmarking with both random and sequential block access (with mixed read and write operations). The information required to determine whether such optimization may be desirable is obtained from the disk controller manual. The time between the availability of two sectors in a block may be computed from the rotational latency of the drive and average "seek" times. This figure may then be compared with an estimate of the time required by the software to get from one block or sector request to the next, assuming sequential block requests and normal system loading. If the software time is longer than the hardware time, some form of interleaving may be desirable.

The goal of the optimization is to be able to read a track's worth of blocks in one revolution. The worst case is reading only one sector per revolution. A good test for overall performance is to **SWEEP** the disk, and then compare the time for the **SWEEP** with the number of revolutions. The sweep time divided by the number of blocks per cylinder should (ideally) equal the number of cylinders covered.

REFERENCES

SWEEP, Section 5.3.4

3.2.8.4 REPORTING DISK STATUS TO polyFORTH

The hardware driver code must store disk status information in **DISK 2+** (**DISK 4+** on 32-bit systems) and subsequent bytes as needed. Two conventions for this information are in common use.

One convention declares if no errors occurred then **DISK 2+** is zero. Usually drivers using this convention perform an exclusive **OR** between a mask and the status data from the disk controller. If multiple cells of status are available, they are logically combined so common hardware failures may be isolated. This method has been used most often where the controller design is such that the disk status has no bits set unless errors have occurred.

The other convention copies status data directly from the disk controller to **DISK 2+**. If multiple cells of status data are available, they are copied to a region starting with **DISK 2+**. Each method has advantages and disadvantages. The "zero means no errors" method simplifies an application's disk error checking, and allows the disk retry block to be standardized and simplified in Forth systems with several different disk controllers. This method also has a higher machine code overhead, and requires more documentation to isolate types of errors. The "copy-status" method is simpler to implement, faster, and more directly understandable, but sometimes much more difficult to apply in systems with multiple controllers. Follow the convention already on your system, if possible. Remember to document the arrangement of error flags in **DISK 2+** in the shadow blocks.

The most recent block number written is stored in **'BUFFER 2+** (**'BUFFER 4+** on 32-bit systems) by the word **?UPDATED**.

3.2.8.5 ASSEMBLING A SYSTEM WITH MULTIPLE CONTROLLERS

Several parts of a polyFORTH system must be modified in order for it to smoothly manage a computer with several types of disk.

1. **BLOCK** must be modified to access all the disk over a contiguous range of block numbers (discussed later in this section).

2. Removable media should be in physical Drive 0 of multiple-drive systems, so the system can be booted from a backup copy of polyFORTH if necessary.

3. The bootstrap must be adjusted to accommodate item 2.

4. **VOLUME**, the disk size in the disking utility, is usually set to the size of the media containing the polyFORTH system to make backing up the system media convenient.

5. The word **PART** may need to be modified to use a look-up table.

On most polyFORTH systems, specifications 2 and 3 are already met, and need not be changed. Specifications 4 and 5 are trivial and will not be discussed further. Meeting specification 1 simply is difficult, and is the subject of the remainder of this section.

The two disk driver words **(BLOCK)** and **(BUFFER)** must be changed. Pre-existing Forth disk drivers almost always make use of a block number range starting at 0 and going up to some number of blocks dependent on the size of the disk. Ideally, one would wish for a method to combine unmodified disk drivers in a simple consistent way.

The two basic problems a disk-driver-combining method must solve are:

1. Choosing the correct driver.

2. Ensuring the driver receives the correct block numbers.

A particularly nice way to solve both problems is to define a word called **PLEAT**. **PLEAT** accepts a block number and a maximum possible block number. If the block number is less than the maximum block number (*i.e.*, acceptable to the disk driver), then **PLEAT** returns the block number, and a Boolean "false" for program control. If the block number is greater than, or equal (*i.e.*, not acceptable), then **PLEAT** returns the block number, less that controller's maximum acceptable block number, with a Boolean "true" for program control. By subtracting the current controller's maximum block number, **PLEAT** eliminates all those possible block numbers, so the next driver can begin its block numbers at zero. If the block number is within the driver's maximum, then the driver gets the block number unchanged. Although the example is high-level, a real driver would have **PLEAT** written in assembler.

In the following example, three different disk controllers are controlled by three separate drivers called **<BLOCK>**, **[BLOCK]** and ***BLOCK***. The drivers have meaningful block number input ranges of 0 to 500, 0 to 3000, and 0 to 9000 respectively. The "l" in the stack arguments of **PLEAT** stands for "limit."

In **(BLOCK)**, **?ABSENT** determines if a hardware operation is necessary—if the block is already in memory, **?ABSENT** performs all the functions of **(BLOCK)**. The phrase:

```
'BUFFER @EXECUTE
```

returns the address of a usable block buffer (and writes out the contents if necessary). The **OVER** brings a copy of the block number over the buffer address. The phrase **500 PLEAT** leaves the block number and a 0 on the stack if the block number is less than 500 (this controller's maximum block number is 499). When the **IF** uses the 0 left by **PLEAT**, **IF** transfers control to the controller's access word, **<BLOCK>**. If the block number were greater than 500, **PLEAT** would subtract 500 from the block number to prepare the block number for the next controller selection, for **[BLOCK]**.

```
: PLEAT ( n l - n-l 1 , n 0 ) 2DUP U<
   IF  DROP  0  ELSE  -  1  THEN ;
```

```
: (BLOCK) ( n - 1)   ?ABSENT  'BUFFER
   @EXECUTE OVER ( - n a n) 500 PLEAT
   IF  3000 PLEAT  IF  *BLOCK*   ELSE
   [BLOCK] THEN ELSE <BLOCK> THEN
   ESTABLISH ;
```

PLEAT can be used in the definition of **(BUFFER)**. **(BUFFER)** has different stack arguments than **(BLOCK)** {(- a) rather than (n -a)} because the block number used in **(BUFFER)** is generated by the word **?UPDATED**. **PLEAT** must be situated so it can use the block number left on the stack by **?UPDATED**.

Therefore, the construction using **PLEAT** in **(BUFFER)** must deal with the write words of the drivers, instead of the whole individual versions of **(BUFFER)**:

```
: (BUFFER) ( - a)   ?UPDATED ( - a n)
   500 PLEAT  IF  3000 PLEAT IF  *WRITE-
   WORDS*  ELSE  [WRITE-WORDS]  THEN ELSE
   <WRITE-WORDS>  THEN ;
```

Otherwise the application of **PLEAT** to constructing a **(BUFFER)** is the same as **PLEAT**'s application to the construction of a **(BLOCK)**.

3.3 LOADING polyFORTH SOURCE BLOCKS

Most compiled languages have a three-step process of constructing executable programs:

1. Compile the program to an object file on disk.
2. Link this program to other previously compiled and/or assembled routines.
3. Load the result into memory.

This often-lengthy procedure has a negative effect on a programmer's creative effectiveness. polyFORTH supports fully interactive programming by shortening this cycle to a single, fast operation: compiling from source to executable form in memory. This process is accomplished by the word **LOAD**.

3.3.1 The LOAD Operation

The word **LOAD** specifies the interpretation of source text from a disk block. **LOAD** expects the block number of the Forth block to be **LOAD**ed on the top of the stack:

```
number LOAD
```

The current contents of **OFFSET** will be added to the block number. The resulting absolute block number is stored in the user variable **BLK**, used by Forth's text interpreter. At this time interpretation of text input from the current input source is suspended and input is taken from the specified disk block. **>IN**, the character counter, starts at 0, and will be incremented by the action of the text interpreter until it reaches 1023 or an **EXIT** command is encountered, whichever comes first.

When all processing specified by the disk block has been completed (assuming that no errors were encountered in processing the block), execution resumes with input from the source that was in control when the **LOAD** was encountered. Note that **LOAD** sets the base to **DECIMAL** before returning.

The block number specified for the **LOAD** command must be a single-precision integer in the range 0 < n < 32767.

Where a block contains definitions, the result of a **LOAD** operation will be to compile these into the dictionary. Remember, though, that this is the result of the execution of defining and compiling words as they are processed by the text interpreter.

The process of **LOAD**ing disk blocks is identical to processing the same information entered from the terminal, with all information in the same disk block entered at once (*i.e.*, there will be no embedded carriage returns).

During the interpretation of a disk block, the processing of text input under program control proceeds as normal, substituting the disk block for the terminal, with the following exceptions and differences:

1. If a **TEXT** operation is executed, the entire remainder of the disk block will be read into memory, starting at the first position of **PAD**, unless the count expires or the delimiter is found. Since there is no guarantee a delimiter will be found at all, this should be avoided.

2. Use of **WORD** is equivalent to use of **TEXT** and follows the same restrictions as given for **TEXT**.

The block to be **LOAD**ed may itself contain a **LOAD** command, at which point the **LOAD**ing of the first block is suspended. When this occurs, the block number of the current block, the current character pointer for the interpreter operating in this block and the address interpreter pointer are saved on the return stack pending loading of the requested block. This nested **LOAD**ing process may continue indefinitely, subject to return stack size.

Where a group of blocks is to be **LOAD**ed, they should all be specified by **LOAD** commands contained in a single block, as opposed to serial nesting, *i.e.*, having each block load the next block in sequence. This is because each nested **LOAD** command requires several cells of additional space on the return stack, as explained below, and the chances of a return stack overflow are significantly greater when each block loads the next block in sequence. From a management viewpoint, moreover, it is best to see all the blocks needed for a well-defined portion of an application grouped together in one place.

The command **THRU** can load a group of sequential blocks. For example, if Blocks 260 through 270 needed to be loaded, **THRU** could be used:

```
260 270 THRU
```

THRU is an elective loaded from the "programmer aids" block.

A **LOAD** operation may also be compiled in a definition. The requested **LOAD** is done when the definition is executed. Following the completion of the **LOAD**, execution of the definition containing the **LOAD** request will resume at the word immediately following the **LOAD**.

If, during the **LOAD**ing process, a Forth error is detected, an error message is produced and all **LOAD**ing ceases. Both the return stack and the parameter stack are cleared and Forth reverts to terminal input. At this point, typing **L** will display the block being **LOAD**ed when the error was detected, with the editing character pointer **CHR** set to point just after the last word examined (*i.e.*, the one producing the error).

During loading, all text interpreter input is taken from the specified disk block. All output, however, proceeds to its normal destination. Thus, the use of **.(** or other output commands will send output to the terminal of the task executing the **LOAD**.

REFERENCES

.(, Section 2.3.6.5
PART, Section 3.2.1
EXIT, Section 2.4.6
L, Section 5.1.1
TEXT, Section 2.3.6.3
WORD, Section 2.3.6.2

3.3.2 Use of the Return Stack by LOAD

Each invocation of **LOAD** places four items on the return stack; they contain the following information:

1. The logical instruction counter for **LOAD**.
2. The input source in effect when the **LOAD** was issued (*i.e.*, the contents of **BLK**).
3. The character position in the input source (*i.e.*, the contents of **>IN**).
4. The address interpreter pointer (**I**) for **INTERPRET**.

The word **LOAD** calls the word **INTERPRET**, which calls the word **EXECUTE** for each word found in the dictionary. At some point during the **INTERPRET**ation of a block of source text, a **LOAD** may be executed. The new, nested **LOAD** pushes the above four items on the return stack, and then proceeds to **INTERPRET** its block of text. A **LOAD** may terminate in one of two normal ways:

1. When the word **-'** in **INTERPRET** has exhausted the input stream.
2. When the **EXECUTE** in **INTERPRET** executes the word **EXIT**.

Either occurrence forces Forth out of the endless loop in **INTERPRET**, and thereby allows **LOAD** to continue past **INTERPRET** and restore all the system data the **LOAD** earlier stored on the return stack.

If the programmer places items on the return stack during loading and fails to remove them before the end of the block, the four items removed from the return stack as **LOAD** finishes will be an incorrect set. This will cause an unpredictable error, depending upon the exact values of the items. In these circumstances, it is possible loading would continue, using an improper disk block.

REFERENCES

EXIT, Section 2.4.6
INTERPRET, Section 1.1.4

3.3.3 Named Program Blocks

The defining word **CONSTANT** may be used to give names to important blocks, such as blocks which in turn load other blocks to form a utility or application. Such a block is often called a "key block." For example,

```
120 CONSTANT OBSERVING
```

will be used as:

```
OBSERVING LOAD
```

The above has the effect of loading Block 120 and any other blocks specified to be loaded by that block.

Alternatively, the word **LOADS** creates a definition where the **LOAD** operation happens automatically. In this case,

```
120 LOADS OBSERVING
```

defines **OBSERVING** so you can merely type:

```
OBSERVING
```

to load this application.

Normally, **LOADS** is the more convenient way to name key blocks. Use of **CONSTANT** is appropriate when you want to use the name in other ways, such as:

```
OBSERVING LIST
```

We recommend the use of a key block for each entire application. Enter definitions using either of the methods described above for named blocks as well as any other brief application-wide definitions. Then you can see at a glance which of your application blocks are loaded and in what order. Your polyFORTH system contains several such key blocks, usually named in Block 10.

Use of this technique is much safer than "chaining" blocks, because "chaining" blocks can cause a return stack overflow. Generally one block on a system names all the key blocks in a system, and is **LOAD**ed immediately after booting. In polyFORTH systems, this is usually Block 10.

Note: There is a special danger with named blocks. They can be successfully **LOAD**ed when in any number conversion base. For this reason, named blocks should have a **DECIMAL** command in the first line to guard against accidental loading with an incorrect base.

REFERENCES

CONSTANT, Section 2.6.3
LOAD and the Return Stack, Section 3.3.2

3.3.4 Overlays

Because of Forth's compilation speed, there is rarely need for a dynamic run-time overlay capability. Many resident applications have several functionally independent subsets, however, and it is conventional to organize these as mutually exclusive overlays, any one of which may be loaded into each terminal's private dictionary. This is done by explicit command. Once **LOAD**ed, such an overlay will remain resident until replaced by another.

Examples of such overlay categories in a business environment might include order entry, payroll, and general ledger. In a scientific laboratory system there may be several different data acquisition and analysis modes.

This section covers two techniques for managing such overlays. To replace the contents of an entire task dictionary with a new overlay, we recommend use of the word **EMPTY**. To create additional levels of overlays within the task dictionary, such that when an overlay is loaded, it will replace its alternate overlay beginning at the appropriate level, we recommend use of the word **FORGET**. This section also discusses the option of allowing an overlay to reset the boundary between system and private definitions.

3.3.4.1 SINGLE-LEVEL OVERLAYS: EMPTY

The command **EMPTY** empties a user's private dictionary. In polyFORTH, most overlays (such as **DISKING** and **PRINTING**) begin with the word **EMPTY** at the top of the load block. For example:

```
0  ( DISK UTILITY)   EMPTY   DECIMAL
1  35 LOAD   37 LOAD
2    etc. ...
```

Any application definitions which are not meant to be replaced by the overlay should have been loaded as system electives (*i.e.*, by Block 9) not as part of the task's private dictionary. System electives, of course, are not affected by **EMPTY**.

The complete definition of **EMPTY** is:

```
: EMPTY   H 2+ @ H !  GOLDEN CONTEXT 20
   MOVE ;
```

Note: **H 2+** is **H 4+** and 20 is 36 in 32-bit systems.

The mechanism used by **EMPTY** to remove only those definitions in the task dictionary without affecting the system dictionary involves an array named **GOLDEN**. The system has one **GOLDEN** array containing the default values for **CONTEXT** and **CURRENT** plus the eight system "link heads"—that is, eight cells, each pointing to the last definition in each of the eight chains in the system portion of the dictionary.

Part of the function of **EMPTY** is to store the values of the system pointers into the user's private link head array. Thereafter, the first word to be compiled into each chain will be linked to the last word in the appropriate chain in the system portion of the dictionary, rather than being linked to the end of whatever chains may have previously existed in the task dictionary.

This function is accomplished by the phrase:

```
GOLDEN CONTEXT 20 MOVE
```

in the definition of **EMPTY**.

The other function of **EMPTY** is to reset the value of **H**. **H** is used at compilation time to point to the next available cell in the dictionary where a new definition may be placed. **H** is reset to the beginning of the user's dictionary. Thus the phrase:

```
H 2+ @ H !
```

resets **H** to the beginning of the task dictionary, so that thereafter any new definitions will be written over any previous definitions residing in the task dictionary.

REFERENCES

CONTEXT, Section 3.4.3
GOLDEN, Sections 3.3.4.3, 3.4.4

3.3.4.2 MULTI-LEVEL OVERLAYS: FORGET

The word **FORGET** is used to discard the most recent portion of a task's dictionary. The command:

```
FORGET NAME
```

will discard the definition **NAME** and all words defined after **NAME** in a user's partition. The user's dictionary pointer **H**, as well as the task's private link heads for the dictionary chains, will be reset to the last definition in the vocabulary before **NAME**.

Since **H** is reset, the dictionary is truncated spatially as well as logically.

FORGET has two uses:

1. To discard only part of your definitions. For example, when testing, you may reload only the last block, not your entire application.

2. To create additional levels of overlays.

Suppose your application includes an overlay called **GRAPHICS** whose load block begins with the command **EMPTY**. Once **GRAPHICS** is loaded, you want to be able to load either of two additional overlays, called **COLOR** and **B&W**, to load after **GRAPHICS**, thus creating a second level of overlay. Here is the procedure to follow.

1. Define a "null definition" as the final definition of **GRAPHICS**, using any word you want as a dictionary marker. For example:

   ```
   : OVERLAY ;
   ```

   Preferably, such a null definition will be placed at the bottom of the **GRAPHICS** load block.

2. Place the appropriate **FORGET** phrase on the first line of the load block of each level-two overlay. For instance,

```
( COLOR)    FORGET OVERLAY    : OVERLAY ;
```

Thus, when you execute the phrase:

```
COLOR LOAD
```

you "forget" any definitions which may have been compiled after **GRAPHICS** and restore the null definition of **OVERLAY** to serve as a marker in the event you want to load an alternate level two definition, such as **B&W**.

By using different names for your null definitions, you may create any number of overlay levels. One of the simplest choices of overlay markers is the word **HELP**, since the first definition in many applications is the definition of the help screen displayed at the beginning of the application.

REFERENCES

Help Screens, Section 1.5

3.3.4.3 RESETTING THE POINTERS FOR AN "EMPTY" DICTIONARY

Application words intended to be available to all users in the system are normally loaded from the electives load block so they will become system definitions. The command **GILD** actually creates the division between system definitions and task definitions as may be seen near the end of the electives load block. **GILD** is defined as follows:

```
: GILD   CONTEXT GOLDEN 20 MOVE
   HERE H 2+ ! ;
```

The first part of this phrase copies the values of the eight link heads plus copies of **CONTEXT** and **CURRENT** in the **OPERATOR** user area, at the moment the above phrase is executed, to the **GOLDEN** array. Since the phrase is executed after all system definitions have been loaded, the **GOLDEN** array will henceforth point to the ends of the system-definition chains.

Note that it is important that **CONTEXT** is **FORTH** when this is done, so the default **CONTEXT** (after **EMPTY**) will be **FORTH**.

The second part of the above phrase stores the present value of **H** (the dictionary pointer) into **OPERATOR**'s **H 2+** as a reference for the future as to where the "empty dictionary" should begin. (Note: On some systems the empty dictionary location is in **H 4+**; see your *CPU Supplement*.) In certain cases it is useful to have an overlay reset the boundary between system and task definitions by executing the above phrase. An example in polyFORTH is the Data Base Support option, whose load block concludes with **GILD**.

This technique has the advantage of making additional capabilities available to all users on the system without recompiling. Each user will be required to execute **EMPTY** to reset the beginning of his private dictionary to the end of the extended system dictionary. This can also be done automatically by **PROMPT**ing them.

REFERENCES

GOLDEN, Section 3.4.4
PROMPT, Section 4.10

3.4 VOCABULARIES

Vocabularies are mutually exclusive collections of definitions residing concurrently within the dictionary. Up to eight vocabularies may exist at any one time. Dictionary searches proceed from one vocabulary to another in a

specified sequence of up to four vocabularies. This mechanism allows you to control which vocabulary or vocabularies are to be searched. Within each vocabulary the search is from newest to oldest.

Vocabularies have three principal uses:

1. In the resident system, to segregate special-purpose words such as those in the **ASSEMBLER**, to allow them to have the same names as standard Forth words.

2. In the Target Compiler, to segregate target versions of **FORTH**, **ASSEMBLER**, and **EDITOR** words from the resident versions.

3. In applications running in the resident system, to protect against accidental misuse of words only intended to be available to programmers.

3.4.1 Vocabulary Selection

The standard vocabularies provided by Forth are:

```
FORTH
ASSEMBLER
EDITOR
```

FORTH is the standard fundamental vocabulary. **ASSEMBLER** contains all assembler mnemonics, addressing modes, and other special assembler commands. **EDITOR** contains the editing commands for editing source text in blocks.

The use of separate vocabularies makes it possible, for instance, for the word **I** to be defined to supply a loop index in one context and insert a string in another context or name a register in yet another.

There are two actions on the dictionary relevant to vocabularies: searches and additions. The sequence of vocabularies to be searched is specified by the contents of the user variable **CONTEXT**. The vocabulary to which any further definitions are to be linked is specified by the contents of the user variable **CURRENT**.

You may change the contents of **CONTEXT** by simply naming the desired vocabulary. For example, the word:

```
ASSEMBLER
```

changes **CONTEXT** so future searches will begin with the **ASSEMBLER** vocabulary. **CONTEXT** is automatically set to **ASSEMBLER** by the defining words **CODE** and **;CODE**.

Similarly, you may employ the word:

```
EDITOR
```

to set **CONTEXT** to begin by searching the **EDITOR** vocabulary. Several of the **EDITOR** commands are found in **FORTH** and automatically set **CONTEXT** to the **EDITOR** vocabulary.

CONTEXT is automatically reset to the contents of **CURRENT** by the defining word **:**. For example, assume **FORTH** is "current" (as is typical) and a user has just finished modifying a block using **EDITOR** commands (**CONTEXT** is now **EDITOR**). When the user loads the block, **CONTEXT** switches back to **FORTH** when the first **:** is encountered.

The contents of **CURRENT** may also be changed. The word **DEFINITIONS** sets **CURRENT** to the "context" vocabulary. For example, the phrase:

```
EDITOR DEFINITIONS
```

first sets the value in **CONTEXT** to be the **EDITOR** vocabulary, then sets **CURRENT** also to **EDITOR**. Thereafter any future definitions will be linked according to the **EDITOR** vocabulary. When the system starts up, or following an **EMPTY**, the default vocabulary for **CONTEXT** and **CURRENT** is **FORTH**.

3.4.2 Creation of a Vocabulary

Each of the three standard vocabularies is associated with an index, as follows:

1 is **FORTH**
3 is **ASSEMBLER**
5 is **EDITOR**

For internal reasons, a vocabulary index may be any odd number from 1 to FH.

If you look up the definitions for the commands associated with the standard vocabularies, you will find:

```
HEX   0001 VOCABULARY FORTH
      0015 VOCABULARY EDITOR
      0013 VOCABULARY ASSEMBLER  DECIMAL
```

(The order of the digits is reversed for some processors.)

When viewed as a hexadecimal number, each of the four 4-bit "nibbles" of the constant represents a vocabulary index. Thus the variable **CONTEXT** may contain as many as four vocabulary indexes. This feature allows vocabularies to be chained; that is, after the system searches the vocabulary specified by the index in the right-most nibble, it will then search the vocabulary specified by the nibble to the left, and so on. The search continues until the word is found, a 0 nibble is encountered, or all four nibbles have been used, whichever comes first.

Although **VOCABULARY** is a defining word, it does not define a vocabulary, but rather a "vocabulary-specifying command" (*e.g.*, **FORTH**, **EDITOR**, **ASSEMBLER**). The word **VOCABULARY** associates a vocabulary-specifying constant (*e.g.*, 0015) with each of these commands. This constant, when placed in **CONTEXT**, specifies the order the appropriate vocabularies are to be searched. When placed in **CURRENT**, the first vocabulary to be searched specifies the vocabulary to which new entries will be linked.

As mentioned above, the order of the digits is reversed for some processors. To determine which order is used by your system, either refer to your listing, or set and display **CONTEXT** by typing:

```
FORTH HEX CONTEXT @ U.
```

If the result is 1 (the leading zeros don't show), searches begin with the vocabulary whose index is specified by the right-most nibble (vocabulary 1, or **FORTH**). If the result is 1000, your system first searches the vocabulary whose index is specified by the left-most nibble.

In all systems (regardless of the direction in which the nibbles are examined), the search order depends on the context vocabulary as indicated in this table:

Command	Search Order		Constant
FORTH	1.	**FORTH**	0001 or 1000
EDITOR	1.	**EDITOR**	0015 or 5100
	2.	**FORTH**	
ASSEMBLER	1.	**ASSEMBLER**	0013 or 3100
	2.	**FORTH**	

Although the same constant that is stored into the variable **CONTEXT** is also stored into **CURRENT** (resetting **CONTEXT** according to **CURRENT**), only the nibble in the first-search position is used to indicate which vocabulary will receive new dictionary entries (see the definition of **CREATE** in your system listing and the references below).

The actual creation of a new vocabulary occurs after a new vocabulary-specifying command has been defined and its associated constant has been placed in **CURRENT** with the phrase:

```
specifying-command DEFINITIONS
```

As new definitions are compiled a new "vocabulary" will be created.

If you are not target compiling, it is possible to add up to five vocabularies of your own (using the odd digits 7-F), provided you write definitions linking any new vocabulary to an existing vocabulary, especially **FORTH**, (otherwise searches will find nothing). The principal use of additional vocabularies is for "sealed" vocabularies in a resident system. Most other situations where vocabularies might be considered are better handled by one of these methods:

1. In an application running on the resident system, to define two classes of words which may have the same names but different meanings, we recommend the use of overlays.

2. In a target compiled application where the Forth interpreter is not present, there is no value in having separate vocabularies since there are no searches.

3. In a target compiled application in which the Forth interpreter is present, words which must be protected against inadvertent misuse should be made headless during target compilation, thereby rendering them unfindable in a search.

REFERENCES

CREATE, Section 2.7.1
Sealed Vocabularies, Section 3.4.5
Vocabularies in the Target Compiler, Section 7.2

3.4.3 Hashed Dictionary Searches

Among the user variables in each terminal task's user area, between **CONTEXT** and **CURRENT**, reside eight consecutive cells called the "link heads." Each link head contains the address of the most recent definition added to one of eight linked lists that comprise the dictionary (see Fig. 3.6).

The space in the dictionary is sequentially allotted, with new entries having greater addresses than older entries. Entries from all the linked lists are mixed together in the dictionary. In systems with multiple tasks, the private space for each task is allotted and named in the public dictionary when electives are loaded immediately after booting.

As new definitions are created, they are linked to one of the eight linked lists. The selection of which linked list depends on a combination of the current vocabulary index and the first letter of the word being defined. The vocabulary index is added to the ASCII character, then the least significant bit of the 4-bit nibble is masked out to yield an even cell offset to the array of link heads.

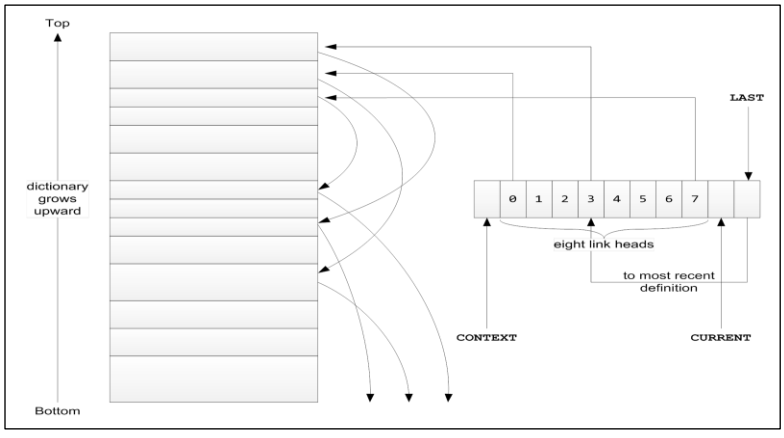


Fig. 3.6

Dictionary Link Heads

Thus a single list may contain words from as many vocabularies as have been defined (up to a maximum of eight). Yet any two words with identical names, belonging to separate vocabularies, will be linked to separate lists. This arrangement more evenly distributes the number of definitions linked to the various chains, and makes it possible to search an entire vocabulary for a given word by actually searching only one-eighth of the total dictionary. This greatly reduces search time and thus, compilation time.

As stated earlier, each terminal task user area contains its own copy of the eight link heads. When a terminal task's dictionary is empty, these heads point to the end of the lists that comprise the public dictionary. As words are added, each task's link heads will point to the new definitions which physically reside within the tasks' private dictionaries.

3.4.4 The GOLDEN Array

The system maintains an array, called **GOLDEN**, which contains a copy of the eight link heads (see Fig. 3.7) that point to the ends of the system dictionary lists (see Fig. 3.6). Thus "system definitions" are those linked through **GOLDEN** and are available on a common, reentrant basis to all users. The **GOLDEN** array begins with a system value of **CONTEXT** (specifying the **FORTH** vocabulary), continues with the eight link head cells and ends with the system value for **CURRENT** (again **FORTH**).

Thus, the definition of **EMPTY** contains the phrase:

```
GOLDEN CONTEXT 20 MOVE
```

which copies the system link heads (as well as **CONTEXT** and **CURRENT**) into the user's private link heads, thus resetting each of the link pointers for the task dictionary back to the end of each of the lists in the system dictionary.

The word **GILD** contains the reversed phrase,

```
CONTEXT GOLDEN 20 MOVE
```

which copies the user's private link heads into the system's link heads. In effect, this relocates the border between system definitions and the user's definitions. Thus, any private definitions which a user has loaded prior to the above phrase become public to other users (provided that they first execute **EMPTY** to reset their private link heads). **GILD** also resets the physical bottom of the dictionary so that the user can no longer forget the "public" section of the dictionary.

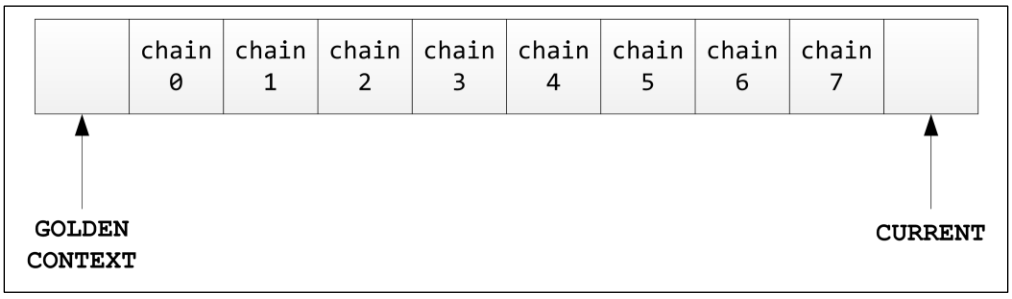


Fig. 3.7

The **GOLDEN** Array

GILD is found near the end of the electives block (usually Block 9). Normally, when any definitions are meant to be available to all users, their blocks should be loaded from the electives block, above **GILD**.

REFERENCES

EMPTY, Section 3.3.4.1
FORGET, Section 3.3.4.2

3.4.5 Sealed Vocabularies

The vocabulary mechanism offers the potential for an exceptionally powerful security technique. You can implement this by setting up a special application vocabulary consisting of a limited number of commands guaranteed to be safe for users. You then ensure no application word can change **CONTEXT**, and that **CONTEXT** is set so the text interpreter will only search the application vocabulary.

This has the effect of sealing a task into its limited vocabulary and rendering all other words "unfindable." This is how a sealed vocabulary is constructed:

1. Define a new vocabulary for the findable words. For example:

   ```
   HEX 0017 VOCABULARY APPLICATION
   ```

   This vocabulary must be linked to **FORTH** at compile time so that **APPLICATION** words may contain references to **FORTH** words. This is done by including **FORTH** (index 1) in the search order as shown in the example.

2. Place all definitions to be available to users in the application vocabulary by declaring:

   ```
   APPLICATION DEFINITIONS
   ```

   before compiling such definitions.

3. Define **SEALED** like this:

   ```
   : SEALED   0007 CONTEXT ! ;
   ```

   so once **SEALED** is executed, only definitions in Vocabulary 7 can be accessed. Thus **SEALED** might be executed as the last item in the application load block (in a resident system), and then only when the application has been completely tested.

Note that **CONTEXT** cannot be changed by a reference to its name, because the variable **CONTEXT** is defined in the **FORTH** vocabulary which is sealed from search, as are all words (such as **!**) that would enable a knowledgeable user to change **CONTEXT**.

An example of the load block for a sealed application might be:

```
 0
 1 : SEALED   0007 CONTEXT ! ;
 2 HEX   0017 VOCABULARY APPLICATION
 3 APPLICATION DEFINITIONS
 4 : GO   ... ;
 5 : STOP   ... ;
 6 : TURN   ... ;
 7 SEALED
 8
 9
10
11
12
13
14
15
```

3.5 CALENDAR SUPPORT

Forth supports two optional output formats for the date:

mm/dd/yy and dd mmm yyyy

The latter uses a three-letter month-name abbreviation and assumes the current year, which may be reset in the calendar block. An application may use whichever of these is most compatible by loading the appropriate calendar block in Block 9. The calendar option currently installed can be seen by typing **SYSTEM** after polyFORTH has been booted and the options loaded. The two calendars are located in your system in consecutive blocks.

The three-letter abbreviation type of calendar is useful because it is simple, easy to read, and does not require double-precision integers. The abbreviation calendar is inconvenient for use in applications entering dates for a variety of years.

For both calendars, the internal date format is the same: a single-precision integer containing the number of days elapsed since January 1, 1900. This format is called a "Modified Julian Day" (MJD). An MJD is a compact representation (2 bytes vs. 6 or 8). Also, arithmetic between MJDs can be performed directly, without complex conversions. Dates prior to March 1, 1900 will not be set or displayed correctly because only centuries divisible by 400 are leap years and the special conditions for 1900 are not tested for by the calendar. To obtain the day of the week from an MJD, simply take that number modulo 7; a value of 0 is Sunday. For example:

```
4/24/88 M/D/Y 7 MOD .
```

gives 4 (Thursday).

The week-day number may be used to index a text array for output.

3.5.1 Date Input

1. **dd mmm Input Format**

The day/month calendar is a convenient, readable format appropriate for applications where the majority of dates entered are current (*e.g.*, transaction or measurement dates).

Where this option is used, the MJD conversion is performed by the month-name abbreviation, which is actually a Forth command. The months are:

```
JAN   MAY   SEP
FEB   JUN   OCT
MAR   JUL   NOV
APR   AUG   DEC
```

These words expect the day-number on the stack; they will add offsets for the current year and month to leave the correct MJD on the stack. For example,

```
25 NOV
```

will leave on the stack the MJD for the twenty-fifth of November of the current year. To set the year type:

```
year A.D.          (e.g., 1988 A.D.)
dd mmm NOW         (e.g., 25 NOV NOW)
```

The current calendar year is the default. It is set in Block 9. The year must be edited every January.

2. **mm/dd/yy Input Format**

The mm/dd/yy format is most convenient for applications where dates are frequently entered for years other than the current one (*e.g.*, birth dates).

The date typed in the mm/dd/yy format is converted to a double-precision integer on the stack by the standard input number conversion routines. A leading zero is not required on the month number but is required on the day number if it is less than ten. Years entered as 00 to 99 are treated as being in the twentieth century; twenty-first-century dates may not be entered by this option.

The double-precision number thus entered must be given as a parameter to the date input conversion routine **M/D/Y**, which computes the MJD. For example:

```
8/03/40 M/D/Y
```

returns the MJD for August 3, 1940.

Normally **M/D/Y** is incorporated in the application command that accepts the date. For example, given the definition:

```
: HIRED ( d)   M/D/Y DATE ! ;
```

then the phrase, **1/15/87 HIRED** would convert the date 1/15/87 to an MJD and store it in the variable **DATE**.

REFERENCES

MJD, Section 3.5
Number Conversion, Section 2.5.2

3.5.2 Date Output

All date output commands expect an MJD (Modified Julian Date) on the stack. This will be formatted into an output string that has an appearance similar to the input string for the input format of the calendar option selected.

The basic output command is:

```
mjd .DATE
```

In systems with the mm/dd/yy calendar installed, **.DATE** works this way:

```
mjd .DATE               5/16/88 ok
```

Systems with the day-month option have another version of the output format:

```
mjd .DATE               16 MAY 1988 ok
```

The actual formatting is done by the word **(DATE)**. **(DATE)** returns the address and length of a formatted string (the arguments appropriate for **TYPE**). These arguments may also be used for other purposes, such as other forms of output. To print only the day and month (not year) using the day-month calendar, for example, one could define:

```
: .DAY ( n)   (DATE) 4 - TYPE ;
```

This would print today's date this way:

```
TODAY @ .DAY            16 MAY ok
```

REFERENCES

MJD, Section 3.5
TODAY, Section 3.5.3

3.5.3 System Date Management

The system date is stored as a single-precision modified Julian date. The system date is near **TICKS** in the system variables region of memory. The address of the system date is given by the word **TODAY**. Thus, **TODAY @** will return the value of the system date in modified Julian form (add **.DATE** for display).

To set the system date, type:

```
date NOW
```

The date need only be set after power-up. When the system is loaded or when the system "help screen" is displayed by the command **SYSTEM**, the date is displayed for verification.

Note that rollover of the date at midnight is not programmed to occur automatically, because this would add tremendous overhead to the clock interrupt routine. Instead, the date is checked for rollover and corrected whenever the time is read by **@TIME**.

REFERENCES

MJD, Section 3.5
Time Overflow at Midnight, Section 3.6.6

3.6 CLOCK SUPPORT

Assuming the presence of a hardware clock, polyFORTH provides standard time-of-day support words. These words allow the user to set and to print the current time of day.

The internal units of time maintained by polyFORTH are *clock ticks*, the value of each one depending upon the frequency of the hardware clock. Part of the standard clock support word-set includes a ratio for converting internal units to milliseconds, and most application-level words operate in milliseconds.

The clock electives also allow each task to establish a timer used to deactivate a task for a user-specified time interval.

3.6.1 Internal Time Representation

The clocks on most computers are simple time bases which generate a pulse at a regular interval. Most often the pulse is used to cause a "clock interrupt" or "tick." When a clock interrupt occurs, the computer leaves the program it was running, and runs a routine to increment a counter. The computer then resumes running the program it was running before it was interrupted.

The polyFORTH clock interrupt routine is made as simple as possible to reduce the clock overhead. Usually the polyFORTH clock interrupt routine merely increments a double-precision variable called **TICKS**. All other functions (setting the clock, hour/minute calculation, time overflow at midnight) are performed by clock words that access **TICKS** at irregular intervals (from applications, for example).

The word **TICKS** pushes onto the stack the address of a double-precision integer containing the current time of day in units of actual clock ticks (milliseconds, sixtieths of a second, or whatever, depending on the actual clock hardware). The stack effect of **TICKS** is (- a).

The most important hardware-dependent variable in the design of clock routines is the number of ticks per second. The two most common rates are power-line frequency, 50 or 60 ticks per second (a popular timebase), and 1000 ticks per second. With a 60 Hz clock, the clock can run about 820 days before **TICKS** will turn over to zero. A 1000 Hz clock can run about 49 days before **TICKS** will turn over.

The word **@TIME** returns the current double-precision value of **TICKS** {stack effect: (- d)}. It disables the clock interrupt while picking up the two cells of the counter, to ensure a tick doesn't occur between fetches. **@TIME**

should always be used to fetch the time of day. In addition to protecting against an interfering interrupt, it also checks for midnight, and updates **TODAY** as needed.

REFERENCES

Time Overflow at Midnight, Section 3.6.6

3.6.2 Setting the Clock

On systems with clock hardware, polyFORTH provides a facility to set the system clock. The time of day is set by the word **HOURS**. **HOURS** is executed with the current time of day on the top of the stack.

Some hardware clocks cannot be turned off when the new value of **TICKS** is stored. Thus, potentially, on these machines a clock interrupt could occur between the operations which store the new low and high order cells of **TICKS**. This is not usually significant because the new low-order value is stored first. On extremely rare occasions, the new low-order cell of **TICKS** will be 65535, and the two-cell "store" operations will be broken by an interrupt, resulting in loss of the "carry", and giving a substantial clock error if the clock interrupt cannot be disabled. On systems with 60-cycle clocks this time occurs for 2 microseconds (the approximate time to store a cell on many machines) every 18 minutes, and the error can occur once a day. All systems that can will circumvent this possible error by momentarily disabling the clock interrupt.

The current time of day is entered as a double-precision integer. In the following example, *hh* is the hour (0-23) and *mm* is the minute (00-59).

```
hh:mm HOURS
```

If the time of day given to **HOURS** contains an invalid hour or minute specification, the results are unpredictable and times printed by the system may be meaningless.

HOURS separates the hours and minutes out of the double-precision number, and computes the corresponding number of ticks (depending upon the hardware clock frequency). The result is stored into **TICKS**. **HOURS** can be easily user-modified to allow setting the time to the nearest second.

REFERENCES

Time Overflow at Midnight, Section 3.6.6

3.6.3 Timed Events

polyFORTH provides the ability to use the system clock to time events, both in the sense of specifying when something will be done and measuring how long something takes.

The word **MS** causes a task to suspend its operations for a specified number of milliseconds, during which time other tasks can run. For example if you have an application word **SAMPLE** which records a sample, and you want it to record a specified number of samples, one every 100 ms (ten times per second), you would write a loop like this:

```
: SAMPLES    ( n)    0 DO  SAMPLE
   100 MS  LOOP ;
```

Since **MS** does its timing using the system clock, the accuracy of the interval measured depends on the resolution of that clock. If the above example were run on a version of polyFORTH (such as the Native system for the IBM-PC) which sets the clock to "tick" at 1 ms intervals, the 100 ms will be very close to 1 ms accuracy. The polyFORTH running on the IBM as a co-resident OS with MS-DOS, on the other hand, is limited by the fact that DOS requires the clock to tick only every 55 ms. With this resolution, the "100 ms" interval may be anywhere between 56 and 110 ms. As a general statement, the error on an interval will be approximately the number of milliseconds in one clock tick.

To obtain better accuracy, you may use a clock with shorter intervals. If you need to respond promptly to an external event, the best way is to associate an interrupt directly with the event.

REFERENCES

Interrupts, Section 6.11; The *CPU Supplement*
PAUSE, Section 4.2

3.6.4 Measuring Elapsed Time

polyFORTH supports two words allowing the user to measure the elapsed time interval between two events in milliseconds:

Command	Action
COUNTER	Returns on the stack the low-order cell of **TICKS**.
TIMER	Repeats **COUNTER**, then subtracts the two values, and displays the time interval since the previous execution of **COUNTER**, in milliseconds.

For example, the following code would be used to measure the execution time of 1000 executions of the user word **xxxx**:

```
: MEASURE   COUNTER  1000 0 DO
   xxxx LOOP  TIMER ;
```

This measurement also includes the time required to handle the **DO ... LOOP** itself. To arrive at a precise value, you may compute the overhead by running the following loop:

```
: OVERHEAD   COUNTER  10000 0 DO
   LOOP  TIMER ;
```

Then subtract one tenth of this time from the time you obtained with **MEASURE** to get the actual time for 1000 executions of **xxxx**.

To figure times with an arbitrary number of iterations, divide the figure given by **TIMER** by the number of iterations. To do the arithmetic in Forth, type:

```
t 100 n */ .
```

where *t* is the time given by **TIME** and *n* is the number of iterations.

This yields the number of 1/100s of a millisecond per iteration. Note the maximum clock error per iteration (in the same units) may be calculated by:

```
1000 100 Hz */ iterations /
```

where **Hz** is the clock frequency (interrupts per second).

High-precision benchmarks with nested loops are left as an exercise for the reader.

3.6.5 Time of Day Output

The current time of day may be printed by the word **.TIME**. An example of the use of **.TIME** is:

```
: CLOCK   BEGIN PAGE  @TIME .TIME
   30000 MS  AGAIN ;
```

This word prints the time as a five-character string followed by a blank. The time of day is printed only to the most recent minute. The appropriate phrase to print the current time of day is:

```
@TIME .TIME
```

.TIME calls the word **(TIME)**. The word **(TIME)** expects a time of day, in internal representation, on the stack. It converts this double-precision number into a string of the format hh:mm and leaves the address and length of this string on the stack. The stack effect is (d - a n). **(TIME)** is used to format a time of day for output using **TYPE** or for some other application use. **(TIME)** may be used to print a stored time of day or to print the current or stored time with the report generator.

Although it is possible to fetch the internal time representation with the phrase:

```
TICKS 2@
```

a much better way, which automatically handles time overflow at midnight, is the word:

```
@TIME
```

Since **@TIME** handles the time rollover at midnight, the phrase **@TIME .TIME** should precede the system date display to ensure accuracy after midnight.

REFERENCES

Time Overflow at Midnight, Section 3.6.6

3.6.6 Time Overflow at Midnight

In polyFORTH, when the clock interrupt routine increments through midnight, no special action is taken and the current time of day becomes 24:00. As the clock interrupts continue, **TICKS** continues to be incremented, causing an invalid time to be maintained.

Of course, the polyFORTH clock interrupt could easily be programmed to prevent this, but execution overhead would increase because of the need for constant checking for day rollover. polyFORTH uses a different scheme to minimize overhead yet ensure correct dates.

The way polyFORTH resets a clock after an overflow past midnight is with the word **@TIME**. **@TIME** returns the internal representation (stored in **TICKS**) on the stack, so it is convenient to use **@TIME** in any routine using the contents of **TICKS**. For example:

```
@TIME .TIME
```

is equivalent to:

```
TICKS 2@ .TIME
```

except that the first time that **@TIME** is used in a new day, **@TIME** automatically adjusts the value in **TICKS** to be the correct time for the new day, and increments the **MJD** in the variable **TODAY**.

REFERENCES

MJD, Section 3.5

3.7 THE TERMINAL DRIVER

Forth supports a variety of means to perform I/O with a terminal, printer, or other serial-type I/O device. In addition, a simplified method is provided to make use of cursor positioning, and other hardware-dependent features, without forcing the use of particular models of terminal.

The general scheme of the **TYPE** and **EXPECT** interrupts is explained in this section, with a Dijkstra diagram (Fig. 3.8) of a representative implementation .

REFERENCES

EXPECT, Section 3.7.1
TYPE, Section 3.7.3

3.7.1 Terminal Input Commands

What follows is a table of specialized words that handle input from serial devices:

Word Stack Function

EXPECT (a n) Gets *n* characters from the task's serial device, echoes each and places them in memory beginning at *a*. The process will also stop if **EXPECT** sees a carriage return. An example of use is

```
PAD 5 EXPECT  12345 ok
```

EXPECT is used for most terminal input. **EXPECT** will "back up" over previously input characters when it receives an ASCII BS (8) or **RUBOUT** (7F). When the character pointer points to *a*, the original address, **EXPECT** stops backing up and will thereafter echo an ASCII BELL (7) for each RUBOUT or BS it receives. The number of characters that have been input by **EXPECT** is available through the phrase:

```
SPAN @
```

STRAIGHT (a n) Gets *n* characters from the task's serial device without echoing them, and places them in memory beginning at *a*. **STRAIGHT** ignores carriage returns, backspace characters, and all special characters. **STRAIGHT** is most often used to transfer binary data over a serial link.

KEY (- b) Accepts exactly one byte of data from a serial link. **KEY** calls **STRAIGHT** so it doesn't echo. **KEY** is sometimes used for input prompting and in serial protocols. **KEY** is also often useful to interactively determine the numeric value of a character:

```
KEY . 67 ok
```

Word Stack Function

?KEY (- b/0) Checks whether a character has been received on the task's serial device since the last call to **EXPECT**, **STRAIGHT**, **KEY**, or **?KEY**. If so, the value of the character received is returned; if not, a zero ("false") is returned. The returned value is primarily used as a truth flag; if you have interest in the actual character value you should use a phrase such as:

```
?KEY ?DUP IF ...
```

Aside from returning the key value, **?KEY** does not retain the value. Use of **KEY** following **?KEY** will await another keystroke.

REFERENCES

String Operations, Section 2.2

3.7.2 Basic Principles of Terminal Input

Most terminal input is accepted by the word **EXPECT**. **EXPECT** is a routine which vectors execution to some version (potentially different for every terminal task) of **(EXPECT)** through the user variable **'EXPECT**, which contains the address of the parameter field of **(EXPECT)**. **EXPECT** takes an address and length from the stack, and inputs a string of characters to memory starting at the byte whose address was on the stack. An example of the use of **EXPECT** is:

```
PAD 5 EXPECT            12345 ok
```

This example puts five (12345) characters into memory starting at **PAD**. If a carriage return is typed, the string that was input will be truncated to the number of characters preceding the carriage return. In the following

discussion, note that the standard text input interrupt has no provision for an "escape from program" function. This omission is intentional, so applications can be written to be totally "user-proof."

On most systems, **EXPECT** and **STRAIGHT** share interrupt code (see Section 3.7.1 for a discussion of **STRAIGHT**). In addition, systems dealing with X-ON and X-OFF must be capable of taking action when those characters are received. Thus, serial interrupt routines must often deal with seven cases of possible input (see Fig. 3.8):

1. Normal characters (for **EXPECT**).
2. Carriage Return ($0D_H$)—resume execution (for **EXPECT**).
3. Backspace or delete (08_H or $7F_H$) with a partially filled input area (for **EXPECT**).
4. Backspace or delete with an empty input area (for **EXPECT**).
5. Binary serial input (for **STRAIGHT**).
6. X-OFF (13_H, CTRL-S)—turn off output stream.
7. X-ON (11_H, CTRL-Q)—turn on output stream.

The first case, normal characters, is processed by the following steps:

1. The character is input (usually from a device to an accumulator).
2. The address of the terminal's task's user area is calculated so that the interrupt routine can access the task's user variables, especially **CTR**, **PTR**, and **SPAN**.
3. Is the user variable **CTR** less than zero? If it is, then the system is performing input. Positive values in **CTR** indicate that the system is performing output. In this case (normal **EXPECT**), the result is always yes.
4. Is the **STRAIGHT** flag set? If the straight flag is set, the CPU may jump directly to step #10—see the case about **STRAIGHT**.
5. The parity bit is cleared.
6. The character is tested to see if it is **BS** or **DEL** (this test fails for the normal case of **EXPECT**).
7. The character is tested to see if it is **CR** (this test fails).
8. The character is echoed.
9. One is added to the user variable **SPAN**. **SPAN** is a count of the number of characters input.
10. The character is stored to the address in the user variable **PTR**. **PTR** is short for PoinTeR.
11. One is added to **PTR** so the next character will be stored properly.
12. One is added to the user variable **CTR**. **CTR** contains the two's complement of the number of characters that may yet be stored without overrunning the input area. Adding one brings **CTR** closer to zero. **CTR** is short for CounTeR.

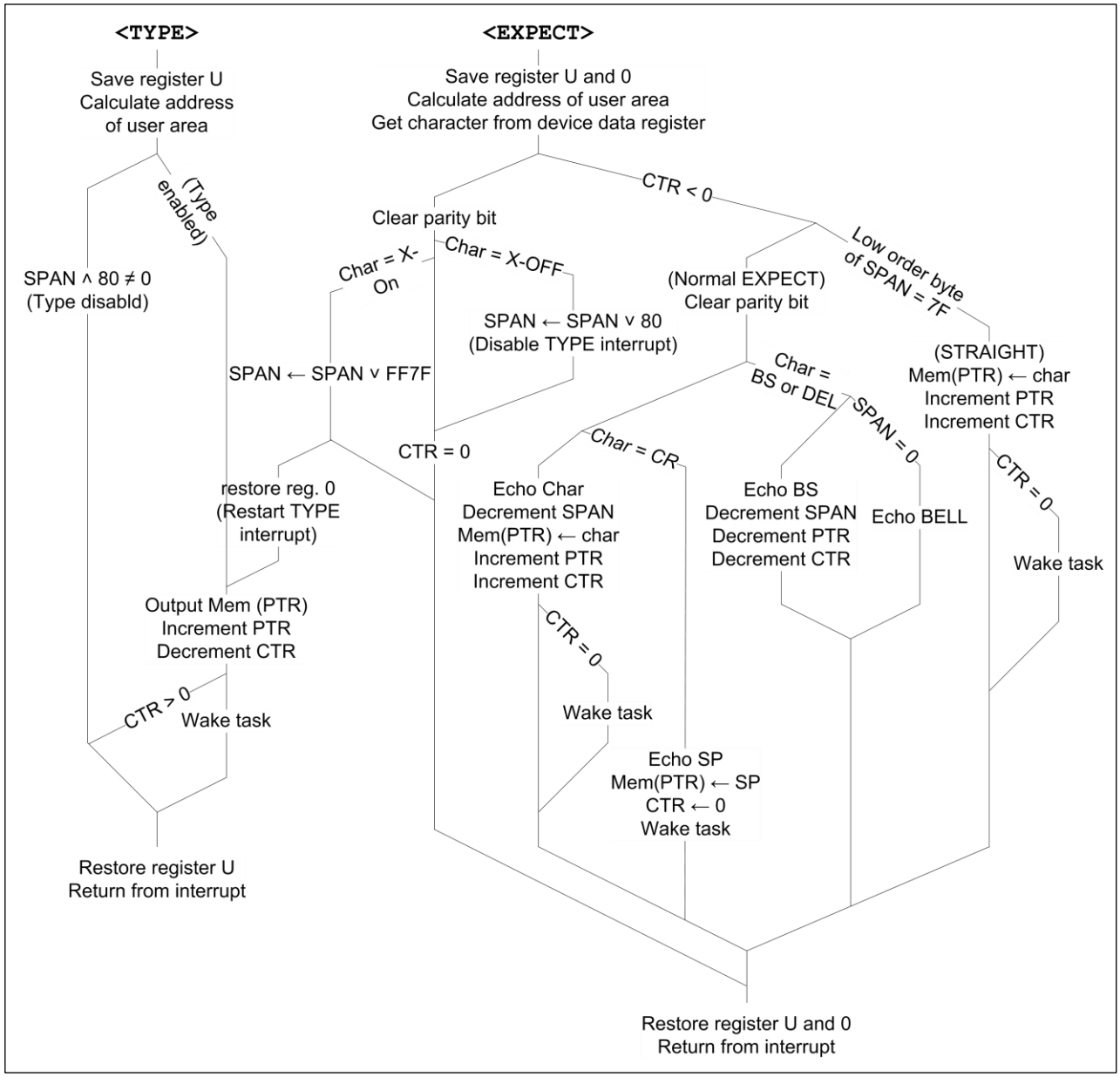


Fig. 3.8

D-Diagram of **<TYPE>** and **<EXPECT>** for PDP-11 polyFORTH with X-ON/X-OFF processing.

13. If **CTR** is zero (the input area is full), then **WAKE** is stored in the status area of the terminal's task, to awaken the terminal task.

14. The CPU returns from the interrupt.

The second case, a carriage return, is processed by the following steps:

1. 1 through 6 are identical to the normal character case, above.

7. The character is tested to see if it is a carriage return. The code is shared, but in this case the test succeeds.

8. A space is echoed.

9. **CTR** is set to zero.

10. **WAKE** is stored in the status area of the terminal's task, to awaken the terminal task.

11. The CPU returns from the interrupt (shared with normal case).

The third case, backspace or delete with a partially filled input area, is processed by the following steps:

1. 1 through 5 are identical to the normal character case.

6. Tests the character to see if it is **BS** or **DEL**. This test is shared with the normal characters case, but succeeds in this case.

7. **SPAN**, the number of characters input, is tested to see if it is zero (this test fails).

8. A backspace is echoed.

9. **PTR**, which points to the next empty byte of the input area, is decremented by one.

10. **CTR**, which contains the two's complement of the number of bytes remaining empty in the input area, is decremented by one.

11. **SPAN**, the number of characters input, is decremented.

12. The CPU returns from the interrupt (shared with normal case).

The fourth case, backspace or delete with an empty input area, is processed with the following steps:

1. 1 through 5 are identical to the normal character case.

6. Tests the character to see if it is **BS** or **DEL**. This test is shared with the normal characters case, but succeeds in this case.

7. **SPAN**, the number of characters input, is tested to see if it is zero. This test is shared with the **BS**-or-**DEL**-and-input-area-partially-full case, but succeeds in this case.

8. An ASCII **BELL** is echoed, to inform the typist there are no more characters to delete.

9. The CPU returns from the interrupt (shared with normal case).

The word **STRAIGHT** is a special modification of **EXPECT**. **STRAIGHT** takes the same stack arguments as **EXPECT**, but completely ignores the content of the data it moves. **STRAIGHT** is often used when transferring binary data over a serial link. On most systems, **STRAIGHT**'s interrupt routine is integrated with **EXPECT**'s interrupt routine by testing for a "**STRAIGHT** flag" in step 4 of **EXPECT**'s "normal character" case. The **STRAIGHT** flag must be in the user variable area, so no interference occurs when several terminal tasks are in operation.

For case 5, the steps that **STRAIGHT**'s interrupt routine perform are:

1. 1 through 3 are as in **EXPECT**'s normal character case.

4. Tests a "**STRAIGHT** Flag" in the user variable area. This test succeeds. In systems where **STRAIGHT** and **EXPECT** do not share interrupt code, this step is not performed.

5. The CPU jumps to step #10 of the "normal characters" case of the **EXPECT** interrupt, skipping all of the input testing steps. In systems where **STRAIGHT** and **EXPECT** share no interrupt code, Steps 10 through 14 are independently reproduced.

In case number six (the "X-OFF" case), the system is performing output when it receives an X-OFF character (13_H). The steps performed are:

1. 1 and 2 are identical to the normal character case.

3. **CTR** is found to be non-negative, meaning that the system is performing output.

4. The parity bit is cleared.

5. The input character is found to be X-OFF (13_H), meaning the device receiving output is getting full.

6. A type-interrupt-disable flag in the terminal task's user area is set. When the type interrupt checks this flag, the type interrupt routine does nothing (a character is not transferred).

7. The CPU returns from the interrupt (shared with the normal case).

In case number seven, the "X-ON" case, the system was performing output, and the output was turned off when the receiving device got full and sent an X-OFF. Now the receiving device has digested its data and it has sent an X-ON to tell the computer to resume transmission. The steps performed are:

1. 1 and 2 are identical to the normal character case.

3. **CTR** is found to be non-negative, meaning the system is performing output.

4. The parity bit is cleared.

5. The input character is found not to be an X-OFF (13_H).

6. The input character is found to be X-On (11_H), meaning output may be resumed.

7. The type-interrupt-disable flag is reset, so the type interrupt will transfer characters.

8. If **CTR**, the count of characters to output, is zero, then return from the interrupt (don't do Step 9).

9. Output a character, and restart the character transfer process.

REFERENCES

Vectored Routines, Section 3.1

3.7.3 Terminal Output—High Level Discussion

All terminal output occurs by means of the word **TYPE**. **TYPE** is a routine vectoring execution to some version (potentially different for every terminal task) of **(TYPE)** through the user variable **'TYPE**. **'TYPE** contains the address of the parameter field of **(TYPE)**. **TYPE** uses an address and a count, storing the address into the user variable **PTR**, and storing the count into **CTR**. After the stores, on systems with interrupts, **TYPE** initiates the **TYPE** interrupt routine **<TYPE>**, and enters the multitasking loop.

TYPE is used by three other routines of interest.

Word Description

>TYPE Uses an address and a count (like **TYPE**) but copies the string into **PAD** before typing it out. **>TYPE** is used in the **EDITOR** to type lines from the block buffer being edited. Because **>TYPE** calls **TYPE**, which enters the multitasking loop (as all input/output should), there is no guarantee that the contents of the block buffer will remain unchanged as the string is typed, so **>TYPE** copies a string into the user's **PAD** and types the string from there.

MSG A defining word used to output short, unchanging strings of characters to the terminal. **MSG** is used on most systems to define (**CR**), **(PAGE)**, and other short bursts of control characters. **MSG** is used as follows:

```
MSG name nn C, cc C, ... cc C,
```

where *name* is the name of the new word, *nn* is a byte telling how many characters to output, and each *cc* is the value of an ASCII character.

EMIT Outputs a single character from the least significant byte of the top of the stack, and then pops the stack. **EMIT** is often useful for initial "cut and try" definitions. After a sequence of characters is proven useful, it may be redefined using **MSG**, usually taking less space.

REFERENCES

String Output, Section 2.3.6.4
Vectored Execution, Section 2.4.8

3.7.4 Terminal Output—Low Level Discussion

On most systems, **TYPE** makes use of an interrupt routine. The **TYPE** interrupt routine (or **TYPE**, on systems without interrupts) transfers a character each time the terminal interface interrupts the CPU (or is ready). The terminal interface interrupts when it is ready for another character. In order to transfer a character, the **TYPE** interrupt routine must:

1. Find the correct task's user area so the **TYPE** interrupt can find **CTR** (the user variable containing the count of the number of characters to output), and **PTR** (the user variable pointing to the next character to output);

2. Output the character pointed to by the task's **PTR**;

3. Add 1 to the task's **PTR**;

4. Subtract 1 from the task's **CTR**;

5. If the task's **CTR** is zero, wake the task, and disable the **TYPE** interrupt.

Some systems use the X-ON/X-OFF convention. The X-ON/X-OFF handshaking sequence allows terminals with slow display-update logic to interface with fast baud rate interfaces over 3-wire transmission lines. The handshaking sequence is as follows (see Fig. 3.8):

1. The terminal's data buffer (which may be as small as three characters) approaches fullness, forcing the terminal to send an X-OFF character (ASCII DC3, 13_H);

2. The computer receives the X-OFF and stops transmitting;

3. The terminal empties its update buffer, and then sends an X-ON character (ASCII DC1, 11_H).

4. The computer receives the X-ON, and resumes transmission.

In a polyFORTH system not using X-ON/X-OFF handshaking, the **TYPE** and **EXPECT** interrupt routines are completely independent. In systems using X-ON/X-OFF, the **EXPECT** interrupt must turn the TYPE interrupt on and off.

In the implementation diagrammed in Fig. 3.8, **<EXPECT>** (the **EXPECT** interrupt routine) controls **<TYPE>** (the **TYPE** interrupt routine) by means of a flag in Bit 7 of the user variable **SPAN**. The bit is set by **<EXPECT>** when **<EXPECT>** receives an X-OFF character. When **<EXPECT>** receives an X-ON, the bit is reset, and if the user variable **CTR** is greater than zero, indicating more characters to **TYPE**, then **<EXPECT>** jumps directly into **<TYPE>**.

<EXPECT> must enter **<TYPE>** because the **TYPE** interrupt occurs when the interface is finished putting out a character to the terminal. Since **<TYPE>** ignored the last **TYPE** interrupt (because of the "type disable" bit in **SPAN**), the interrupt/output … interrupt/output sequence must be restarted by outputting a character again.

REFERENCES

The **EXPECT** Interrupt Routine, Section 3.7.2

3.7.5 Support of Special Terminal Features

Each terminal task defined by **TERMINAL** has unique user variables, including a port address, or other device and system specific interrupt vectoring. The design of polyFORTH assures that each terminal task has an associated, particular terminal. Each terminal may have different control character sequences for the following functions:

```
CR      PAGE      TAB      MARK      CLEAN
```

These functions have their addresses stored in user variables, so public programs (such as the **EDITOR**) can use private definitions of **CR**, **PAGE**, **TAB**, **MARK**, and **CLEAN** by directed execution.

Two user variables are defined to track the location of a terminal's cursor: **L#** and **C#**. **L#** contains the number of the line where printing is occurring (0-23 on most CRTs).

C# contains the column number where the next character will be placed (0-79 on most CRTs). Most special terminal functions read or change **L#** and **C#**.

On most systems, these terminal functions are defined as the most recent definitions when the terminal task was created by **TERMINAL**. When a terminal task is created, the default values of **'CR**, **'PAGE**, **'TAB**, and **'MARK** are compiled into the task's initialization table, so they can be copied into the task's user variable area when the task is constructed. The initialization table is provided so systems can be target compiled into read-only memory. Often it is convenient to initialize all terminals of a particular type by loading the new terminal definitions just before creation of all the terminal tasks servicing that particular type of terminal.

Note the convention followed by the names of the public functions, vectoring user variables, and device-specific primitives. All systems-defined vectored routines follow this naming convention. The terminal functions' behavior is as follows:

Public	Vector	Primitive	Stack	Function
CR	**'CR**	**(CR)**		Adds one to **L#**, sets **C#** to zero, and sends a terminal command equivalent to a teletype's carriage-return line-feed sequence.
PAGE	**'PAGE**	**(PAGE)**		Sets **L#** and **C#** to zero then clears the screen and homes the cursor on CRT-type terminals, and starts an empty page for all others.
TAB	**'TAB**	**(TAB)**	(l c)	Sets **L#** and **C#** with line and column numbers from the stack (with values between 0-23 and 0-79, respectively, for many CRTs) and positions the cursor appropriately, taking these values from the upper left-hand corner of a CRT-screen. For example:

```
5 0 TAB
```

positioning the cursor at the first character of the sixth line printed by the terminal.

Public	Vector	Primitive	Stack	Function
MARK	**'MARK**	**(MARK)**	(a n)	Uses the same arguments as **TYPE** (an address and a count). **.MARK** performs the same function as **>TYPE** except **MARK** also highlights the text it types. **MARK** goes through the following steps:

1. Copies the text to **PAD**, so text from a block buffer can be printed without multitasking interference.

2. Sets the terminal's highlighting (underline or reverse video are preferred where possible) or, on terminals that do not highlight, emits a caret ($5E_H$).

3. Types the text.

4. Unsets the terminal's highlighting.

CLEAN	**'CLEAN**	**(CLEAN)**	Clears to the end of the line.

Your polyFORTH system is shipped with several terminal configuration blocks. In addition, the default terminal configuration block is near the section of source code defining the screen editor. See these blocks and their shadow blocks for more detailed information.

REFERENCES

>TYPE, Section 3.7.3
Target Compilation of Tasks, Section 7.9
Terminal Tasks, Sections 4.8, 4.9, 4.10
Vectored Execution, Section 2.4.8
Vectored Routines, Section 3.1

3.8 THE FORTH BOOTSTRAP

There are two basically different kinds of read/write memory: volatile and non-volatile. Volatile memory loses its information when its power is cut. Most semiconductor memory is volatile.

To combat the problem of loading software into a powered-up computer, computer designers place a small, non-volatile read-only memory in the computer. This ROM contains code which is executed when the CPU powers up. Usually the code reads Track 0, Sector 0 of Disk 0 into memory, and then jumps to the beginning of the data it has read into memory. The ROM is called "the system's bootstrap ROM." The Data on Track 0, Sector 0 is usually called the "disk bootstrap." The term "bootstrap" is derived from the expression "It pulls itself up by its bootstraps."

polyFORTH is designed so the disk bootstrap and any associated device drivers reside in Block 0. The polyFORTH disk bootstrap reads the pre-compiled nucleus into memory and jumps to the first byte of it.

The word **RELOAD** is a convenience available on some polyFORTH systems allowing a programmer to simulate a "cold start" without physically touching a power switch or reset button. There are two ways **RELOAD** works:

1. **RELOAD** enables the bootstrap ROM (if necessary—sometimes the bootstrap ROM can be "turned off" so it is not accessible), then **RELOAD** jumps to the power-up entry point in the bootstrap ROM.

2. On some systems, the Forth system is in ROM, and **RELOAD** jumps to the first byte of ROM.

Whenever possible, **RELOAD** is designed not to erase block buffers, change the date variable **(TODAY)** or change the clock variable **(TICKS)**.

Usually, if a bootstrap installation requires a special command, the command is called **BOOT** and is available in the **DISKING** utility.

REFERENCES

DISKING Utility, Section 5.3
Power-up Initialization, Section 7.11

4.0 MULTITASKING

Multitasking allows a computer to appear to be doing many things at once. In particular, the pF/x multitasker is intended to provide service to multiple programs required to operate without any fixed timing relationship (*i.e.*, asynchronously).

This section explains how tasks are constructed, how they are controlled and how the CPU is shared between them.

Two types of tasks are detailed: background tasks and terminal tasks. Both types are fundamentally identical. A terminal task can be thought of as an extremely elaborate background task tailored to service a terminal and run the polyFORTH development environment. Tasks in some applications require a subset of the facilities available in the development environment. Terminal tasks can be easily "pruned" of unnecessary facilities.

REFERENCES

Target Compilation of Tasks, Section 7.9

4.1 FORTH RE-ENTRANCY AND MULTITASKING

When more than one task can share a piece of code, the code has a property called "re-entrancy." Re-entrancy is a valuable property, because when tasks can share code, memory is conserved.

Non-re-entrant routines contain sections subject to change as the program runs. Self-modifying code is not re-entrant. Routines with "private" variables are not re-entrant. Re-entrant routines can have private constants (a constant's value does not change). Re-entrant routines can always be programmed into a read-only memory.

Forth routines are naturally re-entrant. Most Forth routines keep their intermediate results on either the parameter stack or the return stack. Programs to handle text or other tables can be designed to keep their tables in the section of random access memory allotted for each task. A facility is provided to allow a programmer to define public routines to access variables, but still retain re-entrancy by accessing private versions of these variables in each task (such variables are called "user variables"). Forth routines can be made completely re-entrant with very little effort.

Because Forth routines are naturally re-entrant, pF/x allows tasks to share routines in a single "public" dictionary. This practice conserves large amounts of memory. In most applications, all system and application routines can be shared (with the minor exception of the I/O instructions on certain processors). Therefore, as few as 2048 bytes of read/write memory per terminal can support a satisfactory program development environment. Terminal tasks not requiring a private dictionary (such as printer spooling tasks) can operate with 300 bytes. The minimum size for a useful task with no private dictionary is about 250 bytes. Some applications (PBX's, process control, and some communications systems) naturally use large arrays of small tasks, with each task running a simple shared program.

REFERENCES

User Variables, Section 4.6

4.2 PRINCIPLES OF OPERATION

The polyPF/x multitasker is designed to fulfill several goals:

1. Provide asynchronous execution of code.
2. Be convenient to use.
3. Be fast in execution.
4. Be simple to understand.
5. Use a minimum amount of memory.
6. Be independent of particular hardware configuration (*e.g.*, a clock is unnecessary).

The pF/x multitasker satisfies 2, 3 and 4 above by consisting of only about 13 words. Number 5 is a consequence of Forth's basically re-entrant structure (see "Forth Re-entrancy," referenced at the end of this section). Numbers 1 and 6 are assured by the way Forth schedules tasks. Forth services tasks when an executing task stops to await some form of I/O.

Simplicity and high performance are assured because control is passed from one task to another only at known, programmer-controllable points, and always between Forth words. This principle greatly simplifies the context-switching operation (thus reducing overhead) and simplifies the programmer's task of writing routines for a multi-user environment.

The rest of this section is a detailed discussion of the scheduling algorithm, its associated words and some useful techniques. For a discussion of other words, see their associated sections.

A "round-robin" algorithm (see Fig. 4.1) schedules processor time. Each task has a turn in control until it executes the high-level words **PAUSE** or **STOP**, or the assembler code ending **WAIT**. Most words performing asynchronous hardware operations (*e.g.*, **TYPE**, **EXPECT**, **BLOCK**, and **BUFFER**) contain a **WAIT** or a jump to **PAUSE**, so that while a task is waiting for an I/O operation to be completed other tasks can use the CPU. Since Forth is naturally very fast, tasks tend to spend much of their time awaiting I/O. Tasks that are performing extensive computations may be prevented from impacting overall system performance by using **PAUSE** in a few regularly executed words. The PF/x multitasker is said to be "I/O driven."

The following words control use of the CPU by tasks:

Word	Function
PAUSE	Suspends the task that calls **PAUSE** to allow all other tasks one turn in control of the CPU.
STOP	Puts the task that calls **STOP** to sleep until that task is awakened by an interrupt routine or by some other task.
WAIT	An assembler code ending that behaves exactly as **STOP**.

The round robin is implemented as an endless loop of jump instructions (see Fig. 4.1). Each task has its own jump instruction, which transfers control to the jump instruction of the next task. When a task is going to awaken, the task's jump instruction is replaced by an instruction to transfer control to the machine code to awaken the task. This special instruction is usually called **WAKE**, and is often a trap instruction. The round robin is sometimes called "the **PAUSE** loop," "the idle loop," or "the multitasking loop." The address of the first byte of a task's jump instruction is pushed on the stack by the high-level Forth word **STATUS**. In assembly code, the address of the current task's **STATUS** is available in a register or cell called **U**. The address used by the jump in **STATUS** is located at **STATUS 1+** on most 8-bit processors, at **STATUS 2+** on most 16 and 32-bit processors.

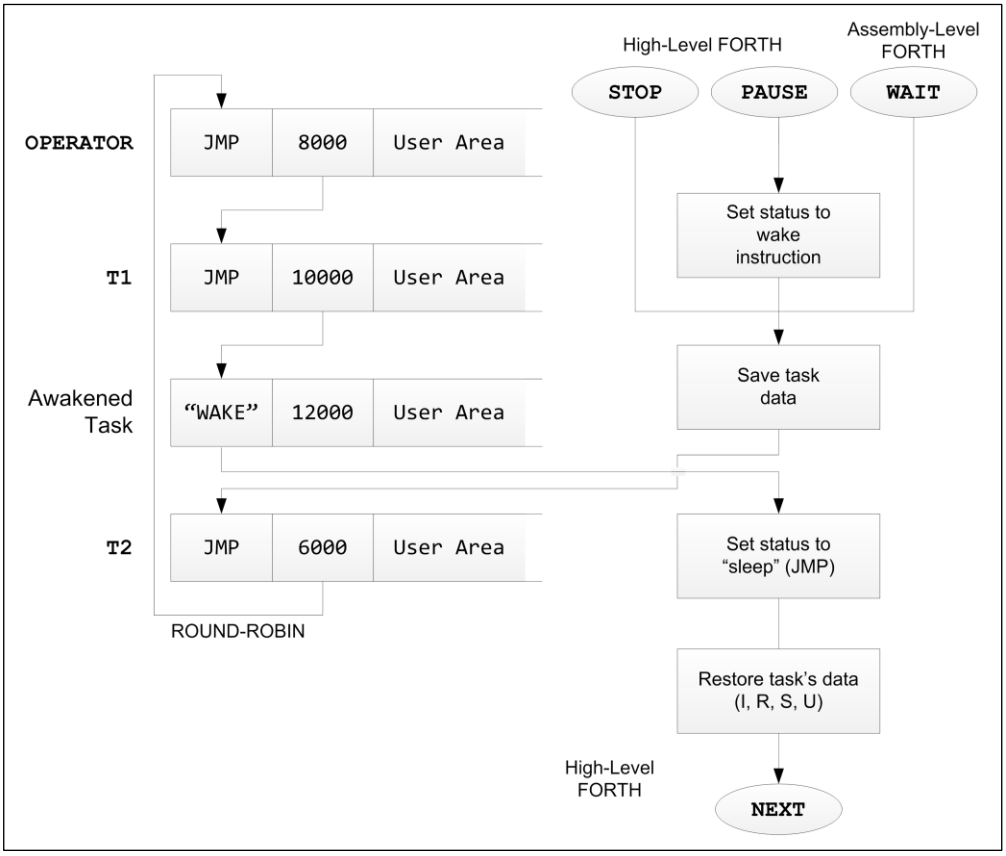


Fig. 4.1

Schematic of the relationship between the word **PAUSE** and the multitasking round robin. **PAUSE** does not enter the round robin at a fixed place. **PAUSE** always runs the following task.

Interrupt routines are often used to awaken tasks. For example, whenever the interrupt routine for **EXPECT** sees a carriage-return, the interrupt routine awakens the terminal task associated with the terminal. A common way to perform complex non-critical interrupt servicing is to have the interrupt routine perform all time-critical operations, and then store **WAKE** into a task's **STATUS**. When the round robin gets around to the task, the task resumes execution after the previous **STOP**, and runs until it executes another **STOP** or **WAIT**s for an I/O operation. Tasks typically perform asynchronous operations such as data reduction and logging. An example of a data reduction loop being run by a dedicated task might be:

```
: COLLECT   BEGIN ACCEPT DATA REDUCE
   STORE STOP AGAIN ;
```

The interrupt routine that services the data source for this example will awaken the task out of the word **STOP**, by storing **WAKE** in the task's **STATUS** when data is ready to be accepted. The specific method for doing this is discussed in the *CPU Supplement* for each system.

The most general case of changing turns in the round robin occurs when control of the CPU is relinquished from high-level Forth, with arrangements for the task to automatically awaken and resume execution on its next turn. **PAUSE** performs this most general case. **PAUSE** can be embedded in complex calculations which perform no I/O and which might otherwise cause a particular task to control the CPU for undesirably long periods. Consider the following collision orbit calculation for example:

```
: POSITION   X STEP  Y STEP ;
```

```
: ?COLLIDE   30000 0 DO  POSITION HIT
   PAUSE LOOP ;
```

In this example the word **STEP** is assumed to have been defined to perform the calculations for integrating the next step in the target X or Y coordinate. **HIT** expects the coordinate of the target (computed by **POSITION**) on the stack and performs appropriate course corrections. Since all of these computations are time consuming and since other functions must be running concurrently, it is desirable to give up the CPU for one turn around the round robin for each step in the integration. Inserting the **PAUSE** in the loop accomplishes this.

The related words, STOP and WAIT (see Fig. 4.1) share much of the code for PAUSE. These are the steps that **PAUSE** performs:

1. The **WAKE** instruction is stored into **STATUS**, replacing the round-robin jump. This step ensures that the task executing **PAUSE** will awaken on its next turn. On many machines, **WAKE** is equivalent to a subroutine jump to the code for step 5, below.

2. The system pointers **I** and **R** are saved by being pushed onto the current task's parameter stack. This portion of the code is the entry point for **STOP** and **WAIT**, which by skipping step 1 do not automatically resume execution of the current task on the next turn.

3. The parameter stack pointer (**S**, in assembler) is saved in a reserved location in the user area. At the completion of this step, all non-recoverable unshared task data for the address interpreter has been saved. Since tasks only relinquish the CPU between Forth words, other registers do not have to be saved.

4. The CPU jumps to the location whose address follows **STATUS** (which is the next task's **STATUS**), and proceeds to jump through the circular round-robin loop until a **WAKE** instruction is encountered. The **WAKE** transfers control to step 5.

5. The address of the new task's **STATUS** is stored in **U**. In machines that use a **JSR** for the **WAKE** instruction, the address of **STATUS** can be calculated using the address that **WAKE** left on the subroutine linkage stack. Some way of obtaining the **STATUS** address is always available.

6. Using **U** to find the task, the parameter stack pointer is restored, and then **I** and **R** are restored from the new task's parameter stack.

7. A jump to the next task is stored into the current task's **STATUS** to make the current task's state "don't awaken."

8. Finally, **NEXT** is executed, which invokes the Forth word indicated by the new task's **I**.

REFERENCES

Assembler Code Endings (**WAIT**), Section 6.2
Forth Re-entrancy and Multitasking, Section 4.1

4.3 DEFINING A BACKGROUND TASK

BACKGROUND tasks have a parameter stack, a return stack, and space for variables which are not shared (these are called "user" variables). **BACKGROUND** tasks do not service a terminal or have a private dictionary.

The word **BACKGROUND** allocates memory in the dictionary and sets up a two-cell task definition table for a **BACKGROUND** task (see Fig. 4.2). **BACKGROUND** is a defining word which expects the sizes of the user variable area (sometimes called "the user area"), the parameter stack, and the return stack. All sizes are in bytes. An example of **BACKGROUND**'s use is:

```
16 64 48 BACKGROUND SCRIBE
```

This defines a task whose name is **SCRIBE**, which has 16 bytes of user area, 64 bytes of parameter stack and 48 bytes of return stack. Task names should be job assignments (as here) or proper person's names.

In the example above, 16 will be the number of bytes reserved for the user area. The smallest possible user area is 16 bytes, and contains the task **JMP** in **STATUS** (one or two bytes), the address of the next task in the round robin (two bytes), the stack pointer (two bytes) and **S0**, the bottom of the parameter stack (two bytes). To determine the minimum size for your system, see your *CPU Supplement*. The three extra cells in **SCRIBE**'s user area could be used by **SCRIBE**'s program to keep a sample count, a virtual array address, and a device I/O address for a data logging application.

The 64 in the example indicates the parameter stack can have a maximum size of 32 single-precision integers or 16 double-precision integers. (On 32-bit systems the maximum size would be 16 single-precision or 8 double-precision integers.) Note that there must be sufficient space to store **I** and **R** on the parameter stack, as well as whatever arguments are present when the task enters the multitasking loop.

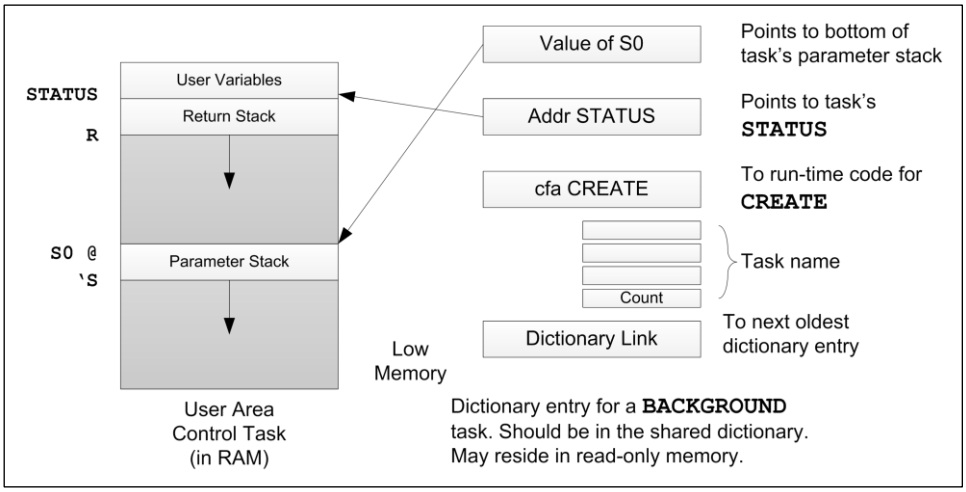


Fig. 4.2

Layout of memory allotted for a background task. The task definition table is compiled by **BACKGROUND**. The executable portion of the task is compiled by **BUILD**.

The 48 indicates the program can nest Forth words, loop parameters, etc. to a depth of 24 return stack entries (12 entries on 32-bit systems).

The total size of **SCRIBE** in this example is 130 bytes, excluding the dictionary head, because a cell containing the address of **SCRIBE**'s **STATUS** is added by the word **BACKGROUND**. In addition, a cell is compiled just after the **STATUS** address that tells where the new task's S0 should be. The data to calculate **S0** is available when **BACKGROUND** is executed to compile the task definition table, but need not be preserved after the new task has been fully initialized. Thus the cell for the value of **S0** is included in the space allotted for the parameter stack. The runtime behavior of words created by **BACKGROUND** is to return on the stack the address of the task definition table, the first cell of which contains a pointer to the first byte of the task's **STATUS**. Thus, the location of **SCRIBE**'s **STATUS** is found by the phrase:

```
SCRIBE @
```

If a task has an associated interrupt routine (as terminal tasks do, for example), the interrupt routine may be defined or linked to the task when the task's memory is allotted by an application word performing **BACKGROUND** plus whatever additional functions are needed at this time. When many generically similar tasks need to be defined, a word combining the functions of memory allotment and interrupt-routine-linkage is the most

convenient way to define them. The combination is convenient because memory allotment produces the addresses needed for linking the interrupt-routine to a task.

BACKGROUND allots space for a task in the dictionary, but does not link the task into the round robin, or make the task run a program. This will be described in the next section.

When target-compiling systems with tasks in read-only-memory, **BACKGROUND** must be executed as part of the target compilation process. **BACKGROUND** must be part of target compilation because **BACKGROUND** allots memory in RAM and creates the table in ROM that will be used after power-up in the target system to build the task's user area and stacks in RAM.

In a resident environment, tasks are usually defined and initialized at the same time, during the loading of system electives (controlled by Block 9).

REFERENCES

Initializing a **BACKGROUND** Task, Section 4.4
I and **R**—Address Interpreter, Section 1.2.5
Making a **BACKGROUND** Task Run a Program, Section 4.5
Target Compiling Multiprogrammed Applications, Section 7.9

4.4 INITIALIZING A BACKGROUND TASK

When pF/x starts up, at least one task exists. This task is a terminal task called **OPERATOR**. If no other task exists, **OPERATOR**'s jump address will contain the address of **OPERATOR**'s **STATUS**. In this way, the round robin first consists of a single jump instruction, which jumps to itself.

The word **BUILD** initializes the user area for a new background task and links the task into the round robin.

1. **BUILD** copies the complete jump instruction, including the address of the next task, from the **STATUS** of **OPERATOR** into the **STATUS** of the new task.

2. **BUILD** stores the address of the new task's status in **OPERATOR**'s jump address, so that the CPU jumps from **OPERATOR** to the new task to the task which **OPERATOR** formerly preceded.

3. **BUILD** gets a copy of the value for **S0** (the bottom of the new task's parameter stack) calculated by **BACKGROUND** and compiled just after the **STATUS** address, and stores the new task's value for **S0** into the user variable **S0**.

In a resident system tasks should be defined and initialized when the electives are loaded from Block 9 by the command **HI**. Conventionally, task definitions are in Block 36, possibly also 37. It is not good practice to define a task in a private terminal partition, since an **EMPTY** would "forget" the task and break the round robin. Definitions in the public dictionary cannot be forgotten, and thus the tasks defined there are safe.

Here is an example of how to define a task, and then link it into the round robin. When the electives are loaded, the task is defined by the phrase:

```
16 64 48  BACKGROUND SCRIBE
```

In a target-compiled system, task initialization is usually part of the power-up sequence. In a resident system, task initialization is usually performed during the electives load, just after task definition:

```
SCRIBE BUILD
```

SCRIBE leaves the address of the task definition table on the stack. This is used by **BUILD** to set up the task's stack and user area in RAM.

Although the task exists and is part of the round robin, note that it is not yet running a program. The words the task executes may be defined much later.

REFERENCES

Defining a **BACKGROUND** Task, Section 4.3
Making a **BACKGROUND** Task Run a Program, Section 4.5

4.5 CONTROLLING A BACKGROUND TASK

After a task has been defined, with memory allotted to it by **BACKGROUND**, and is linked into the round robin by **BUILD**, the new task is asleep. This is necessary, because the task still has not been given the values for I and R that determine what word(s) the address interpreter will execute for the task.

The word that makes a task run a program is **ACTIVATE**. **ACTIVATE** expects on the stack the address of a task definition table, as returned by executing a task name. **ACTIVATE** clears the parameter and return stacks of the task, and then awakens the task with a value of I that points to the word immediately following **ACTIVATE**. **ACTIVATE** must be used in a : definition. The following example performs all initialization required to run a **BACKGROUND** task. The words following **ACTIVATE** must end in **STOP** or an endless loop.

Assume that a background task was defined and initialized during the electives load, by the following phrases (described in previous sections):

```
16 64 48 BACKGROUND SCRIBE
SCRIBE BUILD
```

When the application is loaded, the following definition will use a specified task to record data samples onto disk:

```
: RECORD ( a)   ACTIVATE
   BEGIN  COLLECT >DISK  PAUSE  AGAIN ;
```

The above version of **RECORD** would be used in the following phrase:

```
SCRIBE RECORD
```

The words in the definition of **RECORD** between **ACTIVATE** and ; represent a definition that performs a hypothetical data logging activity. The word **ACTIVATE** uses the address of a task definition table from the stack (put there by executing the task's name **SCRIBE**) and forces that task to execute the words following **ACTIVATE** in the colon definition containing **ACTIVATE**. In the above example **BUILD** is kept separate from **RECORD** because a task can only be built once, but can be activated many times.

For convenience in the following discussion, a "slave" task is the task **ACTIVATE**d by some other task. The "master" task is the one executing **ACTIVATE** to activate the slave task. These are the actions taken when the phrase **SCRIBE RECORD** is executed:

1. The slave task (called **SCRIBE** in the example) has its return stack emptied. If the slave task were put to sleep by **PAUSE**, the value of the slave's return stack pointer would be placed on the slave's parameter stack. **ACTIVATE** simulates **PAUSE**'s action, by storing the address of the bottom cell of the slave's return stack into the bottom cell of the slave's parameter stack. The address of the bottom cell of the slave's return stack is calculated from the address of the slave's **STATUS** cell. These cells are adjacent (see Fig. 4.2).

2. The address in the master task's I when **ACTIVATE** is called is put in the slave task's stack where it would have been saved by **PAUSE**. Thus when the slave awakens its I will point to the address of **COLLECT** in the definition of **RECORD** (no address is compiled for the **BEGIN**).

3. This step essentially leaves the slave's parameter stack empty except for the new values of **I** and **R** stored there. The slave's parameter stack pointer is set to point to the slave's new value for **I** on the parameter stack. The slave's parameter stack pointer is saved in the slave's user area until the slave begins executing.

4. A **WAKE** instruction is stored into the slave's status cell.

5. The master task now finishes running **ACTIVATE**. The last step the master task must perform is to leave the colon definition containing **ACTIVATE** without executing any more of that definition's code. Therefore, the last step of **ACTIVATE** is to jump to **EXIT** instead of **NEXT**.

Note that code following an **ACTIVATE** must never reach the **;** because the **EXIT** which is compiled by a semicolon will attempt to pop an empty return stack. Therefore, the code following an **ACTIVATE** must end either in an endless loop (as in the example) or in the word **STOP**.

Since neither the master nor the slave task ever execute the **EXIT** compiled by **RECORD**'s **;**, it is common to follow such a **;** with the word **RECOVER**. **RECOVER** backs up the dictionary pointer, removing the un-needed **EXIT**.

Another example of a control word, this time using the word **STOP**, is:

```
: HALT ( a)   ACTIVATE STOP ;
```

Because **ACTIVATE** forcibly resets a task's execution environment (empties both stacks and sets **I**), the definition of **HALT** given in the example above will forcibly **STOP** a specified task. Although **HALT** is occasionally useful, it is not a predefined Forth word because a slightly different definition is useful for terminal tasks.

REFERENCES

BEGIN, Section 2.3.1
Definition of **HALT** for a Terminal Task, Section 4.10
PAUSE, **STOP**, and **WAIT**, Section 4.2

4.6 USER VARIABLES

In pF/x, many tasks can share the code for the address interpreter, I/O drivers, etc. Each task will have different data for these facilities: the **EDITOR**, for example, needs to remember which line the user is editing. The fact that all users have private copies of the variables **SCR** and **CHR**, which control the editing cursor, enables them to edit concurrently without conflicts.

pF/x accesses private variables through a facility called **USER** variables. **USER** variables are accessible to programs residing in the public dictionary, yet they are not shared between tasks.

There are two ways to define **USER** variables. This is the "absolute" form:

```
0 USER STATUS   10 USER S0   12 USER 'IDLE
```

The number preceding the word **USER** is an offset in bytes from any task's **STATUS**. The name of the variable follows **USER**. The compile-time behavior of **USER** is like a constant: the number is compiled immediately after the head. The run-time behavior is to add the number to the system pointer **U** to produce an address in the task's private user variable area. **U** contains the address of the first byte of the task's round-robin jump instruction. The run-time code for user variables is written in assembly language and uses the **U** register (or location), rather than the word **STATUS**, but a high-level definition of **USER** could be written as:

```
: USER ( n)   CREATE ,  DOES> ( a)
   @  STATUS + ;
```

The second method of defining **USER** variables is the "relative" method, using the defining word **+USER**. This is most appropriate when a group of user variables is being defined. **+USER** expects on the stack an offset in the user area plus a size (in bytes) of the new **USER** variable being defined. A copy of the offset will be compiled in the definition of the new word, and the size added to it and left on the stack for the next one. Thus, when specifying the series, all you have to do is start with an initial offset and then specify the sizes.

For example, the standard pF/x **USER** variables region on an 8086 begins like this:

```
0   6 +USER STATUS   2+ 2 +USER S0
   2 +USER 'IDLE ...
```

What is compiled is equivalent to the previous example. However, it is convenient to see explicitly that the task status area is 6 bytes long (two bytes for the jump and four for the "follower" task address), followed by two more bytes (for the stack pointer "save" location) and then **S0**, etc.

When you are finished defining **+USER** variables, you should **DROP** the offset. If you are defining an additional group of user variables, you may pick up the offset from the last one by a phrase such as:

```
' STATUS C@
```

(**' STATUS W@** on 32-bit systems) replacing the initial 0 in the previous example.

The absolute method (**USER**) is most appropriate when re-naming an existing user variable; for example, if **TERMINAL** tasks in your application will not be using the polyFORTH editor, you may use the space occupied by its user variables for application user variables.

Terminal tasks generally have about 128 bytes (256 on 32-bit systems) of user area, devoted to I/O, task management, dictionary management, private text interpretation, and the editor. The minimum size for a background task's user area is 16 bytes (see your *CPU Supplement*) for task management. The task management bytes usually consist of:

1. A jump instruction to the next task's jump instruction.
2. A save area for the task's stack pointer while it is inactive.
3. The address of the bottom of the parameter stack (**S0**).

A task may need to initialize another task's user variables, or read or modify them. The word **HIS** allows a task to access another task's user variables. **HIS** takes two arguments: the address of the task of interest, and the address of the executing task's user variable of interest. For example:

```
2 CHUCK BASE HIS !
```

will set the user variable **BASE** of the terminal task named **CHUCK** to 2 (binary). **HIS** subtracts the **STATUS** address of the executing task to get the offset and then adds the offset to the **STATUS** address of the desired task.

HIS is useful for initializations. The sample data logger developed in Sections 4.3, 4.4, and 4.5 would be more useful if the user could specify the number of samples and did not have to specify the task to run the data logging. The user variable definitions and **COLLECT** are included as examples of the use of user variables. **COLLECT** reads data into a block buffer. **SHELF** contains the block number of the next free disk block.

A/D is assumed to be defined to call a code routine to request a sample from an analog-to-digital converter and **WAIT** for an interrupt signaling the completion of the conversion. The interrupt code will read the sample and leave it in a buffer where **A/D** will fetch it and leave it on the stack.

```
10 USER SHELF   12 USER #SAMPLES

: COLLECT   #SAMPLES @ 0 DO  A/D
   SHELF @ BLOCK I 2* + ! UPDATE  LOOP ;

: RECORD  ( n n2)
   512 MIN  SCRIBE #SAMPLES HIS !
   SCRIBE SHELF HIS !
   SCRIBE ACTIVATE  COLLECT  STOP [
```

This version of **RECORD** would be used with the block number and number of samples as its argument; the task address is built in (**SCRIBE**). The following phrase would record 500 samples into Block 30:

```
30 500 RECORD
```

Note that in this more sophisticated version the **PAUSE** to enter the multitasker is replaced by the **BLOCK** in **COLLECT**, which contains a call to **PAUSE**.

Also note the use of **[** instead of **;** in **RECORD** avoids the compilation of an **EXIT** (**;** compiles an **EXIT**) which will never be executed. See Step 5 of **ACTIVATE** (Section 4.5).

The map of user variables in Table 4.1 shows the user variables required by a fully interactive PF/x terminal task. Your CPU's implementation may require variables not listed—see your implementation's system and user variables block.

Although the actual order varies, all systems possess these user variables. You can obtain the absolute address of Location 0 of a task's user area by typing:

```
task-name @
```

See Block 198 for the exact organization of your user variables. Generally, the most-used user variables have smaller offsets, so tasks needing only part of the user variables can have a minimal-sized user variable area. Note also that the user variables that SEND moves from the master to the slave terminal task are grouped together. Names in parentheses are used merely as identification for the reader; they are not Forth words. To address these locations, use one of the Forth words in the table (such as **SCR**) and add or subtract the appropriate number of bytes.

Table 4.1

Map of a Typical User Area

Name	Content
STATUS	Indicates whether the task is ready to become active.
(Follower)	Address of the next user in the multi-tasking chain.
(**S**)	Stack pointer, saved from when the task was last active.
S0	Pointer to the bottom of the parameter stack and the start of the message buffer (for terminals).
'IDLE	Address of the "idle behavior" routine for tasks.
SPAN	Length of the string actually received by **EXPECT**.
CTR	Counter for character-oriented I/O
PTR	Pointer to the address for the current I/O.

The following are needed only for terminal tasks.

Name	Content
'KEY	Contains the most recent character received since the last **EXPECT** or **STRAIGHT** operation or 0 if none.
DEVICE	Terminal device address or other device information.
'EXPECT	Address of the task's **EXPECT** routine.
'TYPE	Address of the task's **TYPE** routine.
'CR	Address of the terminal "new-line" routine.
'PAGE	Address of the terminal screen clear or form-feed routine.

'MARK Address of a routine to mark the editor's cursor on the terminal.

'TAB Address of a routine to position the terminal's cursor.

'CLEAN Address of the task's "clear to end of line" routine.

Name Content

C# Task's current cursor position (column).

L# Task's current cursor position (line).

TOP Top of task's screen scrolling area (line#).

H Dictionary pointer (to the next available byte).

(**H 2+**) Pointer to the "empty" dictionary location. (**H 1+** on 8-bit processors and H 4+ on 32-bit processors.)

OFFSET Offset automatically added to drive-dependent block numbers to calculate absolute block numbers.

BASE Number conversion base (eight for octal, ten for decimal, sixteen for hex).

#TIB The number of characters remaining to interpret in the input stream.

BLK Number of the block being interpreted (zero denotes a terminal).

>IN Pointer to the text interpreter's current position in the input message buffer.

CONTEXT Index of the vocabulary to be searched, followed by eight cells for vocabulary heads.

CURRENT Index of the vocabulary into which new definitions will be put.

SCR Current block number. Used by **LIST** and most **EDITOR** words.

CHR Current character position in **EDITOR**.

EXTENT Contains the number of characters in a block that are affected by character editor functions (normally 64).

WIDTH Maximum width of a name, in bytes.

(**WIDTH 1+**) Default width of a name.

The following user variables are added by the Data Base Support option:

Name Content

R# Current record number.

F# Address of the file definition area for the current file.

L/P Number of lines per page on the task's output device. More than 60 indicates a printer.

P# Current page number for the report generator.

RPT Address of the report heading routine for the report generator.

REFERENCES

Step 5 of **ACTIVATE**, Section 4.5
WAIT, Section 4.2
Data Base Support, Section 8.0

4.7 SHARING RESOURCES WITH GET AND RELEASE

Some system resources must be shared between tasks without giving any one task permanent control. Disk units, tape units, printers, non-re-entrant routines, and shared data areas are all examples of resources limited to use by only one task at a time.

PF/x controls access to these resources with two words that resemble Dijkstra's semaphore operations. (Dijkstra, E.W., Comm. ACM, **18**, 9, 569.) These words are **GET** and **RELEASE**. An example of their use is:

```
: BLOCK ( n - a)   OFFSET @ + DISK GET
   block  DISK RELEASE ;
```

In the example above, the word **BLOCK** requires private use of the disk controller and block buffers while it performs the actual work of ensuring that a disk block is placed in a block buffer. The phrase **DISK GET** waits in the **PAUSE** loop to obtain private access to these resources. The phrase **DISK RELEASE** releases these resources, without awakening another task.

The word **DISK** in the example above is an example of a "facility" variable. A facility variable is a normal **VARIABLE**, but it must be in the public dictionary. When a facility variable contains zero, no task is using the facility represented by the variable. When a facility is in use, then its facility variable contains the address of the **STATUS** of the task that owns the facility. The word **GET** waits in the multitasking loop until the facility is free or owned by the task which is running **GET**. High-level code for **GET** would be:

```
: FREE ( a - a t)   @  DUP 0=  SWAP
   STATUS =  OR ;

: GET ( a)   BEGIN  PAUSE  FREE UNTIL
   STATUS SWAP ! ;
```

GET checks repeatedly whether a facility is free. In conventional operating systems, this time is called a "busy-wait," and heroic measures are taken to conserve it. Experience has shown that queued resource schedulers (the "fast" conventional solution) take more time and are more prone to deadlock than assembly-coded versions of **GET**. **GET** is actually written in code, and the overhead rarely exceeds two or three machine instructions. Maintaining a queue is almost always slower.

RELEASE checks to see whether a facility is free or already owned by the executing task. If so, **RELEASE** stores a zero into the facility variable. Using the definition of **FREE** above, a high-level definition of release would be:

```
: RELEASE ( a)   FREE IF  0 SWAP !  ELSE
   DROP  THEN ;
```

Note that **GET** and **RELEASE** can be safely used by any task at any time, without endangering the system's integrity.

pF/x does not have any safeguards against deadlocks, where two (or more) tasks hang up because each wants a resource that the other has. For example:

```
: 1HANG   DISK GET  TAPE GET ... ;

: 2HANG   TAPE GET  DISK GET ... ;
```

If **1HANG** and **2HANG** are run by different tasks, the tasks could eventually deadlock.

The best practice to avoid deadlocks is to avoid **GET**ting two resources at the same time. In the disk/tape case, for example, one would use **BLOCK**, move data to a local buffer and then to tape. In almost all cases there is a simple way to avoid concurrent **GET**s. However, a poorly-written application might have the conflicting requests occur on different levels of nest, hiding the problems until a conflict occurs.

Remember that it is better to design an application so it **GET**s only one resource at a time—deadlocks are impossible in such a system.

4.8 DEFINING A TERMINAL TASK

The word **TERMINAL** allots memory to a terminal task (see Fig. 4.3) and creates the task definition table for a task that controls a serial port. A **TERMINAL** task can also have a private dictionary, though many don't use one. The serial port is the critical difference. The return stacks of a system's terminal tasks have a standard size, and space is allowed for input message buffers and private dictionaries. Because terminal tasks can have their own dictionary, they have user variable space for dictionary management, disk access, editing and private interpretation of both blocks and terminal input.

TERMINAL uses a machine-dependent set of arguments to describe terminal I/O arrangements and task-size. This example is from the PDP-11 (there may be several examples in your system listing):

```
OCTAL  175610  310  DECIMAL
   2048  TERMINAL  CHUCK
```

The first number is a port address, the second an interrupt vector, the third (in decimal) is the total number of bytes to be allotted to a terminal task which will be called **CHUCK**. The task size argument is required by all implementations of **TERMINAL**, but the other arguments vary. Note that the names of tasks are proper names (as here) or job assignments.

Besides allotting space, **TERMINAL** may also set up interrupt routines for **TYPE** and **EXPECT** for the terminal task. The exact method is dependent upon the serial interface hardware. **TERMINAL** also compiles a task definition table that contains initialization values for certain user variables.

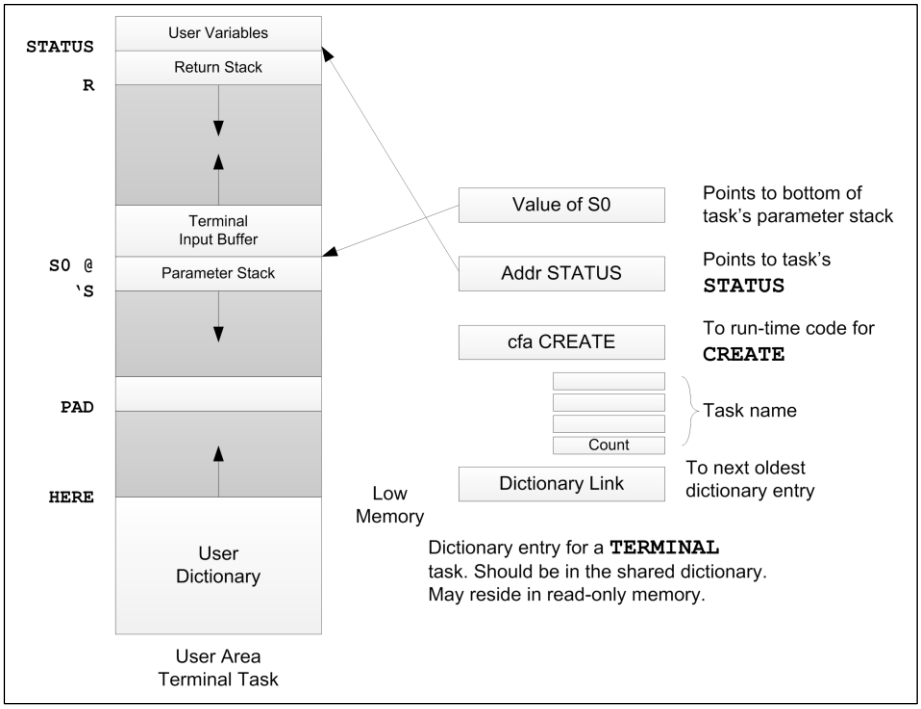


Fig. 4.3

Layout of memory allotted for a terminal task. Read-only memory section is compiled by **TERMINAL**.

The initialization table compiled by **TERMINAL** contains values for the task address, **S0**, **'CR**, **'PAGE**, **'TAB**, **'MARK**, **'TYPE**, **'EXPECT**, **CLEAN**, and usually a hardware-dependent field **DEVICE**. Some of these values are

copied to the terminal task's user variables when the task is initialized by the word **CONSTRUCT**, which is described in the next section.

S0 will be copied to the task's user variable area by **CONSTRUCT**, along with the "tick" variables (**'CR**, **'PAGE**, etc.) which all contain the address of the parameter field of a routine to perform a terminal function (see Special Terminal Functions, Section 3.7.5). Note that the "tick" variables are set using the most recently compiled versions of the terminal-dependent functions. If, for example, an ADM-3A terminal task has been defined with a special **(CR)** routine, and a printer task follows it with no **(CR)** routine of its own, the printer task will be set up using the ADM-3A's **(CR)** routine.

The task definition routines look for words with a name following the parenthesized convention **(NAME)**. For example, the routine to perform a "clear screen" operation should be called **(PAGE)**. Its address will be stored in **'PAGE** when a task is set up and **PAGE** will execute it. For example, **'TAB** will be set to the address for the ADM-3A's **TAB** if the ADM-3A version of **(TAB)** is nearest to the top of the dictionary. Where several of the same model of terminal are to be connected to one Forth computer, a good approach is to define their terminal-dependent functions once and then establish the terminal tasks contiguously in memory.

DEVICE contains the data required by **<TYPE>** and **<EXPECT>** (the terminal interrupt routines) to address the specific terminal associated with the terminal task. Typical values in **DEVICE** might be a port address in a memory-mapped I/O machine, a port number for a multiplexer, or an input instruction followed by a return from a subroutine.

When building a task definition table in read-only memory, **TERMINAL** must be executed at target compilation time, because **TERMINAL** builds the ROM table used to construct the task after power-up in the target system.

REFERENCES

CONSTRUCT, Section 4.9
Support of Special Terminal Functions, Section 3.7.5
Terminal Drivers, Sections 3.7.2, 3.7.4

4.9 INITIALIZATION OF A TERMINAL TASK

After a terminal task has been defined by the word **TERMINAL**, the terminal task must be initialized by the use of the word **CONSTRUCT**, and then be made to run a program by the use of **ACTIVATE**. **CONSTRUCT** is similar to **BUILD** (for background tasks).

Terminal task initialization consists of linking the task into the round robin and setting the task's user variables. The only remaining initialization required before a task is actually running is to set the values of **I** and **R** for the address interpreter, which will be performed by **ACTIVATE**. An example of defining and initializing a terminal task is:

```
OCTAL  175610 310  DECIMAL
   4096 TERMINAL CHUCK
   CHUCK CONSTRUCT
```

The word **CONSTRUCT** performs four separate functions:

1. It links the new terminal task into the round robin (in the same way as **BUILD**).

2. It copies non-device-dependent initialization from **OPERATOR**'s user variables to the new task's user variables.

3. It copies device-dependent data from the task definition table compiled by **TERMINAL** into the user variable area of the new task. Typical device-dependent data includes **DEVICE** and the addresses of the terminal-specific routines for **TYPE**, **EXPECT**, **CR**, **PAGE**, **TAB**, **MARK**, and **CLEAN**.

4. It sets the task's private dictionary pointers **H 2+** (**H 4+** on 32-bit processors) and **H**.

When terminal definition table resides in read-only memory, **CONSTRUCT** must be executed after power-up in the target system.

REFERENCES

Controlling a Terminal Task, Section 4.10
Defining a Terminal Task, Section 4.8
I and **R**, Section 1.2.5

4.10 CONTROLLING A TERMINAL TASK

After a terminal task has been defined and has its user variables initialized, the new terminal task must be made to run a program. This is done by **ACTIVATE**, which controls **TERMINAL** tasks in the same way as **BACKGROUND** tasks.

The usual cautions about using **ACTIVATE** apply:

1. **ACTIVATE** must be in a : definition.

2. **ACTIVATE** uses a task address from the stack, and starts the task addressed to executing the words following **ACTIVATE**. The task must never reach the ; of the definition containing ACTIVATE.

3. The task executing **ACTIVATE** exits from the definition containing **ACTIVATE**, without executing any of the words following **ACTIVATE**. See the reference on **ACTIVATE**, Section 4.5 for more information.

Since most terminal tasks exhibit the characteristic behavior of awaiting input from the keyboard and executing commands thus received, these special words are available which specify this behavior:

Word Stack Function

QUIT Endlessly performs the behavior of **EXPECT**ing keyboard input and **INTERPRET**ing it. This is the program being run by most Forth terminals.

PROMPT (a) Starts a terminal task whose address is on the stack performing its characteristic behavior (**QUIT**) having first initialized its dictionary links to **GOLDEN** by using **EMPTY** and displaying some sort of "prompting" information such as the system "help screen." Terminal tasks that are PROMPTed end up running **QUIT**.

SEND text (a) Used by one terminal task to forward a command string from its input message buffer to another terminal task (whose address is on the stack). The other task will interpret the commands. **SEND** uses **ACTIVATE** to control the specified task, and leaves that task performing **QUIT**.

A simple example of a word using **ACTIVATE** to stop execution of a terminal task is:

```
: HALT ( a)   ACTIVATE QUIT ;
```

HALT uses a task address and forces the task addressed to run **QUIT**. The task that runs HALT leaves the definition of HALT at ACTIVATE, and does not execute **QUIT** or the **EXIT** compiled by **;**.

An example of the use of **PROMPT** is:

```
CHUCK PROMPT
```

where **CHUCK** is the name of a terminal task. **PROMPT**ing should be done after **GOLDEN** is set, at the completion of loading of the electives block. **PROMPT**ing should be done after **GOLDEN** is set because **PROMPT** performs an **EMPTY** and **EMPTY** uses **GOLDEN**. An example of the use of **SEND** is:

```
CHUCK  SEND 7 EMIT
```

which will force the task named **CHUCK** to interpret the phrase **7 EMIT** which will ring the bell on **CHUCK**'s terminal. **SEND** copies the input message buffer and a subset of the user variables of the **SEND**ing task to the slave task, and then uses **ACTIVATE** to force the slave task to interpret its new input message buffer. The dictionary heads are copied to the slave task so that the slave task may use any words that are available to the **SEND**ing task. Note that this convenience implies that you may not **FORGET** a definition that the slave is executing.

REFERENCES

ACTIVATE, Section 4.5
EMIT, Section 3.7.2
EMPTY, Section 3.3.4.1
HALT for **BACKGROUND** Tasks, Section 4.5
PAUSE, **STOP**, and **WAIT**, Section 4.2
The **GOLDEN** Array, Section 3.4.4

4.11 PRINTER TASKS

It is convenient to dedicate a task to the time consuming purpose of printing reports and making listings. Such a task is a terminal task with a hard copy peripheral attached. In many instances, the peripheral has no input facilities at all, being some sort of printer. If the peripheral interfaces differently than a standard terminal (using a parallel port, for example), the printer task will have a private definition of **TYPE**.

Printer tasks rarely require more than 300 bytes of memory, unless a printer task is performing a target compilation to produce a printed log.

Printer tasks are controlled by a word that uses **SEND**. If the printer task were named **TYPIST**, then the word would be defined:

```
: PRINT   PRINTER GET  TYPIST SEND ;
```

and used (for example):

```
PRINT  0 240 INDEX  9 240 SHOW  OK
```

The **OK** at the end form-feeds a page and **RELEASE**s the facility variable **PRINTER**. A facility variable (see references below) is necessary to prevent the printer task from being accidentally re-commanded in the middle of a listing.

The definition of **PRINT** shown above will wait until the printer executes an **OK**. If the printer task does not execute an **OK**, the above definition will wait forever. Another definition which does not wait is:

```
: PRINT   PRINTER DUP RELEASE
   @ ABORT" Not available"
   PRINTER GRAB  TYPIST SEND ;
```

REFERENCES

Device Dependent User Variables, Sections 4.8, 3.1
Facility Variables, **GET** and **RELEASE**, Section 4.7
SEND, Section 4.10
Terminal Drivers, Sections 3.7.3, 3.7.4

5.0 UTILITY FUNCTIONS

A complete programming environment must provide support for all aspects of the software development process. In polyFORTH, many of these functions are integral to the system: multiprogrammed operating system, compiler, interpreters, general purpose command set, and debugging aids such as **DUMP** and **.S**. These have all been discussed in previous sections. Forth's assembler is documented in the next section, as well as in the *CPU Supplement* for your processor. The purpose of this section is to document a number of separate utilities provided with the system. An additional **DOCUMENTOR** utility is provided with the Data Base Support option, described in Section 8.0.

5.1 EDITING CAPABILITIES

This section documents the commands that are used to edit a program block. Before any of the commands described in this section may be issued, however, it is necessary to select a program block for editing. Then, to obtain access to the **EDITOR** vocabulary, type **EDITOR**. Note that for convenience, **T** and **L** are in the vocabulary **FORTH**, and use of either of these will automatically select **EDITOR** for you.

REFERENCES

Disk and Block Layout, Section 5.2.5
Selecting a Program Block, Section 5.1.1

5.1.1 Block Display

To display a block and at the same time select it for future editing, type:

```
n LIST
```

where *n* is the logical block number of the desired block.

Once selected, the current program block may be (re)displayed (and the **EDITOR** selected) by the following command:

```
L
```

The number of the block will be displayed on the first line; the block will be displayed as sixteen lines of text.

Each line of the block will be numbered on the left-hand side, as 0-15 or 0-F depending on whether the user is currently in **DECIMAL** or **HEX**. These line numbers do not actually appear in the stored text but rather are provided by the program for your easy reference.

The characters "ok" will appear at the end of the final line of the block, indicating that the display is complete and that Forth is ready for another command. Regardless of the position in which they appear in the display, these characters do not appear in the actual text of the block.

The current block number is kept in the user variable **SCR**. **SCR** is set by **LIST** or **LOCATE**; all editing commands operate on the block that is specified by **SCR**.

Before a program block has been used, it contains data of an undefined nature. The command **WIPE** will fill the block with ASCII spaces. The block is considered "unused" whenever the entire first line (first 64 characters) of the block are all ASCII spaces.

For convenience, three additional block display commands exist: **N**, **B**, and **Q**. **N** (Next) adds 1 to **SCR**, and then displays the next block. **B** (Back) subtracts 1 from **SCR**, to go back and display the previous block. **Q** adds or subtracts the shadow block offset into **SCR**, so that typing **Q** alternates the block display between a source block and its shadow block. The words **N** and **B** are added to the **FORTH** vocabulary when the **EDITOR** is loaded. The word **Q** is added when the "programmer aids" option block is loaded.

REFERENCES

Shadow Blocks, Section 1.4
Shadow Documentation, Section 5.2.3

5.1.2 String Buffer Management

The **EDITOR** contains two string buffers that are used by most of the editing commands. These are called the "find" buffer and the "insert" buffer. The find buffer is used to save the string that was searched for most recently by one of the three character editing commands **F**, **D**, or **TILL**. It is at least sixty-four characters in length. The insert buffer is also at least sixty-four characters long. It contains the string that was most recently inserted into a line by the character editing commands **I** or **R**, or the line most recently inserted or deleted by the line editing commands **X**, **U**, or **P**. The command **K** interchanges the contents of the find and insert buffers, without affecting the text in the block.

The existence of these buffers allows multiple commands to work with the same string, understanding which commands use which buffers will enable you to use the **EDITOR** more economically. The convention is this: Commands which may accept a string as input will expect to be followed immediately either by a space (the delimiter following the command) and one or more additional characters followed by a carriage return, or by a carriage return only.

In the former case, the string will be used and will also be placed in the string buffer that belongs to the command (find or insert). In the latter case (carriage return only), the string that is currently in the appropriate buffer will be used (and will remain unchanged).

For example, the character editing command:

```
F WORDS TO FIND
```

will place **WORDS TO FIND** in the find buffer and will find the next occurrence of the string **WORDS TO FIND**. Subsequent use of the command F immediately followed by a carriage return will find the next occurrence of **WORDS TO FIND**. The following table summarizes buffer usage:

Buffer:	**Find**	**Insert**
Commands:	**F**	**I**
	S	**R**
	D	**X**
	TILL	**U**
	K	**P**
		K

5.1.3 Line Display

Any single line of the current block (whose number is in **SCR**) may be selected by using the following command:

```
n T
```

where *n* (which must be in the range 0-15) is the line number to be selected.

The **T** (Type) command sets the user variable **CHR** to the character position of the beginning of the line. This value may later be used to identify the line to be changed, using the commands defined in the following section. Since **CHR** is used to store the cursor position for the character editing commands, using **T** (*i.e.*, initializing **CHR**) specifies that any search will start at the beginning of that line. The new cursor position is marked in some convenient way.

The contents of the string buffers and of the block are unchanged by the use of **T**.

5.1.4 Line Replacement

The command **P** ("Place") will replace an entire line with the text that follows it, leaving that text in the insert buffer, or with the current content of the insert buffer (if **P** is followed by a carriage return).

The line number used by the **P** command is computed from the value that is in **CHR**. **P** is normally used after the **T** command, as illustrated by the following example:

```
4 T                              (command)
^THIS IS THE OLD LINE 4          (response)
P THIS IS THE NEW LINE 4         (command + text)
```

A **P** followed by two or more spaces and a carriage return will fill the line with spaces. This is useful for blanking single lines.

P immediately followed by a carriage return will replace the line by the current contents of the insert buffer. Thus, a line may be placed in several locations in a block by the use of:

1. **P** followed by text (the first time).

2. Alternate use of **T** (to select the line and confirm that this is the line to be replaced) and **P** followed by a carriage return.

5.1.5 Line Insertion or Move

The command **U** (Under) is used to insert either the text that follows or the current contents of the insert buffer into the current program block under the line in which the current value of **CHR** falls. Normally **U** is used immediately after the **T** command, where the line number specifies the line under which a new line is to be inserted.

The handling of text and the insert buffer is the same for **U** as for the command **P**.

The word **M** in the editor brings lines in to the block you're currently displaying. If you wish to move a series of lines from one block to another, first list the source block. Note the block number and the first line number. Next, list the destination block and select the line just above where you want the line you're going to bring in to be inserted. Now enter **blk# line# M**. The source line will be inserted below the current line. It won't be removed from the source block. The current line will be one below where it was before. Additionally the source

block number and line number will still be on the stack but the line number will be incremented. This sets you up to do another **M** without entering any additional arguments. The word **M** checks the stack to be certain it contains only two arguments. It **ABORT**s if the depth isn't two. This saves you from accidentally knocking the last line off your block by inadvertently entering an **M**.

5.1.6 Line Deletion

You may use the command **X** to delete the current line (*i.e.*, the line in which the current value of **CHR** falls). You will normally use **X** immediately after a **T** command that specifies the line to be deleted.

When a line is deleted, all higher-numbered lines are "rolled up" by one line and Line 15 is cleared to spaces. In addition, the contents of the deleted line are placed in the insert buffer, where they may be used by a later command. Thus **X** may be combined with **T** followed by **P** or **U** to allow movement of one line within a block. The following sequence would move Line 9 to Line 4, changing only the ordering of Lines 4 through 9.

```
9 T  X  3 T  U
```

Note that if a line is being moved to a position later in the block, the **X** operation will change the positions of the later lines. To move the current Line 4 to a position after the current Line 13, use the following command sequence:

```
4 T  X  12 T  U
```

Line 12 is specified as the insert position since the **X** operation moves the current Line 13 to the new Line 12.

5.1.7 Character Editor

The **EDITOR** vocabulary also includes commands to permit editing at the character level. Except in the case of **F** and **S**, the character editor's commands work within a specified range, controlled by the user variable **EXTENT**. **EXTENT** is normally set to 64, so that the range will be confined to the current line of the current block. The line is selected by the regular **EDITOR** command:

```
line# T
```

A cursor (indicated by a ^ or some form of highlighting) marks the position within the line at which insertions will take place and from which searches will begin. The **T** command sets this cursor to the beginning of the line.

Insertions will cause characters at the end of the line to be lost; they will not spill over onto the next line. Deletions will cause blank fill on the right end of the line.

EXTENT's value may be set to 1024 in some applications (such as word processing) when it's desirable to allow edits to propagate through the entire block.

In the list of commands below, the word "text" indicates a string of text. If the text is omitted, the current contents of the find buffer will be used (for the commands **F**, **S**, **D**, and **TILL**) or the current contents of the insert buffer will be used (for **I**). If text is present, it will be left in the appropriate buffer.

The maximum length of a string is determined by the length of the two string buffers being used, at least 64 characters. In all cases the string is terminated by a carriage return or a caret. If a string that is too long is typed, the string will be truncated to the buffer's size.

The following commands are available:

Command	Function
F text	Finds a match on text anywhere after the current cursor position in the current block and leaves the cursor positioned at the end of the matching text. The search begins at the present cursor position and continues until a match is found or the end of the block is encountered. When text is not found, issues an error message and leaves the cursor where it was before the operation.
n S text	Searches for text anywhere after the current cursor position in the current block, and continues the search through all subsequent blocks up to but not including block number *n*. If a match is found, the block containing the matching text is displayed with the cursor positioned at the end of the matching text (as in **F**). If the search succeeds, the ending block number is left on the stack, so that when you are ready to continue the search you may simply type **S**.
D text	Deletes the matching text or the contents of the find buffer and leaves the cursor where the deletion occurs. When text is not found, issues an error message and leaves the cursor where it was before the operation.
TILL text	Deletes all text from the current cursor position up to and including the text or contents of the find buffer (which must be within the boundary set by **EXTENT**, normally on the current line) When text is not found, issues an error message and leaves the cursor where it was before the operation. If text is found beyond **EXTENT**, the current line will be blanked to the right of the cursor location to the limit of **EXTENT**.
I text	Inserts the text or the contents of the insert buffer at the current cursor position. Any characters that spill off the end of the line when **EXTENT** is 64 will be lost. Any characters that spill off the end of the block when **EXTENT** is 1024 will be lost. The cursor is left positioned at the end of the inserted text.
R text	After text has been found (by the **F** or **S** commands, for example), replaces the found text with the specified text or the contents of the insert buffer, if no text is specified.

Command	Function
K	Exchanges the insert buffer and the find buffer. Useful for correction of bad deletions.

5.1.8 Block COPY Command

The **COPY** command allows a block to be copied, in its entirety, into a different block. It has the following format:

```
s d  COPY
```

where *s* is the source block number and *d* is the destination block number.

COPY moves the entire block; it does not change the contents of the old block. Note that **COPY** uses the word **IDENTIFY** to change the block number to which a block buffer will be written.

5.2 PROGRAM LISTING UTILITY

The **PRINTING** utility is used to list program source blocks on disk. The output is designed for 8-1/2" x 11" paper suitable for keeping in a conventional 3-ring binder. The utility is loaded with the following command:

```
PRINTING LOAD
```

5.2.1 Index Listings

INDEX will produce an index listing that shows the first line of each block in a given range of blocks. It is therefore very helpful to place a parenthesized comment in Line 0 of each block to describe the block's contents in the index.

Generation of an index is specified by the following command:

```
start end+1 INDEX
```

where *start* is the starting block number and *end+1* is the ending block number plus one. The index will be formatted 60 lines to a page. As many full pages will be generated as necessary to cover the requested block range. This command sends the index to the console screen. Preceding the same command with the word **PRINT** routes the index to the printer.

Since you can print 60 lines on a standard page, the large-scale organization of a polyFORTH system source disk tends to follow groups of 60 blocks. For example:

Block	Contents
0-59	polyFORTH Electives
60-179	polyFORTH Utilities
180-239	polyFORTH System Source
240-299	polyFORTH Target Compiler and Drivers

Because of different disk capacities, the layout of your polyFORTH system disk may be different from the example shown above.

5.2.2 Program Block Listings

To list the contents of entire blocks, use the following command:

```
start end+1 SHOW
```

SHOW lists three blocks per page, with the top block number evenly divisible by three. Only entire pages will be printed, in sufficient quantity to cover the requested block range. If either *start* or *end+1* is a number not evenly divisible by three, the printout will contain more blocks than requested, to fill out the pages. On a partially used page, unused blocks will not be listed although space will be left for them. Pages with no used blocks will be skipped entirely. An unused block is defined as one with only ASCII spaces in the first line (first 64 characters), as for example a block that has been **WIPE**d.

To list a single page, use the following command:

```
blk# TRIAD
```

where *blk#* is the block number of any block on the desired page.

5.2.3 Shadow Documentation Blocks

The shadow block system is intended to be an explanatory "shadow" of an entire polyFORTH system source code. Shadow blocks are intended specifically to document words with unusual usages and difficult-to-decipher coding tricks. Shadow blocks can be easily and concurrently edited with code, because the single-letter command **Q** (for

question) flips the editing block number from a code block to its corresponding shadow block and back. The best time to write shadow blocks is at the time the code is written.

At FORTH, Inc. we find it very useful to print source code and the shadow documentation blocks which document that source code on facing pages (when inserted into a binder, for example). This requires that the shadow blocks be printed on the back of the page preceding the source blocks.

The following procedure will allow you to print this type of double-sided listing on your printer.

1. Insert the program disk in Drive 0 and the shadow disk in Drive 1. This step is unnecessary if the shadow and program blocks are on the same disk.

2. Print an index of the blocks to be listed, following the procedure of Section 5.2.1 ("Index Listings"). Do not terminate your command with the word **OK** and do not tear off the paper when the index printing has completed.

   Example command:

   ```
   PRINT 0 60 INDEX
   ```

3. Print a listing of the program blocks you want, as described in Section 5.2.2 above.

   Example command:

   ```
   PRINT 9 45 SHOW OK
   ```

   will print program Blocks 9 through 44 inclusive.

   If you prefer, you may combine Steps 2 and 3 in the same command string, like this:

   ```
   PRINT 0 60 INDEX  9 45 SHOW OK
   ```

4. After the listing has finished, space out one additional blank page by typing:

   ```
   PRINT OK
   ```

   Tear off the paper after that blank page. Remove the paper stack which has been feeding empty pages into the printer.

5. Turn over the printed listings you have just finished and feed the beginning (top) of the paper into the printer. When fed in correctly, the back of the last page of the index you printed in Step 2 above should be the next surface to be imprinted.

6. Now print the shadow blocks for the program blocks just printed. The shadow block for any particular program block should be in the block whose number is the sum of the constant **SHADOWS** plus the program block number. If you follow this convention in writing your own shadow blocks, Forth will compute the shadow block numbers for you. Type in the command to print shadow blocks using the same block numbers that you have just printed the listing for, and the correct blocks will be printed.

   Example command for a particular computer with an offset of 162 blocks between source and shadow (the offset on yours may be different):

   ```
   PRINT 9 45 SHADOW OK
   ```

will print Blocks 171-206 inclusive.

If you have positioned the paper correctly, the first shadow block should be printed on the back of the last page of the index. When the printout is finished and separated into pages, you should find that program and corresponding shadow blocks face each other as you turn the pages. Shadow block listings are useful because they document code conveniently. A useful by-product of the shadow block system is that undocumented code is conspicuous, and there is no physical space for unnecessary documentation.

Two standard styles of shadow blocks exist. The first is a glossary-style format in which the words defined in the code block are explained. The code words are on the left margin with all other text indented three spaces. The second form is standard English text and is used for other explanations.

The command **PAIRS** prints source and documentation blocks in pairs of triads. Each shadow block is printed to the right of its corresponding source block, three pairs per page. **PAIRS**, given the starting and ending block number +1, prints the entire range of blocks and shadows. For example:

```
PRINT 0 300 PAIRS
```

prints source and shadow blocks from Block 0 to Block 299.

REFERENCES

Q, Section 5.1.1
Shadow Blocks, Section 1.4

5.2.4 Double-Sided Listings

Sometimes, for very large applications, it is useful to print source blocks on the front and back of every page. For this purpose the words **FRONT** and **BACK** are defined in the printing utility. To obtain a listing with program blocks on both sides of the paper, follow this procedure:

1. Print an index of the program blocks using the procedure of Steps (1) and (2) in Section 5.2.3 above. Make sure to leave an empty page before starting the printout. Do not end the index printing command with **OK**.

2. Start the program listing with the **FRONT** command. This word lists every other triad of blocks, skipping over the blocks which will be printed on the facing pages. The command format is:

   ```
   PRINT start end+1 FRONT OK
   ```

   where start and end are the starting and ending block numbers of the block range to be printed.

3. After printing has stopped, type:

   ```
   PRINT OK
   ```

   twice to leave two blank pages at the end, then tear off the listing, at the printer's tear-off line.

4. Remove the blank paper feed, and insert the top of your printed listing after turning it over as in Step (5) of Section 5.2.3. Position the paper so that the next page to print will appear on the back of the first triad page.

5. Print the remaining triads using a command of the format:

```
PRINT start end+1 BACK OK
```

where start and end are the same as in Step (2) above.

This will print the program triads skipped during the **FRONT** printing. Your listing, when separated into sheets and put into a binder, will be a front-and-back consecutive listing of the blocks of source code.

5.2.5 Disk and Block Layout Design

There are several conventions for block and disk design that will make your application programs more readable. While these conventions are not dictated by the nature of Forth, we recommend them as good programming practice.

Disk Design

1. Begin Line 0 of each block with a parenthesized comment that describes the contents of the block. The comment is then conveniently listed by the **PRINTING** utility's **INDEX**.

2. In a single block put only source text for words that are related to some one function or isolatable portion of a function. Do not put unrelated words in the same block.

3. Do not overpack a block. Leave several blank lines for expansion. There is seldom an advantage to conserving blocks.

4. Begin sub-applications on block boundaries that are divisible by three. The **PRINTING** utility conveniently puts blocks on a listed page, three blocks per page. This fits nicely on 8-1/2" x 11" paper. By starting sub-application block groups on these block boundaries, your printed listing is separable.

5. Begin major portions of your application on boundaries that are divisible by sixty. Thus, each page generated by **INDEX** will cover one such portion.

6. Write shadow blocks for your application as you program it. Six months later they may save weeks of work.

7. If your application is a resident utility, make the application's key load block a named block (add a constant to Block 10), and add the application to the system help screen in Block 11.

8. The first shadow block (for Block 9) should have a block number divisible by 3, in order to be printed properly (this is already done for you in Block 14).

When the application is listed the application starting blocks (modulo 60) and the sub-application starting blocks (modulo 3) will appear at the tops of the pages.

Block Design

1. Do not define more than one word on a line. An exception might be two or three related constants or variables or a couple of very brief related colon definitions.

2. Start colon definitions at the beginning of a line. A colon definition should never run over a block boundary. A typical colon definition uses three lines or less. Longer definitions can often be factored with groups of words re-defined as individual words, with improved readability and testability of the code. Although Forth allows unlimited nesting of loops, rarely should it be necessary to nest more than two levels without redefining the innermost loops as words.

3. Leave three spaces after the name that is being defined in a colon definition, to set it off from the definition.

4. Break colon definitions up into phrases, separated by double spaces, so that each phrase describes a particular operation:

```
: DOUBLE   X @  2*  X ! ;
```

5. If a definition takes more than one line, indent three or more spaces on the second and succeeding lines.

6. Separate instructions in **CODE** definitions with three spaces. For example:

```
CODE KEY ( - c)   BEGIN   F7 INP
   2 # ANA 0= NOT UNTIL  F6 INP
   A L MOV   0 H MVI HPUSH JMP
```

7. Use standard stack notation in the stack effects comment:

Word	Description
a	An address.
b	A byte (as returned by **C@**).
c	A byte containing an ASCII character (as returned by **C@**).
d	A double-precision signed integer.
ud	An unsigned double-precision integer.
n	A signed single-precision number.
t	A single-precision Boolean truth flag.
u	An unsigned single-precision number.
i	An integer used as an index.

Defining words are more readable when stack argument comments are in both the defining section and the run-time section; for example:

```
: COEFFICIENT ( n)   CREATE ,
   DOES> ( n - n)   @ * ;
```

REFERENCES

Documentation Aids, Section 1.4
INDEX, Section 5.2.1
Stack Arguments Notation, Section 2.1
TRIAD, Section 5.2.2

5.3 DISKING UTILITY

The utility named **DISKING** allows the user to copy ranges of blocks, backup an entire disk, and (on some systems) format a new disk (not all disk controllers support this capability). There is some variation in the actual commands from one system to another; consult the "Help screen" for your **DISKING** utility for specific details, or list the source blocks.

To load this utility, type:

```
DISKING LOAD
```

polyFORTH has a constant named **VOLUME** which normally gives the number of blocks on one floppy disk, cartridge, etc. On some systems with a large disk, however, **VOLUME** may be set to other sizes depending on the particular system. Consult your *CPU Supplement* for details.

5.3.1 Use of BLOCKS and +BLOCKS

The basic word for copying a range of blocks from one part of a disk to another is **BLOCKS**. **BLOCKS** takes three arguments: the source and destination starting block numbers, and the number of blocks to be copied. Thus,

```
60 560 120 BLOCKS
```

copies Blocks 60-179 to 560-679.

BLOCKS will correctly handle blocks which overlap either forward or backward without overwriting any blocks.

+BLOCKS is the same as **BLOCKS**, but copies shadow blocks as well; they are assumed to be offset from source blocks by the value of the constant **SHADOWS**.

5.3.2 Special Commands

The word **OBLITERATE** is defined to blank a region of disk. Its usage is:

```
start  end+1  OBLITERATE
```

which will fill all blocks from start to end entirely with ASCII spaces.

5.3.3 Comparing Disks

An additional facility is provided to ensure that a range of blocks is copied correctly. The word **MATCHES** compares ranges of blocks to discover if they are identical. **MATCHES** takes three arguments: The source and destination starting block numbers and the number of blocks to compare. Thus:

```
60 560 120  MATCHES
```

matches blocks 60-179 to 560-679. **MATCHES** will print the block numbers of any destination blocks which do not match their corresponding source block.

Note that **MATCHES** is not called automatically by **BLOCKS**.

REFERENCES

BACKUP, Section 5.3.2
BLOCKS, Section 5.3.1
Disk Diagnostics, Section 5.3.4

5.3.4 Disk Diagnostics

A simple read-only disk diagnostic is provided. It reads consecutive blocks and reports any read errors encountered. It may be used at any time and does not destroy any disk information. It is used thus:

```
l h+1 SWEEP
```
(Checks blocks *l* through *h* for read errors.)

If an error is detected, the block number is typed out, along with the disk status. Comparing the latter (in the appropriate number base) with the disk controller manual's identification of error bits yields information as to the actual type of error.

Error checking versions of **BLOCK** and **BUFFER** can be made available by loading your system's "Disk Error Handling Block."

REFERENCES

Disk Error Checking, Section 3.2.6

5.3.5 Disk Formatting

Systems that need a disk formatting facility usually have it. This feature is system dependent; check your *CPU Supplement* for details. If provided, it is used by invoking the single command:

```
INITIALIZE
```

This command may take parameters, but normally formats an entire disk. For systems with two removable disks, it is always Drive 1 that is formatted (to prevent inadvertently formatting the master). For systems with fixed and removable disks, the removable disk is the default, and a separate command is provided to format the fixed disk; see the *CPU Supplement* for details.

5.4 DEBUG UTILITY

There are two debugging utilities that let you decompile, breakpoint, or trace any high-level definition. Typing **DEBUG LOAD** will provide access to decompilation routines. Typing **STEPPER LOAD** will provide access to the above plus breakpoint (trap) and single-stepping routines. The utilities are divided in this way because **STEPPER** is relatively large, and should only be loaded if needed.

5.4.1 Definition Decompiling

The source code of a definition is not always available, *e.g.*, if it is in the precompiled nucleus. If you try to locate the source code for **ABORT"**, for example, the following will occur:

```
LOCATE ABORT"  ABORT" Can't
```

You can reconstruct the source code (with some limitations) by typing **DEBUG LOAD** and using the **SEE** name command, which decompiles the colon definition named name. Typing **SEE ABORT"** will produce:

```
COMPILE [4358]  34 STR___ ;  IMMEDIATE
```

Compare this to the original definition of ABORT":

```
: ABORT"  COMPILE abort"  34 STRING ;
   IMMEDIATE
```

The word **abort"** is "headerless," which means it was compiled without a link or a name field and thus only its address is available to the decompiler. Therefore the "run-time" **abort"** has been replaced with its *pfa* (the number you see may be different from 4358). Likewise, **STRING** has been replaced with **STR___** because it was compiled with a **WIDTH** of 3.

SEE uses the command a **PROBE**, which begins decompilation at a specific address *a*. Decompilation is terminated either when the end of the definition is found, or at 128 bytes (64 cell addresses) beyond the destination of the most forward reference (such as an **IF**) found in the definition. The end of a definition is signaled by a semicolon (if it exists—*e.g.*, **RECOVER** has not been used); or by an unconditional exit (such as **ABORT** or **QUIT**) or an unconditional backward branch (such as **AGAIN**) that lies beyond the farthest forward reference previously encountered.

You may often want to execute **PROBE** directly. For example, you could use **PROBE** to examine the "headerless" **abort"** at the given (parameter field) address. Typing: **4358 PROBE** will produce:

```
?R@ HER_ 2+ COU__ TYP_ SPA__ COU__
TYP_ CR BLK 2@ DUP {if 4} SCR 2! ABORT
```

The expression **{if 4}** is produced by an **IF ... THEN** clause and means "branch forward 4 bytes conditionally." Compare this code to the original definition of **abort"**:

```
: abort"  ( t)  ?R@ HERE 2+ COUNT TYPE
   SPACE COUNT TYPE CR  BLK 2@ DUP IF
   SCR 2!  THEN  ABORT ;  RECOVER
```

Another usage of **PROBE** is to determine the action of an execution vector. For example, typing **'IDLE @ PROBE** results in:

```
SUP_____ 0 STA__ ! QUE__ INT_______ ." ok"
   CR {else -14}
```

In this case, **{else -14}** is produced by **AGAIN** and means "branch backward 14 bytes unconditionally." The original code is:

```
: QUIT  SUPPLANT  0 STATE ! BEGIN  QUERY
   INTERPRET  ." ok"  AGAIN ;  RECOVER
```

Most flow-of-control structures ultimately compile the run-time primitives if or else. For example, the original source code for **INTERPRET** is:

```
: INTERPRET  STATE @EXECUTE
   BEGIN  -' IF  NUMBER
   ELSE  DROP EXECUTE
   DEPTH 0< ABORT" Stack empty"
   THEN  AGAIN ; RECOVER
```

The decompiled version, resulting from **SEE INTERPRET**, is:

```
STA__ @EXECUTE
-' {if 5} NUM___ {else 22} DRO_ EXE___
DEP___ 0< ABORT" Stack empty"
{else -35}
```

Of course, not all definitions can be decompiled; **SEE DUP** produces the message:

```
Not a colon definition
```

PROBE however will attempt to decompile any definition; typing **' TYPE @ PROBE** produces:

```
[950] [39914] [11234] ...
```

Here **PROBE** has misinterpreted machine code as a sequence of headerless definitions. As stated above, in this case decompilation stops after 64 headerless terms.

5.4.2 Breakpoint Setting

Breakpoints are useful for finding "intermittent" bugs, that is, bugs which only appear under (un)certain operating conditions. To access the breakpoint or trapping routines, type: **STEPPER LOAD**. The basic command is **TRAP** name, which sets a breakpoint to occur on the next execution of name. Subsidiary commands allow moving and removing the breakpoint, and single-stepping through the definition of name.

Care must be taken when using breakpoints and single-stepping in a multitasking environment.

For example, suppose you wish to investigate the word **CAST** as it is called by the word **Q**. If you type **SEE Q** to decompile it, you will get:

```
SCR @ SHA____ CAS_ LIS_ ;
```

To stop ("trap") execution at **CAST** you "set a breakpoint" with the command **TRAP CAST**, which will decompile **CAST**:

```
SWA_ OVE_ /MO_ 1 XOR ROT * + ;
```

and set the breakpoint to execute ("fire") on the next execution of **CAST**. Having done this, to examine **CAST** you might type:

```
9 LIST
```
(sets **SCR** to 9)

followed by:

```
Q
```

Q will call **CAST**, and the trap will fire:

```
9 321 <-Top nxt-> SWA_
```

When the trap fires, the stack contents are displayed to the left of the brackets, and the next instruction to be executed is displayed to the right. In this case, the *9* is from **SCR @** and the *321* is the value of **SHADOWS**. The

next instruction to be executed is **SWAP**, which is the first instruction in **CAST**. To review the remaining instructions in **CAST**, you may type **SEEN**, which repeats the most recent decompilation:

```
SWA_ OVE_ /MO_ 1 XOR ROT * + ;
   TrOk
```

Notice the change in the Forth prompt from **ok** to **TrOk**. This is a reminder that **TRAP** uses a special text interpreter. From here, you can change the arguments to **CAST**, **DUMP** memory, or take any other (reasonable) action. You can even execute **QUIT** to return to the normal text interpreter. Or you can use **CONT** (continue) which will continue execution of **CAST** (and the remainder of the original command, **Q** in this case).

TRAP works by installing the word **tcode** directly in the location to be trapped (after first saving its contents). When **tcode** runs, it restores these contents and returns to the special text interpreter. **TRAP** displays the decompilation before installing **tcode**, but subsequent uses of **SEEN** will show the new version. Thus **TRAP CAST** shows:

```
SWA_ OVE_ /MO_ 1 XOR ROT * + ;
```

and **SEEN** will display:

```
tcode OVE_ /MO_ 1 XOR ROT * + ;
```

The command **RESTOR** is used to remove the current breakpoint or trap. Thus, in this example, typing **RESTOR SEEN** will display:

```
SWA_ OVE_ /MO_ 1 XOR ROT * + ;
```

If you have trapped a word but have not yet identified a bug, you may wish to move the trap and continue execution. The command **NEST** name will move the trap to the word name and continue execution without leaving the special text interpreter. For example, suppose you have trapped **Q** but then decide that the bug you are looking for must be in **CAST**. Typing **TRAP Q** shows the decompilation:

```
SCR @ SHA____ CAS_ LIS_
```

Then executing **Q** displays:

```
<-Top nxt-> SCR
```

You can then type **NEST CAST** to move the trap to **CAST** and continue execution without leaving the special text interpreter:

```
SWA_ OVE_ /MO_ 1 XOR ROT * + ;
9 321 <-Top nxt-> SWA_
```

NEST combines **TRAP** and **CONT** and leaves you exactly where **TRAP CAST** would have left you—ready to continue or **QUIT**. Don't try to use **NEST** without first **TRAP**ping a word.

5.4.3 Single-Stepping Through a Definition

To access the single-stepping utility, type **STEPPER LOAD** (if not already loaded). This utility provides facilities to step through a definition, with or without skipping over intermediate words, and to cycle through **DO** loops.

For example, suppose you have determined that there is a bug in a definition (such as **CAST**) but you don't know exactly where. You have trapped the definition and would like to execute it one word at a time, examining the stack after each step. As shown in the previous section, you would type **TRAP CAST** followed by **9 LIST** and **Q**, at which point the screen would show:

```
9 321 <-Top nxt-> SWA_
```

Now typing the command **SS** ("single-step") will execute the next word (**SWAP**) and display the following:

```
321 9 <-Top nxt-> OVE_
```

The top two numbers are swapped and the next instruction to be executed is **OVER**. Subsequent uses of **SS** will display:

```
321 9 321 <-Top nxt-> /MOD
```

then

```
321 9 0 <- Top nxt-> 1
```

then

```
321 9 0 1 <-Top nxt-> XOR
```

Now suppose you want to skip ahead in the definition. There are two commands used for this: **GOTO name** and **GO**. The command **GOTO name** will prepare for execution of all steps up to but not including name; the command **GO** will then perform the execution. The skip operation is divided into these two steps in case you need to skip over one or more occurrences of *name* to get to the one you want. In this example, suppose you want to execute all words up to but not including the **+** instruction. You would type:

```
GOTO +
```

Then typing **GO** will execute up to the + instruction, and display:

```
9 321 <-Top nxt-> +
```

Continue single-stepping with **SS** (on the IBM-PC you can use the <alt-F10> key):

```
9 321 <-Top nxt-> +              ( press the <alt-F10> key)
330 <-Top nxt-> ;                ( press the <alt-F10> key)
Trapping done.
```

Single-stepping through a semicolon takes you directly to the normal text interpreter. You may use **CONT** instead to return to the (pending) **Q** command.

Single-stepping through a loop is made easier with the **CYCLE** command. Use of **CYCLE** at some point inside a **DO** loop will cause execution of all remaining instructions in the loop, return to the beginning with index incremented, and execution of all instructions up to the starting point, *i.e.*, one complete cycle. For example, here is a simple definition containing a **DO ... LOOP**:

```
: DOUBLE  1  10 0 DO  DUP . 2*   LOOP  DROP ;
```

Typing **TRAP DOUBLE** will display:

```
1 10 0 2>R DUP . 2* {loop -9} DRO_ ;
```

Then executing **DOUBLE** will display:

```
<-Top nxt-> 1
```

Type **SS**: `1 <-Top nxt-> 10`
Then **SS** again: `1 10 <-Top nxt-> 0`
Then **GOTO 2*** followed by **GO**: `1 <-Top nxt-> 2*`

If you type **SS** at this point, you will single-step through the **2*** instruction to **LOOP**. If instead you type **CYCLE**, you will "cycle" all the way through the loop, that is, you will execute the **2***, follow the loop back to the **DUP**, and execute all instructions in the loop, stopping again just before the **2*** instruction. You may **CYCLE** through the loop as many times as you wish; typing **CYCLE** four times in this case would produce:

```
2 <-Top nxt-> 2*
4 <-Top nxt-> 2*
8 <-Top nxt-> 2*
16 <-Top nxt-> 2*
```

To exit from the loop, **GOTO** the first word past the **LOOP** instruction:

```
GOTO DROP
```

Followed by: **GO**. The display will show:

```
32 64 128 256 512 1024 <-Top nxt-> DRO_
```

From here, you can **CONT** or **QUIT** or continue single-stepping.

Below are summarized all of the commands in the **DEBUG** and **STEPPER** utilities:

Word	Description
SEE name	Decompiles and displays the definition of *name* as far as possible if it is a colon definition, otherwise issues an error message. Name fields are reconstructed to their compiled **WIDTH**; headerless words are replaced by their pfa's.
SEEN	Repeats the decompilation display of the most recent word decompiled by **SEE**.
a PROBE	Begins decompilation at address *a*. Decompilation is terminated when the end of the definition is found, or at 128 bytes (64 cell addresses on 16-bit machines) beyond the destination of the most forward reference found, whichever occurs first. The end of a definition is signaled by a semicolon, or by an unconditional exit or an unconditional backward branch that lies beyond the destination of the farthest forward reference previously encountered.

Word	Description
TRAP name	Decompiles and displays the definition of name, and sets a breakpoint to occur on the next execution of name. When the breakpoint or trap occurs, the current stack contents and the next instruction to be executed are displayed. Execution in **TRAP** mode is controlled by a special text interpreter, as signified by an **TrOk** response rather than **ok**.
CONT	Continues execution of all remaining instructions in a definition that has been **TRAP**ped.
RESTOR	Removes the current breakpoint or trap, restoring the current definition.

NEST name	Moves a trap from its current position to the word name, displaying the decompilation of name. The new name may be at the same level in a definition or lower. At least one execution of TRAP must precede usage of **NEST**.
SS	Single-steps through the currently **TRAP**ped definition. Each use of **SS** displays the current stack contents and the next instruction to be executed.
GOTO name	Prepares for execution of all steps from the current one up to but not including name. **GOTO** name may be used several times in succession to skip over multiple occurrences of name.
GO	Executes all steps from the current one up to the step pointed to by **GOTO**.
CYCLE	When in single-step mode inside a **DO** loop, **CYCLE** causes execution of all remaining instructions in the loop, return to the beginning with the index incremented, and execution of all instructions up to the initial starting point (*i.e.*, one complete cycle). Each use of **CYCLE** displays the current stack contents and the next instruction to be executed. To exit the loop, **GOTO** the first word past the **LOOP** instruction.

5.5 AUDIT UTILITY

The **AUDIT** utility (accessed by **AUDIT LOAD**) compares two blocks or ranges of blocks and can highlight the differences between corresponding blocks. The block ranges may be on the same or different parts of disk, and may have the same or different relative block numbers. The utility is designed as an extension to the standard polyFORTH editor and essentially lets you edit two blocks at the same time. **AUDIT** is often used to compare a source part against a backup part to identify changes. It is also used to coordinate the separate efforts when more than one programmer is working on an application.

As an example, suppose you have the following on disk in Block 100, Part 2:

```
 0 ( Miscellaneous)
 1 : STRING ( n)   -2 ALLOT  WORD C@  2+ 1+ ALLOT ;
 2
 3 : ABORT"   COMPILE abort"  34 STRING ;   IMMEDIATE
 4
 5 : ."   COMPILE dot"  34 STRING ;   IMMEDIATE
 6
 7
 8
 9
10
11
12
13
14
15
```

Now suppose you have an “improved” version in Block 100, Part 4:

```
 0 ( Miscellaneous)
 1 : STRING ( n)   -2 ALLOT  WORD C@  2+ 1+ ALLOT ;
 2
 3 : ABORT"   COMPILE abort"  [ASCII] " STRING ;
 4    IMMEDIATE
 5 : ."   COMPILE dot"  [ASCII] " STRING ;   IMMEDIATE
 6
 7
 8
 9
10
11
12
13
14
15
```

Here’s how you would set up to compare Part 2 against Part 4:

```
2 PART  100 LIST  4 TARGET
```

After **LIST**ing a block to make it current, the phrase **n TARGET** highlights all the character positions where the current block differs from the correspondingly numbered block in Part n. In this case, **4 TARGET** highlights all the differences to Block 100 on Part 4:

```
 0 ( Miscellaneous)
 1 : STRING ( n)   -2 ALLOT  WORD C@  2+ 1+ ALLOT ;
 2
 3 : ABORT"   COMPILE abort"  34 STRING ;   IMMEDIATE
 4
 5 : ."   COMPILE dot"  34 STRING ;   IMMEDIATE
 6
 7
 8
 9
10
11
12
13
14
15
```

The highlighted text shows that there are some differences between the “current” block (the one you can see) and the “other” block (the one you can’t). To list the other block with its differences compared to the first, type **W**:

```
 0 ( Miscellaneous)
 1 : STRING ( n)   -2 ALLOT  WORD C@  2+ 1+ ALLOT ;
 2
 3 : ABORT  COMPILE abort" [ASCII] " STRING ;
 4    IMMEDIATE
 5 : ."   COMPILE dot"  [ASCII] " STRING ;
 6    IMMEDIATE
 7
 8
 9
10
11
12
13
14
15
```

W (like **Q**) toggles between the current block and the other one. Whichever block you are looking at is the current one. **W** always highlights the differences between the two blocks. The command **O** will toggle between blocks without highlighting. If you have edited or re-listed the current block and lost the highlighted differences, use **V** to see the current block with highlights. To summarize:

	Differences are not highlighted	Differences are highlighted
Current block:	**L**	**V**
Other block:	**O**	**W**

IMPORTANT:

The highlighting method uses space at **PAD** to hold one of the blocks and therefore interferes with the editor's Find and Insert buffers. Thus if you wish, for example, to delete a line in the current block (with **X**) and insert it into the other block, you must use the **O** (non-highlighted) command rather than the **W** (highlighted) command to toggle to the alternate block.

Once block differences have been examined, if you decide that you prefer the *current* block to the other block, you can **KEEP** the current block. This copies the current block to the other block and destroys the previous other block:

KEEP (The two blocks are now identical)

If you prefer the *other* block (the one you can't see), you can **TOSS** the current block. This copies the other block to the current block and destroys the previous current block:

TOSS (The two blocks are now identical)

Once editing of this pair of blocks is complete, type **G** to continue the "audit." The **G** command will compare corresponding blocks, stopping and displaying the *next* difference, that is, the first subsequent block which differs between the two parts.

Auditing normally stops at the end of the part. You can limit the last block to be audited with the **n TO** command, which halts auditing at the current block *n*:

```
320 TO
```
(Auditing stops at Block 320.)

You can also audit within a single part, or in different block numbers across parts. The command **n m MATCHING** performs an audit comparing blocks starting at n to blocks starting at m, both in the current part. For example, if you want to compare blocks starting at 100 to blocks starting at 150 within Part 2, you would type:

```
2 PART  100 150 MATCHING
```

The most general command is **n m p AGAINST**, which compares blocks starting at *n* in the current part to blocks starting at *m* in Part *p*. For example, to compare blocks starting at 100 on Part 2 to blocks starting at 200 on Part 4, you would type:

```
2 PART  100 200 4 AGAINST
```

5.6 PROMS UTILITY

The **PROMS** utility is designed to write ("burn") most popular PROM devices on a GTEK PROM programmer, models 7128 or 7228. The **PROMS** utility is called GTEK on some systems. Before using PROMS you should have created a PROM image on a contiguous range of disk blocks. This is usually done with the target compiler (documented separately). You must also have created the serial terminal task **REMOTE**, typically defined (on the IBM-PC) as follows:

```
COM1  4096 CHANNEL REMOTE  1200 BAUD
REMOTE CONSTRUCT
```

The model 7228 runs also at 2400 baud.

The **PROMS** utility, like other utilities, is compiled and accessed by typing:

```
PROMS LOAD
```

There will be a brief pause while the utility initializes the GTEK programmer, you will see the message:

```
SETTING BAUD RATE
```

You may also see the message:

```
Prom HELP  Timeout
```

This means that either the GTEK is not correctly connected or that the **REMOTE** task is not correctly initialized. Otherwise, you will see the message:

```
Prom 2716  Start 0  At 0  Long 2048
   Bus 1 Image 300
```

This means the GTEK is programmed with defaults to:

1. Burn a type *2716 PROM*,
2. *starting* at 0 bytes offset in the disk image;
3. placing the data *at* 0 bytes offset in the PROM;

4. burning a data array 2048 bytes *long* (default is the PROM size);

5. using a *bus* width of 1 byte (8 bits);

6. and getting the data from the disk *image* starting at block 300.

The default PROM and bus width are on Line 15 of the load block. This line, as shipped, reads:

```
CR 1 >2716   HELP
```

The 1 selects the "bus width" where 1 means that the PROM has an 8-bit (1-byte) wide data bus, and 2 means that two PROMs will be used in parallel for an effective 16-bit (2-byte) data bus (see **EVEN** and **ODD** below). Most applications will use bus width 1.

5.6.1 Burning a New PROM

The **1 >2716** default command selected a 2716 PROM. To change PROM types or to see a list of available PROMs, type the command **PROMS**, which will display a list similar to the following:

```
( PROM types:

prom    type   menu
size    name   char )

 2048 PROM >2716  B      2048 PROM >27C16 L
 4096 PROM >2732  C      4096 PROM >27C32 M
 4096 PROM >2732A D
 8192 PROM >2764  E      8192 PROM >27C64 O
16384 PROM >27128 F     32768 PROM >27256 Z
```

Your list may differ. **PROM** is a defining word, so that each line in the list creates a PROM type with a type name as shown, an associated size in bytes which is used to set **LONG**, and an associated menu character which is sent to the GTEK to initialize it properly. To select a different PROM, type: **n typename**, where *n* is the desired bus width in bytes and the *PROM typename* is one of those in your PROMS list. For example:

```
1 >2732A.
```

Every PROM selection command sets **START** and **AT** to zero and **LONG** to the number of bytes in the PROM (the argument to **PROM** in the list above). A **START** of zero means "start the burn with the first (zeroth) byte of the disk image." An **AT** of zero means "start by burning the first (zeroth) byte of the PROM." Use of non-zero values for **START** and **AT** is discussed below.

The starting block of the disk image is set with the command **n IMAGE** where *n* is the block number; for example:

```
300 IMAGE
```

The default image starting block is set to the constant **NEW** in the **PROMS** load block :

```
NEW IMAGE
```

The constant **NEW** is defined with the system electives and is normally such that the image is located near the upper end of the first system part. You may need to change **IMAGE** to suit the size of your PROM.

Assuming you want to burn from the start of the disk image to the start of the PROM, none of these defaults needs to be changed. Just insert a blank (erased) PROM in the GTEK and type **BURN**. You may have to wait several minutes for the PROM to burn, depending on the type and size. When it is done, you will hear a "beep" and will see the "ok" prompt. **BURN** stops immediately if it detects an unerased byte or a defective PROM. To summarize:

1. Select a PROM type, such as: **1 >2716**.
2. Select a disk image starting block, for example: **300 IMAGE**.
3. Insert a blank PROM into the GTEK and type **BURN**.
4. Repeat Step 3 for multiple copies.

5.6.2 Copying a PROM

To copy an existing PROM, you can first read it into the disk image and then burn the image into a new PROM (or PROMs). To read a PROM, select the PROM type, set the disk image starting block (making sure there is enough space on disk for the entire image), insert the PROM in the GTEK, and type **READ**. For example:

```
1 >2716  300 IMAGE  READ
```

This can take several minutes. **READ** reads the PROM image into the disk image blocks, destroying the previous image. **READ** also resets the variables controlled by **START** and **AT** to zero and **LONG**'s variable to the length of the PROM in bytes.

Once the image is read, insert the new (erased) PROM and type **BURN**. You can now burn as many PROMs as you like from the current disk image. If you wish to verify that the new PROM is correct, you can compare it to the disk image with the command **MATCH**. In summary, here's what you do to copy a PROM:

1. Select a PROM type, such as: **1 >2716**.
2. Select a disk image starting block, for example: **300 IMAGE**.
3. Insert the original PROM and type: **READ**.
4. Insert the new PROM and type: **BURN**.
5. Optionally, type: **MATCH** to verify the burn.
6. Repeat Steps 4 and 5 for multiple copies.

5.6.3 Burning Partial PROMs

You can burn part of a PROM provided that the part you burn is erased. Set **START** to the starting offset (in bytes) from the beginning of the disk image and **LONG** to the number of bytes to burn. For example, to burn the first half of a 2716 PROM from an image starting in Block 300, you would need to type:

```
1 >2716  300 IMAGE  1024 LONG  BURN
```

To burn the second half from the same image, you would type:

```
1024 START  BURN
```

The **START** command affects **LONG**, adjusting it to equal the PROM length minus the **START** value. **START** also sets **AT** to be equal to **START**, so that the second half of the disk image will be burned into the second half of the PROM. If you want to burn the *second* half of the disk image into the *first* half of the PROM, you need to explicitly reset **AT**:

```
1024 START  0 AT
```

The **START**, **LONG**, and **AT** commands, in combination, let you burn any part of a disk image into any part of a PROM.

5.6.4 ODD and EVEN PROMs

Some computers, such as the IBM AT, use PROMs in pairs, with the even-numbered bytes of memory in one PROM and the odd-numbered bytes in another. The data bus width is effectively 16 bits (2 bytes) and so has a bus width of 2; for example:

```
2 >2716
```

In this case you use the **EVEN** command to burn the even-numbered bytes of a disk image into one PROM and the **ODD** command to burn the odd-numbered bytes into another:

Insert the first PROM then type: **EVEN BURN**

Insert the second PROM then type: **ODD BURN**

Remember that **LONG** specifies the number of bytes in the *disk image* rather than in the PROM itself. If you need to burn partial PROMs, there are *two* bytes of disk image associated with each byte of PROM, so **LONG** must be twice the length of either PROM.

5.6.5 Images Larger Than One PROM

Often you may need to burn a large program into several smaller PROMs. The **CHIP** command specifies which smaller PROM to burn. For example, suppose you need to burn an 8192-byte program into four 2716 PROMs. You would type, in sequence:

```
0 CHIP BURN
1 CHIP BURN
2 CHIP BURN
3 CHIP BURN
```

The disk image size should be an integer multiple of the PROM size (some parts of the image may be zero). If the bus width is 2, **CHIP** can be combined with **ODD** and **EVEN**:

```
0 CHIP EVEN BURN          (first PROM)
ODD BURN                  (second PROM)
2 CHIP EVEN BURN          (third PROM)
ODD BURN                  (and so on)
```

5.6.6 Other PROM Programmers

If your PROM programmer is not one of the GTEK models described here, you may well be able to adapt the GTEK code to some other type. Most intelligent PROM programmers use the same general format and communicate with you over an RS-232 serial interface line. A typical dialog looks like this:

1. You send spaces or carriage returns to the PROM programmer, allowing it to synchronize with you.
2. It sends a sign-on message and a prompt sequence, which you can discard.
3. You send setup information, discarding all prompts.
4. You send the PROM image.

You should be prepared to respond to any error codes you receive.

5.7 NETWORK UTILITY

The network utility is used to move ranges of blocks between two polyFORTH systems; they do not need to have the same block numbers on each system. Binary data as well as source code and files may be sent. Before using this utility, you should establish that you have a working serial interface between the two computers. If you have a "break-out box" you can verify that signal lines are not crossed and are at the proper voltage levels, and you should use simple **TYPE** and **EXPECT** commands to verify that characters can be sent successfully in each direction. Each system should have the serial terminal task **REMOTE** defined and active, typically running at 9600 baud.

The NETWORK utility has only three commands: **TRANSMIT**, **RECEIVE**, and **C**. The **TRANSMIT** and **RECEIVE** commands each take a range of blocks in "**SWAP**ped" **DO ... LOOP** form; for example:

100 200 TRANSMIT means **TRANSMIT** blocks 100 to 199.

100 101 RECEIVE means **RECEIVE** block 100 only.

100 100 RECEIVE means **RECEIVE** block 100 only.

The **C** command returns the status of the transfer, and may be executed at any time on either the receiving or the transmitting computer. For example, on the transmitting computer before transfer has begun, one would see:

```
C  0 -1
```

The first number returned (0) is the number of the last block successfully transmitted (or received, if the receiving computer). In this case, *no* blocks have been transmitted yet. The second number (-1) is the **CTR** of this computer's **REMOTE** task. In this case, the transmitter's **REMOTE** is waiting to *receive* one "go-ahead" character from the receiving task.

The network protocol operates as follows: First, the transmitting computer sets up the range of blocks to be sent by executing:

```
start  end+1  TRANSMIT
```

as described above. This action sets transmitter's **REMOTE** to expect one handshake character, an ASCII 4 (EOT). Next, the receiving computer executes:

```
start  end+1  RECEIVE
```

where the length of the block range must match the transmitter's range, but the specific block numbers do not need to match. This action sends the EOT to the transmitter, which replies by sending the first block (1024 bytes) and then returning to **EXPECT** mode for another EOT (if more than one block is to be sent). Upon receipt of 1024 bytes, the receiver generates the EOT needed for the second block transmission, and so on.

The transmitter's **CTR** status (as reported by **C**), will start at -1, go to 0 when the EOT is received, then jump to 1024 and count down to 0 as the block's bytes are transmitted. The receiver's **CTR** status will start at -1024 and count up to 0 as the block's characters are received. This sequence repeats for each block that is sent.

For example, suppose you want to transmit Blocks 9-11 to another computer where they will be stored as Blocks 20-23. The transmitting computer goes first:

```
9 12 TRANSMIT          (send 3 blocks:  9 then 10 then 11)
```

The receiving computer goes second:

```
20 23 RECEIVE          (receive 3 blocks: 20 then 21 then 22)
```

The transfer begins immediately. You can use **C** to monitor the progress of the transfer on each computer. Typical values might be:

Receiving Side	**Transmitting Side**
`C  20 -300`	`C   9 300`
`C  20 -100`	`C   9 100`
`C  21 -502`	`C  10 502`

Once the transfer is complete, the receiver's **C** will show the last block number received and that no further characters are expected:

```
C  22 0
```

NETWORK has no "handshaking" logic beyond the initial request to send for each block. If characters are lost, the receiver's **C** will show that characters were expected when transmission ceased, and which block they were for:

```
C  21 -2
C  21 -2
C  21 -2
```

All previous blocks were correctly sent (at least 1024 characters per block were received), and no blocks after this one were attempted. If **C** is "stuck" this way, correct the line problem and re-transmit blocks starting with the first incorrect one.

.

6.0 THE ASSEMBLER

Forth is one of the fastest, most efficient high-level languages available and is used extensively in real-time programming and applications programming. Such programs are usually written in the extensible Forth word set. However, where particular time constraints exist, Forth can assemble machine-language definitions of Forth words. Among the many examples of words defined by machine-language instructions in the Forth nucleus are the operations:

```
+   -   SWAP   DROP   2DUP
```

The assembler for your particular CPU is detailed in the *CPU Supplement* to this manual. This section provides a general overview of the assembler on any Forth system. Note that the assembler is not used in ordinary high-level Forth programming, only in **CODE** definitions. Assembler code is, by definition, machine-dependent. However, there are many characteristics of Forth assemblers that are relatively consistent across all the processors on which Forth has been implemented. It is this set of common characteristics which this section addresses. Examples will be given using code for some of the most popular processors; unfortunately, space will not permit providing versions of each example for all processors. Hopefully, the specific examples will still be comprehensible.

6.1 CODE DEFINITIONS

The Forth defining word **CODE** creates a standard dictionary entry whose code address field contains the address of the byte that follows, which is the first byte of the parameter field where machine instructions are assembled. See Fig. 6.1 for a diagram of this dictionary entry. The form of a **CODE** definition is:

CODE NAME ... instructions ... code ending

CODE creates the definition, whose name is **NAME**. It also selects the **ASSEMBLER** vocabulary, in which the various instruction mnemonics, addressing modes, etc., are defined. These are used to build actual machine instructions, which are laid down in subsequent locations in the dictionary. The code ending is one of several macros, all of which ultimately return to Forth's address interpreter.

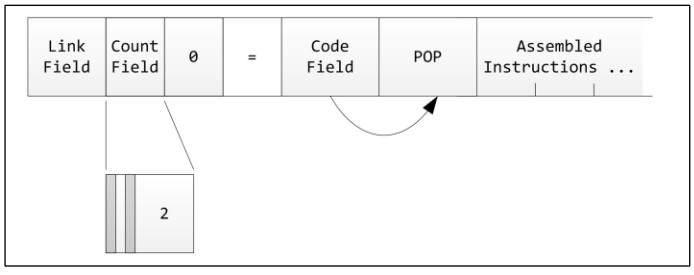


Fig. 6.1

Diagram of a dictionary entry for a **CODE** entry.

Aside from the dictionary entry header there is no high-level language overhead in either space or time within a code definition. All instructions are executed at full machine speed.

As a general rule, Forth programs are written first in high-level language. Then a time analysis is performed to locate the most frequently executed words, which are then **CODE**d. Two examples of such words might be:

1. The portion of an interrupt routine that actually moves data to and from a device.

2. The innermost loop of a routine where the computer spends a significant portion of its time (for example, the word **NEXT** in the Forth nucleus).

REFERENCES

Code Endings, Section 6.2
Macros, Section 6.7

6.2 CODE ENDINGS

Most Forth code routines end with a jump to the address interpreter, sometimes after modifying the stack. Exceptions are interrupt routines (which return to the code that was being executed before the interrupt occurred) and routines which initiate processes whose completion will be signaled by an interrupt. These end with a jump to the part of the multiprogrammer that will deactivate the current task and begin searching for the next one.

The most common code ending is **NEXT**, a macro that assembles the address interpreter return. This, plus the other most common code endings are summarized below. On some processors these endings assemble code or branches to the appropriate code, whereas on others they return addresses which may be used as arguments to a **JMP**. Refer to your *CPU Supplement* for a list of the code endings for your processor.

Word	Description
NEXT	Address interpreter return.
WAIT	Enters the multitasking loop.
' PAUSE JMP	Enters multitasking loop for one turn. The code for **PAUSE** is prefaced by instructions that decrement **I** by 2 so that the word containing **' PAUSE JMP** will be executed over and over again. This is useful for waiting on polled devices
INTERRUPT	Returns from interrupt.

Check your *CPU Supplement* for the appropriate list of code endings for your processor.

6.3 ASSEMBLER INSTRUCTIONS

To compile a colon definition, the interpreter enters a special compile mode in which the words of the input string are not executed (unless designated as **IMMEDIATE**). Instead, their addresses are placed sequentially in the dictionary. During assembly, however, the interpreter remains in execute mode. The mnemonics of the processor instructions are defined as words which, when executed, assemble the corresponding operation code at the next location in the dictionary. Operands (addresses or registers) precede instruction mnemonics in order to leave information on the stack that will be used by the mnemonic to assemble the instruction.

Depending on the processor, several kinds of instructions and addressing are possible. These are defined in the polyFORTH assembler for each processor to assemble instructions in the appropriate format, given the mnemonic operation code and the additional parameters that are necessary to describe the instruction. The instruction is assembled into the next available location in the dictionary.

For example, the Intel 8080 processor has an ALU reference instruction format for instructions that perform arithmetic computations. The Forth assembler defines the command **ALU**, which is used to define mnemonics of the ALU class, which in turn assemble ALU reference instructions. For example, the mnemonic ADD is defined on 8080 systems by:

```
80 ALU ADD
```

ADD is an operation which assembles an ALU-type instruction whose numeric code is 80_H and whose operand will be on the stack. In use,

```
L ADD
```

assembles an instruction which, when executed, will add the contents of Register L into the accumulator.

6.4 NOTATIONAL CONVENTIONS

Although each Forth assembler uses the manufacturer's mnemonics, there are some standard Forth notational conventions that are shared by all assemblers. Fundamental Forth pointers have standard names:

Name	Description
S	Address of the top of the parameter stack.
W	Address of the parameter field or code field of the current definition.
I	Interpreter pointer.
R	Address of the top of the return stack.
U	Beginning of the user area.

These are often registers, but may reside in memory in some computers. Refer to your *CPU Supplement* for a discussion of their locations on your system. Wherever these pointers reside, the standard names may be used in code to refer to them.

Registers are numbered in a way that reflects the manufacturer's usage and the actual bits used in assembled instructions. In addition, for convenience and readability, some registers are given names by using **CONSTANT**. Thus, on the DEC PDP-11:

```
5 CONSTANT S
```

enables you to refer to Register 5 by number or by the name **S** (which identifies this register as containing the stack pointer). Similarly, the Intel 8080 has named Registers H and L.

Forth code routines tend to be extremely short, averaging under a dozen instructions on 16-bit processors. Moreover, Forth assembler code is entirely structured. The conventional vertical format that includes comments on each line is not very helpful. FORTH, Inc. uses a horizontal format, with three spaces between each instruction and one space between each component of an instruction (address specifiers, the mnemonic itself, etc.). In this format, the average code definition occupies only two or three lines and is still readable. Comments that are enclosed by the command **(** and its delimiting **)** may appear anywhere:

```
( in parentheses like this)
```

Shadow blocks are also helpful.

REFERENCES

Disk and Block Layout, Section 5.2.5
Documentation Facilities, Section 1.5

6.5 USE OF THE STACK IN CODE

It is necessary when using code to distinguish how the stack is used at assembly time and at execution time. The words in a code entry are executed at assembly time to create machine instructions which are placed in the dictionary to be executed later. Thus,

```
HERE 2- TST
```

at assembly time places the current dictionary location on the stack (**HERE**) and decrements it by two. The resulting number is the parameter for **TST**, which assembles a machine instruction that is the equivalent of:

```
TST *-2
```

in conventional assembler notation. Similarly, such words as **SWAP** and **DUP** are executed at assembly time to manipulate the parameters being used by assembler words, although such stack words would be compiled into the dictionary in a : definition. For example, in the 8080:

```
0  HERE SWAP  H LXI  JMP
```

assembles an endless loop that loads zero into the accumulator. **HERE** pushes the address of the next free byte of dictionary space onto the stack. The phrase **H LXI** takes the zero from the top of the stack (at assembly time) and assembles a "load index immediate" that will load zero into the **HL** register pair. The **JMP** uses the address left on the stack to assemble a jump to the first byte of the load.

In high-level definitions the run-time use of the stack is implicit: Numbers that you type are placed there automatically, routines naturally leave their results there, etc. Code, however, requires that parameters be handled explicitly, using **S** (the parameter stack pointer) and the code-endings that push or pop the stack before the execution of **NEXT**.

6.6 ADDRESSING MODES

Although in general, Forth's assembler implements the processor manufacturer's mnemonics, there are standard notational conventions used in all processors for specifying addressing modes. Obviously, not all processors have all addressing modes or interpret terms such as "relative" identically. Nonetheless, certain basic concepts do exist and it's helpful when you're working with several processors to have these concepts expressed in standard ways.

Refer to your *CPU Supplement* for the specific addressing modes that are implemented in your system.

Standard Forth addressing notation includes the right parenthesis, which indicates either relative addressing (when it is by itself) or indexing (when it is combined with an index register designation). Some examples:

Word	Function
S)	Addressing relative to the top of the stack.
S)	Indexed by **S**.

1)	Indexed by Register 1.

On machines with automatic incrementing or decrementing, the parenthesis may be combined with **+** or **–**. On the DEC PDP-11 and the TI 9900, for example:

Word	Function
S)+	Refers to the number on top of the stack, "popping" it off at the same time; that is, incrementing the stack pointer.
S –)	Refers to the next available location on the stack, a "push" operation.

The position of the sign indicates when the increment or decrement takes place with respect to the development of the effective address, in this case postincrementing and predecrementing.

Immediate addressing is indicated by **#** and memory-indirect by the right parenthesis; the assembler can determine from the address whether **)** means register-relative or memory-relative (indirect). In addition, there are specific items of notation for each processor—these are described in the *CPU Supplement*.

Parameters may be taken directly from memory if this is permitted by the architecture of the processor. The assembler will automatically check to determine whether the address of the argument permits a short format instruction. If it will not, an extended format will be used. Often parameters may be picked up without being named. As long as an address is on the stack, it doesn't matter how it got there:

```
HERE 55 , ... LDA
```

will enter the literal *55* in the dictionary and leave its address on the stack at assembly time. (The operation puts the number that is on the stack into the dictionary at **HERE** and increments **H** by one cell.) The instruction **LDA** encounters the address on the stack and assembles an instruction to move its contents to Register A.

REFERENCES

, (comma), Section 2.7.2

6.7 MACROS

Macros are easily defined in Forth by using **:** definitions that contain assembler instructions. For example, on the RCA 1802 one frequently uses the operations **DEC** and **STR** successively on the same register. For convenience, the macro:

```
: DST ( r)    [ ASSEMBLER ] DUP DEC  STR ;
```

has been defined. Then **S DST** assembles the two instructions:

```
S DEC  S STR
```

Note the way the **DUP** in the definition of **DST** allows the single parameter S to be used by both the **DEC** and **STR** mnemonics.

Macros are mainly a notational convenience; **DST** assembles two instructions, just as if the expressions had been written out in full.

The words used to implement the assembler structures (loops and conditionals) are defined as macros, as are the code endings.

6.8 PROGRAM STRUCTURES

Control of logical flow is handled by Forth's assembler using the same structured approach as high-level Forth, although the implementation of the commands is necessarily different. The commands even have the same names as their high-level analogues; ambiguity is prevented by use of separate vocabularies. The following are implemented as standard macros:

Word	Function
BEGIN	Puts an address on the stack (**HERE**).
UNTIL	Assembles a conditional jump back to the address left by **BEGIN**. It is preceded by a condition code. The loop is ended if the condition is met. Common condition codes are **0=** and **0<**, as appropriate to the various CPUs.
NOT	Inverts the action taken for a condition code.
IF	Assembles a conditional forward jump, to be taken if the preceding condition is false, leaving the address of this instruction on the stack. It is also preceded by a condition code.

Word	Function
ELSE	Provides the destination of **IF**'s jump (whose address was on the stack) and assembles an unconditional forward jump (whose location is left on the stack).
THEN	Provides the destination for a jump instruction whose location is on the stack at assembly time (left by **IF** or **ELSE**).

The **ELSE** clause may be omitted entirely. This construction is functionally analogous to the **IF ... ELSE ... THEN** construction provided by Forth's compiler. For instance,

0= IF {code for 0} **ELSE** {code for not 0} **THEN ...**

0= IF {code for 0} **THEN ...**

Please note, however, that whereas the **IF** and **UNTIL** in high-level Forth remove an item from the stack and test it, the corresponding assembler words assemble conditional branches whose action will depend on condition codes set by the result of a previous instruction.

Since the locations or destinations of branches are left on the stack at assembly time, the structures **BEGIN ... UNTIL** and **IF ... ELSE ... THEN** may be nested naturally. By manipulating the stack during assembly, however, you can assemble any branching structure.

If you wish to branch forward, use **IF** to leave the location of the branch's address field on the stack. At the branch's destination, bring the location back to the top of the stack (if it is not there already) and use **ELSE** or **THEN** to complete the branch (by filling in the branch's destination at the location that is on the top of the stack). If you wish to branch back to an address, leave it on the stack with **BEGIN**. At the branch's source, bring the address to the top of the stack and use **UNTIL** or a jump mnemonic to assemble a conditional or unconditional branch back. Be sure to manipulate the branch address before the condition mnemonic since each condition code adds one item to the stack.

Suppose, for example, you wish to define a word **LOOK**, which takes two parameters (a delimiter on top of the stack with a starting address beneath it) and which scans successive bytes until it finds either the delimiter or a zero. The number of characters scanned is returned. Here is a definition of **LOOK** for the Motorola 6800:

```
~ CODE LOOK ( a c - n)   B PUL  A PUL  TSX
   0 ) LDX  BEGIN   0 ) TST   0= NOT IF
      0 ) A CMP   0= NOT IF   INX   B INC
      ROT JMP  THEN THEN   A CLR
   TSX   PUT JMP
```

Here the phrase **0= NOT IF** (used twice) assembles two conditional forward jumps which will be executed if the character scanned is the same as one of the delimiters. If the loop is to be repeated, after **B INC** a **JMP** is needed back to the **BEGIN**. Since the intervening **IF**s have left their locations on the stack, the backwards branch must be assembled by **ROT JMP**. The **ROT** (executed at assembly time) pulls the address left by **BEGIN** to the top of the stack where it is used as **JMP**'s destination. Finally, the **THEN**s fill in the destination of the **IF**s.

There are no labels in Forth. Although you could define them, their functions are better performed by the words **IF**, **ELSE**, **THEN**, **BEGIN**, and **UNTIL**. Since **CODE** definitions are usually extremely short, labels are not particularly desirable; they tend to encourage complicated flow patterns that are not appropriate in Forth.

6.9 LITERALS

Some processors allow you to define instructions to reference literals. For these, the standard Forth word for identifying a literal is **#**. Thus the instruction:

```
1000 # 0 MOV
```

would move the literal 1000 into Register 0. A few processors allow a short instruction format for small literals and a long format for larger ones. In such cases the Forth assembler automatically examines the literal and generates the appropriate format.

On processors that do not directly support literals, the main technique for supplying them is to compile a literal directly, and then pass the literal's address to an instruction that references it by **HERE**. For example:

```
HERE 1000 ,
CODE FIX   0 MOV ...
```

In this example the literal *1000* is placed in memory and its address left on the stack. The **MOV** instruction assembles a reference to that address. When executed, the effect will be to move 1000 into Register 0.

6.10 DEVICE HANDLERS

Device handlers should be kept extremely short, including only the instructions required to pass a value to or from the stack or to issue a command. Consider, for example, a self-scan character display that is interfaced to an RCA 1802 as Device 2. This is all that is needed to output one character from the top of the stack:

```
CODE (EMIT) ( c)   S INC   S SEX
   2 OUT   NEXT
```

In this example, **S INC** increments the stack pointer (to get the low-order byte), **S SEX** sets **S** as the output register, and **2 OUT** sends the character to the device, incrementing **S** again to complete a **POP**.

Given this hypothetical definition of **(EMIT)** you could define **(TYPE)** at high-level to display a string of characters whose byte address and length are on the stack:

```
: (TYPE) ( a n)   0 DO  PAUSE  DUP C@
   (EMIT)  1+ LOOP  DROP ;
```

To convert and display a number on the stack, you could define **SHOW**:

```
: SHOW ( n)    (.) (TYPE) ;
```

Here **(.)** performs the conversion, leaving the address and length of the resulting string for **(TYPE)**. The point here is that, given the simple code definition **(EMIT)**, full control of the display is available in high-level Forth.

Device drivers are highly variable in nature, depending upon both the processor and the actual device. You'll find a discussion of drivers for your processor in your *CPU Supplement* and useful examples in the system listings.

6.11 INTERRUPTS

The presence of Forth's multiprogrammer (plus a few standard conventions) makes dealing with interrupts in Forth relatively simple. The principle strategy is to perform only the most time-critical actions at interrupt time, to notify the task responsible for the interrupting device that the interrupt has occurred, and to defer all complex logic to high-level routines executed by that task. "Notification" may take the form of setting a flag, incrementing or decrementing a counter, or modifying the task's status such that it will become active at the next opportunity in the multiprogrammer cycle.

The basic form of an interrupt handler is as follows:

ASSEMBLER BEGIN {code instructions} **dev# INTERRUPT**

where **ASSEMBLER** selects the assembler vocabulary (**CODE** does this for you automatically but should not be used because it builds an unneeded head); **BEGIN** pushes onto the stack the address of the beginning of the code (which will be used by the word **INTERRUPT**); the code instructions perform the necessary work of the routine; dev# stands for the device code or interrupt vector to which the routine will respond, and **INTERRUPT** is a special code ending macro that assembles the appropriate "return from interrupt" instruction and sets the address of the code supplied by **BEGIN** in the interrupt vector.

The actual implementation of **INTERRUPT** is highly processor-dependent. On machines with hardware-vectored interrupts, the implementation merely stores the address of the code in the specified vector address. On such machines, interrupts incur no additional overhead: only the instructions in the interrupt routine itself are executed. On machines in which software must identify the interrupting device, the identification system is system specific—consult your *CPU Supplement*. However, one popular method is to set the polling routines in a chain, with the CPU's interrupt vector pointing to the first polling routine. If the device served by the first routine did not generate the interrupt, the first routine executes a jump to the second routine.

On every system conventions are established for the use of registers at interrupt time. On most systems, you may not use any registers without saving and restoring them. Save and restore only the registers you are actually going to use! The usual place to save registers is on the return stack. On systems with only one hardware stack, the parameter stack becomes the place of choice. On systems with software vectors and few registers, one or two registers are routinely saved and restored so that you may use them freely. Consult your *CPU Supplement* for details.

6.12 EXAMPLE

As an example of the action of the assembler, consider the definition of the high-level comparison operator **0=**. This word expects a value on the stack. If the value is non-zero, it will be replaced by a zero (false); if it is zero it will be replaced by a negative one (true). The code for this routine on the Intel 8086 is:

```
CODE 0= ( n - t)    0 POP    0 0 OR
   0 # 0 MOV    0= IF    0 DEC
   THEN    0 PUSH    NEXT
```

Fig. 6.2 and Table 6.2 show the processor during compilation and execution of this routine. Table 6.3 shows, for the purpose of comparison, the definition of **0=** for other processors.

```
DECODE 0=
   58D          58                0 POP
   58E          0B C0             0 0 OR
   590          B8 00 00          0 # 0 MOV
   593          75 01             596 JNE
   595          48                0 DEC
   596          50                0 PUSH
   597          AD 97 FF 25       NEXT
```

Fig. 6.2

Disassembly of **0=** (8086 version) using the polyFORTH disassembler utility (see the *CPU Supplement* for details).

Table 6.2

Assembly of **0=** on the 8086

Instr.	Action During Assembly	Action During Execution
0	Pushes 0 (Register AX) onto the stack.	Pops the top of the stack into Register AX.
POP	Assembles a POP instruction, referencing Register 0.	
Instr.	**Action During Assembly**	**Action During Execution**
0	Pushes 0 onto the stack.	**OR**'s Register AX with itself, which sets the status bits.
0	Pushes 0 onto the stack.	
OR	Assembles an instruction to "or" Register AX with itself.	
0	Pushes 0 onto the stack.	Puts a zero in Register AX, without affecting the status bits.
#	Pushes 10 onto the stack to indicate immediate addressing mode.	
0	Pushes 0 onto the stack.	
MOV	Assembles an instruction to move a zero to Register 0 (AX).	
0=	Pushes 74_H onto the stack (code for ≠)	Branches if the condition bits do not indicate a zero.
IF	Assembles a JNE instruction with a destination address of zero, and leaves the address of the destination field on the stack.	
0	Pushes 0 onto the stack, for Reg. AX.	Decrements Register AX. Since AX contained 0, this leaves a -1 in AX.
DEC	Assembles an instruction to decrement Register AX.	

THEN	Stores the current value of **H** (the top-of-dictionary pointer) in the address left on the stack by **IF.** Assembles nothing.	No action.
0	Pushes 0 onto the stack.	Pushes the content of Register AX (containing either 0 or -1) onto the stack.
PUSH	Assembles an instruction to push the contents of Register 0 (AX) onto the stack.	
NEXT	Assembles the following sequence: **LODS W 0 XCHG W) LIP**	Returns to the address interpreter.

Table 6.3

Definition of **0=**

On the 68000:

```
CODE 0= ( n - t)   D1 CLR   S ) TST
   0= IF   1 #Q D1 SUB
   THEN  D1 S ) MOV   NEXT
```

On the PDP-11:

```
CODE 0= ( n - t)   0 CLR   S ) TST
   0= IF   0 DEC   THEN
   S ) 0 MOV   NEXT
```

7.0 TARGET COMPILATION

Target compilation was originally developed to cross-compile a Forth system from one type of computer to another. Now target compilation is used to produce stand-alone applications that do not require the full Forth programming environment.

The target compiler is not a single program, but a collection of software tools to solve three problems:

1. How can a system recompile itself? This problem is solved by compiling the new system into an out-of-the-way place. The assembler and compiler are redefined. The process of using polyFORTH to recompile itself is discussed in Sections 7.3, 7.4, 7.5, 7.6, and 7.7.

2. How can the old system locate objects in the new system? This problem is solved with special dictionary entries, which are discussed in Sections 7.1, 7.2, 7.7, and 7.8.

3. How does the old system differentiate between the dictionary entries of itself and the new system? This problem is solved by using vocabularies, as discussed in Sections 7.1, 7.2, 7.7, and 7.8.

Other sections discuss diagnostics, debugging and other useful techniques.

The target compiler is a utility which is loaded by the command:

```
COMPILER LOAD
```

Subsequently, you load your application using a load block which, in turn, loads the source blocks describing the nucleus primitives and other system support words needed, and finally the application itself. More detailed information about recommended target compilation procedures is provided in Sections 7.1 and 7.13.

There is a facility for interactively testing target compilable applications before they are target compiled. This facility is in a block called "The target compatibility block." You may find it helpful to look at its commands to understand the general meaning (but not the actual process) of certain target compiler words.

REFERENCES

The Target Compatibility Block, Section 7.12

7.1 RESIDENT, HOST, AND TARGET WORDS

Target compilation presumes that two machines exist. The machine that already runs polyFORTH and acts as a host for the target compilation is called the "host" machine. The machine which is the target of the compilation effort is called the "target" machine.

Three sets of words exist when performing target compilation.

1. "Resident" words are the ordinary, native Forth words which reside on the host system.

2. "Host" words compile definitions for the target system, but are written using resident words. Host words provide the means by which the host offers its resources to compile the new target system.

3. The words which will execute on the new target system are called "target" words.

Target definitions differ from normal Forth definitions in that they have *two dictionary entries*: one in the dictionary being compiled for the target, and one in the dictionary of the host. A target word's *code* is in the target dictionary, and will be executed by the target computer. The host's definition for the target word is basically a *constant*, with special run-time behavior which compiles the constant's *value* into the target's dictionary (see the reference to **EMPLACE**, below). This value is the address of the target definition's code field in the target dictionary.

The host entry is necessary because during target compilation the resident system's dictionary is the only one that can be searched. Target dictionary entries may not have names and dictionary links, and may have run-time code which cannot execute correctly on the host.

The definitions that make up the new compiler and assembler used to produce definitions for the target system are *host words*. In this section they are called the *host compiler* and *host assembler*.

Resident, host, and target words all have their own vocabularies in the host's dictionary.

REFERENCES

Defining Words, Section 2.7
EMPLACE, Section 7.7
Vocabulary Conventions, Section 7.2
Vocabularies, Section 3.4

7.2 VOCABULARY CONVENTIONS

The host system differentiates between resident, host, and target words by using vocabularies. If you are not familiar with polyFORTH's use of vocabularies, you should review Section 3.4.

All of the vocabularies used in the target compiler reside on the host machine. The original polyFORTH system uses three vocabularies:

```
FORTH          0001
ASSEMBLER      0013
EDITOR         0015
```

The last digit is the first vocabulary searched. The numbers are in hexadecimal with leading zeros shown. Your machine may have the digits in reversed order.

Two new vocabularies are used for host words:

Vocabulary	Search Order	Description
HOST	0017	The **HOST** compiler, sometimes called the "target" compiler. Contains the words which produce the target program.
ASSEMBLER	0179	The **HOST** assembler, sometimes called the "target" assembler.. Contains the assembler words used to produce the target program's nucleus.

In addition to these, **FORTH** is re-defined to have a search order of 0071_H (adding **HOST** to the search order). **HOST** and this version of **FORTH** are also **IMMEDIATE** words, unlike most vocabularies in polyFORTH, meaning that they will be executed immediately when they are invoked in colon definitions.

If the target system is itself programmable, three target vocabularies must reside in the host with index numbers similar to the resident vocabularies **FORTH**, **ASSEMBLER**, and **EDITOR**, but are offset by 10 decimal ($0A_H$). The respective hexadecimal assignments for the target versions of the standard vocabularies are:

```
FORTH           000B
ASSEMBLER       00BD
EDITOR          00BF
```

The vocabulary conventions for the target compiler are intended to provide syntactically equivalent access to both the resident and target vocabularies. The target vocabularies are accessed by the host version of **-'**. Since the host compiler only uses the host **-'**, the host compiler will only compile words which have entries in the host's target vocabularies. The host version of **-'** works by temporarily substituting a vocabulary search pattern from the variable **VOC** into **CONTEXT**, and then performing a resident **-'**.

VOC serves as the target compiler's **CURRENT**, in that it controls the linking of new words. It also functions as its **CONTEXT** (when it is used by the host version of **-'**). The normal value of **VOC** is 000B.

The word **TARGET** unlinks the most recently defined word from a resident vocabulary of the host dictionary and relinks the new word into the primary vocabulary of **VOC** (usually target-FORTH). **TARGET** is used immediately following the definition of a word to create words which will be executed by the target compiler rather than just compiled. **TARGET** is analogous to the resident word **IMMEDIATE**. A good example of **TARGET**'s use is in the definition of the target compiler's compiler directives (see Block 249 of a system listing). The target compiler usually searches only the target vocabularies (according to **VOC**), and thus its compiler directives must be in the target vocabularies.

REFERENCES

-', Section 2.6.1
Host **(CREATE)**, Section 7.5.2
Vocabulary Definitions, Section 3.4.2

7.3 DICTIONARY CONVENTIONS

The target compiler was designed to be usable for as many types of applications as possible. The polyFORTH nucleus is designed to work in either read-only memory or read/write memory. The target compiler can therefore compile applications for two major environments:

1. The "conventional" environment in which most system memory is read/write memory. The system probably has a disk and terminals.

2. The "dedicated" environment, in which the program resides permanently in read-only-memory, with a small block of scratch-pad read/write memory. All types of memory may need to be minimized if the dedicated system will be mass-produced.

The dedicated environment is more demanding of the target compiler, because two types of memory must be managed. Read-only memory can contain programs and constants, but no variable data. The presence or absence of read-only memory is the major difference between conventional and dedicated environments.

7.3.1 Dictionary Conventions for Read-Only Memory

Forth creates code which is naturally non-self-modifying and can be placed in non-volatile read-only-memory. All Forth facilities, including multitasking and the target-compiler, are designed so that they will be usable to produce a system with a minimum amount of read/write memory.

The majority of dictionary entries in Forth contain executable routines. The rest are labels to data areas. In a Forth system using ROM (Read Only Memory) the permanent dictionary resides in ROM. Data areas in RAM (Random Access Memory) are labeled by constants in the dictionary in ROM. The value of the constant is a pointer into RAM.

Because the target compiler must be able to allocate both RAM and ROM, the target compiler maintains two dictionary pointers. **'H** (pronounced "tick-H") is the dictionary pointer most similar to the normal, resident pointer H. **'H** points to the next available byte of read-only-memory. **'R** is the dictionary pointer to the next available byte of read/write (RAM) memory.

These are the words for managing memory. Note that sometimes pairs of words are defined which perform analogous functions for ROM and RAM:

Word	Stack	Function
WINDOW	(a)	The vocabulary to which the target system's **FORTH** definitions are emplaced.
HDS	(- a)	Returns the address of the cell containing the number of bytes in a dictionary head, excluding the cfa. **HDS 2+** is the address of the default head length. The head length in **HDS** is reset to **HDS 2+** by **(CREATE)**. (On 32-bit processors, **HDS 4+** is the address of the default head length.) **HDS** and **HDS 2+** are set by **DICTIONARY**.
DICTIONARY	(n)	Sets the maximum number of bytes in a dictionary name to *n*. A value of zero will let **(CREATE)** compile just the code field, and results in a minimum size dictionary of headless definitions. An interactive target uses 34 bytes, which yields a dictionary link field, a character count, and up to 31 characters. A code field is always included. Individual definitions can be made headless by preceding them with a **|** ("bar" ASCII 124 decimal), or they can be given nontruncated names by preceding them with a **~** ("tilde"). **DICTIONARY** also clears the target space to a fill value (typically -1).
HERE	(- a)	Returns the address of the next byte of the target system's ROM. This word is similar to the resident **HERE** but refers to ROM storage only (not variable data areas, which must be in RAM).
THERE	(- a)	Returns the address of the next byte of the target system's RAM. This word is the RAM version of **HERE** used to refer to variable data areas.
ORG	(a)	Sets the ROM dictionary pointer (**'H**) to *a*.
GAP	(n)	Advances **'H** by n to reserve n bytes of ROM. Similar to **ALLOT**.
Word	**Stack**	**Function**
ALLOT	(n)	Advances **'R** by *n* to reserve *n* bytes of RAM. Using **ALLOT** after **WINDOW** sets **'R** to an initial non-zero RAM address. Similar to **GAP**.
EQU name	(n)	Creates a host constant without a corresponding entry in the target dictionary.
LABEL name		Labels a ROM target address location in the host system only, using **HERE** and **EQU**. When name is executed, it returns the target address.

|	Sets the number of bytes in **HDS** to 0, so that the following dictionary entry has only a code-field address. **HDS** is automatically reset to a default after the target head is created by **(CREATE)**. Pronounced "bar" (ASCII 12410).
~	Sets the number of bytes in **WIDTH** to 31, so that the entire name is compiled for the following dictionary entry. **WIDTH** is automatically reset to a default. Pronounced: "tilde"
RECOVER	Reclaims a cell in the dictionary. Often used after infinite loops or words ending in **STOP**, **ABORT** or **QUIT** to reclaim the two-byte address of **EXIT** compiled by ;.

REFERENCES

~, Section 1.1.1

7.3.2 Dictionary Conventions for Read/Write Memory

A program compiled for ROM using the tools and conventions described above will work satisfactorily in a RAM system as well. If you wish to avoid having to allocate pseudo-ROM and RAM separately, however, running with data and definitions intermingled in the nucleus as they are in a resident dictionary, you may easily modify the polyFORTH target compiler to accommodate this.

The definitions that must change are those for **VARIABLE** (and its relatives) and **ALLOT**. In addition, you must change references to **THERE**, **'R**, etc.

Resident	Target Compiler	Remarks on Target Version
H	H	Top of host dictionary
	‘H	Top of remote (ROM) dictionary
	‘R	Top of allocated RAM
HERE (H @)	HERE (‘H @)	Next available byte of target ROM
	THERE (‘R @)	Next available byte of target RAM
n ALLOT (H + !)	n ALLOT (‘R +!)	Allots target RAM
CREATE name	n GAP (‘H +!)	Allots target ROM
	CREATE tname	‘tname’ returns target ROM address
VARIABLE name	VARIABLE tname	‘tname’ returns target RAM address
FORTH, EDITOR, ASSEMBLER	FORTH, EDITOR, ASSEMBLER, HOST	All but **ASSEMBLER** are **IMMEDIATE**
a n DUMP	a n DUMP	Dumps target address space

Fig. 7.1

Chart showing Forth words used to control memory space in resident and host compilers.

7.4 COMPILATION TO A VIRTUAL DICTIONARY

The target compiler produces an entirely new dictionary tailored by a programmer for the target computer.

The new dictionary is constructed in a space apart from the host dictionary, in RAM or on disk. If the target dictionary is being built in RAM, it will be copied to disk at the end of the target compilation process.

The size of the target dictionary is controlled by the constant **#K**, which gives its size in units of 1024 (1K) bytes. It will be put on disk starting at the block whose number is given by the constant **NEW**.

7.4.1 Words that Differ for Different Types of Target Space

When the target compilation takes place, the code produced is not “mixed in” with the resident dictionary. Instead, space is set aside on disk or in RAM and then accessed by the target compiler through a small, device-independent vocabulary of fetch, store, and compile words. The code is usually compiled to an 8K-byte buffer called **RAM**. This space is set aside for performing target compilation and will be called “target space.” The first byte of target space coincides with the first byte of the target application program. In many (but not all) systems, the first byte of target space will be the target CPU’s memory location zero.

The target compiler is designed so that it can compile to either a RAM buffer or an area on disk. The target compiler load block contains a line which will be commented:

(Compile to RAM) or **(Compile to disk)**

The special words that must be defined for each type of compilation are loaded by that line. You may change that line so that the target compiler compiles to either disk or RAM. To find the exact block numbers refer to the disk index at the beginning of your program listing to find the **(Compile to RAM)** and **(Compile to disk)** blocks in the target compiler section of your system disk.

Certain words are defined or redefined so as to operate in target space:

Word	Stack	Function
>T	(a - a)	Translates a target space address into a host address. See the following subsections for details.
TC@	(a - b)	Fetches a byte from target space.
TC!	(b a)	Stores a byte to target space.
T@	(a - n)	Fetches a cell from target space.
T!	(n a)	Stores a cell to target space.
T+!	(n a)	Adds to a cell in target space.
C,	(b)	Compiles a byte in the next byte of ROM in target space.
,	(n)	Compiles a cell in the next two available bytes of ROM in target space (next four available bytes on 32-bit processors.
CMOVE	(s d n)	Moves *n* bytes starting from a host-system source address *s* to a target space destination *d*.
DUMP	(a n)	Dumps *n* bytes to the display, starting at target space address *a*.
DICTIONARY	(n)	Sets the maximum number of bytes that will be placed in a dictionary name to *n*. A value of zero will let **(CREATE)** compile just the code field address and results in a minimum size dictionary of headless definitions. The default size is 34 bytes, which gives a dictionary link field, a character count, and 31 characters. A code field is always included. Individual definitions can be made headless by preceding them with a **|** ("bar"), or they can be given non-truncated names by preceding them with a **~** ("tilde"). **DICTIONARY** also clears the target space to 0 or -1. The fill value should be -1 to allow patching in most ROM-based systems.

Depending on where the target space is kept, some other words, notably **FLUSH**, are redefined. **FLUSH** occurs just once in the target compilation, at the end. **FLUSH** saves the target space on disk. On systems which are compiling to RAM, **FLUSH** is redefined to copy the target space to the range of blocks whose start is given by the constant **NEW** and length by **#K**.

7.4.2 Compiling to RAM

One of the places into which the target compiler can compile code is a memory buffer **#K** bytes in length called **RAM**. Compiling to RAM is much faster than compiling to disk. For extensive target-compiled applications, **RAM** may need to be larger on some computers.

The compiled program is always saved on disk at the end of the target compilation; usually in the same place as with the "compile-to-disk" option (change Block 240 to alter the "compile-to-RAM" to "compile-to-disk").

Since target compilation compiles to different places, the definition of the word **>T** (pronounced "to-T") changes. **>T** translates a memory address for the target machine into an address for the host machine's target compilation space. The definition for **>T** differs considerably between a target compilation to disk and a target compilation to

the buffer called **RAM**. When target compiling to **RAM**, **>T** basically adds an offset to the target address to displace it into the buffer called RAM. All physical addressing in the target compiler passes through **>T**.

Some other words are defined in special ways for RAM compilation:

Word	Function
DICTIONARY	Clears the RAM buffer, and sets **HDS**, the maximum dictionary length.
FLUSH	Moves the RAM buffer to disk, to the range of blocks starting at **NEW**. **FLUSH** should occur only once in a target compilation: at the very end.
DUMP	Redefined to dump target space to the display.

7.4.3 Compiling to Disk

If memory is limited (for example, if you are one of a number of programmers sharing a system with limited partition sizes), you can compile your program directly to the disk region described in the previous section.

The target compiler can compile to either disk or RAM; the choice is up to the user, based on how much RAM is available. Change Block 240 to alter the "compile-to-disk" option to the "compile-to-RAM" option.

Compiling to disk is slow, but does not require a large region of memory. Compiling to disk is sometimes necessary when several programmers are using one machine.

As target compilation compiles to different places, the definition of **>T** changes. **>T** translates a memory address for the target machine into an address for the host machine's target compilation space. When compiling to disk, **>T** uses **BLOCK** to treat disk as virtual memory. All physical addressing in the target compiler passes through **>T**.

Some other words are defined in special ways for disk compilation:

Word	Function
DICTIONARY	Clears the disk blocks of the compilation area (sets all the values to -1 or 0), and sets **HDS**, the maximum dictionary name length.
CMOVE	redefined in high level. Moves from host memory to target space (on disk) using **TC!**.
DUMP	Redefined to dump target space to the display, using **T@**.

7.4.4 Compiling to a Remote Target

It is possible to target compile directly into a target machine, provided that the machine has sufficient read/write memory (with the correct addresses at RAM), a serial link to the host, and a "talker" program.

The talker program must be able to: read a byte from target memory to the serial link; write a byte from the serial link to target memory; and begin execution at an address given over the link.

An advantage of this form of target compilation is that a target-compiled system may be interactively debugged on the target machine. Another major advantage is that the target-compiled system may use the host system's peripherals, such as disks, printer and display. The disadvantages are that the system interconnection hardware may be clumsy to arrange and in most situations, target compilation to RAM gives almost the same or more interaction.

This is the method of choice for cross compiling Forth to another machine. Typically, a core system is first defined which performs terminal and disk functions over a serial line connected to the host. Then the system I/O functions are added until a free-standing system exists. The final step is the installation of a disk bootstrap.

The words are defined as follows:

Word	Function
>T	Adds an offset (which may be zero).
TC@	Is a serial I/O routine which sends an operation ID and an address, and receives a byte (usually using **STRAIGHT**, see references).
TC!	Is a serial I/O routine which sends an operation ID, an address and a byte.

Target memory reference is written to use **TC@** and **TC!**.

Word	Function
EXECUTE	Is a serial I/O routine. Sends an operation ID and an address. The target machine jumps to the address.
FLUSH	Copies the target system to the host's disk by using **TC@** to fetch data over the serial link.
CMOVE	Moves bytes from the host to the target using **TC!**.
DUMP	Is defined to dump target space to the host's display, using **TC@**.

REFERENCES

STRAIGHT, Section 3.7.1

7.5 HOST DEFINING WORDS

A **HOST** defining word (see Fig. 7.2 and 7.3) creates dictionary entries in both the **HOST** dictionary and the target dictionary. The target dictionary contains the defined word's run-time behavior and parameter fields. The host dictionary contains constants that point to the target dictionary entries.

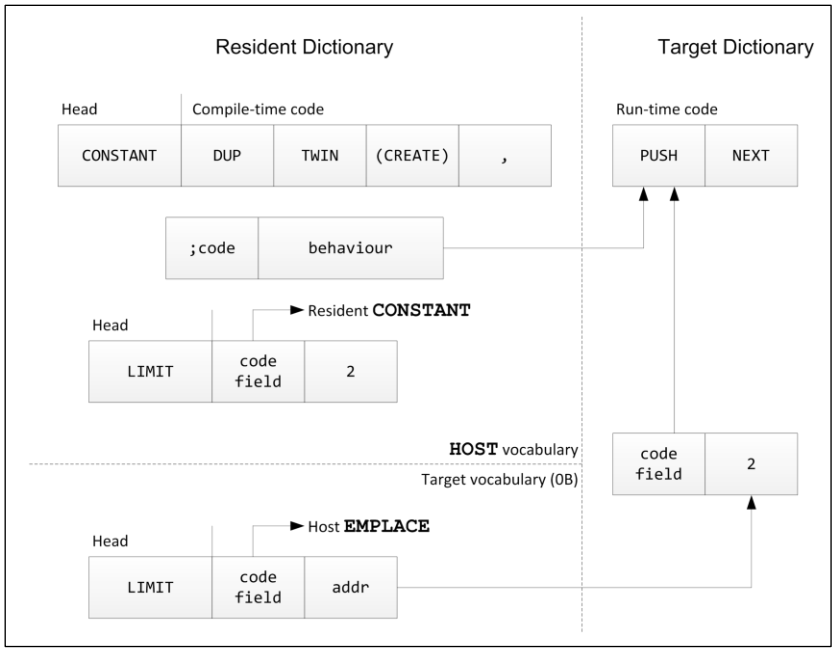


Fig. 7.2

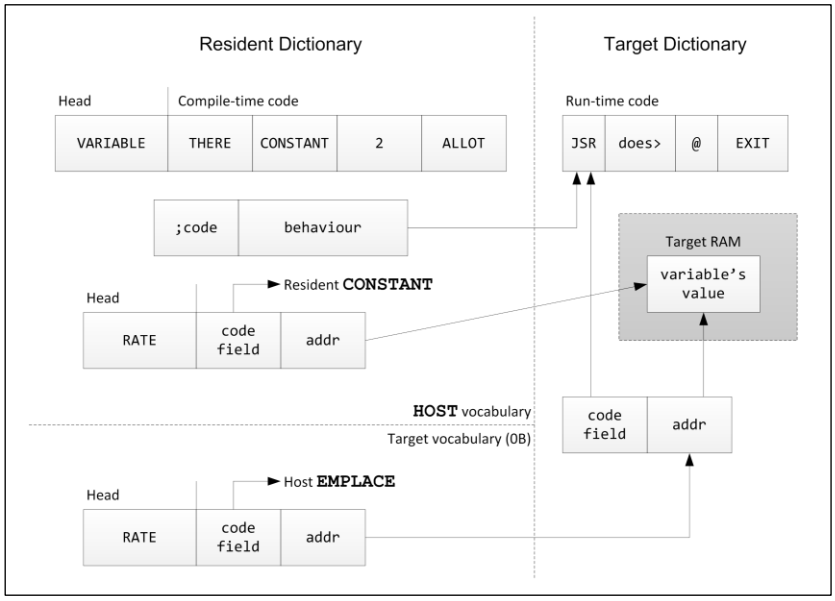


Fig. 7.3

Fig. 7.2 is a diagram of the structures used and generated by a host defining word using **;CODE**:

```
FORTH : 2CONSTANT ( n)
   DUP TWIN   (CREATE),   ;CODE
   2 W) PUSH    NEXT
```

(8086 version). An example of its use is:

```
0 01000 2CONSTANT LIMITS
```

Fig. 7.3 is a diagram of the structures used and generated by a host defining word using the host **DOES>**:

```
FORTH : VARIABLE   THERE CONSTANT
   2 ALLOT  DOES>  @ ;
```

An example of its use is:

```
VARIABLE RATE
```

REFERENCES

Defining Words, Section 2.7

7.5.1 Using HOST Defining Words

In many instances, an application can be written more conveniently and compactly if defining words are used. A defining word for a target compiled application must be loaded after the Forth nucleus and after the host defining words in Blocks 246-250. The nucleus must be loaded so that target run-time code is available for the definitions of **CREATE** and **CONSTANT** in Block 250.

A fairly common type of application defining word defines arrays. A resident definition to define byte arrays would be:

```
: ARRAY ( n)   CREATE ALLOT ;
```

What the above word does is label the beginning of an area of memory (with **CREATE**), and then **ALLOT** memory to form space for the array. To create an array of 30 bytes named **BUF**, the word **ARRAY** would be used:

```
30 ARRAY BUF
```

Note that after this definition **BUF** can be used in colon definitions to provide an address:

```
: @BUF ( n - n)   BUF +  C@ ;
```

A host version of the word **ARRAY** should be usable in exactly the same way as a resident version.

If **ARRAY** were target compiled without change, several things would go wrong:

1. The host **:** would compile **ARRAY** into the target dictionary, making it unavailable for interpretive execution in the host.

2. If **ARRAY** were available in the host dictionary, the host's **CREATE** in **ARRAY** would label a position in target ROM, rather than target read/write memory.

Fixing these two problems might yield the definition:

```
FORTH : ARRAY ( n)   THERE EQU ALLOT ;
```

The word **FORTH** ensures that the resident colon compiler compiles **ARRAY** into the resident dictionary (so that **ARRAY** can be executed). **THERE** returns the address of the next byte of target RAM. **EQU**, generates only a resident dictionary entry. Thus, the following sequence is in error:

```
FORTH : ARRAY ( n)   THERE EQU  ALLOT ;
30 ARRAY BUF

: @BUF ( n - n)   BUF + @ ;
```

The error would occur because **BUF** (as created by **EQU**) has no entry in the target dictionary, and thus no address for **BUF** can be compiled into **@BUF**. No error would occur if **@BUF** and all other words that reference **BUF** were defined in code, because in a code definition **BUF** would be executed interpretively at target compilation time. Also, the erroneous definitions in the example above are one of very few situations in which code will load correctly with the target compatibility block and incorrectly with the target compiler.

The definition of the host word **ARRAY** which has exactly the same usage as the resident word is:

```
FORTH : ARRAY ( n)   VARIABLE 2-  ALLOT ;
```

Note that the target compiler's version of **VARIABLE** for programs in ROM is a **CONSTANT** which returns the address of a cell in RAM. Thus the phrase **VARIABLE 2-** is equivalent to the phrase **THERE CONSTANT**.

Some host defining words need to have special run-time behaviors. For these instances, the target compiler provides versions of the words **;CODE** and **DOES>**. These words may be used exactly as in a resident system, except that **;CODE** must be followed by code in the target's assembly language. The following is an example of a word using **DOES>**:

```
FORTH : COEFFICIENT ( n)   CREATE ,
   DOES> ( a)  @ * ;
```

The host version of **DOES>** works differently from the resident version; **DOES>** is executed because **DOES>** is **IMMEDIATE**. The host **DOES>** compiles a run-time word for itself (see Fig. 7.3) in the host definition of **COEFFICIENT**, then compiles an entry point in the target dictionary. The target code at the entry point jumps to the target's run-time definition for **DOES>** in the target nucleus. After assembling the target entry point, **DOES>** begins compilation, compiling the remainder of the defining word's source definition into the target dictionary, following the defining word's target machine code entry point. The resident portion of the defining word has a compiled value at the end of its resident dictionary entry that is the address of the target run-time portion's machine code entry point.

New target words are defined when the host **CREATE** makes a Target Compilation dictionary entry in the target's dictionary, and the code compiled by the host **DOES>** changes the new target word's code field address to point to the defining word's run-time machine code entry in the target dictionary.

Frequently the most compact solution to a problem uses many defining words because defining words of the same type share code.

REFERENCES

Defining Words, Section 2.7

7.5.2 The Operations of HOST Defining Words

The first host defining word loaded with the target compiler is **(CREATE)**. **(CREATE)** is used by all host defining words to construct dictionary entries. Most of the complexity of **(CREATE)** occurs because **(CREATE)** builds entries in the target dictionary which are either headless or have names and dictionary links. **(CREATE)** has the following steps:

1. It uses **EMPLACE** to make a constant in one of the host's target vocabularies (see References below—**EMPLACE**).

2. It compiles a dictionary link in the target dictionary and links the new entry between **HEAD** (the target's **GOLDEN** array) and the previous value in **HEAD**.

3. It compiles the new word's name in the target dictionary, preceded by a length.

4. The value in **HDS** (which controls the size of the word's head in the target dictionary) is reset to the default value in **HDS 2+** (**HDS 4+** on 32-bit processors).

5. The code field address of the target dictionary entry is set to point to the first byte following the code field address. This value of the code field address implies that machine code will follow the target's head immediately, unless the code field address is changed (as it is after the nucleus is compiled, when target run-time behaviors are available).

When **HDS** has a value of 0, **(CREATE)** skips Steps 2 and 3, thus compiling only a code field address in the target dictionary. **HDS** may be set temporarily (see Step 4) to 0 by the word **|** (pronounced: "bar"). **HDS** is reset to the value in **HDS 2+** at the end of **(CREATE)**. **HDS 2+** was initially set by the value into **DICTIONARY** (**HDS 4+** on 32-bit processors). (See Block 181 of your system listing and the reference below).

Some target constants, addresses, and variables need to be accessed during target compilation. Since the values for these words are in the target's dictionary, they would not normally be available. The word **TWIN** creates a twin dictionary entry of these constants, addresses etc. in the host's dictionary, in a resident vocabulary.

Some host defining words only make entries in the host dictionary. The two most notable are **EQU** and **LABEL**.

Command	Action
n EQU name	Defines a resident constant of value *n* whose name is *name*. Saves space in the target dictionary. Note that a word created by **EQU** may not be used in a target colon definition.
LABEL name	Defines a resident constant with the value of **HERE**. Useful for labeling assembly language entry points. Note that a word created by **LABEL** may not be used in a target colon definition.

REFERENCES

DICTIONARY , Section 7.3.1
EMPLACE, Section 7.7

7.6 THE HOST ASSEMBLER

The host assembler has been called the "target" assembler because it assembles code for the target machine. Do not confuse this with the assembler that will run on the target machine. We will use the following definitions in this section:

Word	Description
Resident Assembler	Resides in the resident vocabulary called **ASSEMBLER** in the host computer, and assembles code for the resident system.
Host Assembler	Resides in the host vocabulary called **ASSEMBLER** , assembles code for the target system during target compilation.

Target Assembler	Resides in the target system, and is not executable in the host. Will be the target system's resident assembler.

If the host and target have the same CPU, the host assembler source is identical to the source for the target and resident assemblers. The host assembler behaves differently because **,** and **C,**, the lowest level compiling words in the assembler, are **HOST** definitions which compile to the target's address space.

At the beginning of target compilation, before the target nucleus is assembled, no target behaviors exist, not even **NEXT**. The host assembler is the part of the target compiler which creates the fundamental target behaviors. The word which first creates the heads (name, length, dictionary link, and code field address) for the host assembler is **(CREATE)**. **(CREATE)** is used in the host assembler's definition of **CODE**, as well as the later host defining words. **(CREATE)** sets the code field address of the target dictionary entry so that it points at the entry's parameter field (see Fig. 7.2 and 7.3).

Note that if a different assembler is loaded by Block 240, and that if the source code is correct for the new assembler, the type of CPU is unimportant to the target compiler. Target compilation forms a very powerful cross compilation facility (not surprising, as cross compilation was the target compiler's original purpose).

REFERENCES

(CREATE), Section 7.5.2
CODE, Section 6.1

7.7 THE HOST COMPILER

The host compiler has been called the "target" compiler because it compiles colon definitions for the target machine. Do not confuse this with the compiler that will run on the target machine.

Word	Description
Resident Compiler	Resides in the resident vocabulary called **FORTH**. Compiles high-level colon definitions for the resident system.
Host Compiler	Resides in the host vocabulary called **HOST** during target compilation, and compiles colon definitions for the target system.
Target Compiler	Resides in the target system and is not executable in the host. Will be the target system's colon compiler.

The host compiler (the word **]**, pronounced: "right bracket") is simpler than a resident colon compiler. The host compiler looks up words and executes their host dictionary definitions. The result is usually to compile something (such as the target address of the word) in the target dictionary. If a word cannot be found, the host compiler attempts to convert it to a number, and compile the number in the target dictionary. The host compiler cannot be loaded until after the target's nucleus is assembled, because the host compiler needs target addresses for the run-time behaviors of literals, **:**, **;**, and all compiler directives such as **DO**, **IF**, etc.

When the host compiler executes the host definition of a word, the run-time behavior of the word actually compiles values into target address space. The most common example of such a run-time behavior is the word **EMPLACE**. **EMPLACE** compiles the addresses in every high-level word in a target compiled system. **EMPLACE** contains the run-time behavior of **(CREATE)** (which creates target dictionary heads, with a parallel entry in the host's target vocabularies). At compile time, **EMPLACE** uses the resident version of **CONSTANT** to compile a named value in a target vocabulary of the host's dictionary. When the word defined by **EMPLACE** is executed (by the host compiler, for example) the run-time behavior of **EMPLACE** compiles the word's value into the next available cell of read-only memory in target address space. The words defined by **EMPLACE** always have values which are the target addresses of target parameter field addresses.

A typical definition of **EMPLACE** is:

```
: EMPLACE ( a)   LOG CONSTANT TARGET
   DOES> @ 2- , ;
```

LOG produces an entry in the compilation log. **CONSTANT** compiles the target address on the stack into a resident vocabulary of the host (probably **HOST**). **TARGET** relinks the dictionary links to the new word, so that the new word may be found in the host's target vocabularies. Note that the comma after **DOES>** compiles to target address space because the comma is from the **HOST** vocabulary.

Compiler directives and similar immediate words are placed directly into a target vocabulary of the host, to be executed by the host compiler.

REFERENCES

(CREATE), Section 7.5.2
IMMEDIATE Words, Section 2.8.8

7.8 TARGET DEFINING WORDS

There are three levels of system that can be produced by means of target compilation:

1. Non-interactive—Usually a dedicated controller of some sort. All dictionary entries are headless.

2. Interactive—Some dictionary entries have heads. The text interpreter and at least one terminal exist in the system. An interactive system is a superset of a non-interactive system.

3. Extensible—Words exist in the target dictionary which can define more words in the target dictionary. An extensible system is a superset of an interactive system.

Target defining words are the words which make the third level possible. Target defining words are essential parts of your polyFORTH development system. This section is about "closing the loop"; how the target's defining words are defined during target compilation.

Target defining words may have both a compile-time and a run-time behavior. The basic target defining words have run-time behavior which is code in the nucleus. To see an example of such a word, see the run-time definition of **CREATE** in Block 184, and the compile-time definition in Block 226. The labeled gap which precedes the run-time code (in Block 184) will contain the high-level colon definition of **CREATE**'s compile-time behavior (from Block 226).

The compile-time and run-time behaviors must be physically adjacent in memory because of the method used to transfer the address of the run-time behavior to the code-field-address of the defined word.

Target defining words are an example of one way to handle forward references in Forth. Before the compile-time definition is compiled **'H** is set to the beginning of the associated labeled gap with an **ORG**. The compilation fills in the gap left when the nucleus was assembled. When all forward-reference defining words have been compiled, the compiler's **'H** is reset to where it pointed before the **ORG** of the first forward-referencing defining word.

Forward referencing is not recommended because it creates unnecessary complexity. Although techniques have been developed to eliminate forward references for the nuclear defining words, the techniques are still controversial because they consume slightly more space and time in critical system functions. Only six forward references exist in most polyFORTH systems, they are: **:** (colon), **CREATE**, **CONSTANT**, **USER**, **X**, and **WORD**.

REFERENCES

'H, Section 7.3
Defining Words, Section 2.7

7.9 TARGET COMPILATION OF TASKS

The target compiler and multitasker are individually complex, so that target compilation of a system with multiple tasks is not a simple process.

A task is set up in three steps. First, the task must be defined and its memory allotted. Second, the task must be initialized and linked into the round-robin. Third, the task must be activated with a program to run.

A **FORTH** task occupies two regions of memory. One region contains:

1. A dictionary head,
2. A pointer to the task's **STATUS** cell,
3. An address of the bottom of the task's parameter stack, and
4. In terminal tasks only, a table of device-dependent initial values for user variables.

The above part is unchanging throughout execution, and may reside in read-only memory. The other part, which must reside in RAM, contains:

1. Both stacks,
2. The status cell,
3. The user variable area, and
4. On terminal tasks, the dictionary and input message buffer.

Since this part resides in read/write memory, it must be initialized every time the target system powers up. Task definition and memory allotment must occur during target compilation. Memory is allotted for background tasks by the word **BACKGROUND**, and for **TERMINAL** tasks by the word **TERMINAL**. The target nucleus and host defining words must be compiled before a target task can be defined with **BACKGROUND**. For **TERMINAL** tasks, not only the nucleus and defining words, but also any device-dependent routines must also be loaded before a target teminal task is defined. All useful multitasking functions must also be loaded.

The easiest method is to modify the multitasking definitions in Blocks 30-32 to make the definitions target compilable. Versions of the task defining words that are suitable for target compiling are given in Blocks 253-254.

The word **PROMPT** needs to be changed so that **SYSTEM** is replaced with a greeting or behavior appropriate to the application.

REFERENCES

Application Defining Words, Section 7.5.1
Multitasking, Section 4.0

7.10 CONSERVING MEMORY

It is particularly important to ROM-based applications that they occupy a modest amount of memory. The cost of each ROM—including burning, testing, and general handling—can strongly impact both the production cost and field reliability of high-volume ROM-based products.

Several techniques for conserving memory are covered in this section, notably: pruning, code-sharing, use of **RECOVER**, and head minimization.

Pruning is the process of removing parts of Forth unneeded by an application. The system source code for Forth is grouped into blocks by function, so that related words are together. When an entire facility is unnecessary (*e.g.*, the target system's compiler), the blocks simply are not loaded by the load block of the target-compiled application. Several thousand bytes may be recovered in this fashion. The facilities most commonly deleted are: assembler, compiler, editor and disk support. Non-interactive systems may also prune the interpreter, dictionary management and possibly the terminal drivers. Note that if a system is sold which contains the compiler or assembler, FORTH, Inc. is entitled to royalties (see your Licensing Agreement).

If the memory size is critical (as in a system which will be mass-produced), pruning can be done within blocks. One way of doing this is to copy the source blocks of the nucleus, and then delete all unused Forth primitives from the copies or move their definitions below an **EXIT** to reserve them for future use.

Pruning is the last step of producing a target compilable program—as much debugging as possible should be done first. Pruning usually only requires a few hours, even on large projects.

Code sharing is another powerful conservation technique. Code can be shared in three ways: by subroutines, by code which jumps into another word's code, and by defining words. Forth is a language of subroutines, so vast amounts of code sharing occur naturally. Sharing code by jumping to the common section is routine Forth practice—use **LABEL**s to mark entry points; they are more convenient than using **HERE**. For example, in an 8086-based system, the two following pieces of code will target compile into the same code:

```
CODE EXECUTE ( a)   W POP  LABEL execute
   W DEC   W DEC  W ) LIP

CODE @EXECUTE ( a)   W POP   W ) W MOV
   W 1 MOV  1NZ IF  execute JMP
   THEN  NEXT
```

With **HERE**, the code is somewhat less readable:

```
CODE EXECUTE ( a)   W POP   HERE   W DEC
   W DEC  W ) LIP

CODE @EXECUTE ( a)   W POP   W ) W MOV
   W 1 MOV  1NZ IF   SWAP JMP
   THEN  NEXT
```

If the application has many words which end with a jump to a particular entry point, a custom code ending (a colon definition in the assembler vocabulary containing the words to assemble the jump) may be appropriate.

The use of defining words is a powerful, frequently neglected technique that allows words to share code. In a control application which uses a few standard delays in many places, the following defining word would save memory:

```
FORTH  : DELAY ( n)   | CREATE  ,
   DOES> ( a)   @ MS ;

 1000 DELAY SECOND
60000 DELAY MINUTE
  300 DELAY 300MS
   40 DELAY 40MS
```

Compare the source above with the following:

```
| : SECOND   1000 MS ;
| : MINUTE   60000 MS ;
| : 300MS   300 MS ;
| : 40MS   40 MS ;
```

Not only is the first solution easier to read and use, it also saves 12 bytes, and gives an equivalent result. The savings increase as the number of similar words increases.

To see why this technique saves space, look at the defining word, **DELAY**. The word compiles a target code field address followed by a single-precision delay. Only two cells per delay word exist in the target dictionary. The **|** (pronounced "bar") automatically sets **HDS** temporarily to zero, ensuring that a name is not compiled to the target for each delay word.

Each time a standardized delay is used in a colon definition, it will require only two bytes in the target, instead of the six bytes required for a literal followed by **MS**. In addition, each new delay word only requires four bytes (a **CFA** and a value), instead of the ten bytes (**CFA**, **LITERAL**, value, **MS**, **EXIT**) required by an ordinary colon definition.

Defining words are also sometimes useful for defining standardized output sequences to particular devices. If defining words are used for output, each defining word should define a very particular output style for a particular class of devices, rather than trying to do everything. The word **MSG** is an example of a specialized I/O defining word.

The bar used in the example above may be placed before the colon or **CODE** of any definition which does not need to be accessed interactively at run-time in the final target. If all the definitions of an application can be headless (as in a calculator, for example), the argument into **DICTIONARY** should be zero. **DICTIONARY** sets the default name length and clears the target compilation space.

Some colon definitions are an endless loop (*e.g.*, **QUIT**, and **INTERPRET**). On these, the **EXIT** compiled by **;** is superfluous. The last cell compiled may be "unallotted" by the word **RECOVER**. Words containing **ACTIVATE** may also use **RECOVER**. **RECOVER** may be used in resident applications as well.

REFERENCES

ACTIVATE, Section 4.5
Application Defining Words, Section 7.5
Code Endings, Section 6.2
LABEL, Target Compatibility Block, Section 7.12
MSG, Section 2.7.6.3

7.11 POWER-UP INITIALIZATION

The point of the power-up initialization routine is to force all I/O devices and critical memory areas (*e.g.*, the Forth system variables and block buffers) into a known state before enabling interrupts and starting the application program.

Power-up code is sometimes difficult to debug. Therefore, the most common strategy for initializing a Forth system is to move from machine code to a high-level definition as quickly as possible.

The following steps are most often followed:

1. Disable interrupts (if necessary).
2. Set up the Forth registers (**S**, **I**, **R** and on a multitasking system, **U**).
3. Perform any necessary hardware initialization.
4. Execute **NEXT**.

NEXT executes the "next" address at **I**, so **I** should be set to the address of the parameter field of a high-level power-up routine. The high-level power-up routine is often application-dependent. The high-level routine should be debugged as thoroughly as possible before it is run on the target machine or as a power-up routine.

For an example of a power-up routine, see the initialization/power-up block of your polyFORTH system listing and the *CPU Supplement*.

REFERENCES

Diagnostic and Debugging Techniques, Section 7.13
The Address Interpreter, Section 1.1.6

7.12 RESIDENT TESTING OF TARGET APPLICATIONS

An important principle in Forth is to keep the programming and debugging cycle as interactive as possible. When a target machine and the host machine have the same CPU, it is usually most convenient to do as much debugging as possible in the host. When applications are target compiled, certain words are available which are not normally available in a resident system. Therefore, to allow target compiled applications to be debugged in the resident system, there is a block called "target compatibility block." On most systems this is Block 252. The target compatibility block is loaded at the beginning of the application load block during resident system testing.

The target compatibility block contains definitions for words which are often used by target-compiled applications but which are not normally defined in a resident system. The definitions in the target compatibility block are minimal definitions that will allow almost all target-style source code to be loaded in a resident system, and still emulate as closely as possible the target system behavior.

The principal difference between the resident and target environments is that during target compilation there are three distinct memory areas available, with allocation pointers to each. These are host RAM, target RAM and target ROM. In a resident system, there is only one memory area available for compiling: host RAM. The target compatibility definitions provide a separate region of host RAM analogous to the target RAM area, where the values of **VARIABLE**s, etc. will reside.

The size of this RAM region must be on the stack when the compatibility block is loaded. Thus, to reserve 2K bytes of RAM, the appropriate phrase would be:

```
2048 252 LOAD
```

This memory will be taken from your user area, so you must ensure that your user area is large enough to contain the application plus the RAM buffer.

The target compatibility block provides a very useful level of interactive debugging not usually available on cross compilation systems. It is not difficult to produce source code that works correctly with both the target compatibility block and the target compiler. The following table gives the words usually present in a target compatibility block, along with a description of their functions and the significant differences, if any, from the corresponding target compiler definitions.

Note that these words behave this way only when loaded from the target compatibility block. The target compiler versions are different, but equivalent. You may wish to refer to your system listing of the compatibility block and the target compiler as you read the following table.

Word	Stack	Function
'R	(- a)	Emulates the target compiler's pointer to the next available byte of RAM. Note that a buffer is allotted to emulate the RAM in a system. The size of the buffer must be on the stack when the target compatibility block is loaded.
THERE	(- a)	Pushes the address of the next available byte of simulated target RAM onto the parameter stack. **HERE** is assumed to put the address of the next available byte of ROM on the stack.
ALLOT	(n)	Allots *n* bytes in the simulated RAM. The corresponding word for ROM is **GAP**. **GAP** is rarely used, as variable arrays cannot be defined in ROM, and tables of constants are defined using commas to compile each value.
ORG	(a)	Sets the value of **H** which is the pointer to the next available byte of simulated ROM. Note that one must be careful using absolute addresses for **ORG**, because addresses during resident testing will be different from those in the target compiled system. Absolute addresses may also lead the compiler to over-write the resident system. Alternatives to absolute addresses are to use labels (defined later in this table) or offsets from **HERE**.
TARGET		Emulates the vocabulary change performed by the target compiler's **TARGET**. **TARGET** changes the precedence of the most recently defined word so that the word is immediate and will be executed at compile time.
HOST		A dummy definition that sets **CONTEXT** to be **FORTH**, the vocabulary containing the resident definitions. **HOST** is **IMMEDIATE** so that it will be executed when the compiler sees it in a colon definition instead of compiling its address.
FORTH		Differs from the resident version in that it is **IMMEDIATE**.
VARIABLE name		Creates a constant whose value is the address of a cell in RAM. Used exactly like the resident equivalent; the difference is that the parameter field and the value are segregated in ROM and RAM.
CVARIABLE name		Creates a constant whose value is the address of a byte in RAM. **CVARIABLE** is not available on processors such as the LSI-11 which require alignment on even byte addresses.
LABEL name		Creates a constant whose value is an address in program memory. **LABEL** is often used to provide entry points in assembler routines. In target compiled code **LABEL** adds nothing to the code because **LABEL** creates an entry in the host dictionary rather than in the target dictionary. In the target compatibility block, **LABEL** produces a dictionary

entry whenever it is used—even in the middle of assembly code! To make this transparent, the compatibility version of **LABEL** assembles a jump just before the dictionary entry to the point immediately after it.

n EQU name (n) Creates a constant with a value of *n*. In the target compiler, **EQU** creates a constant in the host dictionary which is only available during target compilation. The target compatibility block version is a normal **CONSTANT**. The target version of **EQU** may be referenced in code or interpretively, but may not be referenced inside a colon definition, because there is no target definition to compile a reference to. This is concealed by the compatibility version, which, being a normal constant, may be referenced anywhere.

Words defined by **EQU** and **LABEL** can not be referenced inside target compiled colon definitions. Words defined by **EQU** and **LABEL** may be used interpretively and in assembler code, but not in colon definitions, because their target compiler equivalents don't create a target dictionary entry.

If you make use of vectored execution, please note that the resident version of **ASSIGN** must be changed in order to be correctly target compiled.

An example of a resident version of **ASSIGN** is:

```
: assign ( a)   R> 2+ SWAP ! ;

: ASSIGN ( a)   BEGIN  COMPILE assign
   COMPILE [ 2- @ , ] ; IMMEDIATE
```

The resident version must circumvent the fact that in a resident system, the word **:** "colon" is headless, by using the phrase in brackets. When target compiling, an equivalent definition of **ASSIGN** is:

```
: assign ( a)   R> 2+ SWAP ! ;

FORTH : ASSIGN ( a)   COMPILE assign
   colon , ; TARGET
```

ASSIGN stores the address of the behavior following **ASSIGN** into the address on the stack. **ASSIGN** must be used in a colon definition.

REFERENCES

Vectored Execution, Section 2.4.8

7.13 DIAGNOSTIC AND DEBUGGING TECHNIQUES

The basic principle behind speedy debugging is to keep the debugging process as interactive as possible, as long as possible. When the target and host have the same CPU, it is easy to test code interactively.

A facility which is sometimes useful is the target compilation log. Every time that an entry in the target dictionary is created, the log types the entry's name and the address of the first byte of the entry's parameter field. The log can be turned on by taking the **LOAD** command for the log out of its parentheses in the target compiler load block (Block 240). The log is not usually used for routine target compilations because the extra I/O slows compilation and clutters the screen. To use the log with printer output it is necessary to define a **TERMINAL** task for the printer with adequate memory to target compile the application. The printer task is made to perform the target compilation.

The target compatibility block is included in the system specifically to allow a target-compilable application to be compiled and run by a resident system. The resident system should have appropriate interfaces to allow thorough program debugging. Once the application works correctly in the resident system, it can be target-compiled and run in the host again. If it is possible to cycle power (or simulate cycling power with a reset button) without memory loss, the power-up initialization routine can be tested directly. Alternatively, the target-compiled application can be moved to the proper (or the least non-interfering) places in the host's read/write memory to simulate the target system. The application can be initialized by jumping in machine code to the first instruction of the power-up initialization routine. Be sure to keep the "move" routine in a place where it will not be copying on top of itself.

If the system refuses to do anything at all when the power-up code is executed, almost certainly there is a problem in the power-up initialization routine (after all, you've tested the other code already). Invent some simple visible signal, for example: toggling a console light or form-feeding a page on the printer. Place this signal code so that it will be executed as early as possible in the power-up sequence. If the signal does not occur, there is something seriously wrong with either the machine, or your understanding of it. Read manuals carefully before calling a technician. If the signal does occur, you can move the signaling routine to verify that each phase of power-up initialization occurs correctly and isolate trouble spots. A good signal for high-level code is to send consecutive numbers to the already-initialized terminal. Insert in your code:

```
... 1 . words 2 . words 3 . ...
```

and your screen will indicate the latest spot successfully reached.

Once you have tested as much of the software on the host as you can, it is time to mate the software to the target hardware. The most difficult situation is a read-only memory application running on a target which has a different type of CPU than the host.

Desk-check the program until no errors are found, then target-compile it with the log turned on and program the read-only memory. Using the log with a hard-copy device will make it possible to determine where to insert patches, or what the logic analyzer means (if you're using one).

Test the software in the target. If the system does not come up, then go through the same steps described above for bringing the application up on the host (moving a signal through the program). Remember that each iteration reduces the number of errors tremendously. Be sure that you find an error or verify program correctness on every iteration.

One of the problems with read-only memory is that it takes time to erase. It helps to have one EPROM chip set under the ultraviolet lamp and another in the computer (*remember:* as interactive as possible). If you routinely have read-only memory applications, you may find it worthwhile to construct a RAM simulator which uses read/write memory but can be programmed by the host.

Alternatively, rather than iterating with EPROMs, you may find it easier to insert a "talker PROM" and compile to the system through a serial link—choose your strategy and stick to it. Serial link compilation is much easier for large systems containing mostly RAM.

When the system comes up and seems to be working, remember to test the application as exhaustively as you did on the host. The small problems that remain will not be found otherwise.

REFERENCES

RELOAD, The Forth Bootstrap, Section 3.8
Serial Link Compilation, Section 7.4.4

8.0 DATA BASE SUPPORT

The polyFORTH Data Base Support option is a set of tools with which you can design efficient data base applications or components for general applications.

The Data Base Support option includes:

- A simple disk-oriented file manager.
- Commands for defining records within files and fields within records.
- Tools for generating columnar reports.
- Utilities for producing totals and subtotals.
- Techniques for linking subfiles to main files and for chaining records within files, and;
- A set of words for creating ordered indexes (for keeping sorted lists).

8.1 OVERVIEW

polyFORTH presents its Data Base Support option in the form of a "kit," leaving complete flexibility for you, the developer, to create a data base design that reflects the natural organization of the data itself.

Before you begin constructing a data base application, you must understand a few simple premises that underlie the design of the Data Base Support option. First, let's review common data base terminology.

A *data base* is the complete set of organized data that is available to the computer. A data base is divided into related groups of data called *files*. For example, a file might contain the names, addresses, and phone numbers of all your clients.

A file, in turn, is divided into *records*. A record might contain the name, address and phone number for a single client. For every client, there would be one record in the file.

A record is itself divided into a collection of *fields*. For instance, one field might be called "STREET."

In a data acquisition environment, a file might contain a set of readings taken during one experiment. Each record could contain the set of measurements taken at a single point in time during the experiment; each field could contain the reading of a different measurement. In this case, you might have numerous files, each containing the data obtained during one run of the experiment; however, the records in each file would be laid out identically.

Many applications require multiple types of files that relate one to another. Suppose you want to record all your invoices, using an "Accounts Receivable" program. In the course of your business, you bill several invoices to the same client. Rather than duplicate the name and address of the clients every time you bill them, it makes sense to have one file for the client data, and another file for invoices. Each invoice record can point to one of the clients in the client file. In this way, one file can "use" another file.

8.1.1 Contiguous Files and Performance

A premise of the Data Base Support option is that you are a knowledgeable programmer concerned about performance. Its approach allows you to design the data base for optimum efficiency.

In contrast, typical data base packages are intended to simplify data base construction for non-programmers. These packages do not require that you think about how your data is organized. On the down side, you lose the ability to structure your data base in the optimal way. The price for greater generality is impaired performance and increased size of compiled code.

In a disk-oriented data base application, the key determinants of performance are:

1. How many physical disk accesses are required to access a logical data item?

2. How much head motion occurs during normal operation?

If you can minimize the number of physical accesses required and the disk head motion, you can maximize performance.

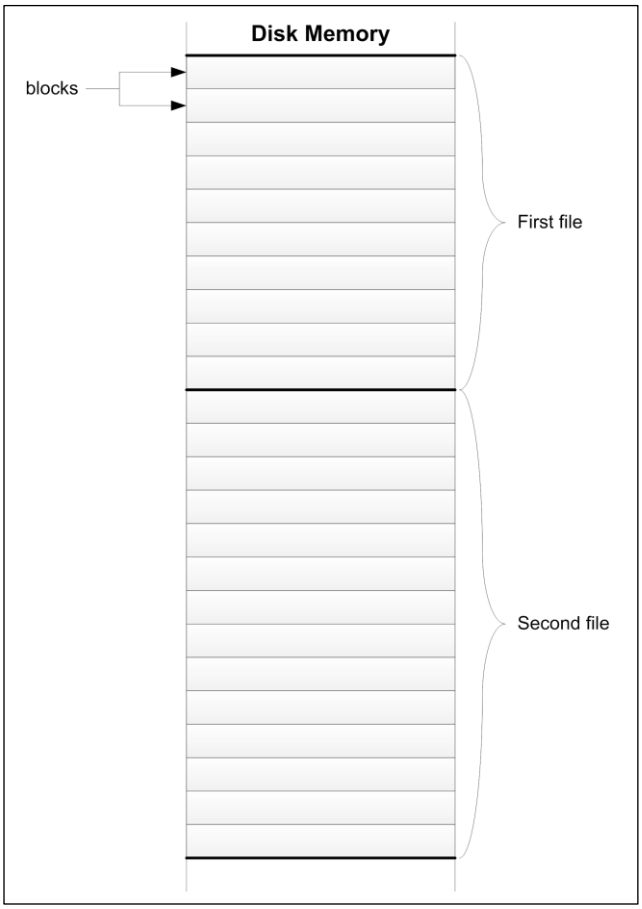


Fig. 8.1

In polyFORTH, a file occupies a contiguous range of blocks on disk. A file may be any size (using whole blocks) up to the capacity of the disk. There is no need for a file allocation table.

The polyFORTH Data Base Support option addresses both issues simultaneously by imposing a single restriction: a file is a contiguous region of blocks on disk.* This means the system does not automatically "manage" files by interweaving them on disk as they expand and contract. Files are not fragmented across the disk, and there is no need for a file allocation table to point to the fragments.

Instead, you specify the maximum size of each file when you create the file, and assign it a contiguous range of blocks on the disk. Although this requires some thought, there are several advantages:

1. Since files, and therefore records, are contiguous, the exact location of any data element can be calculated. Thus any data element can be accessed in a single physical disk access. In traditional operating systems and file managers, an application cannot know from a record number where that record lies physically. The location must either be looked up in a directory, or found by following a set of chains. Reportedly, one popular operating system requires up to six disk reads to access a single logical record.

2. While accessing various elements within a single file, the disk's magnetic head need only travel within the distance occupied by the file. Head motion is minimized.

3. You have control over the arrangement of multiple files in relation to one another. For instance, if one file serves as an index into another file, you can place these two files adjacent to each other on the disk. Again, head motion is minimized.

These benefits assume that you are running on a native polyFORTH system. If your polyFORTH runs under another operating system, performance depends on the way that O/S treats the disk.

Just as files are contiguous and of fixed length, so too are records and fields. Again, although variable-length fields require less thinking on the part of the user, they necessarily degrade performance. Fixed-length fields do not necessarily imply fixed-length amounts of data, because a variable number of subordinate records may be chained together as necessary. (We'll explore this technique further in Section 8.7.)

Since the primary bottleneck in disk-based file systems is disk-access time, minimizing this bottleneck achieves surprising efficiency. For example, one company sells a data base system which uses the polyFORTH Data Base Support option to handle 300 Mbytes of data and support 64 simultaneous users with under one-second response time even at peak load, on a single 68000 microprocessor.

REFERENCES

Disk Block I/O, Section 1.2.2, 3.2

8.1.2 Current Files and Records

Another concept that is fundamental to the Data Base Support option is the notion that at any given moment, exactly one file is current and one record is current. Let's first describe what is meant by a file being current.

We mentioned that a file is simply a contiguous, fixed-length range of blocks on the disk. There is no file allocation table on the disk, nor is there any other indicator of which blocks constitute which file. The knowledge of where each file begins and ends resides within the application code, specifically in a small table that you define for each file (using the defining word **`FILE`**, Section 8.3). This table is called a *File Definition Area* (FDA).

* This means they're physically contiguous on a disk supported by a native polyFORTH. Versions of polyFORTH that are co-resident with another OS use its files to contain blocks, and thus rely on the host OS to manage disk.

The name you give this table is the name of the file itself. The table contains the starting block number, along with sufficient information about the number and size of records for the Data Base Support option to be able to calculate the absolute location of any record in that file.

When you invoke the name of a file, the file definition places the address of its parameter field in a user variable called **F#**. All record-accessing operations in the Data Base Support option use this pointer to indicate the current FDA, which in turn points to the blocks where the desired record resides.

Thus, at any given moment, one and only one file is current. Changing files is a simple matter of invoking the file name, which places a new address in **F#**, taking only microseconds.

Contrast this with the process of "opening" and "closing" files in traditional operating systems. In these systems, each open and close operation requires noticeable disk activity to read in the file directory and write it out again. For this reason, the question of how many files can be open simultaneously is a concern in such systems. This concern disappears with polyFORTH's Data Base Support option.

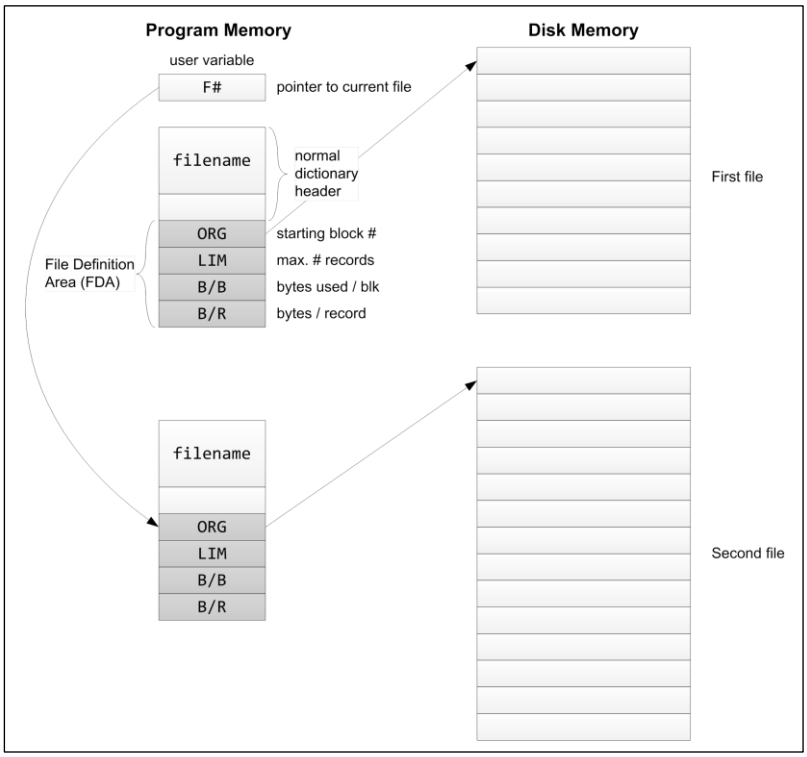


Fig. 8.2

For each file on disk there is a file definition in the dictionary. This definition contains four parameters describing the location and dimensions of the corresponding file. In this figure, the second file is "current."

We can summarize the above discussion by saying, "Files are pointed to, not opened." Analogously, we can say that "Records are pointed to, not read."

Just as there is always a current file, so there is also a current record, the number of the current record found in the user variable **R#**. All the data-access operators refer to specified fields within the *current record* in the *current file*.

The polyFORTH Data Base Support option is once again unique in this con-cept. Many data base packages actually read in an entire record from the disk, then allow access to the fields within it. polyFORTH merely makes a record current; disk access only occurs when a field name is invoked in combination with a field access operator.

This design takes advantage of the behavior of **BLOCK** (Section 1.2.2). Whenever a single field is accessed, **BLOCK** reads the entire block in which that field resides. If multiple fields in the record are required in the same operation (such as displaying all fields in the record for a report), it is unlikely that the block buffer will be reused before all the fields can be displayed. (Should this happen, **BLOCK** will automatically read the block again.) Moreover, it is even likely that references to neighboring records in the same block will also not require physical disk accesses.

An important advantage to not reading the record physically is the certainty that at any given moment only one copy of each record exists. Systems that read a record into memory face the problem of two users accessing the same record, and having different copies of it. Solving this conflict entails various "lockout" schemes, all of which complicate the system and reduce performance.

The file and record pointers are entirely independent of each other. Not only can you select records without re-selecting the file, you can also change files without affecting **R#**.

From time to time in your application you may want to leave your current file and record temporarily (perhaps to examine or display a field from a related file) and return. The following words enable you to "remember" **F#** and **R#** temporarily:

Word	Action
SAVE	pushes **R#** and **F#** onto the return stack.
RESTORE	pops those items off the return stack and places them in **R#** and **F#**.

Naturally, you must use **SAVE** and **RESTORE** as a paired set within the same definition, just as you would use **>R** and **R>**. Similarly, you must use both words within or outside of any **DO ... LOOP** structure in that definition.

Following a **SAVE**, **R#** is on top of the return stack; if you need a copy of it you may get it by using **R@**.

REFERENCES

Return Stack, Section 2.1.3

8.1.3 How Data is Stored

The Data Base Support option allows storage of data in either numeric or alphanumeric form. For instance, a U.S. telephone number, including area code, requires 14 bytes when stored in alphanumeric form:

```
(213) 372-8493
```

This same phone number can be stored in only 6 bytes per record, if it is recorded as a 16-bit area code and a 32-bit local number:

```
213  3728493
```

The appropriate punctuation symbols can easily be inserted when the number is displayed, using pictured numeric output.

The contents of numeric fields travel between the data stack and the disk; the contents of alphanumeric strings travel between the **PAD** and the disk.

For instance, if we have a double-length field named **SALARY**, we can fetch the value of the field (from the current record in the current file) by invoking the phrase:

```
SALARY D@
```

which places its value on the stack in the same way that the word **2@** fetches a double-length value from an ordinary variable. Similarly, the phrase,

```
SALARY D!
```

removes a double-length value from the stack and places it in the current **SALARY** field.

Alternatively, the word **B@** fetches the contents of an alphanumeric field, and copies it to the **PAD**. The word **B!** stores an alphanumeric string at **PAD** into a given field.

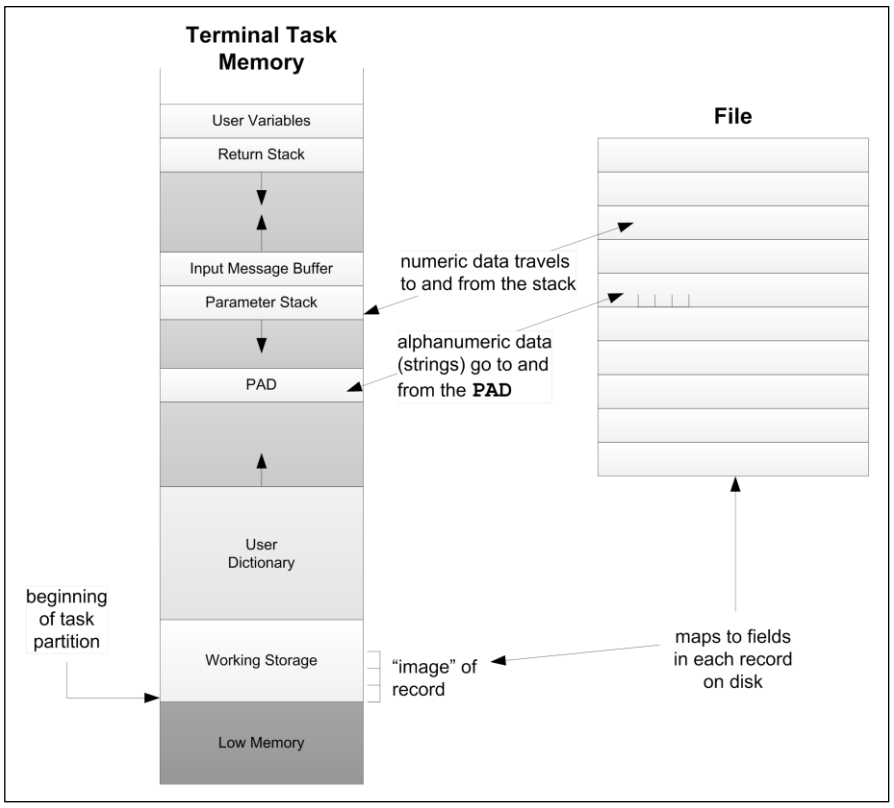


Fig. 8.3

Shows how data travels between disk and memory. Numeric data travels between disk and the parameter stack, and text strings travel between the disk and **PAD**.

REFERENCES

Field Reference Operators, Section 8.5.3
Pictured Numeric Output, Section 2.5.2.1
String Storage in **PAD**, Section 2.3.1

8.1.4 Working Storage

Two features of the Data Base Support option make use of a region of memory called *working storage*. Working storage is allocated at the beginning of a task partition, and serves as a place where record data may remain which is easily accessible, but less volatile than **PAD** or the parameter stack. Since each task has a working storage area, tasks running concurrently may use the same code referring to working storage without conflict.

One use of working storage appears in automatic totaling (Section 8.8.8). Here working storage holds the accumulating registers for each column of data to be added as the report is generated.

The second use of working storage is as an "image" of a record. The same relative positions are maintained both in the record on disk and in working storage. For example, working storage is used to hold the key during a binary search of an ordered index (Section 8.6), in the field in which it will be found in records being searched.

The same field names that let you access fields on disk also may be used to reference the corresponding fields in local working storage. There is only one "record" in the working storage area. This technique lets you map data items as though they were contained in records although they are temporarily in resident memory instead of on the disk.

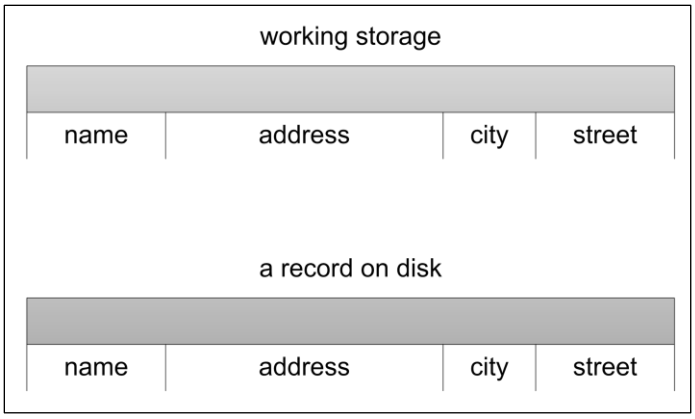


Fig. 8.4

Field names may be used to reference either the individual field in the current record (on disk), or the corresponding field in working storage.

The double use of working storage for both subtotaling and key searches rarely causes a conflict, since the two activities occur at separate times. To be on the safe side, however, the statistics component uses the word **REGISTER** to return the address of the accumulators, which in turn is defined in terms of **WORKING**. If you find that you will encounter a conflict, you may resolve it by simply redefining **REGISTERS** to point to some other area.

How much working storage is necessary? If you are using the subtotaling feature, the amount depends on the number of accumulating registers you need. In total, you will need the sum of:

16	register area management
4	header variables for registers
#registers * 8	8 bytes per accumulator
_____	
total	

For instance, three accumulators will require 44 bytes* (16 + 4 + 3*8).

If you are using ordered indexes, working storage must be as large as the largest record in any ordered index file.

Remember that any task which performs an application that uses working storage must have sufficient room allotted for it—including the printer task. Section 8.1.6 describes methods for allocating working storage.

REFERENCES

Accessing Fields in Working Storage, Section 8.5.5
Ordered Indexes, Section 8.6
Subtotaling, Section 8.8.8

8.1.5 Installing The Data Base Support Option

Before you begin to use the Data Base Support option, you must first decide whether you will be using it in your personal task only, or whether other terminal tasks may need to use it simultaneously.

To load the option into a your private terminal task, list its load block with the phrase,

```
FILES LIST
```

The number of this load block is returned by the constant **FILES**. Make sure that the block begins with the phrase:

```
EMPTY   n ALLOT
```

where n is the amount of working storage required for your files application (see Section 8.1.4).

At the end of the load block, a null definition of **TASK** should appear. This word will be the last word in the dictionary when file applications are loaded, and will mark the point at which overlays will occur.

Now issue the command:

```
FILES LOAD
```

to load the Data Base Support option.

As you create data base applications, each of these should begin with the phrase:

```
FORGET TASK   : TASK ;
```

This makes each application an overlay, which will discard other overlays that use this convention. For instance, if you have an accounts-receivables application, its load block should begin with the above phrase, to forget any other applications without forgetting the Data Base Support option itself.

Finally, if you wish to output a report to your printer, you must allot a working storage area in the printer task. This may be done with the phrase:

```
n TYPIST H HIS +!
```

* On 32-bit machines these sizes should be doubled.

This phrase advances the dictionary pointer **H** for **TYPIST**—the printer task—by the amount *n*. It is most convenient to edit this phrase into the **FILES** load block.

Alternatively, if several terminals require use of the Data Base Support option, the package should be loaded with the system electives by Block 9. In this case, remove the:

```
EMPTY    n ALLOT
```

at the top of the **FILES** load block; also remove the definition of **TASK** and the word **EXIT** at the bottom (by placing parentheses around them). By allowing the final word **GILD** to execute, the Data Base Support definitions will become available to all tasks.

Edit the phrase:

```
FILES LOAD    n TYPIST HIS +!
```

where *n* is the amount of working storage required for your files application (see Section 8.1.4) into the last line of Block 9 (just above the **EXIT**). If you have already loaded the electives before making this addition, type:

```
FLUSH RELOAD
```

then type:

```
HI
```

Using this approach, each files application must begin with the phrase:

```
EMPTY    n ALLOT
```

instead of **FORGET TASK**, where *n* is the amount of working storage needed by that application.

REFERENCES

ALLOT, Section 2.8.1
EMPTY, Section 3.3.4.1
FORGET, Section 3.3.4.2
HIS (User variables), Section 4.6
LIST, Section 5.1.1
Overlays, Section 3.3.4
Parentheses Used for Comments, Section 1.5.1
System Electives, Section 1.4.2

8.2 CREATING A SIMPLE FILE

This section introduces the procedures for creating a simple file by way of an example, and provides a contextual framework for the detailed sections that follow.

In this section we are assuming that we already know how to structure our data; we are concerned here only with the mechanical aspects of file creation and field layout. For a more general discussion of data base design, see Section 8.9.

Our simple example will be a file of names and addresses. To avoid extra detail, we will only use alphanumeric fields. No attempt will be made at keeping a sorted file (ordered indexes are discussed in Section 8.6).

Step 1

Define the fields:

```
0  20 BYTES NAME    20 BYTES STREET
   14 BYTES CITY     2 BYTES STATE
    6 BYTES ZIP     14 BYTES PHONE   DROP
```

In the above lines we have defined six Forth words which will reference the individual fields in each record. The initial zero is the relative position within the record. The defining word **BYTES** creates an alphanumeric field of the specified width (the width must be an even number on cell-aligned processors). The final **DROP** is necessary to discard the final relative position within the record (see Section 8.5.1).

In addition to **BYTES**, several other defining words are available for creating different types of fields (see Section 8.5.2).

Step 2

Determine how many records and blocks the file will need. Two words that are not generally resident are available in the "file initialization block."* **#R** computes the number of records of given size that would fit in a given number of blocks; **#B** computes the reverse: the number of blocks needed to hold a given number of records of given size (see Section 8.3.1).

Step 3

Define the file:

```
74 500 39 400 FILE PEOPLE
```

This statement defines a file called **PEOPLE** which contains records that are each 74 bytes in length. The file will hold a maximum of 500 records, consuming a range of 39 contiguous blocks. The starting block will be 400 (see Section 8.3.1).

Invoking the filename **PEOPLE** makes this file current.

Step 4

Initialize the file. Load the file initialization block (if it's not already loaded) and execute the phrase:

```
PEOPLE INITIALIZE
```

to fill all blocks in the file with zeroes (see Section 8.3.3).

Step 5

Enter data.

Here is the definition of a word that will allow data entry for a single record (person):

* This block is not generally resident, because it is used only in the initial creation of the data base. It may usually be found at **FILES 5 +**.

```
: enter   PEOPLE  SLOT READ
     CR ." Name? "      NAME ASK
     CR ." Address? "   STREET ASK
     CR ." City? "      CITY ASK
     CR ." State? "     STATE ASK
     CR ." Zip? "       ZIP ASK
     CR ." Phone? "     PHONE ASK ;
```

By invoking **PEOPLE**, we select the **PEOPLE** file as the current file.

The word **SLOT** allocates a new record in the current file, and leaves its number on the stack (see Section 8.4.3). The word **READ** sets the current record according to the number on the stack (see Section 8.4.1).

Next, the definition prompts the user to enter the "name" field. The word **ASK** is like **EXPECT**, except that it places the expected text in the given field. The same process is followed for the remaining five fields.

Step 6

Display the data.

We define the following word to display the current record:

```
: person   NAME B?  CR STREET B?  CR
   CITY B?  STATE B?  ZIP B?  PHONE B? ;
```

The word **B?** displays the contents of the given **BYTES** field (see Section 8.5.3).

To display the contents of all records that have been entered, we define:

```
: everyone   PEOPLE RECORDS DO CR
   I READ  person  LOOP ;
```

By invoking **PEOPLE**, makes the **PEOPLE** file current. The word **RECORDS** returns the appropriate arguments for a **DO ... LOOP**, including all records that have been allocated by **SLOT** in the current file (see Section 8.4.3).

Within the **DO** loop, **READ** makes each record current in turn, and **person** displays the information for that record.

Here is a sample of the output of **everyone**:

```
Andrews, Carl
1432 Morriston Ave.
Parkerville PA 17214 (717) 555-9853

Boehning, Greg
POB 41256
Santa Cruz CA 95061 (408) 666-7891

Chapel, Doug
75 Fleetwood Dr.
Rockville MD 20852 (301) 777-1259
```

```
Cook, Dottie
154 Sweet Rd.
Grand Prairie TX 75050 (214) 642-0011
```

To produce columnar output, we would use the "Report Generator" (Section 8.8).

For deleting records, we would use the word **SCRATCH** (see Section 8.4.3).

8.3 FILE DEFINITION AND ACCESS

A polyFORTH file is a contiguous region of Forth blocks. On native versions of polyFORTH this means that the file will be physically contiguous, and that you can arrange for files that are accessed together to be physically near one another. This can significantly speed up a data base application.

Versions of polyFORTH that are co-resident with a "host" operating system (such as MS-DOS or RSX) are identical from the point of view of the programmer, but since allocation of physical disk space is performed by the host operating system you haven't the actual level of control you do on the native versions.

This section discusses how files are defined and referenced on all polyFORTH systems.

8.3.1 The FILE Definition

The word **FILE** is used to define files, given the attributes of the file. The format is:

```
length limit blocks origin FILE name
```

where:

Word	Description
length	is the length of each record in bytes (maximum 1024);
limit	is the maximum number of records (on 16-bit processors the limit is 32767 records per file);
blocks	is the number of 1024-bytes blocks occupied by the entire file;
origin	is the first block number, expressed as a double-precision number on all polyFORTH ISD-4 systems (remember the decimal point!); and
name	is the user-assigned name of the file.

The defining word **FILE** creates a new name (dictionary entry) that, when invoked, will make this file current. The dictionary entry contains the **F**ile **D**efinition **A**rea (FDA) for the file being defined.

We recommend that you define all your files in a single block, making it easy to see which ranges of blocks have been allocated for other files. If a disk will contain source or other data along with files, it's a good idea to indicate these other uses in comments on the same block.

Here is an example of good file definition layout in a block:

```
( BytesRecordsBlocksOrigin          Name)
     26      801       22     500.  FILE   (GLOSSARY)
    340      800      268     522.  FILE   GLOSSARY
      4       10      244     795.  FILE   HITS
```

```
24       42        1     799.  FILE   SECURITY
38     2600      100     800.  FILE   TESTS
```

Note that the number of blocks may be computed from the number of bytes/ record and number of records. In fact, the blocks argument is not actually used by the Data Base Support words; it appears in this list to help you maintain the layout of your disk space.

Generally you will choose an appropriate maximum number of records, based on a reasonable estimate of the needs of the application and allowing for expansion. You will also have worked out the approximate size of each record based on the width and type of fields needed. Then derive the number of blocks from the number of records and size of each record. The word **#B** in the file initialization block is a helpful tool for computing the number of blocks. After loading this block, type:

```
#records #bytes/record #B .
```

For instance, if your application requires 2000 records, and each record is 42 bytes wide, type:

```
2000 42 #B .      84 ok
```

Alternatively you can compute the number of records based on the number of blocks. The word **#R** in the file initialization block does the arithmetic. Type:*

```
#blocks #bytes/record #R .
```

For example:

```
84 42 #R .      2016 ok
```

This shows that you can actually fit an extra sixteen records in the same number of blocks.

By using these tools, you can iterate on various sizes until you get the optimal combination. Sometimes you can increase the size of a record without increasing overall file size. For instance, if your record width is 94 bytes, it takes 200 blocks to store the same number of records; however 200 blocks will store 2000 records even when each record is 102 bytes wide:

```
2000 94 #B .      200
2000 102 #B .     200
```

It's a good idea to leave extra space in records, in case you need to add fields later. Beware, however, of grossly oversizing either your record width or file length, as both of these will increase head motion. Strive for generous but reasonable estimates.

8.3.2 File Definition Area and Access

The word **FILE** establishes a File Definition Area (FDA) for each file in the system. The user variable **F#** always points to the current FDA. Execution of the filename sets **F#** to address the associated FDA.

Each file's FDA contains four values to specify the file. Each of these values may be accessed by the following names, each of which returns the address of the associated value in the current FDA.

* Responses shown in light type.

Name	Description
ORG	Starting disk-block number of the first disk block allocated to the file (Forth logical block number as a double-precision number).
LIM	Number of records, of declared record length, that the file can contain.
B/B	Number of bytes used per block.
B/R	Number of bytes per record.

While these words are used by the Data Base Support option, they are rarely referenced directly in applications. **ORG** and **LIM** can be useful in debugging, however. For instance, the phrase:

```
ORG 2@ D.
```

indicates which file is current; in case of an abort, you can tell which file you were in at the time.

8.3.3 File Initialization Utility

A file that has just been created must be initialized before it can be used. A special utility is available for this purpose. It may be loaded with the phrase:

```
FILES 5 + LOAD
```

This block* is not an overlay; you may load it any time after you load the block in which the files to be initialized are defined. Since initialization is done rarely (only during application development), we recommend that you do not load this block routinely.

To initialize a file, once the initialization block is loaded, type:

```
filename INITIALIZE
```

The word **INITIALIZE** performs the following functions:

1. Writes binary zeros throughout the entire file (including **AVAILABLE**).

2. Writes a -1 in the second two bytes of Record 1 of the file. This serves as a "stopper" for the binary search in an index file. In other kinds of files this has no effect.

Two other words defined in this block—**#R** and **#B**—are useful when designing file layouts.

8.3.4 Shared Files

In polyFORTH files may be either shared or unshared. Shared files are those that are defined in the common dictionary available to all users (loaded by the electives load block). If a file is defined in an overlay, it will be available only to the task or tasks in whose partition it is defined.

* The specific block number may vary on different versions of polyFORTH. It is usually the fifth block after the **FILES** load block.

As we discussed in Section 8.1.3, all users may freely access the file without having to worry about simultaneous access problems, as long as standard polyFORTH accessing methods are used. This is because **BLOCK** ensures that there will be only one copy of a record at a time, and each task does not have its own private copy.

Certain situations require an extra measure of control. For example, one terminal might delete a record that is needed for processing at another terminal at a later point. In such as case, you may use a "status" byte in the record to control access.

REFERENCES

BLOCK, Section 3.2
Installing the Data Base Support Option, Section 8.1.5

8.4 RECORD MANAGEMENT

The process of record management includes selecting records, finding the next free record when a new record is needed, and marking deleted records as available for future use.

Not all applications require special record allocation techniques. For instance, if a file contains 100 records and each record contains information on a permanent piece of equipment which is identified by a two-digit number, there is no need to allocate or deallocate records. You may just use the equipment number as the record number. This is called "direct access."

In an application in which the number of active records changes dynamically, it may be appropriate to use the record allocation techniques described here.

8.4.1 Record Selection

Field reference operators (Section 8.5.3) access fields in the current record. The word **READ** makes a record current.

READ (n --) Makes record n current, having verified that *n* is a valid record within the current file.

The name **READ** is slightly misleading, in that it doesn't perform an actual disk operation, but merely sets a pointer to the current record. **READ** checks that n is not less than zero and not greater than the value of **LIM**. If *n* fails this range test, **READ** aborts.

READ stores the number of the current record in the user variable **R#**.

REFERENCES

Current Files and Records, Section 8.1.2

8.4.2 Available Records

To distinguish allocated records from available records, the Data Base Support option uses the convention that if the first two bytes in a record contain binary zeroes, the record is available for use. When a file is initialized, all records are filled with zeroes. Thereafter, active records may keep any non-zero data in the first two bytes; when a record is released, zero is stored in this area.

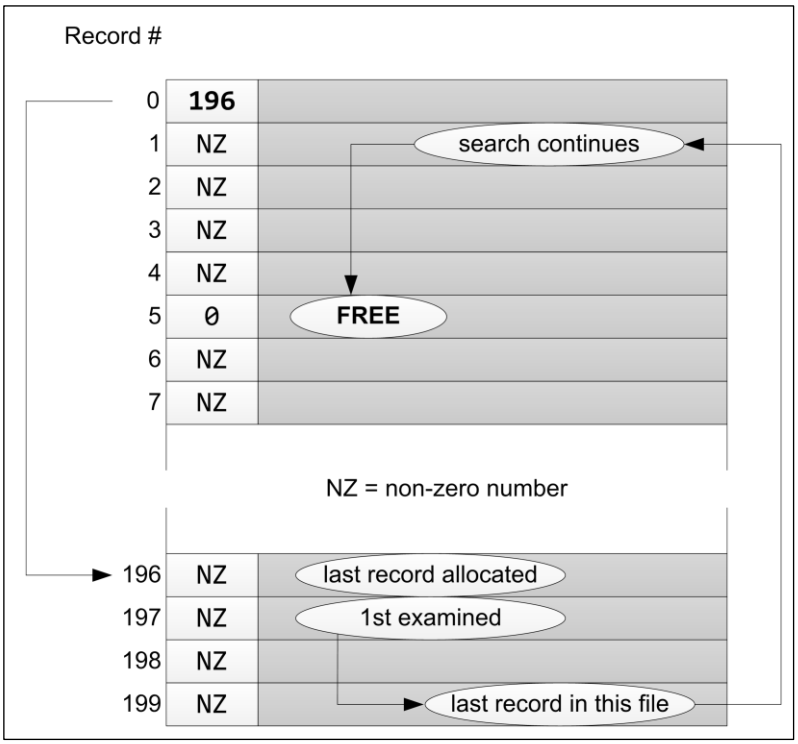


Fig. 8.5

The search for an available record performed by **SLOT** "wraps around" if necessary at the end of the file.

Record 0 of each file contains, in its first two bytes, the record number of the most recently allocated record in that file. The word **AVAILABLE** returns the address of this pointer. When the file is initialized, **AVAILABLE** is zero.

To allocate a new record, the system begins with the record immediately following the "available" record and searches forward for the first free record.

If the search should reach the end of the file without finding a free record, it "wraps" around to the beginning again, so that deleted records will be used. For instance, in Fig. 8.5, **AVAILABLE** points to Record 196; however, there are no more free records between there and the end of the file. But Record 5 is free. By "wrapping around" to the beginning of the file, the search finds the available record.

8.4.3 Record Allocation/Deallocation Operators

Only two words are required for allocating and deallocating records:

Word	Stack	Description
SLOT	(-- n)	Allocates a new record in the current file and returns the number of the allocated record.
SCRATCH	(n)	Deallocates record *n* from the current file, making it available.

SLOT searches the file for the first free record, starting with the record following the one pointed to by **AVAILABLE**. If a free record is found, **SLOT** sets the file's **AVAILABLE** to point to it. **SLOT** then stores a -1 into the first cell of the record to indicate that it is no longer free, and clears the remainder of the record to zeros. If the file is full, an error message occurs and processing is terminated.

SLOT does not make the new record current, it only returns the selected record number on the stack. The reason for this factoring is that you often want to do something with the record number before consuming it with **READ** (which makes this new record the current record). For example, we may wish to cause a link in the current record to point to the new record, as in the phrase

```
SLOT  DUP LINK N!  READ
```

Here the phrase **LINK N!** must come first because after the **READ** we'll be in a different record.

SCRATCH does not change the contents of the record beyond the first two bytes.

REFERENCES

N!, Section 8.5.3
READ, Section 8.4.1

8.4.4 Accessing Files Sequentially

The following words return appropriate stack arguments for a loop which will access the records in a file sequentially:

Word	Stack	Description
RECORDS	(available+1 1)	Typically used before **DO**, returns the content of **AVAILABLE** (the record number of the last record allocated) incremented by one and starting index (1) for a file that has never wrapped around.
WHOLE	(limit 1)	Typically used before **DO**, returns the content of **LIM** and starting index (1) for the entire file.

Since these words return the parameters for the *current file*, it's a good habit to invoke the name of the file just before them, as in **PEOPLE RECORDS**.

When using **WHOLE**, you will probably want to check inside the loop whether each record is currently active. This is normally done by the phrase:

```
LINK N@ IF ...
```

where **LINK** is the generic numeric field comprising the first two bytes of each record. If these bytes contain zero, the record is available for use.

REFERENCES

Available Records, Section 8.4.2
DO Loops, Section 2.4.4
LIM, Section 8.3.2

8.5 FIELD DEFINITION AND ACCESS

A record description is the list of defined fields that appear in the record. Each field is an entry in the Forth dictionary, containing the displacement of the field from the beginning of the record in its parameter field.

A record description is not formally associated with any particular file. It is more like a mask, which is used whenever it is appropriate to access data.

There are several kinds of fields: numbers of various sizes and byte strings of specified length. The following sections discuss the various types of fields and the related operators that are used to access the data stored in them.

8.5.1 Record Description

A record description defines the fields that make up each record in a file. A record description has the following general format:

```
0    field-type field-name
     field-type field-name
     ...
DROP
```

The various field types are described in Section 8.5.2.

A value is carried on the stack throughout the above process to give the relative displacement of the beginning of a record. This value is initialized by the zero at the beginning of the record description, incremented appropriately by each field definition, and finally discarded at the end.

In a complex application the fields in a record description may be defined vertically, like this:

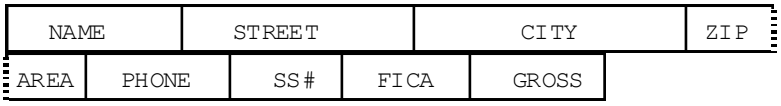
```
 0 ( PEOPLE file records)
 1
 2 0  24 BYTES NAME        ( Last name first)
 3    24 BYTES STREET      ( Street address)
 4    10 BYTES CITY,STATE
 5    DOUBLE ZIP           ( zip code, US only)
 6    AREA                 ( area code)
 7    DOUBLE PHONE         ( phone number )
 8    NUMERIC >DETAIL      ( link to DETAIL file)
 9                ( For employees:)
10    DOUBLE SS#           ( Social sec. number)
11    DOUBLE FICA
12    DOUBLE GROSS         ( Gross income ytd)
13 DROP
14
15
```

As a quick check to verify that the number of bytes used for each record matches the expected value (as specified in the file definition), replace **DROP** with **.** (“dot”). This format allows you to use the shadow block for a general discussion of the file and record.

The field-names defined in the example above—**NAME**, **STREET**, etc.—are now entries in the Forth dictionary. When executed, these words return an address, except for field-names defined with **BYTES**, which return a count and address (see Section 8.5.2).

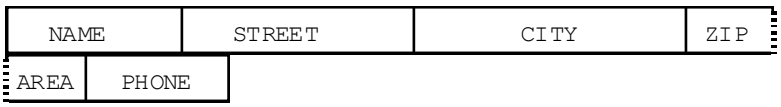
A record description is not formally attached to a particular file and has no name. Use of a field name references the relative location given by that field name in the current record of the current file.

A record in **EMPLOYEES**:



Thus it is possible to use the same field names for two different files, even if the record size used in each file varies. For instance, all the above-defined field names could be used with a file called **EMPLOYEES**, while the first six could be used with another file called **CUSTOMERS**.

A record in **CUSTOMERS**:



NOTE: Due to the Data Base Support option's record-allocation scheme, the first field of an active record may never contain a zero in its first two bytes. In our example, this is not a problem because the first field is alphanumeric (even blanks are stored as decimal 32's). Otherwise, we would have to rearrange the order of the fields so that one which will never contain zero is first.

REFERENCES

Available Records, Section 8.4.2
Field Types, Section 8.5.2

8.5.2 Field Definitions

The following field types are defined:

Word	Description
1BYTE	This field is for an 8-bit value (range 0-255). On processors that do not tolerate odd byte addresses (such as the PDP-11 and 68000), **1BYTE** fields must be used in pairs to avoid wasting space. Example: **1BYTE AGE** Words that are defined by **1BYTE** return an address, suitable for use with the one-byte memory access operators **1@**, **1!**, and **1?**.
NUMERIC	**NUMERIC** fields occupy two bytes of storage (on 32-bit systems also). On cell-aligned processors, **NUMERIC** fields are automatically aligned on even-byte boundaries. Example: **NUMERIC WEIGHT** Words defined by **NUMERIC** return an address, suitable for use with the numeric field access operators **N@**, **N!**, **N?**, and **N?**.

Word	Description
DOUBLE	This field is for a 32-bit (4-byte) value. Example: **DOUBLE SALARY**

Words defined by **DOUBLE** return an address, suitable for use with the double field access operators **D@**, **D!**, and **D?.**

BYTES This field is for alphanumeric text. A count is required to specify the number of bytes in the field.

Example: **24 BYTES NAME**

Words defined by **BYTES** return a length and address, suitable for use with the byte field access operators **B@**, **B!**, **B?**, and **?B**. The width of a **BYTES** field must be even.

FILLER This field reserves space in the record, typically used for future expansion or to skip regions of a record that are to be accessed by other means. **FILLER** requires the number of bytes to be reserved.

Example: **6 FILLER**

FILLER creates no dictionary entry.

At compile time, the numeric field defining words (**1BYTE**, **NUMERIC**, **DOUBLE**) expect the current displacement in the record on the stack. A copy of the displacement is compiled in the parameter field of the definition, and its value on the stack is incremented by the size of the field in bytes. A **BYTES** field also expects the size of the field on the stack. This value is compiled along with the displacement, and used to increment the displacement accordingly.

When a field-name defined by one of these words is *executed*, it pushes onto the stack the address of working storage, incremented by the displacement of the field to give the address of the field in the record image in working storage. In the case of **BYTES** fields, the size of the field is beneath the address on the stack. The working storage address (and size, in the case of **BYTES** fields) is the appropriate input to the field access operators described in the next section.

REFERENCES

Access to the Record Image in Working Storage, Section 8.5.5
Available Records, Section 8.4.2
Direct Access to Fields, Section 8.5.4
Field Reference Operators, Section 8.5.3
Working Storage, Section 8.1.4

8.5.3 Field Reference Operators

Fields in files are referenced with special words. The following operators assume that the desired file and record have been selected. They refer to fields in the current record (as indicated by the value of user variable **R#**; see Section 8.4.1) In all cases, the name of the field precedes the operator; the field-name returns the appropriate address (and length, in the case of **BYTES** fields) to be used by the access operator.

Word	Stack	Action
1@	(a -c)	Fetches the contents of a **1BYTE** field to the top of the stack.
1!	(c a -)	Stores a byte into the **1BYTE** field whose address is on top of the stack.
1?	(a)	Fetches and displays the contents of a **1BYTE** field.
?1	(a)	As for 1?, except the results are right-justified by the report generator.

N@	(a - n)	Fetches the contents of a **NUMERIC** field to the top of the stack.
N!	(n a -)	Stores a number into the **NUMERIC** field whose address is on top of the stack.
N?	(a)	Fetches and displays the contents of a **NUMERIC** field.
?N	(a)	As for **N?**, except the results are right-justified by the report generator.
D@	(a - d)	Fetches the contents of a **DOUBLE** field to the top of the stack (two cells).

Word	**Stack**	**Action**
D!	(d a -)	Stores two cells into the **DOUBLE** field whose address is on top of the stack.
D?	(a)	Fetches and displays the contents of a **DOUBLE** field.
B@	(n a)	Reads a **BYTES** field, according to the declared length, into **PAD**.
B!	(n a)	Stores a **BYTES** field, according to the declared length, from **PAD**.
B?	(n a)	Fetches and displays the contents of a **BYTES** field, according to the declared length. **PAD** is used as intermediate storage of the field data.
?B	(n a)	As for **B?**, except the results are right-justified by the report generator.

Example of usage:

GROSS D@	Fetches the contents of the **DOUBLE** field **GROSS** onto the stack.

Two other words are included for storing data into **BYTES** fields:

Word	**Stack**	**Action**
PUT	(n a)	Copies the remainder of the input stream into a **BYTES** field. For example:

```
NAME PUT Fred Ferguson ok
```

A string that is too long will be truncated when it is stored. If it is shorter than the field size, it will be blank-filled. A copy of the entire string is left in **PAD**.

ASK	(n a)	Awaits (via **EXPECT**) input from the keyboard, and copies it into a **BYTES** field using **PUT**.

The word **ENTIRE** may be used in place of a field name:

ENTIRE	(-- n a)	Returns parameters for the "pseudo-field" that occupies the entire record in **BYTES** format. For example:

Word	**Stack**	**Action**
ENTIRE	(Cont..)	**ENTIRE B?**

types the contents of the current record as though it were a single **BYTES** field.

REFERENCES

EXPECT, Section 3.7.1
Fetching Input to **PAD**, Section 2.3.6.3
PAD, Section 2.3.1
Report Generator, Section 8.9

8.5.4 Direct Access to Fields

The Data Base Support option is set up so that field names may be used with field access operators in a transparent way, although in fact more is going on with these words than meets the eye. In the event that you need to directly access fields in a file (for instance, if you wish to use **MOVE**, **ERASE**, etc. instead of **N!**, etc.), you should understand the details explained in this section.

The addresses returned by user-defined field names are intended to be consumed by the field reference operators (Section 8.5.3). These addresses, however, are not the addresses of the actual data in a block buffer, but rather addresses within working storage (Section 8.1.4). The field reference operators perform the necessary offset correction, call the appropriate block and access the data. In the case of "fetch" operators, the operators move the data elsewhere (numbers are pushed onto the stack; strings are moved to **PAD**). This allows the field-name words, which return the address, to be used transparently with either working storage or the file data itself; the difference depends solely upon the operator that fetches or stores the data.

Each field reference operation is an implied disk access, since it calls **BLOCK**. It is important not to carry the address of a field in a block buffer on the stack across any I/O operation (such as displaying the content of a field or accessing another field), since in a multitasking environment another task may perform disk activity that changes the content of the disk buffer.

Occasionally it may be useful to bypass the protection of the field reference operators, and determine the actual address of a field in a disk buffer. This can be done by the following phrase:

```
field-name ADDRESS
```

This phrase places the actual memory address of the field on top of the stack. For example, the following phrase will move an array of 100 2-byte data elements from working storage to disk much faster than it would take to calculate addresses repeatedly using **N!**:

```
DATA  DATA ADDRESS  200 MOVE  UPDATE
```

The first use of **DATA** returns the address of the image of the field in working storage. The phrase **DATA ADDRESS** returns the location of the field in virtual memory. **200 MOVE** moves the image in working storage to the disk buffer. **UPDATE** is necessary after writing to a disk buffer.

For **BYTES** fields (since invoking the name of a **BYTES** field pushes both the location and length onto the stack), the phrase:

```
field-name ADDRESS
```

returns the length and virtual memory address (note that the order is reversed from the standard "address, count" order).

If direct addressing is used, you must remember that the content of the buffer can change at any time the task either requests I/O from any source or causes execution of **PAUSE** or **WAIT**. Furthermore, if you modify the contents of any field directly (without using **N!**, **B!**, etc.), you must invoke **UPDATE** after the modification.

REFERENCES

Disk Buffer Management, Section 3.2.1
MOVE, Section 2.3.4
Multitasking Overview, Section 1.2.3

8.5.5 Access to the Record Image in Working Storage

Because field names return addresses within local working storage, you can directly access the working storage image of a record. This lets you map data items as though they were contained in records, although they are kept in resident memory instead of on the disk. There is only one "record" in the working storage area.

Using ordinary memory-access operators in conjuction with field names provides access to working storage locations:

Word	Action
C@	Fetches an 8-bit number. Example: **AGE C@**
C!	Stores an 8-bit number. Example: **39 AGE C!**
@	Fetches a single-length number. Example: **LINK @**
!	Stores a single-length number. Example: **16 LINK !**
2@	Fetches a double-length number. Example: **PRICE 2@**
2!	Stores a double-length number. Example: **196.75 PRICE 2!**
S@	Fetches a string from working storage to **PAD**. Example: **NAME S@**
S!	Stores a string from **PAD** into working storage. Example: **NAME S!**

The word **WORKING** returns the address of the beginning of the task's working storage.

REFERENCES

Memory-Stack Operations, Section 2.1.2
PAD, Section 2.3.1
Working Storage, Section 8.1.5

8.6 ORDERED INDEX FILES

An ordered index file is one in which the records are kept in ascending order depending upon the ASCII values of a *key*. A key is an item of data that is used in a match or comparison.

There are two purposes for an ordered index file. First, it greatly speeds up searches based on the key data. Second, it allows you to display the main file alphabetically without having to sort it.

Each record in the index file contains a key together with a link address to an associated main file. This link resides in a 16-bit field called **LINK**. In Fig. 8.6, the index file **(NAMES)** contains the names of people, ordered alphabetically, along with links to the main file.

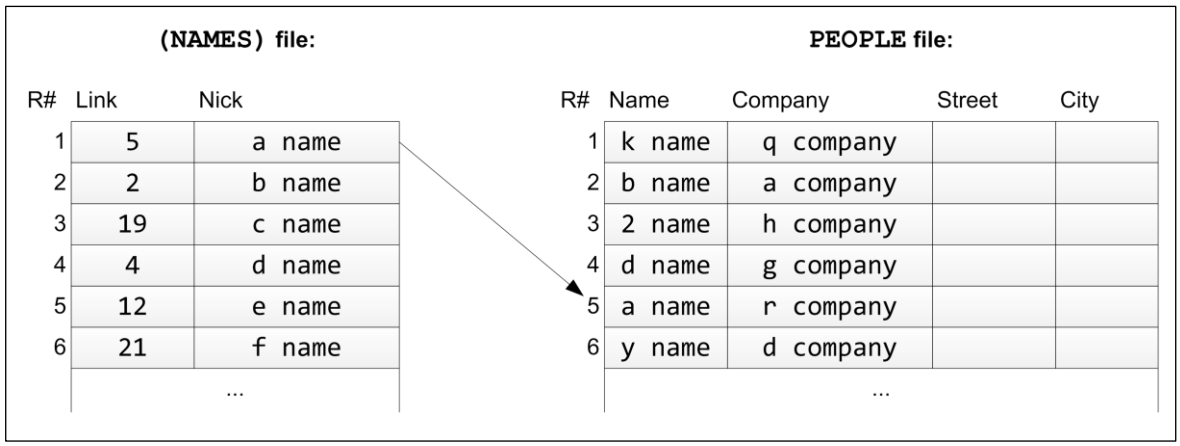


Fig. 8.6

An ordered index file (on the left) showing links to the corresponding records in the main data file (right).

You may have several index files addressing the same main file. For example, a file of scientific data could be indexed by both sample name and observation number, using two separate index files. In Fig. 8.7, a second index file (**COMPANIES**) points to the same main file **PEOPLE**, but uses the company field as a key, and keeps the records ordered alphabetically by company.

Searches on an ordered index are performed using a "binary search," which locates a record (or the place that it should go if it is not in the file) with only $\log_2 n$ steps rather than n/2 (which is the average for a "brute force" or sequential search).

A binary search works by taking the occupied part of the file and dividing it by two, then comparing the desired key with the field in the middle record. If the key is larger, then the high half of the file is halved again. This process is repeated until the size of the remaining set of records is one. This remaining record must match the key, if the key is in the file; otherwise, it is the record *before* which the key would be inserted. For a file of 128 records, a binary search requires only seven comparisons, as compared with an average of 64 for a sequential search.

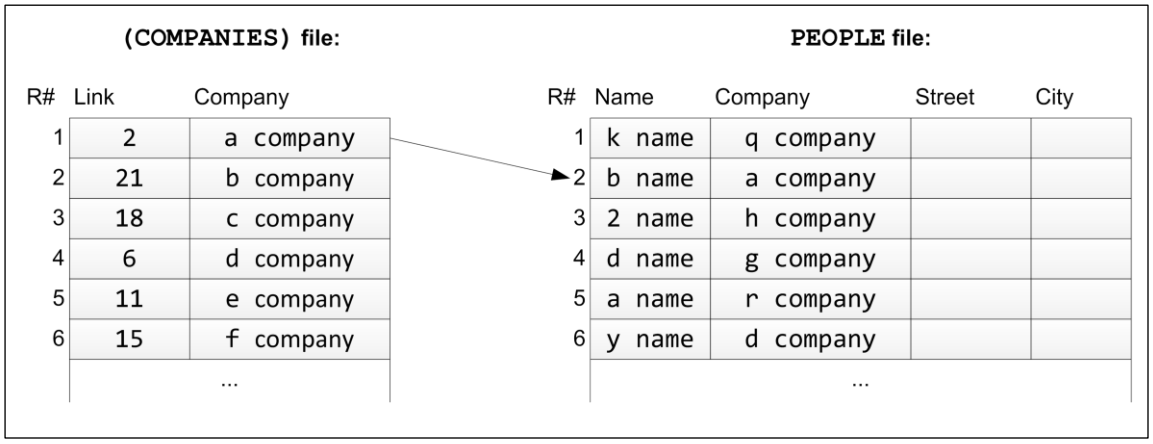


Fig. 8.7

Another index to the same main file shown in Fig. 8.6, this time using the company name as key.

An ordered index is a "dense file." That is, there are no gaps between active records. Therefore, **AVAILABLE** always reflects the number of records in the index file, and all records in the index file can be accessed with a **DO LOOP** with the knowledge that all records are active. With files maintained using **SLOT** and **SCRATCH**, you must check the **LINK** field (first two bytes of every record) to see whether each record is active.

8.6.1 Index File Records

At minimum, an ordered index file must contain the key and the link that associates the key with its main data record. The link is a 16-bit record number residing in the first two bytes of the record, and the key field immediately follows.

You can keep data other than keys in an index file and process this data in the same manner as data in other types of files. Such a technique should be avoided, however, if more than one user will have simultaneous access to the file, because record numbers may change due to insertion or deletion by other users.

The time required to search an index depends upon the length of each record as well as the number of records, because longer records will require more blocks to store the file, and hence more disk accesses to search it. Therefore, you should keep these records as small as possible.

The first two bytes of each record in an index file contain the link to the associated record in the main file. polyFORTH 2012 predefines this field as **LINK**. The phrase:

```
LINK N@
```

reads the link field of the currently selected record and returns it on the stack.

When creating the record description (Section 8.5.1) for an index file, you must skip over the **LINK** field by using the phrase **2 FILLER** at the beginning of the layout, or by starting with a displacement of two rather than zero. For example:

```
0  2 FILLER              ( Link to PEOPLE file)
   10 BYTES NICK         ( Last name key)
DROP
```

The key may be ASCII or binary. In order to make it possible to use binary integers as keys, as well as to speed up the search, the comparison made in the search routine compares *cell-by-cell*, rather than byte-by-byte. To accommodate this, you must make your key fields an even number of bytes in length. On machines which use a byte order that would render the most significant byte the second one in a string, the operators **B@** and **B!** reverse bytes when fetching and storing from disk such that the data on disk is in a compatible order.

Be aware that the order of the records in the index file is subject to frequent change as a result of file insertion or deletion. Because the record number of an index record may change, it should not be used directly for any purpose.

You must also take special care when sharing ordered files. We suggest you limit the index file to keys, and keep all other data in an associated main file record. Otherwise, a task may be pointing at a current record in an index, but before it accesses the data in the record the index record changes position.

8.6.2 Ordered File Maintenance

An "ordered index" file in polyFORTH is one in which the keys are maintained in ascending ASCII sequence. For instance, an index to a file of records of people might be ordered by last names.

An ordered file allows quick searching on key fields. For instance, given a name, we can search the index file looking for a match. From the index record where the match was found, we can obtain the link to the main file.

8.6.2.1 SEARCHING AN ORDERED INDEX

In polyFORTH, this routine is called **BINARY** (named because it performs a binary search). Here's how it works:

As we've seen (Section 8.5.5), field names return the address of the field in the "image" of the record in working storage. **BINARY** expects to find the match criteria for the desired field in this image (Fig. 8.8).

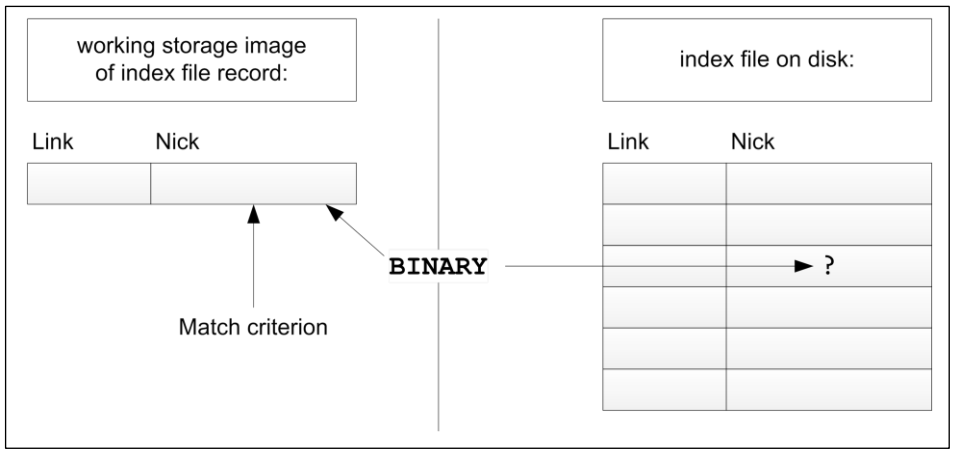


Fig. 8.8

BINARY searches the ordered index for a match to the key in working storage. It returns the content of the **LINK** field of the matching record, and aborts if there is no match.

Make sure that you have allocated enough room in working storage for all tasks (including the printer task) to hold the image of any record on which you use this technique (Section 8.1.6).

For instance, suppose we want to take a name from the input stream, then search for it in the **NICK** (short for "nickname") field of our **(NAMES)** file. The phrase:

```
1 TEXT  nickname NICK S!
```

captures the name from the input stream and stores it into the image of the **NICK** field in working storage.

Now we use **BINARY** to search the index file for this name, first ensuring that the index file is current. **BINARY** expects on the stack the arguments returned by a **BYTES** field:

```
(NAMES)  NICK BINARY
```

Here's what **BINARY** does:

Word	Stack	Action
BINARY	(n a - n)	Searches the current file looking for a match between the criteria in working storage and the given field in the data. Issues a system abort if it cannot find the record requested. On the stack is the record number of associated record in the main field (that is, the contents of the link field of the matching index record). The matching index record number is in **R#**.

(A related word, **-BINARY**, is discussed in Section 8.6.2.2.)

REFERENCES

Access to the record image in working storage, Section 8.5.5
Binary search principles, Section 8.6
R#, Section 8.4.1
TEXT, Section 2.3.6.3
Working Storage, Section 8.1.4

8.6.2.2 INSERTING A RECORD IN AN ORDERED INDEX

Inserting a new record in an ordered file involves two steps. First, we must determine the location in the index file for a new key to be inserted. This ensures that the index file will always be properly "sorted."

Second, we must be able to insert the new key into the file at the appropriate place, moving all subsequent records one notch down in the file.

Using the example in Fig. 8.6, let's consider what must happen when we add a new person to our data base. First, must insert a new index record
into the **(NAMES)** file in the appropriate place, then allocate a new record in the **PEOPLE** file for the data itself. Finally, we must point the **LINK** field in the index record to the data record in the main file.

We've already seen in Section 8.4.3 that the word **SLOT** is used to allocate new records in data files. Adding a record to the index file is more complicated, because we must insert the new record at the appropriate place to keep the keys ordered. For this purpose, we use the words **-BINARY** and **+ORDERED**.

Word	Stack	Action
-BINARY	(n a - t)	Searches the current file looking for a match between the criteria in working storage and the given field in the data. A zero result ('false') means that a match was found; a non-zero flag means that no record in the file contains the indicated key. On exit, if a match is found **R#** contains the number of the first matching index file record; otherwise **R#**

contains the number of the index record before which an insertion will be made. Pronounced "not-binary," because it returns 'true' if a match is not found.

+ORDERED (-) Inserts the record whose image is in working storage into the current record in an ordered index. Subsequent records in the index file are advanced one position relative to the start of the file.

-BINARY expects the same conditions as **BINARY** (Section 8.6.2.1):

1. The current file is the ordered index to be searched.
2. The match criterion is in the key field in working storage.
3. The arguments produced by a **BYTES** field name are on the stack.

+ORDERED expects the following conditions:

1. The current file is the index to be modified.
2. The record before which the insertion is to take place has been previously selected by **-BINARY**.
3. The key and **LINK** fields to be inserted are in their respective fields in working storage.

Using our example, then, the standard procedure is:

```
1 TEXT                  (scan the input stream for the name)
NICK S!                 (store it into the image of NICK)
NICK -BINARY            (search the index file, using the NICK field as the key)
IF                      (no match:)
  SAVE PEOPLE SLOT RESTORE              ( obtain available record number in main file)
  LINK !                (store the record number into working storage)
  +ORDERED              (insert the new index record)
ELSE                    (duplicate entry)
  ORDERED RELEASE
  1 ABORT" Already in file "
THEN  ...
```

Because your code must provide the location into which the insertion will take place (using **-BINARY**), you have the option of determining how to handle duplicate keys if **-BINARY** returns a false (zero) indication. This is normally handled as an abort condition, as shown above.

During execution of the **-BINARY ... +ORDERED** sequence, the index file should not be accessed by any other task, since the record numbers of all records following the insertion point are changing.

To prevent conflicts, the Data Base Support option includes a facility management variable called **ORDERED**. **-BINARY** issues an **ORDERED GET**. This phrase protects the file from being accessed by other tasks on the system until the current task releases it. In this way, file integrity is maintained. **+ORDERED** issues an **ORDERED RELEASE**. If you exit from the operation in any other way, you must do this yourself. The intent is for the task that performed the search to retain control of the file from the moment when the insertion point has been found until the expected insertion has taken place, or until it has decided not to do one.

The word **BINARY** also performs an **ORDERED GET**, so that searches cannot be performed while another task is using this facility. **BINARY** performs an **ORDERED RELEASE** immediately after the search, however, so it "holds" the facility only during the period of the search itself.

REFERENCES

Binary Searches, Section 8.6.2.1
Facility Variables (**GET** and **RELEASE**), Section 4.7
SAVE and **RESTORE**, Section 8.4.2
TEXT, Section 2.3.6.3

8.6.2.3 DELETING A RECORD FROM AN ORDERED INDEX

-ORDERED is used to delete a record from an index file. It may only be issued immediately after the record has been selected (normally by a prior use of **BINARY**).

Word	Stack	Action
-ORDERED	( )	Deletes the current record (**R#**) from an ordered index which is the current file. Subsequent records move back one position, relative to the start of the file.

Because the actual space that was occupied by the deleted record will be occupied by the record that until now followed it, the record is completely obliterated by this operation (unlike **SCRATCH**, which only changes the first two bytes of the record).

Here is an example using **-ORDERED**.

```
1 TEXT              (scan the input stream for the name)
NICK S!             (store it into the image of NICK)
NICK BINARY         (search the index file, using the NICK field as the key; return main file record
                      number)
ORDERED GRAB        (regain control of ORDERED, which BINARY released)
-ORDERED            (delete the index record)
PEOPLE SCRATCH      (de-allocate the record in the main file whose number is on the stack from BINARY.)
```

In this example we had to **GRAB** the facility variable **ORDERED** to prevent another task from accessing the file during the moving of records that will occur during the **-ORDERED** operation. **GRAB** is used instead of **GET** because **GET** releases the CPU so other tasks can run (and potentially alter the file). **-ORDERED** performs an **ORDERED RELEASE** when it is finished.

REFERENCES

Binary Searches, Section 8.6.2.
Facility variables (**GRAB**, **GET**, and **RELEASE**), Section 4.7
SCRATCH, Section 8.4.3

8.6.3 An Example—A Simple Mailing List

The following pages show an example of a simple mailing list application. It demonstrates the use of an ordered index to provide easy access into a file based on a key, such as last name and first initial, and a report which is in alphabetic order based on that key.

This application is a good example of the layout of a polyFORTH application, with a "help screen" at the top of a triad, followed by the application load block and relevant file definitions. The help screen is automatically displayed when the application is loaded, and may be displayed any time by the command **HELP**.

Ensuing blocks are shown with their corresponding "shadow blocks" on the facing page. This was necessary due to the reduced size of this book; on most computer systems you would print the source and shadow blocks side-by-side on the same page, using the **PAIRS** command in the polyFORTH **PRINTING** utility. The last pair of blocks are shown vertically, due to space limitations.

REFERENCES

Disk Organization, Section 5.2.5
PRINTING Utility, Section 5.2.3
Shadow Blocks, Sections 1.5.1, 5.2.3

828

```
 0 ( Sample file system application)   EMPTY
 1 (Working storage)   128 ALLOT
 2
 3 : HELP   828 HELPS ;   HELP
 4
 5 ( File allocation)   830 LOAD
 6 ( Record description)   831 LOAD
 7 ( Data entry)   832 833 THRU
 8 ( Data display)   834 LOAD
 9
10
11
12
13
14
15
```

829

```
 0 This application assumes FILES is loaded in Block 9
 1
 2 HELP  Display these PERSONNEL instructions.
 3 enter name     Enter a new person into the file with
 4                access key of 'name'
 5
 6 remove name    Delete 'name' from the data base
 7
 8 fix name       Enter new information replacing all
 9                current data for 'name'
10
11 see name       Display a person whose key is 'name'
12
13 s              Display current person
14
15 all            Display all records in the file.
```

830

```
 0 (Bytes records blocks origin      name   )
 1    16      300     6   1500 FILE  (PEOPLE)
 2   128      300    43   1530 FILE  PEOPLE
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

1233

```
 1 The record layout for both the PEOPLE and (PEOPLE)
 2    files.  The LINK is predefined, and subsequent
 3    fields are offset from the previous 4 fields.
 5    For example, the NICK name is 14 bytes long
 6    starting in the 2nd byte.
 7 ZIP  is a 32-bit number, as is PHONE.
 8 AREA  code is single precision.
 9
10 The offset for the field types is carried on the
11    stack so that it may be either displayed or
12    dropped at the end of the load.  We use it in
13    this case to display the record size.
14
15
```

1234

```
 0 PERSON  parses the input stream following it for the
 1    NICK field.  It leaves us pointing at the NICK
 2    field in the (PEOPLE) file.
 3
 4 DIGITS  Prompts the terminal for input and converts
 5    it to binary on the stack.
 6
 7
 8 >DOUBLE  Takes the output of DIGITS and converts the
 9    number to 32-bit.  PTR is negative if the
10    NUMBER conversion was 16-bit, in which case the
11    high-order part of the number may be found at
12    'NUMBER 2+ .
13
14 !LABEL  Prompts for each field in order.
15
```

1235

```
 0 enter  creates a new entry for the person whose
 1    nickname follows in the input stream, prompting
 2    for entry of additional data. If there is already
 3    an entry for that nickname, an error message is
 4    issued.  In either case, the record remains the
 5    current one for future editing.
 6
 7 fix  accepts new data for the pre-existing entry
 8    whose nickname follows in the input stream.
 9
10 remove  deletes the person whose nickname follows
11   from the data base.
12
13
14
15
```

831

```
 0 ( Record descriptions)
 1 2 ( LINK)
 2 14 BYTES NICK     ( Nickname, used as the key.)
 3 32 BYTES NAME     ( Full name, first name first.)
 4 32 BYTES STREET   ( Street addr. or PO Box, etc.)
 5 32 BYTES CITY
 6    DOUBLE ZIP     ( Note:  can only handle US zips)
 7    NUMERIC AREA
 8    DOUBLE PHONE
 9
10  CR .( Main file: ) .  .( Bytes )
11
12
13
14
15
```

832

```
 0 ( Data storage)
 1 : PERSON ( - n a)   1 TEXT  NICK S!  (PEOPLE) NICK ;
 2
 3 : DIGITS ( - d/n)   QUERY  32 WORD  NUMBER ;
 4
 5 : >DOUBLE ( n/d - d)   PTR @ 0< IF 'NUMBER 2+ @
 6    THEN ;
 7 : !LABEL   CR ." Name: "   NAME ASK
 8     CR   ." Street: "   STREET ASK
 9     CR   ." City, State: "   CITY ASK
10     CR   ." Zip: "   DIGITS  >DOUBLE  ZIP D!
11     CR   ." Area: "   DIGITS AREA N!
12          ." Phone: "   DIGITS >DOUBLE  PHONE D! ;
13
14
15
```

833

```
 0 ( Record management)
 1 : enter   PERSON -BINARY IF  SAVE  PEOPLE SLOT DUP
 2       READ  NICK S@ NICK B!  RESTORE  DUP LINK !
 3       +ORDERED  PEOPLE READ !LABEL
 4       ELSE  ORDERED RELEASE  ABORT" Already known "
 5    THEN ;
 6 : fix   PERSON BINARY  PEOPLE READ !LABEL ;
 7
 8 : remove   PERSON  BINARY -ORDERED  PEOPLE SCRATCH ;
 9
10
11
12
13
14
15
```

```
          1236
 0 .PHONE  displays the AREA and PHONE numbers as one
 1    would expect to see them.
 2
 3 .ZIP  forces the zip code to be displayed in
 4    nnnnn format.
 5 n .PERSON  displays the data from the nth record in
 6    the PEOPLE data file.
 7 see  Parses the input stream and displays the proper
 8    record.  s does the same thing using R#
 9    (the current record).
10 all  uses the RECORDS word which returns the
11    initial value and number of records+1 in the
12    data file.  The loop counter is used to access
13    each record in the ordered index (PEOPLE),
14    where the LINK field points to the data in the
15    PEOPLE file.
          834
 0 ( Data Display)
 1 : .PHONE   AREA N@ 0  <# 41 HOLD # # #  40 HOLD #>
 2    TYPE SPACE  PHONE D@ <# # # # # #  45 HOLD ( -)
 3    # # # #> TYPE ;
 4 : .ZIP   ZIP D@ <# # # # # #>  TYPE ;
 5
 6 : .PERSON ( n)   PEOPLE READ  CR  NAME B?  5 SPACES
 7    ." (" SPACE   NICK B?  ." )"  CR   STREET B?
 8    CR  CITY B? CR   .ZIP  10 SPACES  .PHONE  SPACE ;
 9
10 : see   PERSON BINARY  .PERSON ;
11
12 : s   R# @ .PERSON ;
13
14 : all   (PEOPLE) RECORDS DO  I (PEOPLE) READ
15    LINK N@ .PERSON  CR  LOOP  SPACE ;
```

8.6.4 Hierarchical Ordered Files

polyFORTH's ordered indexes have the property than whenever a record is inserted or deleted all records following the point at which the action occurs are physically moved to accommodate the change. Although this form of maintenance is somewhat slower than maintaining order by updating chains or pointers (as some data bases do) it is substantially more reliable.

The assumption is that in most applications an index is searched frequently, and insertions and deletions occur relatively infrequently. As a result, we have optimized search time and reliability above maintenance time.

The actual time an insertion or deletion will take depends upon the position in the file at which the action occurs (if it is near the beginning of the file more records must be moved), the number of records in the file, and the size of each index record. In practice, indexes of several thousand records may be maintained on a hard disk without unacceptable delays.

Some applications, however, involve tens of thousands of records that must be searched and maintained in order. In order to deal with such applications, the recommended approach is to divide the total index into several sub-indexes, each of which will be a manageable size. For example, a company with 40,000 employees might separate them into departments. The department code can index a table in memory giving the appropriate origin block number for the index of employees in each department. This block number may be put into the ORG field of the

FDA of a private copy of a generic file definition for the index. Or, the first letter of the employee's last name may be used to select one of 26 indexes.

Such a multi-layered approach is called a *hierarchy*. If you are designing a hierarchical file structure, the important considerations include keeping the decision-making process simple and independent of any frequently changing conditions. If possible, try to base the initial choice on something that can be evaluated without need for a special file search. Above all, you should avoid keeping record numbers of records in an ordered index in a higher-level index, as ordered index record numbers are subject to change.

REFERENCES

File Definition Areas, Section 8.3.2

8.7 CHAINING

Chaining is the linkage of one record to another, whether in the same or a different file. Generally, chaining is appropriate when an unknown amount of data must be associated with a piece of information.

There are as many ways to chain records as there are varieties of applications. In this section, we'll cover most of the situations that require chaining, and present general solutions to each case.

8.7.1 Chaining Techniques

Before you begin coding, make sure that you study the exact requirements carefully. Reviewing this section for considerations will be helpful.

Here are some design considerations to take into account:

1. Will the chaining occur within the same file, or to an auxiliary file?

2. Must there always be at least one auxiliary record chained to a main record, or may a main record have no auxiliary records?

3. When you traverse the chain, should it be in the order in which its elements were added (first-in, first-out), or in reverse (last-in, first-out), or should the chain be maintained in order by a key (such as date and time)?

Let's explore these issues one by one.

In some applications, it is possible to chain records within a single file. Naturally, this is easier than chaining to another file.

For example, suppose that we have a file of customer names and addresses. Some of our customers have several addresses: one for invoicing, one for shipping, and so on. Because multiple addresses are the exception, not the rule, and because address fields are large, we'd prefer not to allow room for multiple address fields within each customer record.

So, we use chaining instead. At this point, we must examine how much information each auxiliary record must contain. It turns out that each auxiliary record must contain almost as much information as the main record. If we create a separate file for the auxiliary records, each record would need to be nearly as large as a record in the main file.

If there is relatively little in the main record (the one all customers have) beyond the primary address, you may as well use additional records in the same file to contain additional addresses. As Fig. 8.9 shows, this approach lets

us re-use the field layout structure that we created for the main file records, even though there are some fields in the primary record that we don't use in the auxiliary records.

For another example of chaining within a single file, we turn to the **DOCUMENTOR** application included with polyFORTH ISD-4 (see Section 8.10). This application lets you enter descriptions of the commands in your applications and produces alphabetized glossaries.

For each word that you enter into the system, the **DOCUMENTOR** saves its name, vocabulary, stack effects (before and after) as text strings, the source block, the date this entry was created or updated, plus as many lines of descriptive text as you care to include.

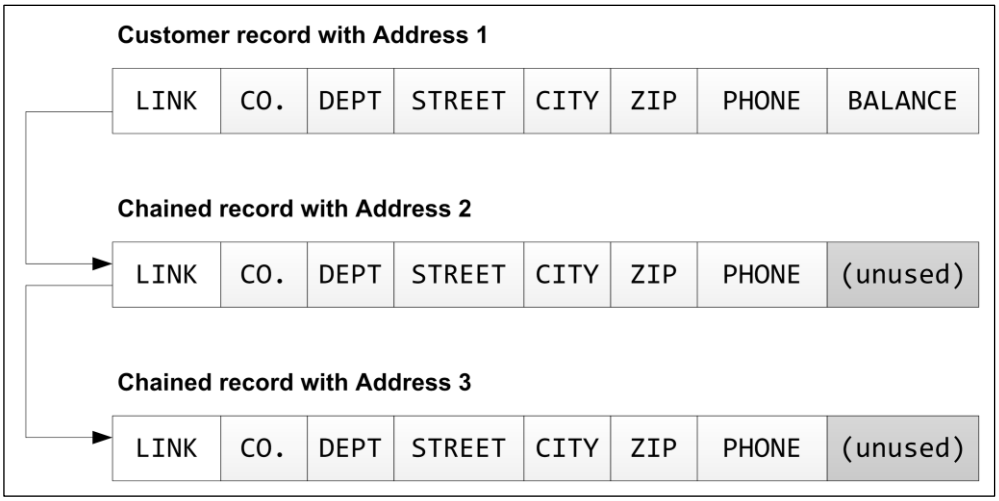


Fig. 8.9

Example of a chain with all records in the same file.

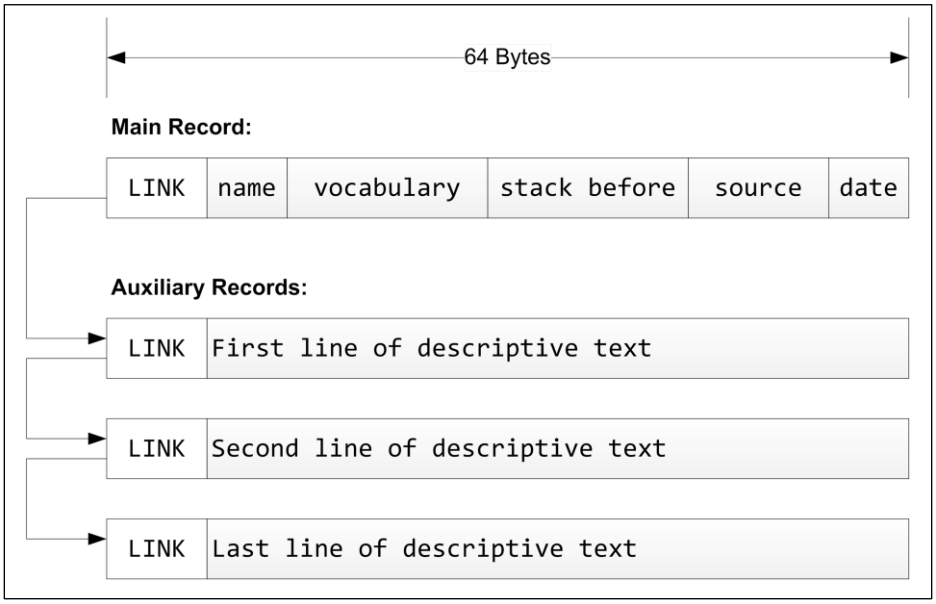


Fig. 8.10

Record chaining in the polyFORTH **DOCUMENTOR** utility.

Fig. 8.10 shows the record structure for the **DOCUMENTOR**. All data except the text is stored in the main record for each command. This record points to an auxiliary record that contains the text description. This record may in turn point to a second text record, and so on. A separate index file contains the alphabetized keys that point to main records in this file.

Although the main records and auxiliary records share *no* fields in common (except **LINK**), they are the same size. Thus it is most efficient to keep both types of records in the same file.

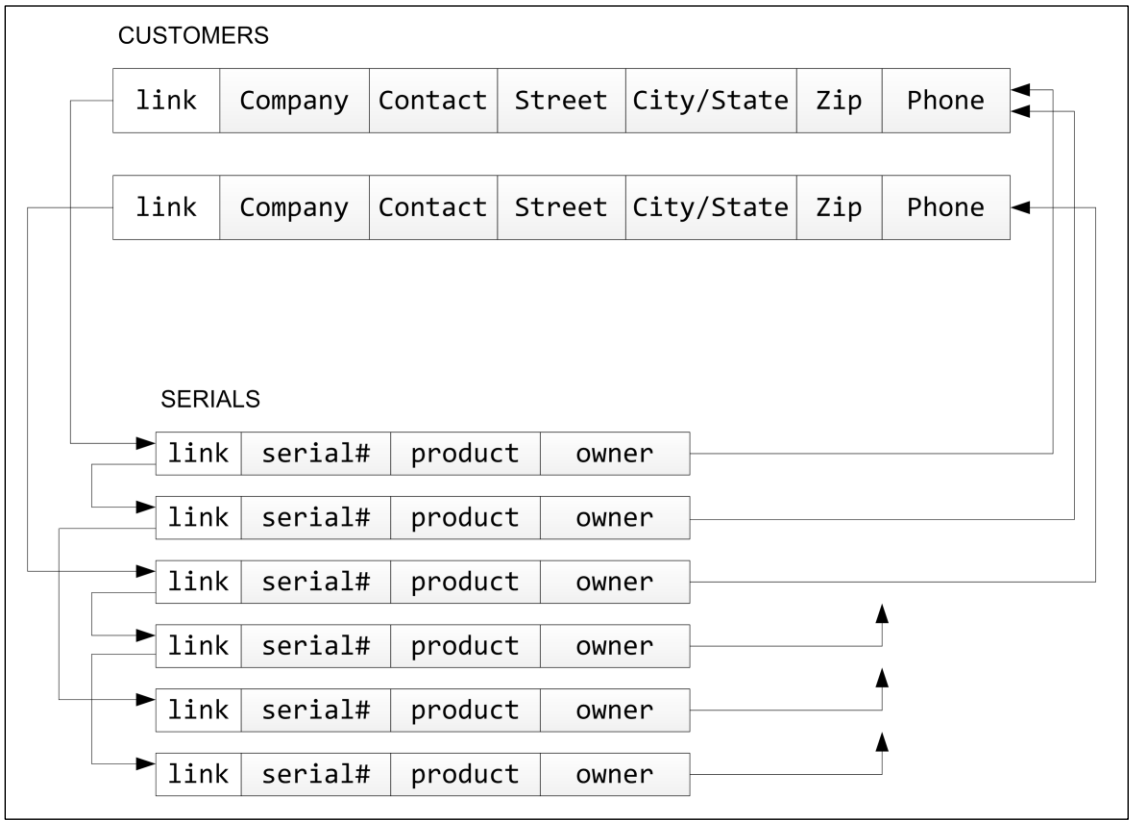


Fig. 8.11

A variable number of serial number records for products purchased by each customer. Note that each serial number record contains a pointer back to the "owner" record. This is important for maintaining file integrity.

A third example illustrates the opposite situation. Suppose we have a list of customers who have purchased our products. For each customer, we also have a list of the serial numbers of the units they received. For some customers, there are no serial numbers; for others, as many as twenty.

You can see in Fig. 8.11 that a serial number record takes much less space than a customer record. Because of this size variance, it's better to create two separate files, one called **CUSTOMERS** and the other **SERIALS**. Each main record in the **CUSTOMERS** file may chain to one or a series of records in the **SERIALS** file. A record in **CUSTOMERS** can also contain an empty link, which would be represented by a value of -1 in the **LINK** field. A -1 **LINK** also identifies the last serial number for a particular customer.

This last example raises the second consideration: whether the application must be able to handle the case of no auxiliary records, or whether the minimum number of auxiliary records attached to a main record must be one.

In the first case, when a main record is created, its link can be left alone (-1) and no auxiliary record need be **SLOT**ted. However, the routine that appends a new auxiliary record to the chain must check whether it is linking from the main record or an auxiliary record.

In the second case, when a main record is created, an auxiliary record must also be slotted, and its number saved in the main record's pointer. Furthermore, the routines for advancing through the chain will differ, as we'll see in the next section.

A third consideration is whether chaining must be last-in, last-out; last-in, first-out; or both. In the case of the **DOCUMENTOR** described earlier, obviously chaining must be first-in, first-out. In such cases, the process of adding a new record to the chain involves:

1. Finding the end of the existing chain;

2. Allocating a new record;

3. Setting the link in the last record of the existing chain to point to the new record.

An example of the opposite situation is a bookkeeping database in which each customer record chains to a series of auxiliary records containing transactions. Because we are almost always more interested in recent transactions than ancient ones, we chain in a last-in, first-out manner. In this case, the process of adding a new record to the chain involves:

1. Allocating a new auxiliary record;

2. Setting the main record to point to it;

3. Placing the main record's previous link number into the link field of the new record.

If the application demands that both directions of chain-following be allowed, then each auxiliary record must contain two link fields: one to the next record in the chain, and one to the previous.

Each chained record should contain a pointer back to the record that is the head of the chain (which may or may not be in the same file as the chain). Some applications use this directly. For instance, suppose in our serial number example we keep an ordered index file using the serial number itself as the key. If records in **SERIALS** contain a pointer to the owner of the chain as shown in Fig. 8.11, then by entering a serial number the user can see which customer has received that instrument.

The most important reason for including a pointer to the owner, even if the application doesn't otherwise demand it, is for ensuring integrity of the data. If through some mischance of hardware failure a link in the main file becomes lost, the chains can be reconstructed and attached to the main records.

8.7.2 Chaining Commands

As we have seen in the previous section, the choice of chaining techniques depends on application needs and on performance tradeoffs. Rather than attempt to decide for you, the developers of polyFORTH 2012 provide a collection of commonly-used chaining tools. You may leave them as is, or you may modify them. The table below gives the general set of commands in the chaining toolbox. Some words appear more than once; this is because several implementations may be useful, depending on how you've answered the design questions in Section 8.7.1. The version shipped with the system is marked with an (*). The others are minor variants; code for some of the alternate versions is given elsewhere in this chapter.

Word	Stack	Action
HEAD	(-- a)	A user variable that points to the first record (head) of the current chain.
LINK	(-- a)	A pre-defined field (the first two bytes of any record) which may be used for chaining. This same field is used in ordered index records to link to the main file records.

Word	Stack	Action
FIRST	( )	(*) **READ**s the **HEAD** record in the chain. This version is used in applications in which there is always at least one auxiliary record and all are within the same file.
FIRST	(-- t)	Returns a flag indicating whether the main record is chained to any auxiliary records, and if it is, **READ**s the record. This version is used in applications in which the **HEAD** record may have no auxiliary records, and when auxiliary records are in a different file.
-NEXT	(-- t)	(*) Reads the next record, assuming that the chain is linked through the field called **LINK**. Returns 'true' if there is *not* a next record in the chain. Pronounced "not-next."
-NEXT	(-- r/0)	Alternate version of **-NEXT**; returns the record number of the next record in the chain, if any, 0 ('false') otherwise. Does not read the record.
-LOCATE	(n - t)	Searches the chain, starting from **HEAD**, for the nth record, returning true if the chain isn't that long. Otherwise, it returns false, having left **R#** pointing to the specified record.
CHAIN	(n)	Inserts a new record at the nth position. If *n* is larger than the length of the chain, inserts the new record at the end. Alternate versions might take no argument and chain at the beginning (last-in, first-out), end (first-in, first-out) or according to a key.
UNCHAIN	(n)	Removes the nth record from the chain.
SNATCH	(a r - r)	Given a field address and record number, fetches the record number from that field and replaces it with the record number given. It is used to update chains.

The arguments for **-LOCATE**, **CHAIN**, and **UNCHAIN** count from zero, where zero is the first record in the chain, and count sequentially down the chain. An argument of -1 is conventionally used to specify the end of the chain (since you don't necessarily know how long the chain is).

The standard versions of **FIRST** and **-NEXT** assume there is always at least one record in the chain, and it's also the **HEAD** record (i.e., it will be subject to the same processing as the others). The chain may be processed in a **BEGIN ... UNTIL** loop:

```
FIRST BEGIN ... -NEXT UNTIL ...
```

The alternate versions allow for the possibility that there are no auxiliary chains, and are optimized for a **BEGIN ... WHILE ... REPEAT** loop:

```
FIRST BEGIN  ?DUP WHILE  READ ...
   NEXT REPEAT ...
```

If you have only one set of chained records and the top of the chain is in a different file from the members, you may incorporate the selection of the file in the words **FIRST**, etc. If you have several sets, you will need to select the file externally. Still another set of variations might allow for the fact that you have more than one chain attached to your main file, and therefore not all chains start with the **LINK** in the main file record. Moreover, there may even be multiple chains through the auxiliary records. In these cases, you would remove the references to **LINK** in these words and specify the field externally.

The intent here is to present a design concept that has worked in many applications, but which presumes that you will tailor a basic vocabulary to your specific application needs—a practice that is consistent with the overall design of Forth in general. Assuming you are adding custom versions of the chaining words for your application, don't forget to remove from the **FILES** load block the reference to the standard ones.

REFERENCES

BEGIN ... UNTIL, Section 2.4.2
BEGIN ... WHILE ... REPEAT, Section 2.4.3
FILES Load Block, Section 8.1.5

8.7.3 Application Examples

This section offers coded solutions to two application problems.

We introduced the **DOCUMENTOR** program, which is included with your polyFORTH system, in Section 8.7.1. The use of this utility is more thoroughly documented in Section 8.10.

The word **(SHOW)** includes this sequence:

```
...                    ( display data from the main record)
BEGIN +L  -NEXT NOT WHILE
   10 SPACES  PHRASE B?
REPEAT ;
```

The word **+L** is similar to **CR**; see Section 8.8.4.

The word **(SHOW)** displays all information about a command. The code fragment shown above displays the list of description lines for the command. When it begins, the main record is still current.

As we saw in Section 8.7.1, the main record's link field points to the first descriptive record, if there is one, which resides in the same file. When the loop begins, **-NEXT** determines whether the main record is linked to an auxiliary record. If not, the loop ends and nothing is displayed. If so, the **WHILE** portion is executed, which displays the first line of text and repeats the loop. Now **-NEXT** indicates whether there is another auxiliary record.

When the last record is reached, **-NEXT** indicates this and the loop ends.

The word **?LINES** is defined as:

```
: ?LINES   1 BEGIN 1+  -NEXT UNTIL  ?PAGE ;
```

The purpose of **?LINES** is to determine whether the current command's description will fit entirely on the page, or whether it is necessary to advance the page first to keep all of its lines together. The loop counts the number of lines (the head plus an unknown number of auxiliary records, at one line each). The word **?PAGE**, introduced in Section 8.8.4, takes an argument from the stack, starting a new page if that many lines will not fit on the current page.

Here is a definition using **SNATCH**:

```
: DELETE ( r#)   ...   BEGIN READ
   LINK 0 SNATCH  DUP 0< UNTIL  DROP ;
```

The part of **DELETE** shown here removes both the main record and all auxiliary records chained to it. The code begins on the main record. The phrase **LINK 0 SNATCH** fetches the record's link field, then replaces it with zero. This has the effect of "scratching" the record, but also provides a pointer to the next record to scratch.

The phrase **DUP 0<** tests whether the pointer indicates that the record just scratched was the last in the chain. If so, the loop ends; otherwise, it reads the next record, and so on.

You may also wish to study the definitions of **T**, **P**, and **U**, which use **-LOCATE**, **CHAIN**, and **UNCHAIN** in straightforward ways.

Our second coding example is another that we introduced earlier in this section: the customer file and associated serial numbers. Here we will present two versions of the application. The first, in Fig. 8.12, uses the versions of **FIRST**, **-NEXT**, and **-LOCATE** that are provided with your polyFORTH system.

In the first block we've defined the record structures for the two files. In the second block, we have words for entering new customers and serial numbers. The word **add** makes use of chaining.

As we saw in our earlier discussion of this application, it is legitimate for a **CUSTOMERS** record to have no serial number attached to it. In this case, the **CUSTOMERS** record will contain –1 in its **LINK** field. If auxiliary records are chained, they will reside in a separate file called **SERIALS**.

The process of adding a new serial-number record is not as simple as it would be if all records were contained in the same file. Here, add must make a decision. If there is no chaining yet, it must go to the **SERIALS** file and use **SLOT** to allocate a record.

Since this is the first record in the chain, it must also store this in the main record's **LINK** field. But if a chain has already been started, it will go to **SERIALS** and use **CHAIN** to add a new record.

The problem is that we cannot use **CHAIN** unless a chain exists already. If all records existed in the same file, then the main record would be the first record in the chain; and we could simply use **CHAIN** in all cases. We would not need a conditional. Or, even if records existed in separate files, but a minimum of one auxiliary record was always present, we could use **CHAIN** and avoid the conditional.

The phrase **-1 CHAIN** is a cliché that means "attach a new record onto the end of the chain." The -1 serves as a number that never gets reached, and **CHAIN** is defined so that if it never reaches n it adds the new record to the end of the chain.

```
 0 ( Customers and Serial Numbers)
 1 ( CUSTOMERS records:)
 2 0  2 FILLER  ( LINK to 1st serial#)
 3   20 BYTES COMPANY
 4   16 BYTES CONTACT
 5   30 BYTES STREET
 6 DROP
 7
 8 ( SERIALS records:)
 9 0  2 FILLER  ( LINK to next serial#)
10   10 BYTES SERIAL#
11    NUMERIC PRODUCT  ( product code)
12    NUMERIC OWNER  ( link to owner CUSTOMERS record)
13 DROP
14
15
```

```
 0 ( Customer/serial number file)
 1 : edit   CR  ." Company name? "  COMPANY ASK
 2    CR  ." Contact? "  CONTACT ASK
 3    CR  ." Address? "  STREET ASK ;
 4 : new   CUSTOMERS  SLOT DUP . READ  edit ;
 5
 6 ( Assumes a serial# chain linked thru HEAD)
 7 : (add)   CR  ." Serial# ? "   SERIAL# ASK ;
 8 : add   SAVE  LINK N@ DUP 0< IF  ( empty chain) DROP
 9          SAVE  SERIALS SLOT  RESTORE
10          DUP LINK N!  SERIALS READ
11       ELSE  HEAD !  SERIALS -1 CHAIN ( add at end)
12   THEN (add)  RESTORE ;
13
14 : edit ( n)   SAVE SERIALS  1- -LOCATE ABORT" Can't"
15    CR  Serial# B?  (add)  RESTORE ;

 0    ( Customer/serial number display)
 1 : .company   COMPANY B?  CONTACT B?  STREET B? ;
 2
 3 : .companies   CUSTOMERS  RECORDS DO
 4      CR  I .  I READ  .company  LOOP ;
 5
 6 : .serials   0  SERIALS  FIRST BEGIN
 7   CR 1+ DUP .  Serial# B?  -NEXT UNTIL ;
 8
 9 : serials   CR  .company  LINK N@  0>  IF
10    LINK N@  HEAD !  SAVE  .serials  RESTORE  THEN ;
11 : show ( n)   CUSTOMERS READ  serials ;
12
13 EXIT   Usage:
14 To enter a new customer:  new  then  add  as needed.
15 To edit an old one:  n show  then  add  or  n edit .
```

Fig. 8.12

An application example using the standard polyFORTH chaining operators.

The word **edit** may be used to change an existing serial number. From its purpose we can assume that a chain exists, and therefore it doesn't have to check the main record's **LINK** to make sure it points to a valid auxiliary record. It simply goes to the **SERIALS** file and uses **-LOCATE** to make the desired record current (aborting if the argument is not valid and **-LOCATE** terminates before reaching it). Then it displays the current contents of the field and lets the user re-enter it. Finally it restores the file pointers to the main file.

In the next block, the word **serials** displays the current company data, followed by a list of all associated serial numbers. Again, since there may be no chain at all, serial must make a decision. The test **LINK N@ 0>** returns 'true' if the link is positive (that is, not -1 or 0), indicating the first record in the chain. In this event, serials saves this link in the variable **HEAD**, selects the **SERIALS** file, and invokes .serials which uses **FIRST** and **-NEXT** to loop through all records in the chain.

To give you an idea of some of the many possibilities, we've coded the same application using different versions of the words **FIRST**, **NEXT**, and **-LOCATE**. While these definitions themselves are more complicated, they reduce the complexity of the application words that use them. These versions are sensitive to the possibility that a main record may not have any auxiliary records attached to it.

Here are the re-definitions, followed by the new versions of the affected application commands:

```
: VALID ( n - t)   0 OVER < DUP  IF
   SWAP READ  ELSE  SWAP DROP  THEN ;
```

```
: FIRST ( - t)   HEAD @ VALID ;

: NEXT ( - t)   LINK N@ VALID ;

: -LOCATE ( n - t)   FIRST IF BEGIN  DUP WHILE
      1-  NEXT NOT IF  DROP -1 EXIT  THEN
   REPEAT  ELSE DROP -1 THEN ;

: add   LINK N@ HEAD !  SAVE  SERIALS
   FIRST IF  -1 CHAIN  ELSE ( no chain)
      SLOT DUP RESTORE  LINK N!  SAVE
      SERIALS READ  THEN
   (add)  RESTORE ;

: serials   LINK N@ HEAD !  SAVE  0
   SERIALS  FIRST BEGIN  WHILE  CR  1+ DUP .
      SERIAL# B?  NEXT REPEAT  RESTORE  DROP ;
```

In the first block, **FIRST** returns a flag that is true if a chain exists at all. If so, the first record in the chain is made current. The word **NEXT** returns a flag that is true if another record exists in the chain. If so, that record is made current.

As you can see, both words make use of the same code, which we have factored into the definition called **VALID**.

We have also re-coded **-LOCATE** in this block. As usual, **-LOCATE** returns a "true" flag if the requested element of the chain cannot be found. In this version, it also returns a "true" flag if no chain exists.

These changes simplify our application definitions. **add** still has to make a decision, but it uses **FIRST** for the test.

Because of the way we have rewritten **FIRST**, **serials** no longer needs an **IF** statement at all. The only conditional is **WHILE**, which gets its argument the first time around from **FIRST**, and henceforth from **NEXT**. Thus, if a first record is absent, the **WHILE** phrase never gets executed. We eliminated the need for a subordinate word **.serials** completely.

REFERENCES

Data Base Design, Section 8.9

8.8 REPORT GENERATOR

The polyFORTH Report Generator is a set of words that assist you in the preparation of formatted output reports. Once you have specified the page format and column headings, and indicated the layout of a single record as a row of data, the Report Generator performs all required output formatting and also controls paging, the heading of each page and related operations.

An optional feature of the Report Generator allows subtotals and grand totals to be accumulated in a simple manner; these totals can then be printed on a separate line with a minimum of effort.

The following example will serve as a quick introduction to the Report Generator. It assumes the fields defined in the example in Section 8.1. Here is the code:

```
: .person   NAME ?B  STREET ?B  CITY ?B
   STATE ?B   ZIP ?B ;

[R People\ Name          Address
       City          St. Zip ]

: all   LITERAL LAYOUT  +L
      PEOPLE RECORDS DO  I READ  .person
   +CR  LOOP ;
```

This produces:

```
FORTH, Inc.                                Page 1  26 AUG 1986

                                        People
Name             Address                City           St   Zip

Andrews, Carl    1432 Morriston Ave.    Parkerville    PA   17214
Boehning, Greg   POB 41256              Santa Cruz     CA   95061
Chapel, Doug     75 Fleetwood Dr.       Rockville      MD   20852
Cook, Dottie     154 Sweet Rd.          Grand Prairie  TX   75050
```

In the example above, the word **.person** is defined similarly to the version given in Section 8.1, except that the field reference operator **?B** is used instead of **B?**. **?B** is the Report Generator version of **B?**, and takes the same stack arguments. The difference is that it performs "tabbing" based on a table of columns created by the word **[R** (third line of example). Section 8.8.3 lists all the output operators that use this table.

The word **[R** specifies both a title (the word "People," centered) and the column headings (the row of labels above each column). It also creates the column table mentioned above, leaving this address on the stack. See Section 8.8.2 for more on **[R**.

The final word, **all** prints the tabulated report. It begins by invoking **LITERAL** so that the address passed from **[R** will become part of this definition, then calls **LAYOUT**, which consumes this address.

+L (short for "plus-line") forces an extra carriage return into the report above the first row of data. Next, **PEOPLE** guarantees that the **PEOPLE** file is current whenever we display this report. **RECORDS** supplies the appropriate arguments for **DO**. Each time through the loop, the next record is made current with **READ**, and the row is displayed with **.people**. Then, **+CR** forces a new-line and checks whether the page is full—if so, the report generator automatically heads the next page.

The report also contains a page banner which includes some text at the upper-left hand corner of the page and the page number and date in the upper right. These are formatted automatically by **LAYOUT**, but are user-configurable.

REFERENCES

Controlling Paging, Section 8.8.4
Page Banner, Section 8.8.5

8.8.1 Specifying a Title/Column-Heading Pair

A single word, **[R**, lets you specify both the title and the column headings. The set-up phrase must appear in the source block, usually just preceding the definition of the report for which they are designed. This source text

must be available for the application when it is running (*i.e.*, mounted on a disk drive); if you need a ROMable version, contact FORTH, Inc.

The format for a title/column heading pair is:

```
[R title-text \column-heading-text ]
```

The entire title/heading pair statement may extend over multiple lines, since the Forth interpreter treats source blocks as contiguous 1K-byte strings.

All characters up to the backslash, except the first blank that follows **[R**, are used for the title text. The title text will be displayed approximately at the center of an 8-1/2" x 11" page, assuming that the title-text is about 30 characters in length (actually, the system outputs 20 spaces before typing this string to approximate the centering; this occurs in the word **+PAGE**).

All characters that follow ****, ending with the delimiter **]**, are used for the heading. The first character (blank or non-blank) that follows the **** corresponds to the first column of the report page.

In addition to being displayed, the heading text is parsed at the time the block that contains the heading is loaded, to produce a table of column widths and locations of the text to be displayed. This table is used by the set of words that output the contents of fields for the Report Generator; this wordset includes: **?B**, **?N**, and **?1**. Thus, each column "knows" where it should appear on the page and how wide it should be.

When displaying **BYTES** fields, it is necessary to ensure that the width of the heading text for that field matches the width of the storage field, plus a few extra spaces as desired for column separation. Any fewer spaces, or significantly more spaces, will result in a skewed output.

With numeric fields, caution should be exercised that the length of the field to be printed does not exceed the width of the column to be used. Should the actual size of a string exceed the column width, it will nonetheless be printed in full and the remaining columns permanently shifted right to accommodate it.

Also, make sure that no blank space intervenes between the last column heading and the delimiting]; this will cause an extra column entry in the table and may skew the output.

Finally, make sure that each heading does not contain a space, such as "Acct. No." because this will be interpreted as two column headings.

The address of the columns table is left on the stack at load time by **[R**; this is the address that must be passed to the word **LAYOUT**. **LAYOUT** initiates the printing of a report and specifies the type of page heading routine to be invoked. It also saves the address of the title/column heading table (in user variable **RPT**) so that each page of the report will display the same header information.

If the title/column-heading pair is to be used in several reports, the address of the table for the title/heading pair may be used as the value for a **CONSTANT**, thus giving a name to the title/heading pair:

```
[R A Report \Col1 Col2 Col3 ]
   CONSTANT 'SHOW'
: SHOW   'SHOW' LAYOUT  ... ;
```

Otherwise, it is more efficient to just keep this address for a **LITERAL** to compile as a literal in the definition that uses this report:

```
[R A Report \Col1 Col2 Col3 ]
: SHOW   LITERAL LAYOUT ;
```

This address may, of course, be **DUP**ed if more than one reference is required, provided the **DUP** appears outside any definition (and thus is executed):

```
[R A Report \Col1 Col2 Col3 ] DUP
: SUMMARY   LITERAL LAYOUT ... ;
: SHOW   LITERAL LAYOUT ;
```

REFERENCES

+PAGE, Section 8.8.4

8.8.2 Formatting Lines

To the report generator, a line consists of a series of columns, each of which has as fixed width. These columns are used to align the data to be printed, with all data right-justified in the current column.

The following words are provided by the Report Generator to display fields within the columns determined by the title/column heading pair:

Word	Stack	Action
.N	(n)	Displays the single-length integer *n* right justified in the next column, in the format used by **.** (dot).
?N	(a)	Displays the contents *a* address as a single-length integer *n* right justified in the next column, in the format used by . (dot).
?1	(a)	Displays the contents of the specified **1BYTE** field, right justified in the next column.
.D	(d)	Displays the double-length integer *d* right justified in the next column, in the format used by **D**.
?B	(n a)	Reads and displays a **BYTES** field, according to the declared length, left-justified in the next column. **PAD** is used as intermediate storage of the field.
.M/D/Y	(n)	Given a Julian date, displays it in the next report column. Since this routine invokes **(DATE)**. it will work with either calendar. Most data base applications prefer to use the mm/dd/yy calendar (Block 43).

Each of these operators advances the columns table to the next column, determines the width of the new field, then right-justifies the output string in this column.

You may also build your own formatting words to display columns, using the word **RIGHT**.

RIGHT	(a n)	Displays an alphanumeric string of length *n*, beginning at address *a*, right-justified in the next column.

The stack arguments are identical to those of **TYPE**.

In fact, **.N**, **?N**, **?1**, **.D**, and **?B** are defined using **RIGHT** and behave according to its rules:

1. If the length of the output string exceeds the width of the column, the results are unpredictable but will include loss of format control.

2. If the length of the output string equals the width of the column, the string is displayed and the column pointer is advanced.

3. If the length of the output string is less than the width of the column, the difference is output as blank spaces, so that the string will be right-justified.

4. Text strings are also right-justified; however their trailing blanks are included, making them appear left justified.

Here is an example:

```
 0 ( Accounts example)     FORGET TASK : TASK ;
 1
 2 0  10 BYTES NAME   NUMERIC ACCT#   DOUBLE BALANCE
 3
 4 : (.$) ( d - a n)   SWAP OVER DABS
 5    <#  # #  46 HOLD  #S  SIGN  #> ;
 6 : .ACCOUNT   ACCT# ?N  NAME ?B
 7    BALANCE D@ (.$) RIGHT ;
 8
 9 [R Account Balances\   Account# Name          Balance]
10 : balances   LITERAL LAYOUT
11    ACCOUNTS RECORDS DO  I READ  .ACCOUNT  LOOP ;
12
13 : enter ( n d)   ACCOUNTS  SLOT READ  BALANCE D!
14    ACCT# N!  NAME PUT ;
15  ( Example:  456 100.00 enter John Doe <RETURN> )
```

The word **BALANCES** produces:

```
FORTH, Inc.                          Page 1  26 AUG 1986
                                    Account Balances
   Account#  Name                                Balance
        456  John Doe                               100.00
        489  Mary Smith                            2970.00
        620  Ed Poore                                 2.59
```

Notice that the first column heading, “Account#” appears in the title/ column-heading pair three spaces after the backslash. This causes the heading on the output report to be indented three spaces (the first space after the backslash counts). On the corresponding formatted lines, the first field is formatted with **?N**, which right-justifies the string against the end of the “Account#” heading.

The middle column is formatted with **?B**; as a text string this field is effectively left-justified. To make the output more pleasing, we have forced the “Name” column heading to be flush left to match. The use of **RIGHT** in **?B** causes the column table to be adjusted the first time a row is displayed (see Section 8.8.6).

In the third column, the data is once again right-justified under the last character of the "Balance" column heading. In this case, we wished to display the double-length field in dollars-and-cents format, requiring the use of a pictured numeric output routine (Lines 4 and 5 of the listing). On Line 6, this pictured numeric output string is displayed, but with **RIGHT** rather than **TYPE**.

If the previous column displayed was the final column on a line, **RIGHT** automatically advances to the next line and resets the column table to begin with the first column on the line.

The following words are available for special formatting requirements:

Word	Stack	Action
0COL	( )	Resets the column table pointer to point to the first column width. Exercise care with this word, since it can cause the output to be misaligned if it is not issued when the actual output print position is at the beginning of a line.
COLS	(-- n)	Advances the column pointer and returns the width of the new column.
Word	**Stack**	**Action**
SKIP	( )	Skips one column.
SKIPS	(n)	Skips *n* columns.

8.8.3 Controlling Paging

The report generator counts each output line; when no available lines remain, it forces a page heading cycle by placing standard titling information at the top of each page.

The maximum number of lines which may be placed on a page is stored in the user variable **L/P** (lines/page). For the printer task, its value is normally sixty-six, which corresponds to the standard page size of 8-1/2" x 11". In areas of the world where the standard page is not eleven inches, this value must be set appropriately.

For terminal task, the value of **L/P** is usually twenty-four.

To change this value for a terminal task, simply store the new value into **L/P**:

```
60 L/P !
```

To reset the value for the printer task, you may either place a similar phrase inside a definition that is executed by the printer task (as in the definition of a report that you are printing), or with the following phrase (see Section 4.6):

```
60 TYPIST L/P HIS !
```

When using the Report Generator, it is not necessary to explicitly invoke a "new-line" function at the beginning of each row of data. As the field-display operators cycle through the columns table, after the last column has been displayed, the next operator resets the column pointer to the beginning of the column table again and issues a "new-line."

The following words control pagination:

Word	Stack	Action
+PAGE		Starts a new page, incrementing the page count in **P#** and displaying the headings for the new page.

Word	Stack	Action
?PAGE	(n)	Starts a new page if there are fewer than *n* spaces remaining; otherwise issues a "new-line."
+L		Issues a **CR** and increments the line count in **L#**. Also resets the column pointers using **0COL**.
+CR		Checks whether the page is full (using **1 ?PAGE**), then issues a **+L**.

You can ensure that a set of related lines will appear together on the same page, by using **?PAGE** before each set to guarantee that they will all fit. For example, if the report consists of groups of three lines that should always appear together, issue the following phrase immediately before displaying each three-line unit:

```
3 ?PAGE
```

Within the three lines, you would use **+L** rather than **+CR** to avoid a possible page break within the group.

You can unconditionally force a new page by providing an argument that appears to be greater than the largest possible number of lines per page:

```
-1 ?PAGE
```

If you wish to suppress automatic paging totally (for example, in cases where the output is directed to a terminal that is not a printer, where page headings are not desired), you may use the word **+L** in place of **+CR**. The words are the same except that **+L** does not check for page full.

Consider the code example in Section 8.8. Here **+L** is used at the beginning of the report, but **+CR** is used after each line inside the loop, to enable automatic pagination. Neither **+L** nor **+CR** is inside **.person**, because this definition (which simply displays the current record) may be used for a single-record terminal inquiry, as well as the many-record report **all**. The terminal inquiry might look like this:

```
: show ( n)   PEOPLE READ  .person +L ;
```

Here **+L** is preferred because we do not want to paginate every few inquiries!

REFERENCES

0COL, Section 8.8.3

8.8.4 The Page Banner

At the top of each page of the report appears the "page banner" which includes:

```
<optional text>                    Page nn  <date>
```

where *nn* is the current page number, and **<date>** is the current system date.

If you wish to modify or eliminate the optional text, simply change the **."** string in the definition of **+PAGE** (Block 133).

It is possible to eliminate the page banner entirely by replacing the word **LAYOUT** with **HEADING**. Like **LAYOUT**, **HEADING** takes as an argument the address of a title/column heading table as provided by the word **[R**, and

establishes this table as current. It then displays the "title" line, without attempting to center it, and on the next line displays the column headings.

HEADING (a) Saves the address of a title/heading table, and outputs the title and column headings.

8.8.5 How the Columns Table Works

The format of the columns table is:

Byte	Contents
address + 0, 2	Disk location of the title/heading pair (character position and block number).
+ 4	Address of the page heading vectored routine.
+ 6	First column width.
8 *et seq.*	Subsequent column widths.

A zero entry in the table indicates the end.

A heading line can contain up to 128 characters. These lines are used to establish a table of column widths at load time in the following manner.

Starting from the backslash in the title/column-heading pair, **[R** scans forward using **32 WORD** in a loop. Each time it encounters a heading, it continues to scan to the first blank character following it, computes the difference from the starting point or previous heading (the width of the field), and compiles this into the table. This loop repeats until the **]** delimiter is encountered. At this point, the indicator for the end of the line (a column width of zero) is inserted and the scan is complete.

For example, suppose the following is the set-up string for a set of column headings (the numbers across the top are your guide to indicate column positions):

```
0         1         2         3         4
01234567890123456789012345678901234567890123456789
\   Account# Name      Balance]
```

The first blank after the "Account#" heading occurs at relative position 12; thus the number 12 is compiled into the table as the width of the first column. The first blank after the "Name" heading occurs at relative position 17; the difference, 5, is compiled as the width of the second column. The delimiting **]** occurs at 34, and the difference of 16 is compiled as the width of the third column. Finally, a zero is compiled to indicate the end of the table.

The finished column table, as constructed by **[R**, contains:

```
12  5  16  0
```

The total width of all columns equals the position number of the last non-blank character.

Notice that some hidden "sleight-of-code" is taking place with the middle column. In our **BALANCES** report, the full 10 characters of the name field appear. What happens is that the first time **BALANCES** invokes **?B**, the word **RIGHT** (which is invoked by **?B**) determines that a width of five is not sufficient for the 10-character **BYTES** field plus the single space between columns. Accordingly, **RIGHT** adds six to the five, to achieve the necessary eleven spaces needed. Finally, to keep the rest of the columns lined up, **RIGHT** subtracts six from the next column ("Balance").

Thus, by the time the first row of data has been displayed, the column table contains these entries:

```
12  11  10  0
```

If you crowd the headings too close together in the title/column-heading pair, **RIGHT** will "steal" so many spaces from the next column that it will contain zero. If this happens, the column table will appear to be terminated at that point, and there will be more output operations on each line than there are columns in the table. The result will be a skewed output.

A line may contain as many columns as required for the output format. Due to the method of establishing columns, the minimum width of a column is two characters.

8.8.6 Non-standard Report Headings

By default, the "new-page function" performs the following steps at the beginning of each page, including the first page:

1. Displays the page banner as described in Section 8.8.5;
2. Performs a **+L** and skips 20 spaces (to approximately center the title);
3. Executes a word called **TITLE**. **TITLE** is defined as:

```
: TITLE   RPT @ HEADING ;
```

RPT is the user variable that points to the current title/column heading table. **HEADING** displays the title and column-heading lines from the given table (Section 8.8.5).

However, the Report Generator lets you vector the third function above. This feature lets you execute your own definition instead of, or in addition to, **TITLE**. For instance, you might add other lines of information below the page banner.

This vectoring is possible without recompiling the **FILES** utility because the third cell of the title/column-heading table contains the address of the routine to be executed at the top of each page. When **[R** generates this table, it copies in the address of the routine **TITLE** by default. By re-setting this address to point to your own definition, you can change the output of the new-page function.

Here is an example:

```
0 ( A Non-standard Report Heading)
1 VARIABLE WHICH
2 [R   \Col1   Col2   Col3]  DUP  4 +
3 : 'ITEM'   LITERAL ASSIGN  ." Report on Item No. "
4    WHICH ?  +CR  TITLE ;
5 : SHOW ( n)   WHICH !  'ITEM'  LITERAL LAYOUT ;
```

This example shows a report for some particular item that is selected numerically, like this:

```
2500 SHOW
```

Stores *2500* into **WHICH** so that you can see a report for Item 2500.

The report generator will print the item number at the top of each page, with headings:

```
Acme Manufacturing Co.              Page 1  26 AUG 1986
```

```
                    Report on Item No. 2500

Col1      Col2      Col3
```

where the top line is the standard page banner, and the text "Report on Item No.___" is formatted by user-defined code.

Here are the steps used in the above example to vector the user-defined code into the new-page routine:

1. Create a title/column heading pair as usual (in this case, we have left the "title" blank).

2. Following the creation of the title/column heading pair, invoke the phrase **DUP 4+**. This computes the address of the execution vector in the table, in addition to the starting address of the table.

3. Define a word that vectors this address to the code you wish to execute. In the definition of **'ITEM'** above, the word **LITERAL** makes the vectored-execution address part of the definition. (**LITERAL** consumes the "address 4 +" during compilation.) **ASSIGN** causes that address to point to the code following **ASSIGN** itself (Section 2.3.8.2).

4. The rest of the definition is the code that will be executed as the third step of the new-page routine. It includes the message "Report on Item No.," followed by the display of the chosen item number. It then invokes **+CR** to move to the beginning of the next line. Finally it invokes **TITLE**, which displays the title/column-heading pair. In this case, we have specified a blank "title," so the title line is blank.

5. Define the report-generating word (the word SHOW) in the usual way, using the address remaining on the stack (the beginning of the table) as the argument to this second occurrence of LITERAL.

Having once executed **'ITEM'** in the above code, the user-defined routine is now a permanent part of the **SHOW** report. However, other reports may be co-resident; since each has its own title/ column-heading table, each has its own new-page execution behavior.

8.8.7 Totals and Subtotals

The Data Base Support option includes a simple utility for computing subtotals and totals of numeric fields as the report is being displayed. In general, the following steps must be followed:

1. Allot enough "working storage" for the registers. Working storage is created by invoking **n ALLOT** immediately after the word **EMPTY** at the beginning of the **FILES** load block (Section 8.6). The value of *n* is calculated by this Forth phrase:

   ```
   ( # of registers needed) 8 *  4 +  16 +
   ```

   See the shadow block associated with the word **REGISTER** on your system.

2. At the beginning of your report word, simultaneously define and clear as many accumulator-pairs as there are fields you wish to total, using the word **ZERO** (see below).

3. As the fields are being displayed, accumulate the values in the subtotal registers by using either **SUM** or **FOOT**.

4. When you wish to display the subtotals (if at all), use the word **SUB**, followed by an appropriate numeric output command.

5. When you wish to display the grand totals, invoke **GRAND**. This copies the grand totals to the subtotals registers. Then use **SUB** as in Step 3.

Here are the relevant words in detail:

Word	Stack	Description
ZERO	(n)	Defines *n* subtotal accumulators, and *n* grand-total accumulators, and sets all to zero. Each accumulator is double-length. For example, if you are totaling three fields, the phrase: **3 ZERO** creates three subtotal accumulators and three grand-total accumulators, and sets all to zero. **ZERO** must be used at the beginning of a report if any of the following words are used.
SUM	(d n)	Adds *d* to the subtotal accumulator for the nth relative field.
FOOT	(d - d)	Advances to the next subtotal register and adds *d* to it. If at the last register, wraps around to the first. For instance, suppose you have a **DOUBLE** field called **SALARY** that you want to both display and add to the running total. The phrase: **SALARY D@ FOOT .D** fetches the contents, adds it to the corresponding subtotal register, then displays it.
SUB	(-- d)	Advances to the next subtotal register and fetches its contents. Also adds the contents into the corresponding grand-total accumulator and clears the subtotal register. If at the last register, wraps around to the first.
GRAND	( )	Copies the grand totals to the subtotal accumulators. For example, the phrase: **SUB .D** will display the subtotal of the next field. The phrase: **GRAND SUB .D** will display the grand total of the next field.

The following example shows how subtotals and grand totals can be easily computed and displayed:

```
          Wine Inventory by Store
Location                  Chablis      Rose   Champagne

Northern California
   Palo Alto                   25        42          78
   San Jose                    16        32          50
   Mill Valley                 31        29          36
   San Francisco               70        59          82
                              142       162         246

Southern California
```

```
   Chatsworth                  35         48          29
   Woodland Hills              32         40          60
                               67         88          89
Grand Total:                  209        250         335
```

Here is the code that produced this display:

```
 0 ( Totals and Subtotals)   FORGET TASK   : TASK ;
 1 0  16 BYTES Location   NUMERIC Chablis   NUMERIC Rose
 2     NUMERIC Champagne   DROP
 3
 4 : .amounts   Location ?B  Chablis N@ 0 FOOT .D
 5    Rose N@ 0 FOOT .D  Champagne N@ 0 FOOT .D
 6 : .subs   SUB .D  SUB .D  SUB .D  +CR ;
 7 [R Wine Inventory by Store\Location         Chablis
 8 Rose  Champagne]
 9 : INVENTORY   LITERAL LAYOUT  3 ZERO  +CR
10    ." Northern California"  +CR
11    WINES RECORDS DO  I READ  .amounts  I 4 = IF  +CR
12      SKIP .subs +CR  ." Southern California " +CR
13      THEN LOOP  +CR
14    SKIP  .subs  ." Grand Total:     " COLS DROP
15    GRAND .subs ;
```

The phrase **3 ZERO** appears in the definition of **INVENTORY** on Line 9. This creates and clears three sets of accumulators, one set for each field we wish to total.

The word **FOOT** appears in the definition of **.amounts** on Line 4:

```
Chablis N@ 0 FOOT .D
```

In this case, the field is **NUMERIC** (single-length), so we fetch it with the operator **N@**. **FOOT**, however, expects a double-length number; the zero preceding it supplies the high-order part. **FOOT** will add the value to the first subtotal accumulator. **FOOT** also returns a copy of the value (as a double-length number). Finally **.D** displays the value in Report Generator format.

The second invocation of **FOOT** in:

```
Rose N@ 0 FOOT .D
```

will cause the value of the **Rose** field to be added to the second accumulator, and so on.

The word **SUB** appears in the definition of **.subs** on Line 6. This definition displays the contents of the three subtotal accumulators in turn. Notice that the three uses of **.D** correspond to the second, third, and fourth columns in the report generator; thus we can only invoke **.subs** when we are about to display the second column (after having output or **SKIP**ped the first column).

In **INVENTORY**, we display the standard header with **LAYOUT**, below which we display the category heading "Northern California."

Inside the loop we display the fields in the usual way, except that we check to see if the index is 4. If so, then it is time to display the subtotals for Northern California and the category heading for Southern California; the code for this appears on Line 12. Here we **SKIP** the first Report Generator column, then issue **.subs**.

After the loop has been completed and the second set of records displayed, the phrase:

```
SKIP .subs
```

(Line 14) displays the subtotals for Southern California, and issues a **+CR**.

Finally we display the text "Grand Total." The trick here is that we also want to display the grand totals on the same line. We cannot use **SKIP**, because it outputs the necessary number of spaces to get to the next report column; after printing the text, we're half the way there already. Our solution is to pad the message with trailing blanks so that the message is 17 characters long (the width of the first field plus one); this leaves us in position to display the second column.

However, the columns table must also be advanced to point to the second column. The phrase:

```
COLS DROP
```

is the same as **SKIP** without issuing the spaces.

Finally, **GRAND** copies the grand-total accumulators to the subtotal registers, and **.subs** displays these.

8.9 DATA BASE DESIGN

Before building a house, it is best to have a blueprint. So too, before defining files and records, it is best to map-out the overall data base needs.

In general, we can formulate two simple rules for planning your data base:

1. Look at the kinds of information you have.

2. Arrange like kinds of information into files.

8.9.1 A Hospital Patient Management Data Base

Our goal in this example is to create a data base for tracking patients in a large hospital. For each patient there is a set of information: items such as address, height, weight, date-of-birth, and so on. (Note that we save date-of-birth and not age. We can always compute age if that's what we need in a report, but a date-of-birth is never obsolete.)

Clearly, this information all belongs in a single record, one per patient. However, there is also a variable number of information items that may be associated with each patient. For instance, each patient may have a different number of tests, and each type of test may have a different amount of information that it produces. In short, the amount of information that we need to keep for each patient is variable in length.

At this juncture, many data base designers would opt for variable-length records and fields. But variable-length records are complex and slow, as we saw in Section 8.1. With nearly the same convenience we can achieve the same results by using a fixed header plus a variable number of subordinate records.

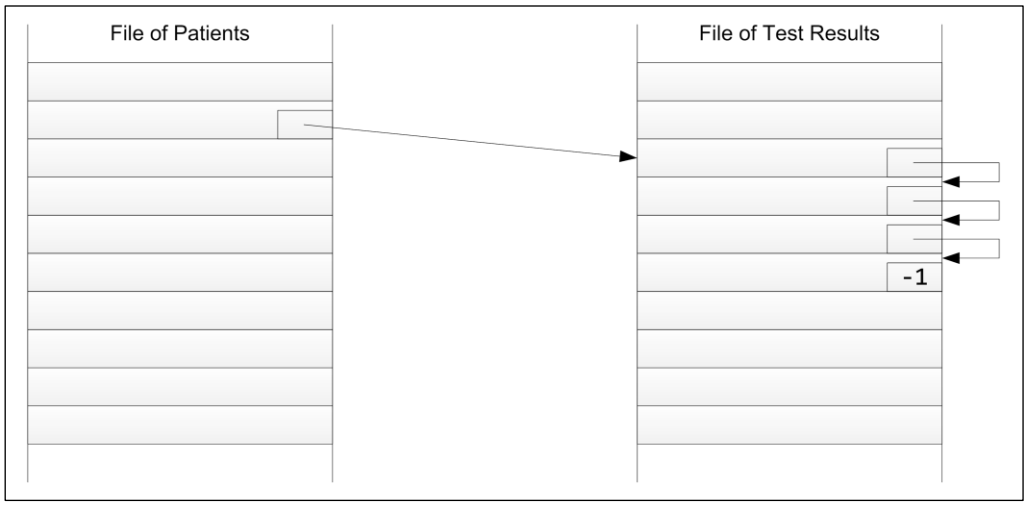


Fig. 8.13

Patient records chained to test results.

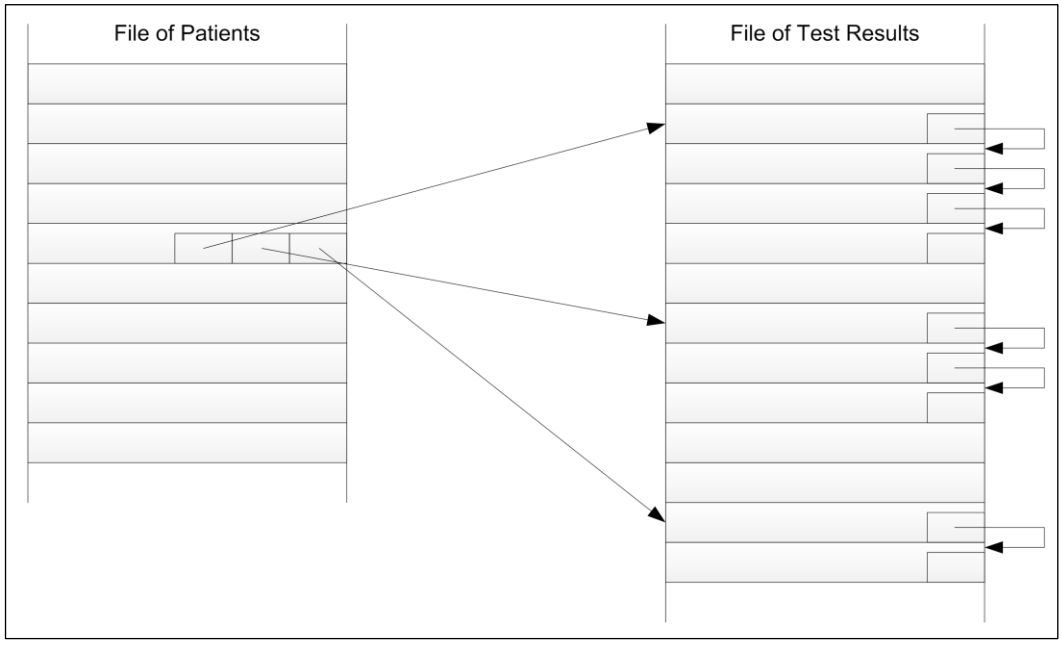


Fig. 8.14

Three “test results” chains for each patient.

This is where chains come in. Fig. 8.13 shows that a field in each **`PATIENTS`** record can point to the first in a series of **`RESULTS`** records, each of which is chained to the next. We achieve the same effect, but at much less expense.

Now suppose that we need to record particular test results for three different tests for each patient. We can accomplish this by providing three fields in each **`PATIENTS`** record, each pointing to a different result record or chain of records (Fig. 8.14).

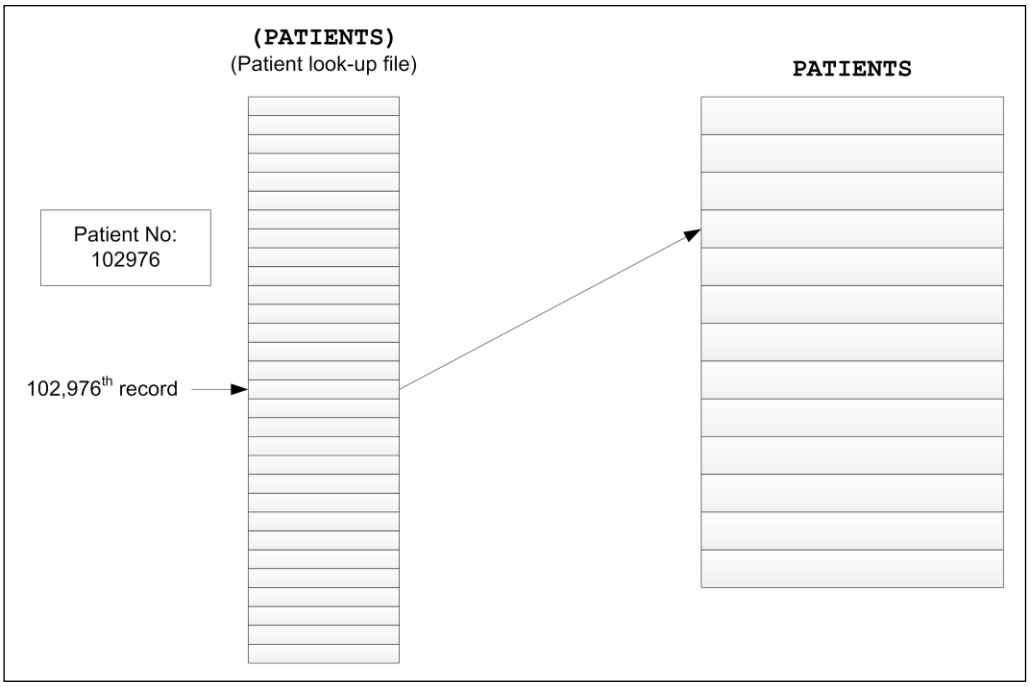


Fig. 8.15

Patient numbers directly index a record in **(PATIENTS)**, which contains a link to the main **PATIENTS** file.

Here's another intriguing problem. The application demands that a patient record can be found on the basis of a "patient number." A patient number is a very large number issued in sequence; in other words, the patient number last issued reflects the total number of patients that have ever been admitted to the hospital in its history. This number could reach 200,000 during the lifetime of our system. However, the department for which we are designing this application expects to see only 30,000 patients during the lifetime of this system. Unfortunately, maintaining an ordered index even of 30,000 records, indexed on "patient numbers," is unmanageable.

Is there any way we can translate a patient number directly into a record number for our **PATIENTS** file? Let's try this: we'll create a file of 200,000 records, each record being only two bytes long. This gives us one record per potential patient number. The two-byte field will contain a record number, pointing to the record in the **PATIENTS** file corresponding to the patient number (Fig. 8.15).

This elegant scheme requires 400 blocks for the look-up file, and yet gives immediate access to a patient record, with only one intermediate disk access. No searching is needed. Furthermore, when new patients are added, **SLOT** is not needed in the look-up file.

In general, direct access is much faster than searching, and should be used whenever appropriate.

8.9.2 An Integrated Business System

Our goal in this example is to create a package that will track income (sales and accounts receivable) and expenses (purchase orders, accounts payable, and payroll), and from these inputs will produce general ledgers, income statements and balance sheets.

Although many commercial business packages treat these functions as separate programs, our goal is to integrate them into a single system. By doing so, we will make the system simpler to use and reduce the opportunity for

error. For instance, when a sales order is entered, the order should be forwarded to the accounts receivable component, and the sale automatically posted to the general ledger without further manual entry.

How shall we organize our data base? Let's begin by identifying the entities and operations that are part of our business, and the reports that we wish to obtain:

```
Entities:   Operations:          Reports:

customers   sales:                 general ledger:
vendors       order entry            income statement
employees     accounts receivable    balance sheet
              payments received
            purchases:
              purchase orders
              accounts payable
              checks written
            payroll:
```

Looking first at the left column, clearly we will need to store information about the entities in a file structure. The question to ask is, "What do we need to know about these entities?" It turns out that for each of our three types of entity, the answer is remarkably similar. In each case we need to know:

name
address (street, city, state, zip)
phone number

This observation suggestions the possibility of using shared code, an opportunity for program simplification. At the very least, this means we can use the same field definition names (**NAME**, **STREET**, etc.) for three different files.

In fact, though, we never have more than several hundred people and companies that we do business with in any year. As a result, we can mix all people and companies in single file, called **PEOPLE**, and add an extra field called **KIND** to indicate whether the entity is a customer, vendor, or employee.

This reduces the number of files for "entities" from three to one, and simplifies the program accordingly.

Because we will need to search and order this file on an alphabetical basis, we must also create an index file, called **(PEOPLE)**. This index will contain simply a link field—to point to the corresponding record in **PEOPLE**—and a "nickname" field, which contains the name in a form that we want it alphabetized by.

We can establish the following rules for entry of the "nickname" field:

```
for human beings:  last name, first name, initial
for companies (customers or vendors):  company name
   (sometimes somewhat abbreviated)
```

As for the additional fields that employees need, we find it simplest to create an additional file called **AUXILIARY**. Each employee record contains a pointer to a record in **AUXILIARY**.

Now let's turn to the operations. Each operation results in a transaction that must be saved. These transactions will become records in a file of *events*. What do we need to know about these events? In the case of a sale, we have:

```
customer
date of sale
amount
check number
```

In the case of a purchase order, we have:

```
vendor
date of order
amount
purchase order number
```

In the case of payroll, we have:

```
employee
date of paycheck
amount
check number
commissions (for commissioned salespeople)
tax contributions, etc.
```

Once again, it appears that many fields exist in common. With the exception of the extra information needed for payroll, we can summarize the above requirements as:

```
WHO
WHEN
AMOUNT
NO.
```

We decide to keep all events in a single file. We will call this file **DETAIL**. Besides the fields described above, we will add a field called **KIND** to indicate whether the event is a sale, an order, etc.

When we organize our data needs in this way, we see that entities and events can be organized together for simplicity. With this understanding, it will be easier to integrate the entire system.

Let's look at the **WHO** field. What should it contain? Perhaps the name of the person or company.

On the other hand, we know there will be many more **DETAIL** records than anything else, so we want to make each record as small as possible. Were we to keep a name field in the **DETAIL** file, it would take up considerable space and require that we look up the name in an index in order to get the address or other information on the name.

Instead, we will keep the record number of the related person or company in the **WHO** field. This occupies only two bytes, and requires no searches.

Now let's study some of the operations we'll want to perform. Suppose it is the end of the month and time to write checks. This is easy. We simply look through the **DETAIL** file looking for accounts payable entries that are due now. From the record in **DETAIL** we can follow the pointer into **PEOPLE** to get the name and address of the payee.

Let's take another example. We want to be able to determine the current balance owed by a particular customer or to a vendor. But we have not included a "Balance" field in the **PEOPLE** records. All we have to do is let each **PEOPLE** record point to the most recent transaction, then let each transaction record point to the next-most-recent transaction, etc. Here we are using chains.

```
ACME Widgets, Inc.                Page 1  31 OCT 1986

                    So. Bay Office Supply Account Status
 #  Job   Ref     Due    DR#   CR#   Amount   Paid Balance
3344    1086 29 NOV   5220  2100   189.24   0.00  189.24
334347   626 29 NOV   1210  2100    10.74   0.00   10.74
3205    2270 10 OCT   2100  1030   779.74 779.74    0.00
277361   930 30 SEP   1210  2100    59.04  59.04    0.00
```

Fig. 8.16

Portion of a report showing a vendor account. The first column shows the number by which each detail item is referenced; it is actually the number of the record in the **DETAIL** file. The report title is a 'custom' one, showing the subject account. Custom report titles are described in Section 8.8.7.

Chaining is appropriate in cases such as this, in which there is no way to predict how many elements there will be, and it makes it easy to generate reports of activity for a vendor such as the one in Fig. 8.16.

There are at least three ways that chaining can be done:

1. Chaining from most recently entered transaction to least recent.
2. Chaining from least recent transaction to most recent.
3. Chaining by something other than order of entry, such as date field, etc.

In this case, we prefer to list transactions starting with the most recent events. This makes possible reports such as shown in Fig. 8.16. As we saw in an Section 8.7, the polyFORTH Data Base Support option includes a block of chain manipulation words that you can customize for your particular application.

So far we have a **PEOPLE** file and a **DETAIL** file. Now let us look at our desired reports.

The general ledger is produced monthly, organized by account. Under each account are itemized all transactions, both credits and debits involving that account during the month. In the balance sheet (Fig. 8.17), we show year-to-date summaries for each account.

```
ACME Widgets, Inc.                Page 1  31 OCT 1986

                                Balance Sheet
CURRENT ASSETS
    CASH
    Continental Bank                 24,165
    Amalgamated Bank                104,965
    Short Term Investments          248,000
    Petty Cash                        5,000  382,130
```

Fig. 8.17

Portion of a Balance Sheet report.

The traditional data base approach to General Ledger might involve running, once each day, some program that looks through the latest events and posts them to another file containing the general ledger data. To produce the general ledger at month's end, this approach would require sorting the transactions file by accounts.

But daily posting not only requires creating another file, it also involves an extra step for the bookkeeper. And sorting would take longer because it requires handling each record several times. As we've mentioned, the polyFORTH Data Base Support option offers many techniques that reduce the need to sort.

Still, each transaction record must indicate the account it affects. When we produce the general ledger and income statements, we will need to display accounts by name, *e.g.* "Advertising." It doesn't make sense to place an account name in each transaction record. Following our dictum, "Arrange like kinds of information into files," we create a new file called **ACCOUNTS**, containing the name of each account. Now our transaction records can simply use a record number to point to an account record.

Actually, each transaction affects two accounts, one as a debit and one as a credit. So, each transaction really needs two pointers. We will call one **DR#** (debit record number) and the other **CR#** (credit record number). As each transaction is made, we know what accounts are involved. For example, when we enter an invoice, the amount will be credited as a sale and debited as an account receivable.

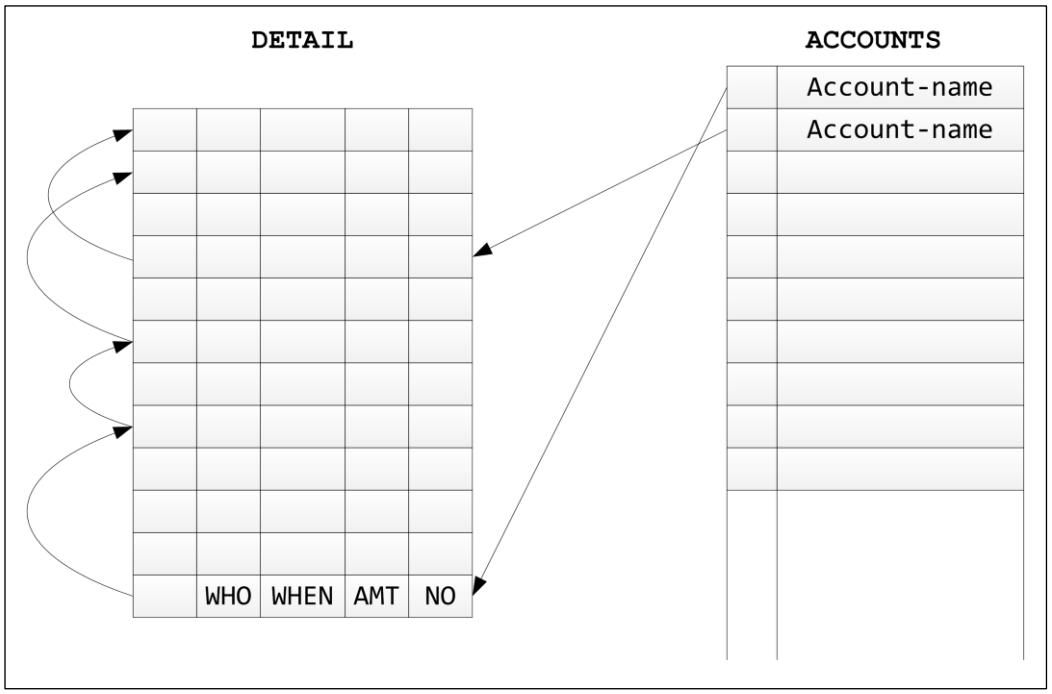


Fig. 8.18

Chains can be used to link transactions affecting each general ledger account.

Since each transaction points to a pair of account records, when displaying a transaction we can also display the account names.

What about a general ledger file? Upon further analysis, we realize that numbers don't need to be transferred to another file, when they are in one file already. It will be easier to run a program once a month that computes the account balances.

To run the general ledger, we need to start with each account, then look at the transactions that affected that account to produce an account summary. We could loop through the **ACCOUNTS** file, and for each account record, step through each transaction that affected the current account during the month. This approach could also use chains. Each **ACCOUNTS** record would point to the most recent transaction affecting it, and each transaction would point to the previous transaction affecting the same account (Fig. 8.18).

However, chaining is somewhat complicated, and better avoided whenever possible. Rather than chaining from **ACCOUNTS** to **DETAIL**, we can simply loop through our **DETAIL** file for this month. Each **DETAIL** record points to a pair of **ACCOUNTS** records. For each transaction, we can *add* the amount to an accumulator for a credit account, and *subtract* the amount from an accumulator for a debit account. In this way, we can tally *all* our account totals by looping through the **DETAIL** file only once.

But where do we keep these accumulators? Since we need one and only one for every account, it makes sense to add a field called **BALANCES** to our **ACCOUNTS** records.

Is this idea really better than following chains? By following chains from **ACCOUNTS** to **DETAIL**, we would have to handle each transaction record twice: once while following a credit-account chain, and once for a debit-account chain. By keeping balances, we can loop through our transactions only once.

By using a one-pass posting algorithm with no chaining, we improve performance a great deal by avoiding sorting, and by about a factor of two by not using chains.

Our **ACCOUNTS** file can use some embellishments. In addition to the two fields it already has:

```
Account No.
Balance
```

we can add **HISTORY**, which is an array of balances for the past 12 months.

In addition, the **ACCOUNTS** file needs an index, which we will call **(ACCOUNTS)**. At first it would appear that we could use the account numbers themselves to sort the accounts when preparing the balancing statement. In fact, however, accountants prefer to subclassify accounts into groups for their own reasons. For instance, taxes are an expense account, but they are usually listed at the end of the list of expense accounts. For this reason, the **(ACCOUNTS)** file is numbered according to the order in which we want accounts to appear on the balance sheet.

Our next step is to write words that reflect the kinds of high-level actions the bookkeepers want to record. Let's start with the operation of placing an order. How must this order affect our data base? What do we need to know?

Clearly we are going to create a new **DETAIL** record. This record will include a **WHO** field to indicate the company from which we are ordering. Since we have our vendors in the **PEOPLE** ordered index file, we need supply only the name of the company. The program can then look up the company, find the record number and place it in the **WHO** field of the new transaction record. The program must also link this new transaction into the chain for that vendor.

We also need to supply the amount of the purchase, and our purchase-order number.

The program itself can place the current system date into the **WHEN** field, and by default, place the date 30 days hence into the **DUE** field. Since this is an order, the program must place the code for a purchase in the **KIND** field.

So what should our "program" for entering an order look like to the bookkeeper? We know the bookkeeper must supply:

1. The amount.

2. The purchase-order number.

3. The name of the vendor.

The simplest, most Forth-like solution is to call the word **BOUGHT**, precede it with the two numeric data items and follow it with the string data. This gives us the syntax:

```
200.00 5134 BOUGHT ACME
```

We can now take a similar approach with a program to record a sale:

```
3998.00 7409 SOLD CROFT
```

The word **SOLD** is preceded by amount and their purchase-order number, and followed by the name of the customer.

We can record the receipt of a check with the word **FROM**:

```
amount check# line# FROM Conway
```

In the above, **line#** is a number that identifies the sale for which this is a payment received. The bookkeeper finds this number on a report of outstanding balances (see Fig. 8.16). While this is simple for the bookkeeper, it is also simple for the program because **line#** just happens to be the record number of the **DETAIL** record showing the sale.

The same syntax can be used for writing a check:

```
amount check# line# TO ROSS
```

Thus, each "program" is simply a Forth word. This approach allows our application to use the Forth interpreter. The problem of how the bookkeeper selects a given operation is effectively eliminated.

To appreciate the significance of this, consider the typical alternative. Most business applications are menu-based. From the main menu, the bookkeeper might select Accounts Payable. Then, from the Accounts Payable menu the bookkeeper might choose Purchase. From there, an entry form might appear, wherein the bookkeeper can select or enter the customer, then fill in the data.

While popular, this menu-based approach can be more laborious for the user. To avoid the switching application modes, the bookkeeper may separate all the purchases from the sales, etc., and do each group one at a time. This requires more paper shuffling.

Our approach, with no hierarchy, lets the user enter various transactions in any order, leading to a more pleasant, efficient working environment. A "help screen" can display the syntax of the commands on request during the learning curve.

In retrospect, we seem to have designed the data base very efficiently. The file with the most records, **DETAIL**, also has the smallest records. Each record in **DETAIL** is only 16 bytes long, and contains no text at all. (This means that 64 such records will fit in a block.)

8.9.3 A Facility Management System

In this example we will see how to organize and simplify a massive data problem by studying the data and looking for a natural hierarchy.

The example involves the problem of controlling digital and analog input/ output with a distributed computer system, where there are several thousand I/O points in dozens of buildings and other locations at a large industrial plant.

Digital "points" include switches, buttons, pressure-sensitive floor plates, pulses to unlock doors, and so on. Analog points include thermocouples, meters on control panels, heating levels, lighting levels, and so on. Our task is to install a distributed computer system to control all these points.

We begin by studying the points as the architects and engineers designated them. The ID for an individual point has the form:

```
ABC-123-1234
```

Experience has taught us that numbers such as this are usually encoded, and that usually the coding scheme presents a goldmine of information on how to organize the system. Upon further investigation we discover these relationships:

```
ABC     -     123     -     1234

a facility   a control panel     a point number on a
                                 control panel in a facility.
```

This information provides the key for our establishing a hierarchical data base, a necessary strategy when dealing with thousands of anything. Another example in which coded numbers can reveal hierarchy is with inventory or parts numbers.

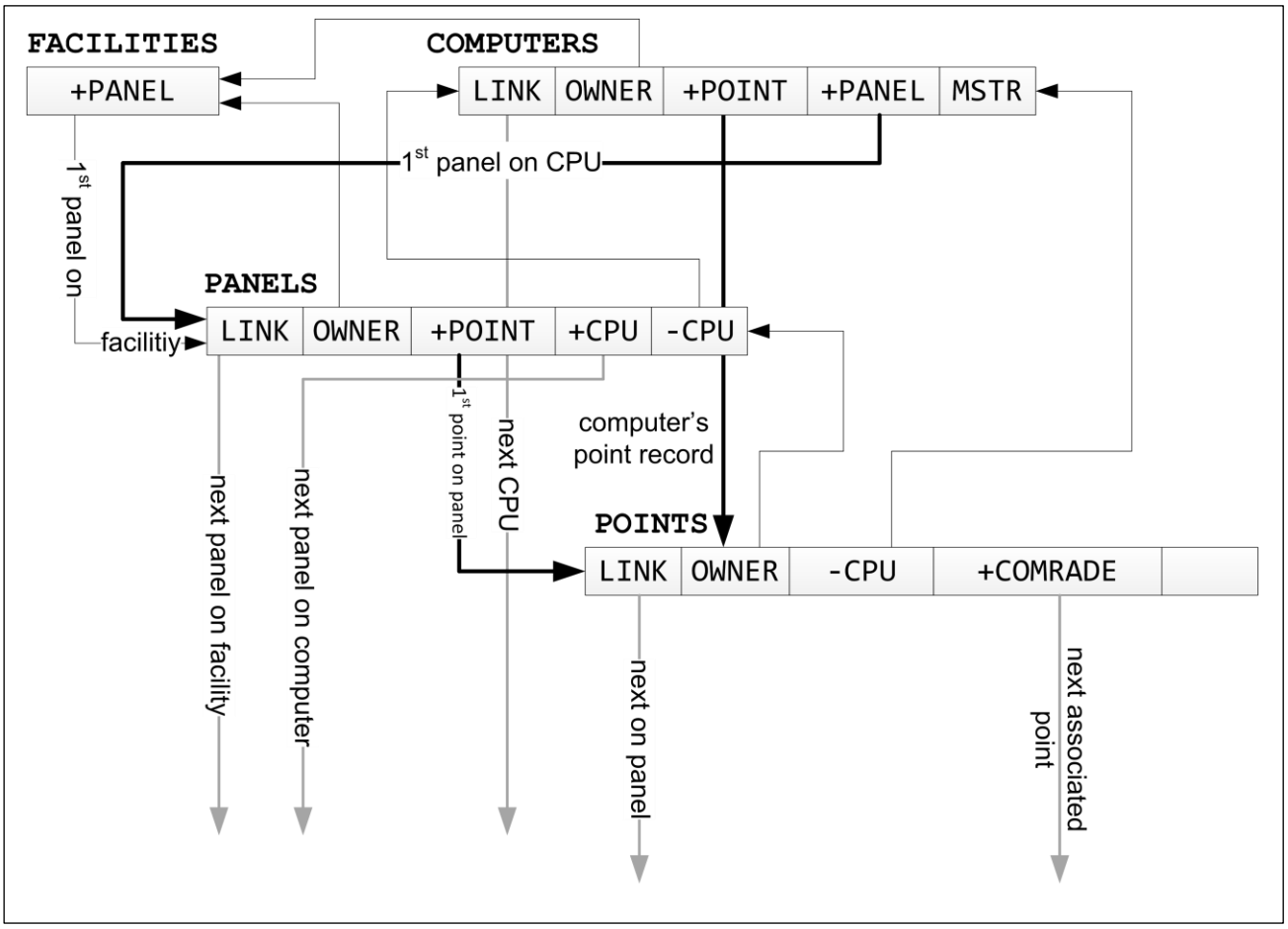


Fig. 8.16

Diagram of the data base for the Facility Management example.

One of the things we must do is allow the control of individual points from the central computer. The operator can enter a point ID and indicate some action. So one of our problems is to associate a point ID with a physical hardware location. One approach would be to have a points index contain-ing the ID of each point in the system. The nature of the coding scheme makes it a candidate for an ordered index, but with over 20,000 records even a binary search is more cumbersome than we can afford. The logical hierarchy will help deal with the volume, and thus the performance. It will also improve the information content; for example, in reports we can make use of the implicit meaning of the data.

The first part of the code is a "facility:" A physical building or location on the plant, like the parking garage, the fire station, etc. A "panel" may be a manually-attended console, a switchbox in a closet, or it may be imaginary, as in the case of points attacked directly to the computer.

A single panel may control many points. And a computer may control many points in many buildings (for instance, the fire station computer needs fire alarms in all buildings). But what is the relationship between computers, facilities, panels and points?

Further digging uncovers fact that a panel is attached to only one computer. This means that each computer can be responsible for its own private data base of panels. Each panel can be chained to any number of points. The master computer can have a file of facilities. Each facility can be chained to any number of panels.

We can now identify four files:

```
FACILITIES
COMPUTERS
PANELS
POINTS
```

We have solved the problem of chaining from computers down to points. Now let's consider the reverse problem. Inputs generate "events." An event has to be dealt with quickly; for instance, the event may be an alarm. If a fire alarm is triggered, a computer will need to display information about the point, such as which building it is in. In other words, we must have linkage from point to panel to building.

The first step of this linkage is the association between an electrical event within the computer and the corresponding point in the **POINTS** file. For instance, the pressing of a button might cause execution of an interrupt routine. This routine must be able to determine which point caused the interrupt.

At first it may seem logical to keep a table that associates hardware addresses with point names. But this would require an extra search. It is more direct to create a table that associates points' hardware addresses with record numbers within the **POINTS** file. Another benefit is that the point record number only requires two bytes, so the table is small.

With this scheme, an electrical event is associated with a record in **POINTS**, which in turn contains the information we need to know about the point, including its code name. Because the code name contains meaningful information, we can now determine which building the point is in.

We can now rest assured that we have found a good solution to the problem, since we have achieved good performance while at the same time reducing complexity. This sort of win/win situation provides the positive feedback that tells us we're on the right track as we iterate through our design.

8.9.4 A Filing Scheme for Image Processing Applications

Our final example illustrates the flexibility the Data Base Support option provides—including the freedom to *not* use some of its features when the application dictates otherwise!

Conceptually, a filed image has two elements, a header and the pixel data. The header indicates what the image is, when it was recorded, who made the image, the dimensions of the image (in pixels), and so on.

There are a variety of ways to index into images. But the real problem is managing the pixel data. Image processing is a prime example of an application in which speed is critical, because there is simply so much pixel data to handle. An array of 512x512 points contains 262,144 pixels, which at 8 bytes per pixel occupies 256 blocks. Just reading this many blocks will take some time.

Now imagine trying to access these pixels one at a time using **1@** (or **N@**). This approach involves the invocation of **BLOCK** plus the record and field accessing computations for each and every pixel. This will be unacceptably slow.

An approach that has proven effective is an interesting hybrid of the Data Base Support option tools, plus ordinary direct disk-access techniques. In this approach, we use the Data Base Support commands for header information, but we keep the pixel data elsewhere on the disk. In other words, we reserve three regions on the disk: a file for headers, another for an index to our headers, and a region of blocks that are *not* files for pixel data. Within the header, a field points to the *block number* where the pixel data begins for that image. Another field indicates how many blocks are used.

We also recommend keeping the data in the form used by the image processing device (usually binary integers). You may want to process an image using floating point (although in the absence of a hardware floating point processor the fixed-point routines supplied with polyFORTH will be much faster). But a 64-bit floating point number is eight bytes long, which means an image will require eight times as many blocks and take eight times as long to read and write off the disk. It is faster to float the numbers after fetching them.

Some users believe that saving pixel data in floating point form retains better resolution. In fact, however, the typical A/D converter on a Vidicon camera (for instance) does not possess many bits of resolution. In industrial vision applications, these devices rarely provide more than one byte of precision. The extra bits that floating point provides simply represent noise.

On the other hand, some applications do utilize greater precision, but store a much smaller number of pixels. In astronomy, for example, an image size may be only 64x64. But the image might be recorded with a highly sensitive detector over a four-hour period with atmospheric correction. Thus, each pixel has already been integrated and may contain as 16 bits or more of information.

8.10 DOCUMENTOR UTILITY

The **DOCUMENTOR** is a Forth utility that allows maintenance of a file that contains descriptions of Forth words. This provides a convenient way to document polyFORTH programs. It is also an excellent example of the use of all data base management features.

Each word defined in the glossary has the following information associated with it:

1. The block in which the word is defined.
2. The glossary vocabulary.
3. Stack usage.
4. One or more lines of text that describe the word and its use.

The **DOCUMENTOR** utility provides commands to maintain this file and to print reports that include either selected glossary vocabularies or the complete file of words.

8.10.1 File Structure

The **DOCUMENTOR** uses one or more glossary files that are specified by the user. Each glossary file is physically composed of two separate files; a data file and an index file. Index file support in the polyFORTH system is required to implement the documentor.

You must preallocate the two polyFORTH files required for a glossary. The data file is composed of 64-byte records, several of which may be chained together to provide multiple lines of text, 64 characters per line. It is named **GLOSSARY**.

The index file used for the glossary is compiled of 26-byte records, with a 24-byte key length comprised of the word-name (12 bytes) and a vocabulary-name (12 bytes). One index record is required for each glossary entry. The name of the index file is **(GLOSSARY)**. It is an ordered index, ordered by word-name and vocabulary; this has the effect of maintaining the glossary in alphabetic order.

Here is a sample definition for a glossary file that contains 450 entries.

```
( BYTES    RECORDS    BLOCKS     ORG            NAME)
     26        450        12     160     FILE (GLOSSARY)
     64       2300       144     172     FILE GLOSSARY
```

8.10.2 Loading Instructions

The **DOCUMENTOR** requires a 1000-byte partition for execution. It is loaded with the following command:

```
DOCUMENTOR LOAD
```

The **DOCUMENTOR** will empty the user's partition, replacing any other overlay.

The procedure for entering a word into a **DOCUMENTOR** glossary consists of making the block number and glossary vocabulary current, entering the stack usage and the word-name, and then entering associated text. You can change block number, stack usage, and text lines easily. The following sections are interdependent; reading through them at one sitting will provide a helpful overview.

8.10.3 Source Block Identification

When you begin to document your application, you will usually specify a source block to be documented and then enter all the words that are defined in that block.

To specify a source block, use the following phrase:

```
blk# SOURCE !
```

Until changed by re-use of the phrase above, this current block number will automatically be stored with each succeeding word entry.

REFERENCES

Entry Changes, Section 8.10.8

8.10.4 Glossary Vocabulary Identification

Along with each word, the system stores the name of the entry's application vocabulary. This usually means the name of the portion of the application in which the word is used, such as the name of its load block. These vocabularies are not necessarily the same as program vocabularies. Glossary vocabularies exist only for logical grouping of words and to enable the same word to be variously defined several times in different blocks.

Before you begin entering words for a new glossary vocabulary, make it the current vocabulary by typing:

```
VOCAB vocabulary-name
```

Note that the name cannot be longer than ten characters. Until changed, this name is kept in memory and copied into each succeeding data record entered.

In order to search for a previously entered word, you must make its vocabulary the current one.

The glossary vocabulary name serves as a secondary key for searches. This means that the same word may be entered in numerous vocabularies, with each entry unique.

The vocabulary is also set by the report command **/VOCABULARY**.

REFERENCES

/VOCABULARY, Section 8.10.7
Finding Previously Entered Words, Section 8.10.8

8.10.5 Glossary Entries

Words are entered into the glossary through the **ENTER** command. This command sets the basic entry into the file. It has the following format:

ENTER word-name (*e.g.,* **ENTER NAME**)

The program will prompt you for brief (16-character) descriptions of stack entries before and after execution. Any valid Forth word name may be used; the maximum length recognized by the **DOCUMENTOR** is twelve characters. If a longer word name is entered, its length will be truncated to twelve characters.

Following **ENTER**, the new word is made the current word, with which will be stored the current block number, current glossary vocabulary name, and up to four lines of associated text. Immediate subsequent use of **AT**, **STACKS**, **T**, **U**, or **P** will affect this entry.

REFERENCES

Entry Changes, Section 8.10.8
T and **P**, Section 8.10.6

8.10.6 Text Specification

The **DOCUMENTOR** provides commands which allow up to four lines of text to be associated with each entry and also allow modification of previously entered text.

The following command is used to enter a line of text that is associated with a definition:

```
U new text line
```

The command U inserts "new text line" under the current text line (which begins at 0 after a new entry). The new text line may be composed of one to 64 characters, including embedded blanks.

Following the use of **ENTER** or **FIND**, the current text line is initialized to zero. Use of **U** not only inserts a new text line, it also increments the current line number. Thus subsequent usage of **U** adds additional text lines.

The command **P** is used to modify existing text. You do this by displaying the line to be changed and then using **P** to replace the old text with new text. Remember that you can only work on the current word in the current vocabulary. You display the appropriate line of text (lines are numbered starting from zero) by typing:

line# T (*e.g.,* **3 T** to display the fourth line)

After a line of text has been displayed, you can modify it by using the following command:

```
P replacement-text-line
```

The command **X** is used to delete a text line previously selected by the **T** command. Thus, to delete Line 2 you would type:

```
2 T
X
```

Lower-case versions of **U**, **T**, **P**, and **X** are provided for convenience.

REFERENCES

ENTER, Section 8.10.5
FIND, Section 8.10.8

8.10.7 Definition Display

To display the current entry, type:

```
F
```

To print all definitions in all the vocabularies in the glossary, in ASCII alphabetical sequence, use the word **SUMMARY**. The same information as for **FIND** is printed for each word entered in the glossary. The printed report is paged and numbered.

The command:

```
/VOCABULARY
```

will print definitions as for **SUMMARY** but only in the glossary vocabulary whose name is specified.

REFERENCES

Glossary Vocabularies, Section 8.10.4
Making an Entry Current, Section 8.10.8

8.10.8 Changes

Changes always affect the current word. Words are made current in two ways.

1. A word just entered is the current word.

2. A previously entered word in the current vocabulary may be made current by using the command:

```
FIND word-name
```

This displays the requested word, with its vocabulary name, block number, stack usage, and text description.

Following the use of **ENTER** or **FIND**, the current line number is initialized to zero.

The current word's stack entries may be changed by typing:

```
#in-#out STACK
```

The current word's source-block# may be changed by typing:

```
new-blk# AT
```

You may not change vocabulary and word names except by deleting and re-entering the entry, since these two items form the index keys.

To redisplay the complete entry for the current word, type:

```
F
```

REFERENCES

Changing Description Lines, Section 8.10.6
Making a Vocabulary Current, Section 8.10.4

8.10.9 Text and Definition Deletion

The following command is used to remove all text lines associated with the current definition and then to delete the current definition from the glossary:

```
DELETE word-name
```

REVISION HISTORY

REVISION	DESCRIPTION
120825	First Release. Page layout, default font, headers and footers updated. Unusable figures in source documents received from FORTH, Inc. have been renovated.

GreenArrays®

Product Data Book DB005 *Revised 8/25/12*

IMPORTANT NOTICE

GreenArrays Incorporated (GAI) reserves the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to GAI's terms and conditions of sale supplied at the time of order acknowledgment.

GAI disclaims any express or implied warranty relating to the sale and/or use of GAI products, including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright, or other intellectual property right.

GAI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using GAI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

GAI does not warrant or represent that any license, either express or implied, is granted under any GAI patent right, copyright, mask work right, or other GAI intellectual property right relating to any combination, machine, or process in which GAI products or services are used. Information published by GAI regarding third-party products or services does not constitute a license from GAI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from GAI under the patents or other intellectual property of GAI.

Reproduction of GAI information in GAI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. GAI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of GAI products or services with statements different from or beyond the parameters stated by GAI for that product or service voids all express and any implied warranties for the associated GAI product or service and is an unfair and deceptive business practice. GAI is not responsible or liable for any such statements.

GAI products are not authorized for use in safety-critical applications (such as life support) where a failure of the GAI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of GAI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by GAI. Further, Buyers must fully indemnify GAI and its representatives against any damages arising out of the use of GAI products in such safety-critical applications.

GAI products are neither designed nor intended for use in military/aerospace applications or environments unless the GAI products are specifically designated by GAI as military-grade or "enhanced plastic." Only products designated by GAI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of GAI products which GAI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

GAI products are neither designed nor intended for use in automotive applications or environments unless the specific GAI products are designated by GAI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, GAI will not be responsible for any failure to meet such requirements.

For current information on GreenArrays products and application solutions, see www.GreenArrayChips.com

Mailing Address: GreenArrays, Inc., 774 Mays Blvd #10 PMB 320, Incline Village, Nevada 89451
Printed in the United States of America
Phone (775) 298-4748 fax (775) 548-8547 email Sales@GreenArrayChips.com