DataBase Support System

Elizabeth Rather and Dennis Ruffer

08 July 2012

Abstract

To describe a simple database support system. This system originated as a loadable module in polyFORTH. It was converted to be compatible with the ANS Forth standard with a goal to become an IEEE 1275 binding for a database package. Later, it was put into the VentureForth development system for various purposes, including the Timing Diagrams. Now, it is being used again in the colorForth assembler/disassembler and converted to be compatible with 64-bit gforth, using Literate Programing in LyX

Contents

1   Overview & License:

This document specifies a proposal for a database package.

1.1 Revision History

Rev.	Date	Description
0.1	4/26/05	First draft
0.2	5/5/05	First formatting pass, with [DaR] markers for where more work is required
0.3	5/18/05	Test all examples and fix [DaR] markers
0.4	1/15/09	Clean up for release as a VentureForth plug in
0.5	8/5/12	Convert to Literate Programming style

1.2 License

2 DATA BASE SUPPORT

The polyFORTH Data Base Support package is a set of tools with which you can design efficient data base applications or components for general applications.

The Data Base Support package includes:

- A simple memory-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 sub-files to main files and for chaining records within files, and;

- A set of words for creating ordered indexes (for keeping sorted lists).

4   ⟨*support* 4⟩≡   (14)
         ⟨*ordered* 57⟩
         ⟨*bind-fields* 40⟩
         ⟨*user* 9a⟩
         ⟨*working* 12⟩
         ⟨*save* 9b⟩
         ⟨*file* 29⟩
         ⟨*block* 32⟩
         ⟨*read* 36⟩
         ⟨*address* 48b⟩
         ⟨*indexed* 44b⟩
         ⟨*origin* 34⟩
         ⟨*initialize* 35⟩
         ⟨*available* 38a⟩
         ⟨*slot* 38b⟩
         ⟨*records* 39⟩
         ⟨*strings* 50⟩
         ⟨*numeric* 45b⟩
         ⟨*long* 45c⟩
         ⟨*float* 46a⟩
         ⟨*double* 46b⟩
         ⟨*bytes* 47a⟩
         ⟨*byte* 45a⟩
         ⟨*put* 47c⟩
         ⟨*ask* 47d⟩
         ⟨*filler* 44a⟩
         ⟨*fields* 43⟩
         ⟨*copies* 44c⟩
         ⟨*entire* 48a⟩
         ⟨*link* 67b⟩
         ⟨*access* 47b⟩

2.1 Overview

polyFORTH presents its Data Base Support package 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 package. 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.

2.1.1 Contiguous Files and Performance

A premise of the Data Base Support package 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.

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 package addresses both issues simultaneously by imposing a single restriction: a file is a contiguous region of data. This means that the data blocks are 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 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:
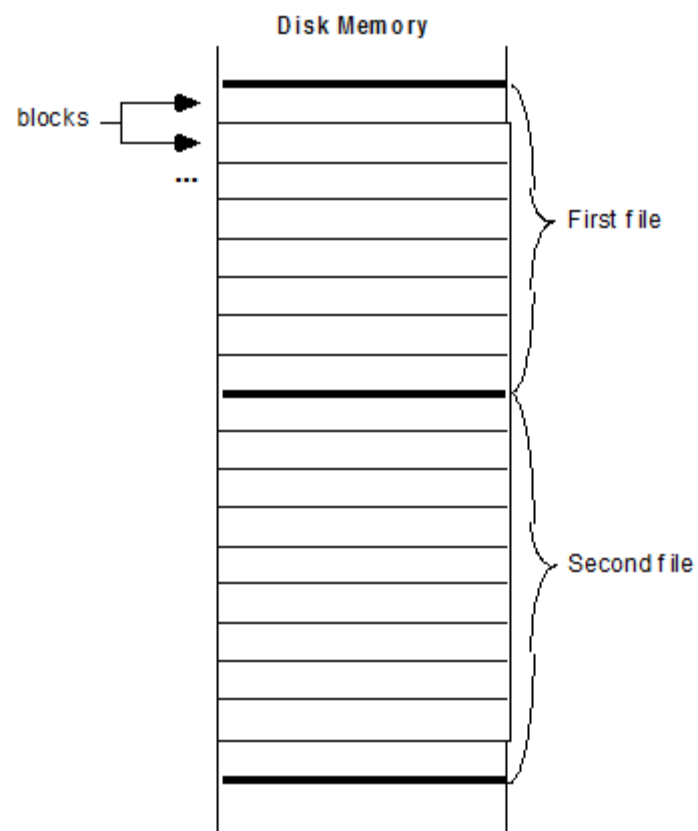
Figure 1:

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 2.7).

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 package 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.

2.1.2   Current Files and Records

Another concept that is fundamental to the Data Base Support package 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**, 2.3). This table is called a *File Definition Area* (FDA).

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 package 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 package 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 package.

Program Memory

user variable

| F# |
pointer to current file

| filename |
normal
dictionary
header

File Definition
Area (FDA)

ORG   (starting block #)
LIM   (max. # records)
B/B   (bytes used / blk)
B/R   (bytes/ record)

| filename |

ORG
LIM
B/B
B/R

Disk Memory

first
file

second
file

Figure 2:

8
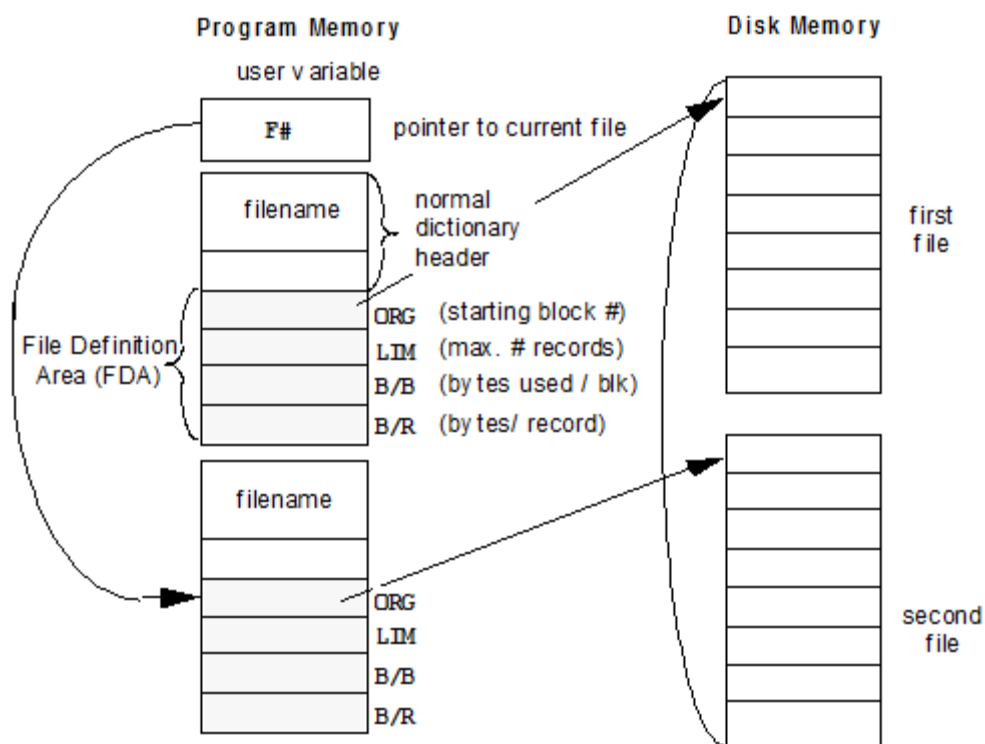
For each file on disk there is a file definition in the dictionary. This definition contains four parameters describing the location and dimensions of the corresponding file. In this figure, the second file is "current."

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 package is once again unique in this concept. 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**. 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#**.

9a   ⟨*user* 9a⟩≡   (4)

```
  #USER CELL +USER R#          \ holds the current record number.
        CELL +USER F#          \ holds the current file pointer.
        CELL +USER DB#         \ holds the current database pointer.
        CELL +USER FLD#        \ holds the current field pointer.
  TO #USER
```

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. Therefore, these words can not tollerate tail recursion enhancements, if your system uses those techniques.

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@**.

9b   ⟨*save* 9b⟩≡   (4)

```
  : SAVE ( -- ) ( R: -- f# db# r# )   R>  DB# @ >R F# @ >R R# @ >R  >R ;
  \ NO-TAIL-RECURSION
  : RESTORE ( -- ) ( R: f# db# r# -- )   R>  R> R# ! R> F# ! R> DB# !  >R ;
  \ NO-TAIL-RECURSION
```

2.1.3   How Data is Stored

The Data Base Support package 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.

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, 2.5.3

2.1.4   Working Storage

Two features of the Data Base Support package 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 (2.8.7). 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 (2.6), 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.

Field names may be used to reference either the individual field in the current record (on disk), or the corresponding field in working storage.

Figure 3:

Terminal Task
Memory

Data Stack

numeric data trav els
to and from the stack

file

PAD

alphanumeric data
(strings) go to and
from th PAD

User
Dictionary

beginning
of task
partition

Working Storage

"image" of
record

maps to fields
in each record
on the disk

Low Memory

Figure 4:

working storage

| name | address | city | st |

a record on disk

| name | address | city | st |
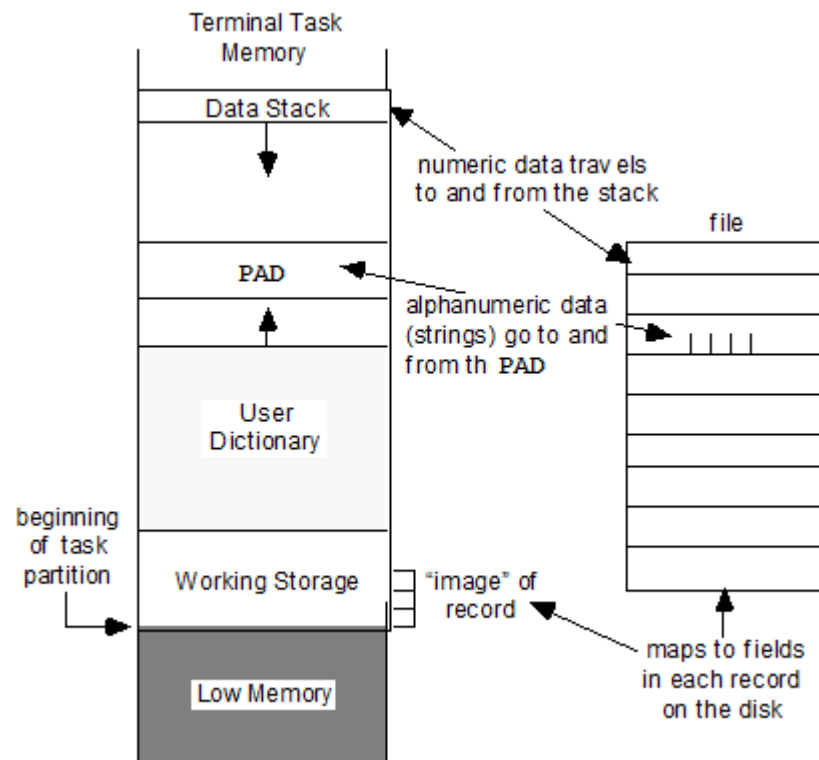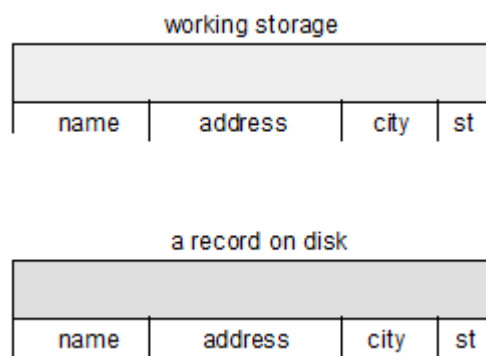
The double use of working storage for both subtotaling and key searches rarely causes a conflict, since the two activities occur at separate times. To be on the safe side, however, the statistics component uses the word **REGISTER** to return the address of the accumulators, which in turn is defined in terms of **WORKING**. If you find that you will encounter a conflict, you may resolve it by simply redefining **REGISTER** 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). On 32-bit machines these sizes should be doubled and on 64-bit machines, multiply times 4.

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.

12   ⟨*working* 12⟩≡   (4)

```
  #USER 1024 +USER WORKING  TO #USER
```

REFERENCES

Accessing Fields in Working Storage, 2.5.5
Ordered Indexes, 2.6
Subtotaling, 2.8.7

2.1.5 Installing The Data Base Support Package in polyFORTH

Before you begin to use the Data Base Support package, 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 package into a your private terminal task, list its load block with the phrase,

`FILES LIST`

The constant FILES return the number of this load block. 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 2.1.4).

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 package.

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 package 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 +!`

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 package, Block 9 should load the package with the system electives. 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 2.1.4) 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.

2.1.6   Installing The Data Base Support Package in gforth

While the previous section described how to install the original Data Base Support Package on the 16-bit polyFORTH block-based system where it was written, the current package has been enhanced to work on 32-bit, file-based systems. It has been tested, at various points in time, on the 32-bit pF32-386/pMSD system using its Text File Support (**TEXTFILE**) Option, on SwiftForth 2.00.2, and on Gforth 0.7.9. However, since each system has its own, unique dependencies, the current system is configured to run on gforth, using an ANS Forth compatibility layer (**pFDatabase.fth**). The primary compatibility issue these routines solve is the issue of locating the support files in a multi-file application package, such as this.

While this could be called an ANS Forth Application, you are not left with an ANS Forth System after it loads, due to many naming conflicts. So, it has ben placed it into a **files-wordlist** that is made current by the word **Files**. Thus, all of the naming conflicts, and most of the application usages are buried away from normal user interaction. Only those words which are considered to be application interface words should be exposed to the user. The developer, however, will need to execute **Files** before the words in this plug in can be used.

The following definitions are required by this system, but are not considered part of it. They may all, eventually, be moved to the supporting Forth system.

The Data Base Support Package can be compiled into gforth by loading the following file:

14   ⟨*pFDatabase.fth* 14⟩≡
```
  [defined] 2+! 0= [IF] : 2+! ( d a -- )   DUP >R 2@ D+ R> 2! ; [THEN]

  ⟨append 23f⟩
  ⟨place 23g⟩
  : prepend ( from len to -- )   dup count  dup 1+ allocate throw  dup >r place
     dup >r place  r> r@ count rot append  r> free throw ;

  ⟨,string 24a⟩
  ⟨string-c, 24b⟩

  : null? DUP 0= IF NIP DUP THEN ;

  : left-parse-string ( str len char -- rstr rlen lstr llen ) \ IEEE 1275 parser from left
     OVER IF
         >R 2DUP R> ROT ROT OVER + SWAP 2DUP = IF  2DROP  ELSE  DO
             DUP I C@ = IF
                 DROP 2DUP + I 1+ SWAP OVER - null?  ( rstr rlen | 0 0 )
                 2SWAP DROP I OVER - null?        ( lstr llen | 0 0 )
                 UNLOOP EXIT  THEN
         LOOP THEN  DROP 0 0 2SWAP
     ELSE  DROP 0 0 2SWAP  THEN ;
```

14

⟨*endians* 22b⟩
⟨*links* 23a⟩

```
wordlist constant DBfiles-wordlist

: files   forth-wordlist  DBfiles-wordlist
    2 set-order  definitions ;  immediate
files
```

⟨*spin* 24c⟩
⟨*get* 20b⟩
⟨*text* 54a⟩
⟨*reverse* 16a⟩
⟨*keep-closed* 16b⟩

```
: >MOVE< ( s d n -- )
    REVERSE @ IF
        2/ 0 ?DO
            >R  COUNT R@ 1+ C!  COUNT R@ C!  R> 2 +
        LOOP  2DROP
    ELSE  MOVE  THEN ;

: xTYPE ( a n -- )
    BOUNDS ?DO
        I C@ DUP BL < IF
            DROP SPACE
        ELSE  EMIT
    THEN  LOOP ;
```

⟨*-text* 24e⟩
⟨*date-format* 17a⟩
⟨*time* 19d⟩
⟨*title″* 19e⟩
⟨*company″* 19f⟩
⟨*app″* 20a⟩
⟨*number-user* 20e⟩
⟨*+user* 21a⟩

⟨*support* 4⟩   \ DataBase Support System
⟨*reports* 73⟩   \ Report Generator
⟨*structure* 33⟩ \ Structured files
⟨*memory* 31⟩   \ Memory based data
⟨*index* 52⟩   \ Ordered Index
⟨*people* 25⟩   \ People example
⟨*personnel* 59⟩ \ Personnel example

⟨*customers* 70⟩ \ Customers example
⟨*accounts* 79⟩   \ Accounts example
⟨*wines* 88⟩   \ Wine Inventory example
⟨*glossary* 100⟩ \ Sample FILE application for documenting Forth words

⟨*test* 108b⟩   \ Test the examples

FORTH DEFINITIONS

Once loaded, the following, optional configuration settings may be invoked to cause the system to behave in specific ways.

FALSE REVERSE !

Causes the system to not store strings with every other byte reversed. This is described in 2.6.1 below, regarding Indexed File Records. The default setting is **TRUE**, since most existing Data Base Support files have been constructed that way. However, if you are not concerned with compatibility with existing data files, you will most likely set this flag to **FALSE**.

16a   ⟨*reverse* 16a⟩≡   (14)
    VARIABLE REVERSE   (Default) FALSE REVERSE !

KEEP-CLOSED when false, file handles are never explicitly closed unless a new file name is bound to a file. When true, file
       handles are closed after each major operation.

16b   ⟨*keep-closed* 16b⟩≡   (14)
    VARIABLE KEEP-CLOSED   (Default) FALSE KEEP-CLOSED !
    [defined] ForTimbre [IF] TRUE KEEP-CLOSED ! [THEN]

LITTLE-ENDIAN
BIG-ENDIAN

These options establish the byte order of values stored in **NUMERIC**, **LONG** and **DOUBLE** fields. The default setting is **LITTLE-ENDIAN**, but you may want to use **BIG-ENDIAN** on systems where that is the native data format. Neither one has any speed benefit, since both are coded using byte fetch operators. However, if you are accessing data in **WORKING** storage, as described in 2.5.5, this setting can be essential.

' DD-MMM-YYYY DATE-FORMAT !
' MM/DD/YYYY DATE-FORMAT !

The format of a date can be adjusted, using this option. The default setting is **MM/DD/YYYY**. The primary usage is in **+PAGE** as is described in 2.8.4.

Modified Julian Dates in the range 01 Jan 1900 through 05 Jun 2079 may be input and output converted, although dates in the first two months of 1900 are not consistent with the rest of the calendar because 1900 was not a leap year.

17a ⟨*date-format* 17a⟩≡ (14)
   ⟨*dy* 17b⟩
   ⟨*days* 17c⟩
   ⟨*fetch-mth* 17d⟩
   ⟨*dmy* 18a⟩
   ⟨*a.d.* 18b⟩
   ⟨*mth* 18c⟩
   ⟨*mdy* 18d⟩
   ⟨*y-dd-dm* 18e⟩
   ⟨*dd-mmm-yyyy* 18f⟩
   ⟨*mmddyyyy* 19a⟩

```
   VARIABLE DATE-FORMAT    ' MM/DD/YYYY DATE-FORMAT !
```

   ⟨*.date* 19b⟩
   ⟨*fetch-date* 19c⟩

D/Y is the number of days per year for a four-year period.

17b ⟨*dy* 17b⟩≡ (17a)
```
   365 4 * 1+ CONSTANT D/Y
```

DAYS is the lookup table of total days in the year at the start of each month.

17c ⟨*days* 17c⟩≡ (17a)
```
   CREATE DAYS   -1 ,   0 ,   31 ,   59 ,   90 ,   120 ,   151 ,
          181 ,  212 ,  243 ,  273 ,  304 ,  334 ,  367 ,
```

@MTH returns the value from days for the given month.

17d ⟨*fetch-mth* 17d⟩≡ (17a)
```
   : @MTH ( u1 -- u2 )   CELLS DAYS + @ ;
```

D/M/Y converts day, month, year into MJD.

18a   ⟨*dmy* 18a⟩≡   (17a)

```
: D-M-Y ( d m y -- u )   >R  @MTH
   58 OVER < IF  R@ 3 AND 0= - THEN + 1-
   R> 1900 -  D/Y UM*  4 UM/MOD SWAP 0<> - + ;
```

A.D. sets the current **YEAR**.

18b   ⟨*a.d.* 18b⟩≡   (17a)

```
VARIABLE YEAR

: A.D. ( n -- )   YEAR ! ;   TIME&DATE A.D. 2DROP 2DROP DROP
```

MTH defines words that convert date to internal form in month of the same name using year in **YEAR**.

18c   ⟨*mth* 18c⟩≡   (17a)

```
: MTH ( n _ -- )   CREATE , DOES> ( d -- n )   @ YEAR @ D-M-Y ;

 1 MTH JAN       2 MTH FEB       3 MTH MAR       4 MTH APR
 5 MTH MAY       6 MTH JUN       7 MTH JUL       8 MTH AUG
 9 MTH SEP      10 MTH OCT      11 MTH NOV      12 MTH DEC
```

M/D/Y takes a double number mm/dd/yyyy and converts it to MJD.

18d   ⟨*mdy* 18d⟩≡   (17a)

```
: M/D/Y ( ud -- u )   10000 UM/MOD  100 /MOD  ROT D-M-Y ;
```

Y-DD DM split the serial number back into its components.

18e   ⟨*y-dd-dm* 18e⟩≡   (17a)

```
: Y-DD ( u1 -- y u2 u3 )   4 UM* D/Y  UM/MOD 1900 +  SWAP 4 /MOD 1+
   DUP ROT 0= IF  DUP 60 > +  SWAP DUP 59 > +  THEN ;

: DM ( u1 u2 -- d m )   1 BEGIN  1+  2DUP @MTH > 0= UNTIL  1-
   SWAP DROP SWAP  OVER @MTH - SWAP ;
```

DD-MMM-YYYY converts internal form to text formatted as dd mmm yyyy.

18f   ⟨*dd-mmm-yyyy* 18f⟩≡   (17a)

```
CREATE MTHS   s" JanFebMarAprMayJunJulAugSepOctNovDec" $c,

: DD-MMM-YYYY ( n -- a n )   BASE @ >R  DECIMAL  Y-DD
   ROT 0 <#  # # # #  2DROP  BL HOLD  DM 1- 3 * MTHS +
   3 + 3 0 DO  1- DUP C@ HOLD  LOOP  DROP
   BL HOLD  0 # #  #>  R> BASE ! ;
```

MM/DD/YYYY converts internal form to text formatted as mm/dd/yyyy.

19a   ⟨*mmddyyyy* 19a⟩≡   (17a)

```
: MM/DD/YYYY ( u1 -- c-addr u2 )   BASE @ >R  DECIMAL  Y-DD
   ROT 0 <#  # # # #  2DROP  [char] / HOLD  DM SWAP
   0 # #  2DROP   [char] / HOLD  0 # #  #>  R> BASE ! ;
```

.DATE displays the system date (u) as mm/dd/yyyy.

19b   ⟨*.date* 19b⟩≡   (17a)

```
: .DATE ( u -- )   DATE-FORMAT @ EXECUTE TYPE SPACE ;
```

@DATE gets the current system date.

19c   ⟨*fetch-date* 19c⟩≡   (17a)

```
: @DATE ( -- n )   TIME&DATE D-M-Y NIP NIP NIP ;

: DATE ( -- )   @DATE .DATE ;
```

19d   ⟨*time* 19d⟩≡   (14)

```
: :00 ( ud1 -- ud2)   DECIMAL  #  6 BASE !  # [char] : HOLD ;
: (TIME) ( secs -- c-addr u)   BASE @ >R  0 <#  :00 :00
   DECIMAL # #  #>  R> BASE ! ;
: .TIME ( secs -- )   (TIME) TYPE SPACE ;

: @TIME ( -- secs )   TIME&DATE 2DROP DROP 60 * + 60 * + ;

: TIME ( -- )   @TIME .TIME ;
```

TITLE" defines the string to be used for listings. When its children are executed, they leave the string address in **'TITLE**.

19e   ⟨*title"* 19e⟩≡   (14)

```
VARIABLE 'TITLE

: TITLE" ( -- )   CREATE  HERE 'TITLE !  ,"
   DOES> ( -- )   'TITLE ! ;
```

COMPANY" sets the company name used for report titles.

19f   ⟨*company"* 19f⟩≡   (14)

```
VARIABLE 'COMPANY

: COMPANY" ( -- )   HERE 'COMPANY !  ," ;

COMPANY" Software Specialists"
```

19

APP" sets the application name used for screen titles.

20a   ⟨*app″* 20a⟩≡   (14)
```
  VARIABLE 'APP

  : APP" ( -- )   HERE 'APP !  ," ;

  APP"  "
```

REFERENCES
B!, 2.5.3
WORKING, 2.5.5
+PAGE, 2.8.4

2.1.7   Installing The Data Base Support Package in Open Firmware

This documentation, 1275 Binding for a Database Package, was never actually published. It is the last time that I worked on the documentation with Elizabeth Rather, with the hope to make this tool public, but the effort died when Apple switched to Intel. Since the Data Base Support Package was originally written to run on polyFORTH, there are dependencies, above and beyond simple ANS Forth requirements. These have all been bracketed with the following:

[DEFINED] <name> 0= [IF] <definition> [THEN]

Here is a list of the non-ANS Forth words that this system requires:

Word Description

GET (a --) Multi-tasking means to get ownership of a facility.

20b   ⟨*get* 20b⟩≡   (14)
```
  [defined] GET 0= [IF] : GET DROP ; [THEN]
  ⟨grab 20c⟩
  ⟨release 20d⟩
```

GRAB (a --) Multi-tasking means to get ownership without pausing.

20c   ⟨*grab* 20c⟩≡   (20b)
```
  [defined] GRAB 0= [IF] : GRAB DROP ; [THEN]
```

RELEASE (a --) Multi-tasking means to release ownership of a facility.

20d   ⟨*release* 20d⟩≡   (20b)
```
  [defined] RELEASE 0= [IF] : RELEASE DROP ; [THEN]
```

#USER (-- a) Multi-tasking variable containing the size of the user area.

20e   ⟨*number-user* 20e⟩≡   (14)
```
  [defined] #USER 0= [IF]  0 VALUE #USER  [THEN]
```

+USER (o n -- o') Multi-tasking means to define new user variables.

21a   ⟨*+user* 21a⟩≡   (14)

```
[defined] +USER 0= [IF]
    : +USER ( o n _ -- o+n )   CREATE DUP ALLOT + ;
[THEN]
```

The following are used to fetch data that is in known Endian format. E.g. in file system structures or network packets. These words work on un-aligned entities.

c@-le (a n -- ...) Little endian fetch of a multi-byte value.

21b   ⟨*c@-le* 21b⟩≡   (22b)

```
[defined] c@-le 0= [IF]
: 1c@-le ( a n -- x )   0 SWAP BEGIN ?DUP WHILE
      1- ROT 2DUP +     C@ >R ROT 8 LSHIFT R> + ROT
   REPEAT NIP ;
: c@-le ( a n -- ... )
   CELL /MOD ROT OVER CELLS + -ROT SWAP ?DUP
   IF  ROT DUP ROT 1c@-le -ROT SWAP  THEN  ?DUP
   IF  0 DO  1 CELLS - DUP CELL 1c@-le SWAP  LOOP
   THEN  DROP ;
[THEN]
```

c!-le (... a n --) Little endian store of a multi-byte value.

21c   ⟨*c!-le* 21c⟩≡   (22b)

```
[defined] c!-le 0= [IF]
: 1c!-le ( x a n -- )   BEGIN ?DUP WHILE
      1- ROT DUP 8 RSHIFT SWAP 2OVER DROP C! ROT 1+ ROT
   REPEAT 2DROP ;
: c!-le ( ... a n -- )   CELL /MOD SWAP >R  ?DUP
   IF  0 DO  SWAP OVER CELL 1c!-le CELL+ LOOP
   THEN  R> ?DUP IF  1c!-le  ELSE DROP THEN ;
[THEN]
```

c@-be (a n -- ...) Big endian fetch of a multi-byte value.

21d   ⟨*c@-be* 21d⟩≡   (22b)

```
[defined] c@-be 0= [IF]
: 1c@-be ( a n -- x )   0 SWAP BEGIN ?DUP WHILE
      1- ROT DUP 1+ SWAP C@ >R ROT 8 LSHIFT R> + ROT
   REPEAT NIP ;
: c@-be ( a n -- ... )
   CELL /MOD ROT OVER CELLS + -ROT SWAP ?DUP
   IF  ROT DUP ROT 1c@-be -ROT SWAP  THEN  ?DUP
   IF  0 DO  1 CELLS - DUP CELL 1c@-be SWAP  LOOP
   THEN  DROP ;
[THEN]
```

21

c!-be (. . . a n --) Big endian store of a multi-byte value.

22a   ⟨*c!-be* 22a⟩≡   (22b)
```
[defined] c!-be 0= [IF]
: 1c!-be ( x a n -- )   BEGIN ?DUP WHILE
      1- ROT DUP 8 RSHIFT SWAP 2OVER +    C! ROT    ROT
   REPEAT 2DROP ;
: c!-be ( ... a n -- )   CELL /MOD SWAP >R  ?DUP
   IF  0 DO  SWAP OVER CELL 1c!-be CELL+ LOOP
   THEN  R> ?DUP IF  1c!-be  ELSE DROP THEN ;
[THEN]
```

22b   ⟨*endians* 22b⟩≡   (14)
```
⟨c!-le 21c⟩
⟨c!-be 22a⟩
⟨c@-le 21b⟩
⟨c@-be 21d⟩

VARIABLE c@-xx
VARIABLE c!-xx

: nC@ ( a n -- u|d )   c@-xx @ EXECUTE ;
: nC! ( u|d a n -- )   c!-xx @ EXECUTE ;

: LITTLE-ENDIAN ( -- )   ['] c@-le c@-xx !  ['] c!-le c!-xx ! ;
:    BIG-ENDIAN ( -- )   ['] c@-be c@-xx !  ['] c!-be c!-xx ! ;

: LITTLE-ENDIAN? ( -- flag )   ['] c@-le c@-xx @ = ;

BIG-ENDIAN \ Default
```

@LINK !LINK ,LINK Do relocation translation, if needed.

22c   ⟨*@!,link* 22c⟩≡   (23a)
```
: @LINK ( a -- a )   [defined] @REL [IF]
      @REL  [ELSE]  @  [THEN] ;
: !LINK ( a1 a2 -- )   [defined] !REL [IF]
      !REL  [ELSE]  !  [THEN] ;
: ,LINK ( a -- )   [defined] ,REL [IF]
      ,REL  [ELSE]  ,  [THEN] ;
```

LINKS (a -- a') Searches the linked list until it finds the last entry in the list (the one with a 0 link).

23a   ⟨*links* 23a⟩≡   (14)
             ⟨*@!,link* 22c⟩

             : LINKS (a -- a')   BEGIN DUP @LINK ?DUP WHILE NIP REPEAT ;

             ⟨*>link* 23b⟩
             ⟨*<link* 23c⟩
             ⟨*unlink* 23d⟩
             ⟨*calls* 23e⟩

>LINK (a --) Adds the top of the dictionary to the given linked list.

23b   ⟨*>link* 23b⟩≡   (23a)
             : >LINK (a --)   ALIGN HERE OVER @LINK ,LINK SWAP !LINK ;

<LINK (a --) Adds the top of the dictionary to the end of the given linked list.

23c   ⟨*<link* 23c⟩≡   (23a)
             : <LINK (a --)   LINKS >LINK ;

UNLINK (a a' -- a) Breaks the link of the given entry, resetting it to the link pointed to by the element on top of the stack.

23d   ⟨*unlink* 23d⟩≡   (23a)
             : UNLINK (a a' -- a)   @LINK OVER !LINK ;

CALLS (a --) Runs down a linked list, executing the high level code that follows each entry in the list.

23e   ⟨*calls* 23e⟩≡   (23a)
             : CALLS (a --)
                 BEGIN
                     @LINK ?DUP WHILE
                     DUP >R 1 CELLS + @LINK EXECUTE R>
                 REPEAT ;

append (a1 n a2 --) Add a string to a counted string.

23f   ⟨*append* 23f⟩≡   (14)
             : append (from len to --)   2DUP >R >R COUNT + SWAP MOVE
                 R> R@ C@ + R> C! ;

place (a1 n a2 --) Place a counted string.

23g   ⟨*place* 23g⟩≡   (14)
             : place (from len to --)   0 OVER C! SWAP 255 MIN SWAP APPEND ;

,string (a n --) Compile a counted string.

24a   ⟨*,string* 24a⟩≡   (14)
```
: ,string ( str len -- a )   HERE DUP >R OVER 1+ ALLOT place R> ;
```

$c, (a n --) Compile a string.

24b   ⟨*string-c,* 24b⟩≡   (14)
```
: $c, ( str len -- )   OVER + SWAP ?DO  I C@ C,  LOOP ;
```

spin (--) Show activity.

24c   ⟨*spin* 24c⟩≡   (14)
```
[defined] spin 0= [IF] : spin ; IMMEDIATE [THEN]
```
           ⟨*-spin* 24d⟩

-spin (--) Stop showing activity.

24d   ⟨*-spin* 24d⟩≡   (24c)
```
[defined] -spin 0= [IF] : -spin ; IMMEDIATE [THEN]
```

-TEXT takes the address of two strings and a length and returns +1 if the 1st string has a higher ASCII value, -1 if it has a
lower value, and 0 (false) if they are equal. The string length must be even. If the length is odd, it will be rounded down
to the next even value.

24e   ⟨*-text* 24e⟩≡   (14)
```
: -TEXT ( a1 n a2 -- flag )
   REVERSE @ IF
       0  ROT 2/ 0 DO
            ROT DUP 2 nC@ >R 2 +
            ROT DUP 2 nC@ >R 2 +
            ROT R> R> - ?DUP IF
                NIP 0< IF
                    1  ELSE  -1
        THEN  LEAVE  THEN  LOOP  NIP NIP
    ELSE  OVER COMPARE  THEN ;
```

In addition, Open Firmware does not presently contain the following ANS Forth words:

SAVE-INPUT RESTORE-INPUT M+ ALLOCATE FREE RESIZE

Additionally, the **BYTE-LOAD** method is used to conditionally compile support for Open Firmware specific features. For example, the 1275 tokenizer does not directly support defining words, so extensions to the tokenizer must be defined every time one of these words is defined. The disk based file support words are also not directly supported, so they are not loaded into the Open Firmware package.

The Data Base Support Package is compiled with the rest of the Open Firmware kernel, so there may be other dependencies, beyond the 1275 specified words, which have not been identified at this point. The package is optionally byte-loaded into the **/packages** node and it optionally byte-loads the applications which use it. This creates a compile time binding (i.e. early binding) between the kernel, this package, and the applications, which is required at the present time. Future work will be needed to remove this dependency.

Systems that use this package should compile their applications in **THE-FILES.of**, following the compile of the **database.of** file

The following modules rely on the Database operators:

```
-1 -1 tokenize-list OF/OFSources/Common/Database.of
Encode-Driver BuildResults/Build/Obj/Database.lzss
-1 -1 tokenize-list OF/OFSources/IO/smu/sdb-parser.of
Encode-Driver BuildResults/Build/Obj/sdb-parser.lzss
```

Add their applications to **The-Drivers.fo**:

```
LoadDriver Database
LoadDriver sdb-parser

AddDriver Database
AddDriver sdb-parser
```

And the byte load of their applications to **Database.of**:

```
" sdb-parser" ['] byte-load-driver CATCH IF 2DROP THEN
```

Then, their application interfaces can be called as needed:

```
0 0 " database" $open-package ?dup
if >r [char] B " distribute-sdb-properties" r@ $call-method
r> close-package
then
```

2.2   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 2.9.

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 2.6).

25   ⟨*people* 25⟩≡   (14)
            ⟨*people-fields* 26a⟩
            ⟨*people-file* 26b⟩

⟨*people-initialize* 26c⟩
⟨*enter-person* 27a⟩
⟨*display-person* 27b⟩
⟨*display-everyone* 27c⟩
⟨*all-people* 74⟩

Step 1

Define the fields:

26a   ⟨*people-fields* 26a⟩≡   (25)

```
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 2.5.1).

In addition to **BYTES**, several other defining words are available for creating different types of fields (see 2.5.2).

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."[1] **#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 2.3.1).

Step 3

Define the file:

26b   ⟨*people-file* 26b⟩≡   (25)

```
  FILE People.dbf

  : /People ( -- ) \ Runtime file setup
     People.dbf  S" People.dbf" >FILE
     TRUE REVERSE !  LITTLE-ENDIAN ;

  76 500 400 BLOCK-DATA 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. The starting block will be 400 (see 2.3.1).

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:

26c   ⟨*people-initialize* 26c⟩≡   (25)

```
  \ PEOPLE INITIALIZE
```

---

[1] In polyFORTH 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 +**.

to fill all blocks in the file with zeroes (see 2.3.3).

Step 5

Enter data.

Here is the definition of a word that will allow data entry for a single record (person):

27a   ⟨*enter-person* 27a⟩≡   (25)

```
: enter-person   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 2.4.3). The word **READ** sets the current record according to the number on the stack (see 2.4.1).

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:

27b   ⟨*display-person* 27b⟩≡   (25)

```
: display-person   CR 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 2.5.3).

To display the contents of all records that have been entered, we define:

27c   ⟨*display-everyone* 27c⟩≡   (25)

```
: display-everyone   PEOPLE RECORDS DO CR
    I READ  display-person  LOOP ;
```

27

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 2.4.3).

Within the **DO** loop, **READ** makes each record current in turn, and **display-person** displays the information for that
record.

Here is a sample of the output of **display-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" (2.8).

For deleting records, we would use the word **SCRATCH** (see 2.4.3).

2.3   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.

2.3.1   The FILE Definition

The word **FILE** is used to group a collection of Data Bases under a single name. Each Data Base within the group will use the
same access operators. Presently, there are 2 types of access operators, disk based (**>FILE**) and memory based (**>MEMORY**).
The disk based operators can not be used within Open Firmware, since the creation of files has not been fully developed yet.

However, in gforth, they are used extensively by passing the pathname string to **>FILE**. The word **FILE=** simply parses the input stream for the pathname string before passing it to **>FILE**.

29   ⟨*file* 29⟩≡   (4)

```
: (FILE) ( o n -- o+n ) \ Usage: (FILE) <name>
   CREATE  OVER , +  DOES> ( -- a )   @  F# @ + ;

0 VALUE #FILE                          \ holds offset of next FCB field.

#FILE 2 CELLS (FILE) FILE-HANDLE    \ holds OS/Memory handle for this file, if open.
       CELL  (FILE) FILE-INIT       \ holds initialization routines for this file.
       CELL  (FILE) 'NAME-FILE      \ holds routine to name the file
       CELL  (FILE) 'BIND-FILE      \ holds routine to bind the file to a path.
       CELL  (FILE) 'UNBIND-FILE    \ holds routine to unbind the file.
       CELL  (FILE) 'READ-RECORD    \ holds routine to read a record from the file.
       CELL  (FILE) 'WRITE-RECORD   \ holds routine to write a record to the file.
       CELL  (FILE) 'DESTROY-FILE   \ holds routine to delete the file.
       256   (FILE) FILE-NAME       \ holds path and name of the file.
TO #FILE

: SET-FILE ( addr -- )   F# ! ;
: SET-DATA ( addr -- )   DB# ! ;

: (NAME-FILE) ( str len -- )   FILE-NAME DUP 256 ERASE place ;

: NAME-FILE ( str len -- )   'NAME-FILE @ ?DUP IF  EXECUTE
   ELSE  2DROP  THEN ;

: READ-RECORD ( d a n -- n' ior )   'READ-RECORD @ ?DUP IF  EXECUTE
   ELSE  NIP NIP NIP 0  THEN ;

: WRITE-RECORD ( d a n -- ior )   'WRITE-RECORD @ ?DUP IF  EXECUTE
   ELSE  2DROP 2DROP 0  THEN ;

: DESTROY-FILE ( -- ior )   'DESTROY-FILE @ ?DUP IF EXECUTE ELSE 0 THEN ;

: BIND-FILE ( -- )   FILE-HANDLE @ 0=
   IF  'BIND-FILE @ ?DUP
       IF  EXECUTE
       THEN  FILE-HANDLE !
   THEN ;

: -FILE ( -- )   FILE-HANDLE @ ?DUP
   IF  'UNBIND-FILE @ ?DUP
       IF  EXECUTE
       THEN  0 FILE-HANDLE !
   THEN ;
```

29

```
: FILE-ALLOT ( -- )
   HERE DUP SET-FILE  #FILE DUP ALLOT ERASE
   ['] (NAME-FILE) 'NAME-FILE ! ;

: FILE ( -- ) \ Usage: FILE <name>
   SAVE-INPUT  CREATE  RESTORE-INPUT THROW
   FILE-ALLOT  BL WORD COUNT NAME-FILE
   DOES> ( -- )   SET-FILE ;

: FILES-NAME ( str len -- )
   [defined] ?FileName [IF]  0 TO ?FileName  ,ObjName
   [THEN]  (NAME-FILE) ;

: FILES-BIND ( -- handle )
   FILE-NAME COUNT 2DUP R/W OPEN-FILE
   IF  DROP R/W CREATE-FILE THROW
   ELSE  -ROT 2DROP
   THEN ;

: FILES-UNBIND ( handle -- )   CLOSE-FILE THROW ;

: FILES-READ ( d a n -- n' ior )
   FILE-HANDLE @ 0= DUP >R IF  BIND-FILE  THEN
   2SWAP FILE-HANDLE @ REPOSITION-FILE ?DUP IF  ROT DROP
   ELSE  FILE-HANDLE @ READ-FILE
   THEN  R> IF  -FILE  THEN ;

: FILES-WRITE ( d a n -- ior )
   FILE-HANDLE @ 0= DUP >R IF  BIND-FILE  THEN
   2SWAP FILE-HANDLE @ REPOSITION-FILE ?DUP IF  -ROT 2DROP
   ELSE  FILE-HANDLE @ WRITE-FILE
   THEN  R> IF  -FILE  THEN ;

: FILES-DESTROY ( -- ior )   FILE-NAME COUNT DELETE-FILE ;

: >FILE ( str len -- )   -FILE
   ['] FILES-NAME 'NAME-FILE !
   ['] FILES-BIND 'BIND-FILE !
   ['] FILES-UNBIND 'UNBIND-FILE !
   ['] FILES-READ 'READ-RECORD !
   ['] FILES-WRITE 'WRITE-RECORD !
   ['] FILES-DESTROY 'DESTROY-FILE !
   NAME-FILE  BIND-FILE
   FILE-INIT CALLS
   KEEP-CLOSED @ IF
```

```
        -FILE
    THEN   ;

: FILE= ( -- )    BL WORD COUNT  DUP 1+ ALLOCATE THROW
    DUP >R place R@ COUNT >FILE  R> FREE THROW ;

: <FILE ( xt -- )    FILE-INIT <LINK ,LINK ;
```

In Open Firmware, the Data Base groups are typically created at compile time and compiled into Open Firmware with
encode-file. Since this puts the data into the dictionary, the word **HERE>MEMORY** is used to put the data into the
buddy memory manager which can be used by the **>MEMORY** access operators.

31   ⟨*memory* 31⟩≡   (14)

```
: MEMORY-BIND ( -- handle )   2 B/R @ #B 1024 * \ 2 records minimum
    DUP FILE-HANDLE CELL+ !  DUP ALLOCATE THROW
    DUP ROT ERASE ;

: MEMORY-UNBIND ( handle -- )   FREE THROW ;

: MEMORY-READ ( d a n -- n' ior )   BIND-FILE
    2SWAP IF  DROP NIP -24 EXIT  THEN      \ invalid numeric argument, file too large
    FILE-HANDLE CELL+ @ 2DUP >             \ limit record position to file size
    IF  SWAP  THEN  DROP SWAP OVER +
    FILE-HANDLE CELL+ @ 2DUP >             \ limit record size to file size
    IF  SWAP  THEN  DROP OVER -
    DUP >R  SWAP FILE-HANDLE @ +
    ROT ROT MOVE  R> 0 ;

: MEMORY-WRITE ( d a n -- ior )   BIND-FILE
    2SWAP IF  DROP 2DROP -24 EXIT  THEN    \ invalid numeric argument, file too large
    SWAP OVER +  DUP FILE-HANDLE CELL+ @ > \ record position + size > file size
    IF  FILE-HANDLE @ OVER RESIZE ?DUP
        IF  >R 2DROP 2DROP R> EXIT         \ can't resize the memory
        THEN  FILE-HANDLE !
        DUP FILE-HANDLE CELL+ !
    THEN  OVER - SWAP FILE-HANDLE @ +
    SWAP MOVE  0 ;

: >MEMORY ( addr len -- )   -FILE
    ['] MEMORY-BIND 'BIND-FILE !
    ['] MEMORY-UNBIND 'UNBIND-FILE !
    ['] MEMORY-READ 'READ-RECORD !
    ['] MEMORY-WRITE 'WRITE-RECORD !
    SWAP FILE-HANDLE 2!  BIND-FILE
    FILE-INIT CALLS ;
```

The words **BLOCK-DATA** or **STRUCTURE** are used to define files, given the attributes of the file. The format is:

length limit origin BLOCK-DATA name

where:

Word Description

length is the length of each record in bytes (maximum 1024 using **BLOCK-DATA**);

limit is the maximum number of records (on 16-bit processors the limit is 32767 records per file);

origin is the first block number, and

name is the user-assigned name of the file.

The defining words **BLOCK-DATA** and **STRUCTURE** create 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. **BLOCK-DATA** will make sure that records do not overlap across 1K boundaries, while **STRUCTURE** does not impose this additional overhead.

32 ⟨*block* 32⟩≡ (4)

```
  : (DATA) ( o n -- o+n ) \ Usage: (DATA) <name>
     CREATE  OVER , + DOES> ( -- a )   @  DB# @ + ;

 0 VALUE #DATA                 \ holds offset of next DCB field.

 #DATA CELL (DATA) ORG        \ holds offset to first record in the file.
       CELL (DATA) LIM        \ holds number of records in file.
       CELL (DATA) B/B        \ holds bytes used per block.
       CELL (DATA) B/R        \ holds bytes per record.
       CELL (DATA) DATA-BFR   \ holds address of record buffer.
       CELL (DATA) 'RECORD    \ holds the vector for record access.
       CELL (DATA) 'TOUCH     \ holds the vector for field update.
       CELL (DATA) #REC       \ holds the record number last accessed in this file.
                              \  This is used with BOUND-FIELDS.
       CELL (DATA) #INDEX     \ holds index of COPIES field.
        256 (DATA) DATA-NAME \ holds name of the database.
  TO #DATA

 : NAME-DATA ( str len -- )   DATA-NAME DUP 256 ERASE place ;

 : BLOCK-POSITION ( n -- d )   DUP #REC !
    B/R @  B/B @ */MOD  1024 UM*  ROT M+
    ORG @ M+ ;

 : (RECORD) ( d -- a )
    DATA-BFR @ B/R @ READ-RECORD IF
       DROP DATA-BFR @ B/R @ ERASE
    ELSE  B/R @ OVER - ?DUP IF
            DATA-BFR @ ROT + SWAP ERASE
```

32

```
          ELSE  DROP  THEN
       THEN  DATA-BFR @ ;

    : (TOUCH) ( d -- )   DATA-BFR @ B/R @ WRITE-RECORD THROW ;

    : BLOCK-RECORD ( n -- a )   BLOCK-POSITION (RECORD) ;
    : BLOCK-TOUCH ( -- )   #REC @ BLOCK-POSITION (TOUCH) ;

    : (BLOCK-DATA) ( b r o -- )
       HERE DUP SET-DATA  #DATA DUP ALLOT ERASE
       ORG !  LIM !  0 #REC !  0 #INDEX !
       1024 OVER / OVER *  B/B !  B/R !
       HERE DATA-BFR !  B/R @ ALLOT
       ['] BLOCK-RECORD 'RECORD !
       ['] BLOCK-TOUCH 'TOUCH ! ;

    : BLOCK-DATA ( b r o _ -- ) \ Usage: BLOCK-DATA <name>
       SAVE-INPUT  CREATE  RESTORE-INPUT THROW
       (BLOCK-DATA)  BL WORD COUNT NAME-DATA
       DOES> ( -- )   SET-DATA ;
```

33   ⟨*structure* 33⟩≡   (14)
```
    : POSITION-FILE ( r -- d )   DUP #REC !  B/R @ UM*  ORG @ M+ ;

    : STRUCTURE-RECORD ( r -- a )   POSITION-FILE (RECORD) ;
    : STRUCTURE-TOUCH ( -- )    #REC @ POSITION-FILE (TOUCH) ;

    : (STRUCTURE) ( b r o _ -- )
       HERE DUP SET-DATA  #DATA DUP ALLOT ERASE
       ORG !  LIM !  0 #REC !  0 #INDEX !
       DUP B/R !  HERE DATA-BFR !  ALLOT
       ['] STRUCTURE-RECORD 'RECORD !
       ['] STRUCTURE-TOUCH 'TOUCH ! ;

    : STRUCTURE ( b r o _ -- ) \ Usage: STRUCTURE <name>
       SAVE-INPUT  CREATE  RESTORE-INPUT THROW
       (STRUCTURE)  BL WORD COUNT NAME-DATA
       DOES> ( -- )   SET-DATA ;
```

We recommend that you define all your files in a single source area, 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:

```
( Bytes Records Origin Name)
26 801 500 STRUCTURE (GLOSSARY)
340 800 522 STRUCTURE GLOSSARY
4 10 795 STRUCTURE HITS
24 42 799 STRUCTURE SECURITY
38 2600 800 STRUCTURE TESTS
```

Note that the number of blocks may be computed from the number of bytes/ record and number of records. 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:[2]

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 over sizing either your record width or file length, as both of these will increase head motion. Strive for generous but reasonable estimates.

34   ⟨*origin* 34⟩≡   (4)
```
 : #B ( nr b/r -- nb )   1024 SWAP /   DUP 1- ROT +   SWAP / ;

 : #R ( nb b/r -- nr )   1024 SWAP / * ;

 : +ORIGIN ( -- o )   LIM @ B/R @ B/B @ IF #B 1024 THEN * ORG @ + ;
```

---

[2]Responses shown in light type.

2.3.2   File Definition Area and Access

The word **STRUCTURE** 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.

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 package, 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.

2.3.3   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.

To initialize a file, type:

filename INITIALIZE

The word **INITIALIZE** performs the following functions:

1. Writes binary zeros throughout the entire file (including **AVAILABLE**).

2. Writes -1 in the entire data area 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.

35   ⟨*initialize* 35⟩≡   (4)
```
: STOPPER ( -- )   1 RECORD CELL+  B/R @ CELL -  -1 FILL  TOUCH ;

: INITIALIZE ( -- )
\   CR ." Initializing "
    FILE-HANDLE @ 0= DUP >R IF  BIND-FILE  THEN
\   DATA-NAME COUNT TYPE ." ..."
    LIM @ 0 DO
        I RECORD  B/R @  ERASE  TOUCH  spin
    LOOP  STOPPER  -spin
    R> IF  -FILE  THEN ;
```

Two other words also defined---**#R** and **#B**---are useful when designing file layouts.

2.3.4   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.

As we discussed in 2.1.3, 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

Installing the Data Base Support Package, 2.1.5

2.4   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.

2.4.1   Record Selection

Field reference operators (2.5.3) 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#**.

36   ⟨*read* 36⟩≡   (4)
```
  : SAFE ( n -- n )   DUP LIM @ U< 0= ABORT" Outside file " ;

  : READ ( n -- )   SAFE DUP R# !  #REC ! ;
```
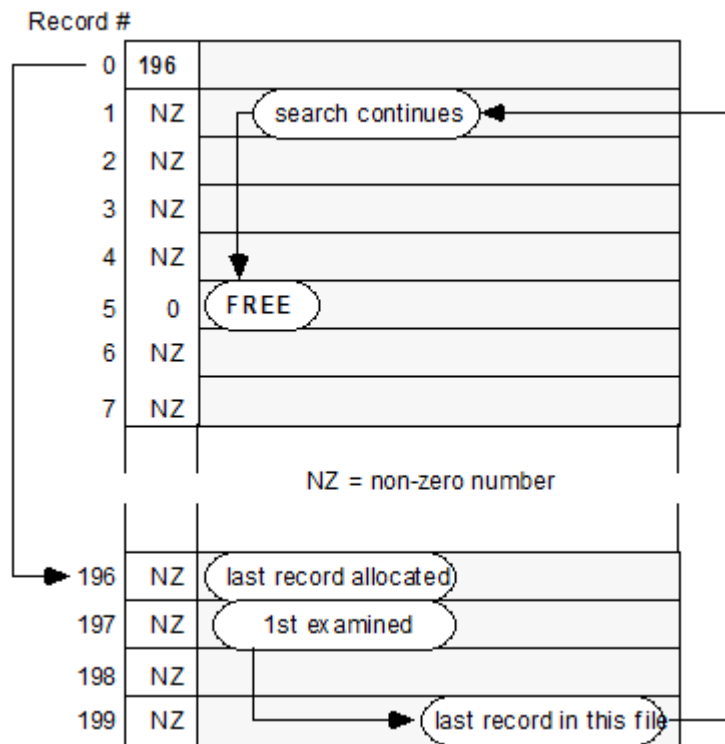
Figure 5:

REFERENCES

Current Files and Records, 2.1.2

2.4.2   Available Records

To distinguish allocated records from available records, the Data Base Support package uses the convention that if the first four 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 four bytes; when a record is released, zero is stored in this area.

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 four 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. Figure 5 on page 37, **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.

38a   ⟨*available* 38a⟩≡   (4)
```
: AVAILABLE ( -- a )   0 RECORD ;
```


2.4.3   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.

38b   ⟨*slot* 38b⟩≡   (4)
```
: SLOT ( -- r )
   AVAILABLE 4 nC@ DUP  BEGIN
        1+  LIM @ MOD  2DUP = ABORT" File full"
        DUP RECORD  DUP @ WHILE  DROP
   REPEAT  DUP  B/R @ ERASE  -1 SWAP !  TOUCH
   DUP AVAILABLE 4 nC!  TOUCH  NIP ;

: SCRATCH ( r -- )   SAFE  RECORD  0 SWAP !  TOUCH ;
```

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 L! READ

Here the phrase **LINK L!** 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!, 2.5.3
READ, 2.4.1

2.4.4   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.

39   ⟨*records* 39⟩≡   (4)
```
  : RECORDS ( -- l f )   AVAILABLE 4 nC@ 1+  1 ;

  : WHOLE ( -- l f )   LIM @ 1 ;
```

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 L@ IF ...
```

where **LINK** is the generic long field comprising the first four bytes of each record. If these bytes contain zero, the record is available for use.

REFERENCES

Available Records, 2.4.2
LIM, 2.3.2

2.5 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 (unless **BIND-FIELDS** contains **TRUE**). It is more like a mask, which is used whenever it is appropriate to access data.

40   ⟨*bind-fields* 40⟩≡   (4)

```
   VARIABLE BIND-FIELDS   FALSE BIND-FIELDS !
```

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.

2.5.1   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 2.5.2.

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:

```
( PEOPLE file records)
24 BYTES NAME ( Last name first)
24 BYTES STREET ( Street address)
10 BYTES CITY,STATE
DOUBLE ZIP ( zip code, US only)
NUMERIC AREA ( area code)
DOUBLE PHONE ( phone number )
NUMERIC >DETAIL ( link to DETAIL file)
( For employees:)
DOUBLE SS# ( Social sec.  number)
DOUBLE FICA
DOUBLE GROSS ( Gross income ytd)
DROP
```

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") or use it to define a **CONSTANT**. 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 2.5.2).

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**:

41

NAME	STREET	CITY	ZIP

AREA	PHONE	SS#	FICA	GROSS

Figure 6:

NAME	STREET	CITY	ZIP

AREA	PHONE

Figure 7:
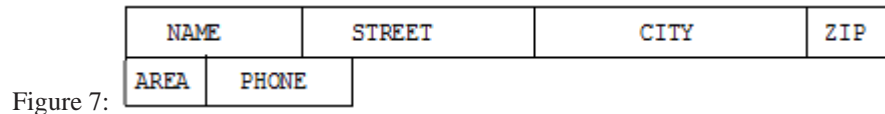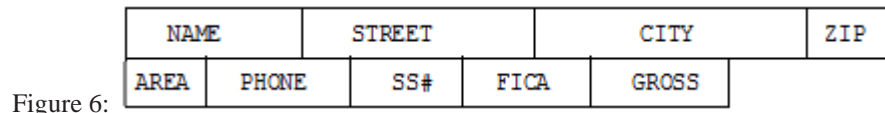
NOTE: Due to the Data Base Support package's record-allocation scheme, the first field of an active record may never contain a zero in its first four bytes. In our example, this is not a problem because the first field is alphanumeric (even blanks are stored as decimal 32's). Otherwise, we would have to rearrange the order of the fields so that one, which will never contain zero, is first.

REFERENCES

Available Records, 2.4.2
Field Types, 2.5.2

2.5.2   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!**, **1?**, and **?1**.

NUMERIC These fields occupy two bytes of storage (on 32-bit systems also).

Example: **NUMERIC WEIGHT**

Words defined by **NUMERIC** return an address, suitable for use with the numeric field access operators **N@**, **N!**, **N?**, and **?N**.

LONG These fields occupy four bytes of storage (only available on 32-bit systems).

Example: **LONG WEIGHT**

Words defined by **LONG** return an address, suitable for use with the numeric field access operators **L@**, **L!**, **L?**, and **?L**.

DOUBLE This field is for a 64-bit (8-byte) value.

Example: **DOUBLE SALARY**

Words defined by **DOUBLE** return an address, suitable for use with the double field access operators **D@**, **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.

43 ⟨*fields* 43⟩≡ (4)

```
: (FIELD) ( o n -- o+n )   CREATE  OVER , +  DOES> ( a -- a' )   @  FLD# @ + ;

0 VALUE #FIELD

#FIELD CELL (FIELD) FIELD-OFFSET \ holds offset within the record.
       CELL (FIELD) FIELD-SIZE   \ holds the size of the field.
       CELL (FIELD) FIELD-COPIES \ holds numbers of times this field is repeated.
       CELL (FIELD) BOUND-FILE   \ holds addr of FCB this record is bound to, or 0.
TO #FIELD

: CREATE-FIELD ( o n -- o+n )
   HERE DUP FLD# !  #FIELD DUP ALLOT  ERASE
   2DUP FIELD-SIZE !  FIELD-OFFSET !  +
   0 FIELD-COPIES !  BIND-FIELDS @ IF
       DB# @ BOUND-FILE !
   ELSE  0 BOUND-FILE !
   THEN ;

: FIELD-DOES ( a -- a' )
   FLD# !  WORKING FIELD-OFFSET @ +
   BOUND-FILE @ ?DUP IF
       SET-DATA  #REC @ R# !
   THEN  FIELD-COPIES @ IF
       #INDEX @  FIELD-SIZE @ * +
   THEN ;

: 1BYTE ( o _ -- o+1 ) \ 1BYTE fields occupy 8 bits.
   CREATE 1 CREATE-FIELD  DOES> ( -- a )   FIELD-DOES ;

: NUMERIC ( o _ -- o+2 ) \ NUMERIC fields occupy 16 bits.
   CREATE 2 CREATE-FIELD  DOES> ( -- a )   FIELD-DOES ;

: LONG ( o _ -- o+4 ) \ LONG fields occupy 32 bits.
   CREATE 4 CREATE-FIELD  DOES> ( -- a )   FIELD-DOES ;

[DEFINED] SFALIGN [IF]
: FLOAT ( o _ -- o+4 ) \ FLOAT fields occupy 32 bits.
   CREATE 4 CREATE-FIELD  DOES> ( -- a )   FIELD-DOES ;
[THEN]
```

```
: DOUBLE ( o _ -- o+4 ) \ DOUBLE fields occupy 64 bits.
   CREATE 8 CREATE-FIELD  DOES> ( -- a )   FIELD-DOES ;

: BYTES ( o n _ -- o+n ) \ BYTES fields occupy a specified number of bytes.
   CREATE CREATE-FIELD  DOES> ( -- n a )   FIELD-DOES  FIELD-SIZE @ SWAP ;
```

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.

44a   ⟨*filler* 44a⟩≡   (4)
```
[defined] -warning [IF] -warning [THEN]
: FILLER ( o n _ -- o+n )   BL WORD DROP + ;
[defined] +warning [IF] +warning [THEN]
```

Example: **6 FILLER <any name>**

FILLER creates no dictionary entry, but simply discards the word that follows.

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.

INDEX sets the current field index. Used to select one of the fields in a **COPIES** field.

44b   ⟨*indexed* 44b⟩≡   (4)
```
: INDEX ( n -- )   #INDEX ! ;
: INDEXED ( -- n )   #INDEX @ ;
```

COPIES makes an array of the previous field. Use **INDEX** to access the individual elements.

44c   ⟨*copies* 44c⟩≡   (4)
```
: COPIES ( o n -- o' )
   FIELD-COPIES @ ABORT" Invalid COPIES"  DUP
   FIELD-COPIES !  1- FIELD-SIZE @ * + ;
```

REFERENCES

Access to the Record Image in Working Storage, 2.5.5
Available Records, 2.4.2
Direct Access to Fields, 2.5.4
Field Reference Operators, 2.5.3
Working Storage, 2.1.4

2.5.3   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 2.4.1). 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 - b) Fetches the contents of a **1BYTE** field to the top of the stack.

1! (b a -) Stores a byte into the **1BYTE** field whose address is on top of the stack.

45a   ⟨*byte* 45a⟩≡   (4)
```
   : 1@ ( a -- n )   ADDRESS C@ ;
   : 1! ( n a -- )   ADDRESS C! TOUCH ;
```

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.

45b   ⟨*numeric* 45b⟩≡   (4)
```
   : NU@ ( a -- n )   ADDRESS 2 nC@ ;
   : N@ ( a -- n )   ADDRESS 2 nC@
      DUP 32768 AND IF  -65536 OR  THEN ;
   : N! ( n a -- )   ADDRESS 2 nC! TOUCH ;
```

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.

L@ (a - n) Fetches the contents of a **LONG** field to the top of the stack.

L! (n a -) Stores a number into the **LONG** field whose address is on top of the stack.

45c   ⟨*long* 45c⟩≡   (4)
```
   : LU@ ( a -- n )   ADDRESS 4 nC@ ;
   : L@ ( a -- n )   ADDRESS 4 nC@
      DUP 2147483648 AND IF  -4294967295 OR  THEN ;
   : L! ( n a -- )   ADDRESS 4 nC! TOUCH ;
```

L? (a) Fetches and displays the contents of a **LONG** field.

?L (a) As for **L?**, except the results are right-justified by the report generator.

FL@ FL! move 32-bit floats between the stack and the file.

46a   ⟨*float* 46a⟩≡   (4)

```
[DEFINED] SFALIGN [IF]

SFALIGN HERE 1 SFLOATS ALLOT CONSTANT DBFLOAT

: FL@ ( a -- r )   ADDRESS 4 nC@ DBFLOAT ! DBFLOAT SF@ ;
: FL! ( r a -- )   DBFLOAT SF! DBFLOAT @ SWAP ADDRESS 4 nC! TOUCH ;

: FL@-le ( a -- r )   ADDRESS 4 c@-le DBFLOAT ! DBFLOAT SF@ ;
: FL!-le ( r a -- )   DBFLOAT SF! DBFLOAT @ SWAP ADDRESS 4 c!-le TOUCH ;

: FL@-be ( a -- r )   ADDRESS 4 c@-be DBFLOAT ! DBFLOAT SF@ ;
: FL!-be ( r a -- )   DBFLOAT SF! DBFLOAT @ SWAP ADDRESS 4 c!-be TOUCH ;

: FL@+ ( a -- r )   ADDRESS SF@ ;   \ These are fast, but endian dependant
: FL!+ ( r a -- )   ADDRESS SF! TOUCH ;
[THEN]
```

D@ (a - d) Fetches the contents of a **DOUBLE** field to the top of the stack (two cells).

D! (d a -) Stores two cells into the **DOUBLE** field whose address is on top of the stack.

46b   ⟨*double* 46b⟩≡   (4)

```
: D@ ( a -- d )   ADDRESS  CELL CASE
       2 OF  4 nC@   ENDOF
       4 OF  8 nC@   ENDOF
       8 OF  8 nC@ 4294967296 /MOD SWAP   ENDOF
       DUP OF  1 ABORT" CELL size not supported"  ENDOF
    ENDCASE ;
: D! ( d a -- )   ADDRESS  CELL CASE
       2 OF  4 nC!   ENDOF
       4 OF  8 nC!   ENDOF
       8 OF  SWAP 4294967296 + 8 nC!   ENDOF
       DUP OF  1 ABORT" CELL size not supported"  ENDOF
    ENDCASE  TOUCH ;
```

D? (a) Fetches and displays the contents of a **DOUBLE** field.

?D (a) As for **D?**, except the results are right-justified by the report generator.

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**.

47a   ⟨*bytes* 47a⟩≡ (4)
```
: B@ ( n a -- )   ADDRESS S@ ;
: B! ( n a -- )   ADDRESS S! TOUCH ;
```

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.

47b   ⟨*access* 47b⟩≡ (4)
```
: N? ( a -- )   N@ . ;
: NU? ( a -- )   NU@ . ;
: L? ( a -- )   L@ . ;
[DEFINED] SFALIGN [IF]
: FL? ( a -- )   FL@ F. ;
[THEN]
: D? ( a -- )   D@ . ;
: 1? ( a -- )   1@ . ;
: B? ( n a -- )   2DUP B@ S.  SPACE ;
```

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.

47c   ⟨*put* 47c⟩≡ (4)
```
: PUT ( n a -- )   1 TEXT B! ;
```

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 **ACCEPT**) input from the keyboard, and copies it into a **BYTES** field using **PUT**.

47d   ⟨*ask* 47d⟩≡ (4)
```
: ASK ( n a -- )   FILE-PAD #TB 2DUP BLANK ACCEPT DROP B! ;
```

The word **ENTIRE** may be used in place of a field name:

Word Stack Action

ENTIRE (-- n a) Returns parameters for the "pseudo-field" that occupies the entire record in **BYTES** format.

48a ⟨*entire* 48a⟩≡   (4)
```
: ENTIRE ( -- n a )   B/R @ WORKING ;
```

For example: **ENTIRE B?**

types the contents of the current record as though it were a single **BYTES** field.

REFERENCES

Fetching Input to **PAD**, 2.1.3
Report Generator, 2.8

2.5.4 Direct Access to Fields

The Data Base Support package 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 (2.5.3). These addresses, however, are not the addresses of the actual data in a block buffer, but rather addresses within working storage (2.1.4). 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 can be an implied disk access, since it may call file I/O operators. It is important not to carry the address of a field in an I/O 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 I/O 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.

48b ⟨*address* 48b⟩≡   (4)
```
: RECORD ( n -- a )   'RECORD @ ?DUP IF EXECUTE THEN ;

: TOUCH ( -- )   'TOUCH @ ?DUP IF EXECUTE THEN ;

: ADDRESS ( a -- a' )   R# @ RECORD  WORKING -  + ;
```

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 TOUCH

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. **TOUCH** 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 **TOUCH** after the modification.

2.5.5   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 conjunction 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!**

Figure 8:

50   ⟨*strings* 50⟩≡   (4)
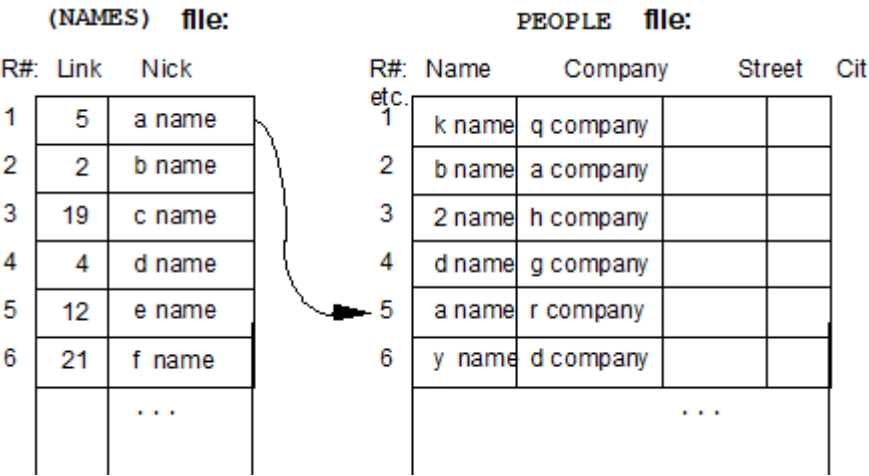```
: S@ (n a --)   FILE-PAD ROT >MOVE< ;
: S! (n a --)   FILE-PAD SWAP ROT >MOVE< ;
: (S.) (n a -- a' n')   DROP FILE-PAD SWAP -TRAILING ;
: S. (n a --)   (S.) xTYPE ;
```
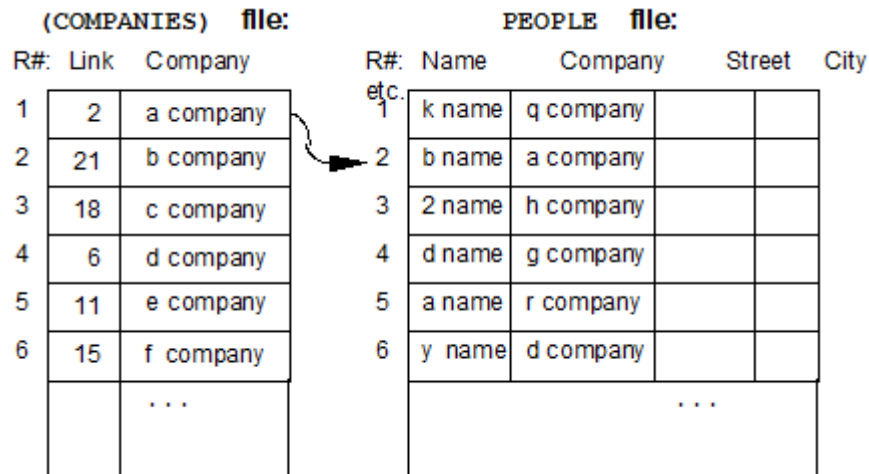
Figure 9:

The word **WORKING** returns the address of the beginning of the task's working storage. Note that there is no common standard for manipulating half-cell quantities (i.e. 16-bit values on a 32-bit system), so it is best to avoid using them within this context.

**REFERENCES**

Working Storage, 2.1.4

## 2.6   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 32-bit field called **LINK**. In Fig. Figure 8 on page 50, the index file **(NAMES)** contains the names of people, ordered alphabetically, along with links to the main file.

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, both sample name and observation number, using two separate index files, could index a file of scientific data. In Fig. Figure 9 on page 51, 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.

Another index to the same main file shown in Fig. Figure 8 on page 50, 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 four bytes of every record) to see whether each record is active.

52      ⟨*index* 52⟩≡                                                                                          (14)
           ⟨*head* 67a⟩
           ⟨*+ordered* 56⟩
           ⟨*-ordered* 58⟩
           ⟨*-binary* 55⟩
           ⟨*binary* 54b⟩
           ⟨*snatch* 68c⟩
           ⟨*-next* 67d⟩
           ⟨*first* 67c⟩
           ⟨*-locate* 67e⟩
           ⟨*chain* 68a⟩
           ⟨*unchain* 68b⟩

### 2.6.1   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 32-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 four bytes of each record in an index file contain the link to the associated record in the main file. polyFORTH ISD-4 predefines this field as **LINK**. The phrase:

**LINK L@**

reads the link field of the currently selected record and returns it on the stack.

When creating the record description (2.5.1) for an index file, you must skip over the **LINK** field by using the phrase **4 FILLER <name>** at the beginning of the layout, or by starting with a displacement of four rather than zero. For example:

```
0 4 FILLER LINK ( 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 *word-by-word*, rather than byte-by-byte. To accommodate this, you must make your key fields an even number of bytes in length. To allow the exchange of data with 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. The can be defeated by setting **REVERSE** to **FALSE**.

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.

2.6.2   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.

2.6.2.1   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 (2.5.5), 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. Figure 10 on page 54).

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.
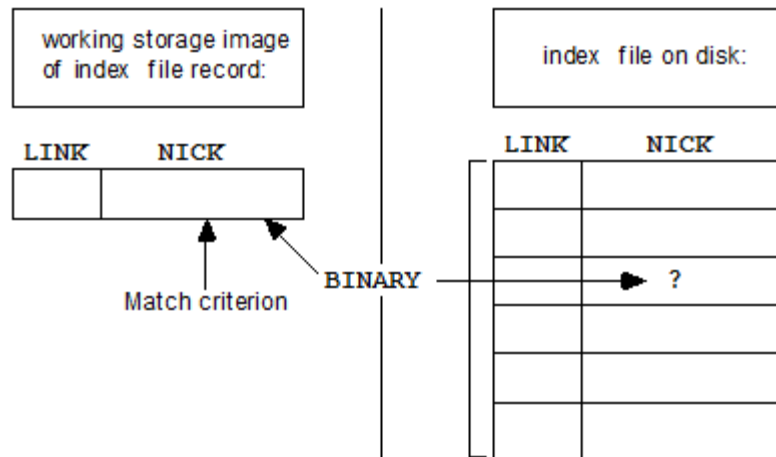
Figure 10:

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 (2.1.5).

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.

54a   ⟨*text* 54a⟩≡   (14)

```
  256 VALUE #TB        \ #TB is the size of the text input buffer.

  CREATE FILE-PAD   #TB ALLOT     \ Make a PAD for our exclusive use

  : >TEXT ( a n -- )   FILE-PAD #TB BLANK  FILE-PAD SWAP MOVE ;
  : TEXT ( c -- )   WORD COUNT  >TEXT ;
```

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#**.

54b   ⟨*binary* 54b⟩≡   (52)

```
  : BINARY ( n a -- n )   -BINARY  ORDERED RELEASE  ABORT" Unknown"  LINK L@ ;
```

(A related word, **-BINARY**, is discussed in 2.6.2.2.)

REFERENCES

Access to the record image in working storage, 2.5.5
Binary search principles, 2.6
R#, 2.4.1
TEXT, 2.6.2.1
Working Storage, 2.1.4

2.6.2.2 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. Figure 8 on page 50, let's consider what must happen when we add a new person to our database. First, must insert a new index record into the **(NAMES)** file in the appropriate place, and 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 2.4.3 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.

55 ⟨*-binary* 55⟩≡ (52)

```
: -BINARY ( n a -- f )     FILE-HANDLE @ 0= DUP >R IF  BIND-FILE   THEN
   SWAP  AVAILABLE 4 nC@ 2/ 1+ DUP READ  ORDERED GET  BEGIN
      DUP 1+ 2/  2OVER OVER ADDRESS  -TEXT 1- IF
         NEGATE
      THEN  R# +!  2/ DUP 0=
   UNTIL  DROP  2DUP OVER ADDRESS  -TEXT 0> ABS  R# +!
   OVER ADDRESS -TEXT  R> IF  -FILE  THEN ;
```

+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.

56   ⟨+*ordered* 56⟩≡   (52)

```
: RSWAP ( n a ra -- n a )    >R 2DUP R>  ROT 2/ 0 DO
      OVER 2 nC@  OVER 2 nC@  2OVER  >R 2 nC!  R> 2 nC!
      SWAP 2 +  SWAP 2 +
   LOOP  2DROP ;

: DIRECTION ( n -- n a rh rl )
   AVAILABLE 4 nC@ +
   AVAILABLE 4 nC! TOUCH  ENTIRE
   AVAILABLE 4 nC@ 2 +
   SAFE R# @ ;

: +ORDERED ( -- )
   FILE-HANDLE @ 0= DUP >R IF  BIND-FILE  THEN
   1 DIRECTION DO
        I RECORD  RSWAP TOUCH
   LOOP  2DROP  ORDERED RELEASE
   R> IF  -FILE  THEN ;
```

-BINARY expects the same conditions as **BINARY** (2.6.2.1):

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 package 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.

57   ⟨*ordered* 57⟩≡   (4)

```
  VARIABLE ORDERED   0 ORDERED !
```

REFERENCES

Binary Searches, 2.6.2.1
SAVE and **RESTORE**, 2.4.2
TEXT, 2.6.2.1

2.6.2.3   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.

58   ⟨*-ordered* 58⟩≡   (52)
```
: -ORDERED ( -- )
    FILE-HANDLE @ 0= DUP >R IF  BIND-FILE  THEN
    ENTIRE SWAP ERASE  -1 DIRECTION SWAP DO
        I RECORD RSWAP TOUCH
    -1 +LOOP  2DROP  ORDERED RELEASE
    R> IF  -FILE  THEN ;
```

Because the record that until now followed it will occupy the actual space that was occupied by the deleted record, 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, 2.6.2
SCRATCH, 2.4.3

2.6.3   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 Data Base application, with a "help screen" at the top, followed by the relevant file definitions. The help screen may be displayed any time by the command **PERSON-HELP**.

59   ⟨*personnel* 59⟩≡   (14)

```
: PERSON-HELP ( -- )
   CR ." PERSON-HELP   Display these PERSONNEL instructions."
   CR ." enter name    Enter a new person into the file with"
   CR ."                    access key of 'name'."
   CR
   CR ." remove name   Delete 'name' from the data base."
   CR
   CR ." fix name      Enter new information replacing all"
   CR ."                    current data for 'name'."
   CR
   CR ." view name     Display a person whose key is 'name'."
   CR
   CR ." v             Display current person."
   CR
   CR ." all           Display all records in the file."
   CR ;

 FILE Personnel.dbf
```

```
: /Personnel ( -- ) \ Runtime file setup
   Personnel.dbf  S" Personnel.dbf" >FILE
   FALSE REVERSE !  BIG-ENDIAN ;

( Bytes   records   origin           name   )
    16      300         0 BLOCK-DATA (PERSONNEL)
    128     300  +ORIGIN BLOCK-DATA  PERSONNEL
```

⟨*personnel-layout* 60a⟩
⟨*person* 60b⟩
⟨*digits* 61a⟩
⟨*!label* 61b⟩
⟨*enter* 61c⟩
⟨*fix* 61d⟩
⟨*remove* 61e⟩
⟨*.phone* 61f⟩
⟨*.zip* 62a⟩
⟨*.person* 62b⟩
⟨*view* 62c⟩
⟨*all* 62d⟩

The record layout for both the **PERSONNEL** and **(PERSONNEL)** files. The **LINK** is predefined, and subsequent fields are offset from the previous 4 fields. For example, the **NICKNAME** name is 12 bytes long starting in the 5th byte. **ZIP-CODE** is a 32-bit number, as is **PHONE-NUMBER**. **AREA-CODE** is single precision.

The offset for the field types is carried on the stack so that it may be either displayed or dropped at the end of the load. We use it in this case to display the record size.

60a   ⟨*personnel-layout* 60a⟩≡   (59)
```
    4 ( LINK)
   12 BYTES NICKNAME        ( Nickname, used as the key.)
   32 BYTES FULL-NAME       ( Full name, first name first.)
   32 BYTES STREET-ADDR     ( Street addr. or PO Box, etc.)
   32 BYTES CITY-STATE
      LONG ZIP-CODE         ( Note: can only handle US zips)
      NUMERIC AREA-CODE
      LONG PHONE-NUMBER
 .( Personnel.dbf records = ) .  .( Bytes ) CR
```

PERSON parses the input stream following it for the **NICKNAME** field. It leaves us pointing at the **NICKNAME** field in the **(PERSONNEL)** file.

60b   ⟨*person* 60b⟩≡   (59)
```
   : PERSON ( - n a)   1 TEXT  NICKNAME S!  (PERSONNEL) NICKNAME ;
```

60

DIGITS Prompts the terminal for input and converts it to binary on the stack.

61a   ⟨*digits* 61a⟩≡   (59)

```
: DIGITS ( - n)   QUERY  32 WORD COUNT  0 0 2SWAP
    BEGIN  >NUMBER  DUP
    WHILE  1 /STRING
    REPEAT  2DROP DROP ;
```

!LABEL Prompts for each field in order.

61b   ⟨*!label* 61b⟩≡   (59)

```
: !LABEL   CR ." Name: "   FULL-NAME ASK
     CR    ." Street: "   STREET-ADDR ASK
     CR    ." City, State: "   CITY-STATE ASK
     CR    ." Zip: "   DIGITS ZIP-CODE L!
     CR    ." Area: "   DIGITS AREA-CODE N!
           ." Phone: "   DIGITS PHONE-NUMBER L! ;
```

enter creates a new entry for the person whose nickname follows in the input stream, prompting for entry of additional data. If there is already an entry for that nickname, an error message is issued. In either case, the record remains the current one for future editing.

61c   ⟨*enter* 61c⟩≡   (59)

```
: enter   PERSON -BINARY IF  SAVE  PERSONNEL SLOT DUP
       READ  NICKNAME S@ NICKNAME B!  RESTORE  DUP LINK !
       +ORDERED  PERSONNEL READ !LABEL
    ELSE  ORDERED RELEASE  1 ABORT" Already known "
    THEN ;
```

fix accepts new data for the pre-existing entry whose nickname follows in the input stream.

61d   ⟨*fix* 61d⟩≡   (59)

```
: fix   PERSON BINARY  PERSONNEL READ !LABEL ;
```

remove deletes the person whose nickname follows from the data base.

61e   ⟨*remove* 61e⟩≡   (59)

```
: remove   PERSON  BINARY -ORDERED  PERSONNEL SCRATCH ;
```

.PHONE displays the **AREA-CODE** and **PHONE-NUMBER** as one would expect to see them.

61f   ⟨*.phone* 61f⟩≡   (59)

```
: .PHONE   AREA-CODE N@ 0  <# 41 HOLD # # #  40 HOLD #>
    TYPE SPACE  PHONE-NUMBER L@ 0 <# # # # #  45 HOLD ( -)
    # # # #> TYPE ;
```

.ZIP forces the **ZIP-CODE** to be displayed in **nnnnn** format.

62a ⟨*.zip* 62a⟩≡ (59)
```
: .ZIP   ZIP-CODE L@ 0 <# # # # # # #>  TYPE ;
```

n .PERSON displays the data from the nth record in the **PERSONNEL** data file.

62b ⟨*.person* 62b⟩≡ (59)
```
: .PERSON ( n)   PERSONNEL READ  CR  FULL-NAME B?  5 SPACES
    ." (" SPACE   NICKNAME B?  ." )"  CR   STREET-ADDR B?
   CR  CITY-STATE B? CR   .ZIP  10 SPACES  .PHONE  SPACE ;
```

view Parses the input stream and displays the proper record. **s** does the same thing using **R#** (the current record).

62c ⟨*view* 62c⟩≡ (59)
```
: view   PERSON BINARY  .PERSON ;
: v   R# @ .PERSON ;
```

all uses the **RECORDS** word which returns the initial value and number of records+1 in the data file. The loop counter is used to access each record in the ordered index **(PERSONNEL)**, where the **LINK** field points to the data in the **PERSONNEL** file.

62d ⟨*all* 62d⟩≡ (59)
```
: all   (PERSONNEL) RECORDS DO
        I (PERSONNEL) READ  LINK L@ .PERSON   CR
     LOOP  SPACE ;
```

Here is a sample of the output of **all**:

```
Andrews, Carl      ( Carl )
1432 Morriston Ave.
Parkerville, PA
17214          (717) 555−9853

Cook, Dottie       ( Dot )
154 Sweet Rd.
Grand Prairie, TX
75050          (214) 642−0011

Chapel, Doug       ( Doug )
75 Fleetwood Dr.
Rockville, MD
20852          (301) 777−1259

Boehning, Greg       ( Greg )
POB 41256
Santa Cruz, CA
95061          (408) 666−7891
```

2.6.4 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, 2.3

2.7 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.

2.7.1 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)?
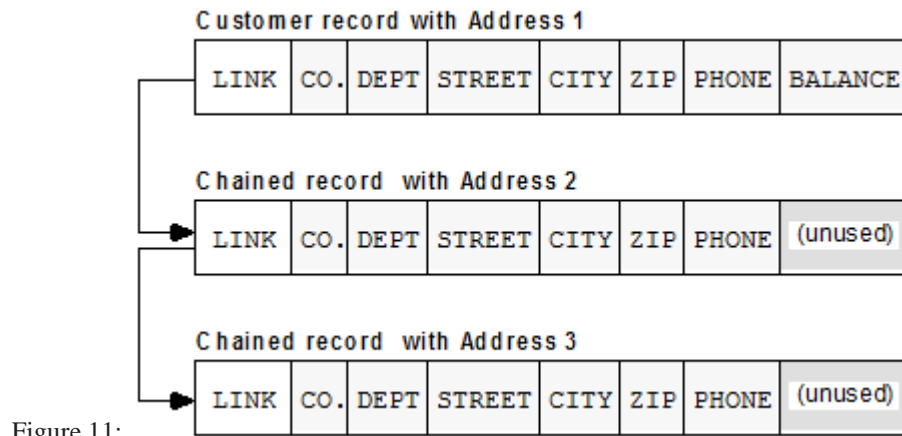
Figure 11:

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. Figure 11 on page 64 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 **Glossary** application included with polyFORTH ISD-4 (see 2.10). 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 **Glossary** 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. Figure 12 on page 65 shows the record structure for the **Glossary**. 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.

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. Figure 13 on page 65 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
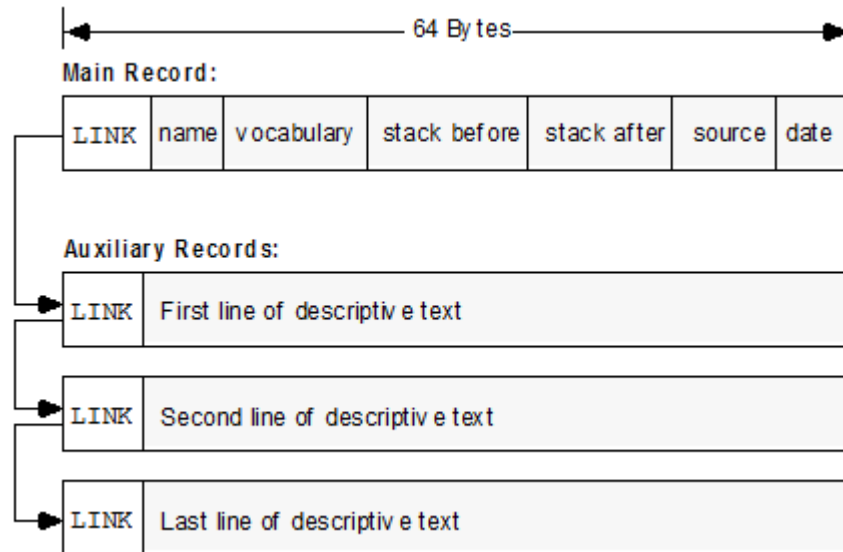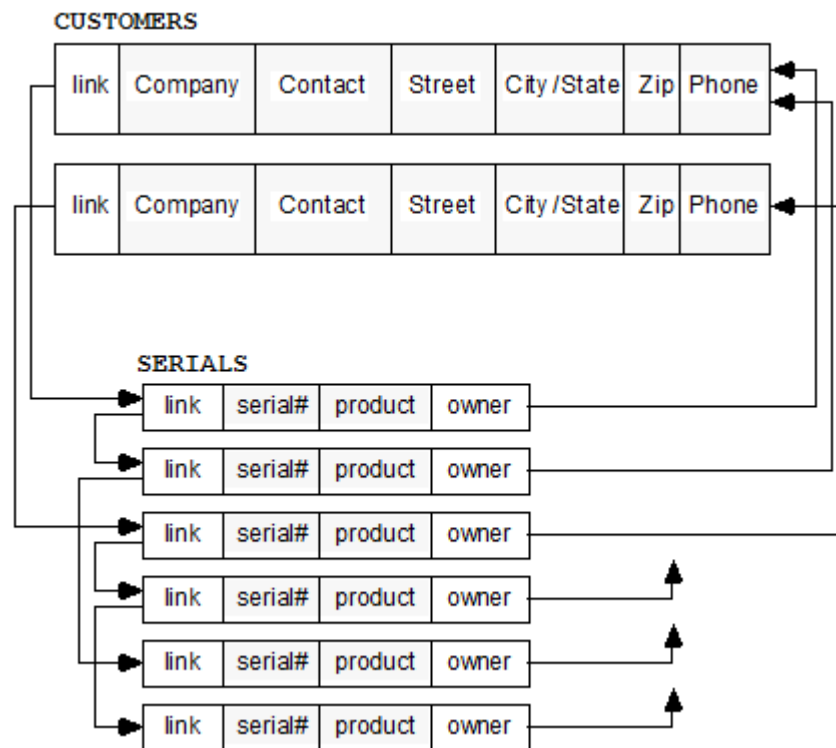
Figure 12:

Figure 13:

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 **Glossary** 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. Figure 13 on page 65, 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.

2.7.2   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 ISD-4 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 2.7.1. 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.

67a   ⟨*head* 67a⟩≡   (52)
```
   #USER CELL +USER HEAD TO #USER
```

LINK (-- a) A pre-defined field (the first four 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.

67b   ⟨*link* 67b⟩≡   (4)
```
   0 LONG LINK DROP
```

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.

67c   ⟨*first* 67c⟩≡   (52)
```
   : FIRST ( -- )   HEAD @ READ ;
```

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."

67d   ⟨*-next* 67d⟩≡   (52)
```
   : -NEXT ( -- t )   LINK L@  DUP 0> IF  READ 1  THEN  1- ;
```

-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.

67e   ⟨*-locate* 67e⟩≡   (52)
```
    : -LOCATE ( n -- r t )
       1+ FIRST  BEGIN
          1- DUP WHILE
          -NEXT IF  EXIT
       THEN  REPEAT ;
```

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.

68a   ⟨*chain* 68a⟩≡   (52)
```
: CHAIN ( n -- )     -LOCATE DROP  ( nth record or end )
   SLOT LINK OVER  SNATCH  SWAP READ  LINK L! ;
```

UNCHAIN (n) Removes the nth record from the chain.

68b   ⟨*unchain* 68b⟩≡   (52)
```
: UNCHAIN ( n -- )    DUP 0= ABORT" Won't"  -LOCATE ABORT" Not found"
   SAVE  LINK L@ READ  LINK 0 SNATCH  RESTORE  LINK L! ;
```

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.

68c   ⟨*snatch* 68c⟩≡   (52)
```
: SNATCH ( a r -- r )   OVER L@  SWAP ROT L! ;
```

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

FILES Load Block, 2.1.5

2.7.3 Application Examples

This section offers coded solutions to two application problems.

We introduced the **Glossary** program, which is included with your polyFORTH system, in 2.7.1. The use of this utility is more thoroughly documented in 2.10.

The word **(SHOW)** includes this sequence:

```
...  ( display data from the main record)
BEGIN +L -NEXT 0= WHILE
10 SPACES PHRASE B?
REPEAT ;
```

The word **+L** is similar to **CR**; see 2.8.4.

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 2.7.1, 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 2.8.4, 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, and 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 the **Customers** example, 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.

An application example using the standard polyFORTH chaining operators.

70   ⟨*customers* 70⟩≡   (14)

```
  FILE Customers.dbf

  : /Customers ( -- ) \ Runtime file setup
     Customers.dbf  S" Customers.dbf" >FILE
     TRUE REVERSE !  LITTLE-ENDIAN ;

  70 500       0 BLOCK-DATA CUSTOMERS
  46 500 +ORIGIN BLOCK-DATA SERIALS

  ( CUSTOMERS records:)
  0  4 FILLER LINK  ( LINK to 1st serial#)
     20 BYTES COMPANY
     16 BYTES CONTACT
     30 BYTES ADDR
  DROP

  ( SERIALS records:)
  0  4 FILLER LINK  ( LINK to next serial#)
     10 BYTES SERIAL#
      NUMERIC PRODUCT  ( product code)
      NUMERIC OWNER  ( link to owner CUSTOMERS record)
  DROP

  : edit-company   CR  ." Company name? "  COMPANY ASK
     CR  ." Contact? "  CONTACT ASK
     CR  ." Address? "  ADDR ASK ;
  : new-company   CUSTOMERS  SLOT DUP . READ  edit-company ;

  VARIABLE ALT  TRUE ALT ! \ Another approach

  ⟨alt-locate 72a⟩

  : (add)   CR  ." Serial# ? "   SERIAL# ASK ;
```

```
: add    ALT @ IF   ⟨alt-add 72b⟩
    ELSE   SAVE   LINK L@ DUP 0< IF  ( empty chain) DROP
           SAVE   SERIALS SLOT   RESTORE
           DUP LINK L!   SERIALS READ
        ELSE   HEAD !   SERIALS -1 CHAIN ( add at end)
        THEN (add)   RESTORE
    THEN ;

: edit-serial ( n)   SAVE SERIALS  1- -LOCATE ABORT" Can't"
    CR   Serial# B?  (add)   RESTORE ;

: .company   COMPANY B?   CONTACT B?   ADDR B? ;

: .companies   CUSTOMERS   RECORDS ?DO
    CR   I .   I READ  .company  LOOP ;

: .serials   0   SERIALS   FIRST BEGIN
    CR 1+ DUP .   Serial# B?  -NEXT UNTIL   DROP ;
: all-serials   ALT @ IF   ⟨alt-serials 72c⟩
    ELSE   LINK L@  0>  IF
           LINK L@  HEAD !  SAVE  .serials  RESTORE
    THEN   THEN ;

: show ( n)   CUSTOMERS READ  CR .company  all-serials ;
```

71

The word **edit-serial** 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 **show** displays the given company data, followed by a list of all associated serial numbers. Again, since there may be no chain at all, **all-serials** must make a decision. The test **LINK L@ 0>** returns 'true' if the link is positive (that is, not -1 or 0), indicating the first record in the chain. In this event, **all-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:

72a   ⟨*alt-locate* 72a⟩≡   (70)
```
   : VALID ( n – t)    0 OVER < DUP   IF
      SWAP READ  ELSE  SWAP DROP   THEN ;
   : ALT-FIRST ( – t)   HEAD @ VALID ;
   : ALT-NEXT ( – t)   LINK L@ VALID ;
   : ALT-LOCATE ( n – t)   ALT-FIRST IF BEGIN  DUP WHILE
         1-  ALT-NEXT 0= IF  DROP -1 EXIT  THEN
      REPEAT  ELSE DROP -1 THEN ;
```

72b   ⟨*alt-add* 72b⟩≡   (70)
```
   LINK L@ HEAD !  SAVE  SERIALS
      ALT-FIRST IF  -1 CHAIN  ELSE ( no chain)
         SLOT DUP RESTORE  LINK L!  SAVE
         SERIALS READ  THEN
      (add)  RESTORE
```

72c   ⟨*alt-serials* 72c⟩≡   (70)
```
   LINK L@ HEAD !  SAVE  0
      SERIALS  ALT-FIRST BEGIN  WHILE  CR  1+ DUP .
         SERIAL# B?  ALT-NEXT REPEAT  RESTORE  DROP
```

In the first block, **ALT-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 **ALT-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 **ALT-LOCATE** in this block. As usual, **ALT-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 **ALT-FIRST** for the test.

Because of the way we have rewritten **ALT-FIRST**, **all-serials** no longer needs an **IF** statement at all. The only conditional is **WHILE**, which gets its argument the first time around from **ALT-FIRST**, and henceforth from **ALT-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, 2.9

2.8   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.

73   ⟨*reports* 73⟩≡   (14)
    ⟨*cols* 80⟩
    ⟨*page* 81a⟩
    ⟨*register* 86a⟩
    ⟨*totals* 86b⟩
    ⟨*sum* 86c⟩
    ⟨*foot* 87a⟩
    ⟨*sub* 87b⟩
    ⟨*total* 87c⟩
    ⟨*grand* 87d⟩
    ⟨*heading* 82⟩
    ⟨*title* 84⟩
    ⟨*title-heading* 76b⟩
    ⟨*rlc* 78⟩
    ⟨*layout* 76a⟩
    ⟨*output* 77⟩

The following example will serve as a quick introduction to the Report Generator. It assumes the fields defined in the example in 2.1. Here is the code:

74   ⟨*all-people* 74⟩≡   (25)

```
: display.person   NAME ?B  STREET ?B  CITY ?B  STATE ?B  ZIP ?B ;

[R               People\Name        \Address            \City          \St.\Zip ]
   CONSTANT PEOPLE-TITLE

: all-people   PEOPLE-TITLE LAYOUT  +L
   PEOPLE RECORDS ?DO  I READ  display.person  +L  LOOP ;
```

This produces:

```
Page 1   07/30/2012
                    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 **display.person** is defined similarly to the version given in 2.1, 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). 2.8.3 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. Note that the line is shown wrapping to the next line here, but that it must be on the same line in the source code. See 2.8.2 for more on **[R**.

The final word, **all-people** prints the tabulated report. It begins by invoking **PEOPLE-TITLE** 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 **all-people**. Then, **+L** forces a new-line.

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, 2.8.3
Page Banner, 2.8.4

2.8.1   Specifying a Title/Column-Heading Pair

A single word, **[R**, lets you specify both the title and the column headings. The set-up phrase usually appears just preceding the definition of the report for which they are designed.

The format for a title/column heading pair is:

[R title-text \column-heading-text]

The entire title/heading pair statement must not extend over multiple lines.

All characters up to the backslash, except the first blank that follows **[R**, are used for the title text.

All characters that follow the first ****, ending with the delimiter **]**, are used for the heading. The first character (blank or non-blank) that follows the first \ corresponds to the first column of the report page. The following backslashes determine where the actual column positions are located.

In addition to being saved in the dictionary, the heading text is parsed at the time the sourcce 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 word set 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 will be shifted right to accommodate it.

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.

76a   ⟨*layout* 76a⟩≡   (73)
```
: LAYOUT ( a -- )   RPT !  0 P# !  +PAGE ;
```

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 ;
```

An additional word, **[R+** provides additional functionality by parsing the word that immediately follows it and executing this word on the first line of each page. Otherwise, its behavior is identical to **[R**.

76b   ⟨*title-heading* 76b⟩≡   (73)
```
: (R) ( -- a )   ['] TITLE ,  HERE  HERE 0 ,  HERE 0 ,
    [char] \ WORD COUNT ,string DROP  HERE SWAP !  [char] ] WORD COUNT ,string
    HERE DUP ALIGNED  SWAP ?DO  0 C, LOOP
    HERE ROT !  COUNT  DUP ROT 0  ROT 0 DO
        OVER I + C@  [char] \ = IF
            I SWAP - ,  BL OVER I + C!  I 1+
    THEN  LOOP  NIP - ,  -1 , ;

: [R+ ( -- a ) \ Usage: [R+ <name> <title> \ <col> \ ... \ <col> ]
    ' , (R) ;
: [R ( -- a ) \ Usage: [R <title> \ <col> \ ... \ <col> ]
    0 , (R) ;
```

REFERENCES

+PAGE, 2.8.4

2.8.2   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).

.L (n) Displays the single-length integer *n* right justified in the next column, in the format used by **L.**.

.D (d) Displays the double-length integer *d* right justified in the next column, in the format used by **D.**

?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).

?L (a) Displays the contents *a* address as a single-length integer *n* right justified in the next column, in the format used by **L.**.

?D (a) Displays the contents *a* address as a single-length integer *n* right justified in the next column, in the format used by **D.**.

?1 (a) Displays the contents of the specified **1BYTE** field, right justified in the next column.

?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/yyyy format.

.D/M/Y (n) Given a Julian date, displays it in the next report column. Since this routine invokes **(DATE)**, it will work with either calendar. Some data base applications prefer to use the dd-mmm-yyyy format.

.WHEN (n) Given the time in seconds, displays the hh:mm:ss in the next report column.

Each of these operators advances the columns table to the next column, determines the width of the new field, and then right-justifies the output string in this column.

77   ⟨*output* 77⟩≡   (73)

```
: (D.) ( d -- a n )
    SWAP OVER DUP 0< IF DNEGATE THEN
    <#  #S ROT SIGN  #> ;

: .N ( n -- )   DUP 0< (D.) RIGHT ;
: .L ( n -- )   DUP 0< (D.) RIGHT ;
```

```
[DEFINED] SFALIGN [IF]
: .FL ( F: r -- )   PAD 4 REPRESENT IF  <#
        >R 1- DUP ABS 0 #S 2DROP SIGN  [CHAR] e HOLD      \ exponent
        3 0 DO  PAD 3 + I - C@ HOLD  LOOP  [CHAR] . HOLD
        PAD C@ HOLD  R> IF  [CHAR] - HOLD  THEN  0 0 #>
    ELSE  S" ?.?e?"  THEN  RIGHT ;
[THEN]
: .D ( d -- )   (D.) RIGHT ;

: ?N ( a -- )   N@ .N ;
: ?L ( a -- )   L@ .L ;
[DEFINED] SFALIGN [IF]
: ?FL ( a -- )   FL@ .FL ;
[THEN]
: ?D ( a -- )   D@ .D ;
: ?1 ( a -- )   1@ .N ;
: ?S ( n a -- )   (S.) LEFT ;
: ?B ( n a -- )    2DUP B@  OVER FILE-PAD SWAP 4 MIN nC@ IF
        ?S  ELSE  2DROP SKIP-COL
    THEN ;

: .M/D/Y ( n -- )    ?DUP IF
        MM/DD/YYYY RIGHT  ELSE  SKIP-COL
    THEN ;
: .D-M-Y ( n -- )    ?DUP IF
        DD-MMM-YYYY RIGHT  ELSE  SKIP-COL
    THEN ;
: .WHEN ( n -- )    ?DUP IF
        BASE @ >R  0  <#
        DECIMAL #  6 BASE ! #  [char] : HOLD
        DECIMAL #  6 BASE ! #  [char] : HOLD
        DECIMAL # #  #>  R> BASE !
        RIGHT  ELSE  SKIP-COL
    THEN ;
```

You may also build your own formatting words to display columns, using the words **RIGHT**, **LEFT** and **CENTER**.

RIGHT (a n) Displays an alphanumeric string of length *n*, beginning at address *a*, right-justified in the next column.

LEFT (a n) Displays an alphanumeric string of length *n*, beginning at address *a*, left-justified in the next column.

CENTER (a n) Displays an alphanumeric string of length *n*, beginning at address *a*, centered in the next column.

78 ⟨*rlc* 78⟩≡ (73)

```
: RIGHT ( a n -- )   COLS OVER -  SPACES TYPE  SPACE ;
: LEFT ( a n -- )   COLS OVER - >R  TYPE  R> 1+ SPACES ;
: CENTER ( a n -- )   COLS OVER - DUP 2/  DUP >R - SPACES
    TYPE  R> 1+ SPACES ;
```

The stack arguments are identical to those of **TYPE**.

In fact, **.N**, **.L**, **.D**, **?N**, **?L**, **?D**, **?1**, 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 the string's trailing blanks are included, making them appear left justified.

Here is an example:

79   ⟨*accounts* 79⟩≡   (14)

```
FILE Accounts.dbf

: /Accounts ( -- ) \ Runtime file setup
    Accounts.dbf  S" Accounts.dbf" >FILE
    TRUE REVERSE !  LITTLE-ENDIAN ;

76 500 0 BLOCK-DATA ACCOUNTS

0  10 BYTES NAMES   NUMERIC ACCT#   DOUBLE BALANCE   DROP

: (.$) ( d - a n)   SWAP OVER DABS
    <#  # #  46 HOLD  #S  SIGN  #> ;
: .ACCOUNT   ACCT# ?N  NAMES ?B
    BALANCE D@ (.$) RIGHT ;

[R Account Balances\   Account#\Name          \Balance]
    CONSTANT ACCOUNTS-TITLE

: balances   ACCOUNTS-TITLE LAYOUT
    ACCOUNTS RECORDS ?DO  I READ  .ACCOUNT  LOOP ;

: enter-bal ( n d)   ACCOUNTS  SLOT READ  BALANCE D!
    ACCT# N!  NAMES PUT ;
```

The word **BALANCES** produces:

```
Page 1   05/12/2005
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.

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.

SKIP-COL ( ) Skips one column.

SKIP-COLS (n) Skips *n* columns.

80 ⟨*cols* 80⟩≡   (73)
```
  #USER CELL +USER RPT  \ holds the address of the current report title block.
      CELL +USER #COL \ holds the address of the current column width.
  TO #USER

  : 0COL ( -- )   RPT @ @ #COL ! ;
⟨line 81b⟩
  : COL ( -- n )   #COL @  DUP CELL+  #COL ! ;
  : COLS ( -- n )    COL @  DUP 0< 0= IF  EXIT
     THEN  DROP +L  COL @ ;

  : SKIP-COL ( -- )   COLS 1+ SPACES ;
  : SKIP-COLS ( n -- )   0 DO  SKIP-COL  LOOP ;
```

2.8.3   Controlling Paging

The report generator does not count each output line, since this capability tends to be too environmentally dependent. Instead, it assumes that the output can be captured and paginated appropriately. This does not, however, stop an application from doing its own page control.

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.

81a   ⟨*page* 81a⟩≡   (73)
```
  #USER CELL +USER P# TO #USER

  VARIABLE 'PAGE    ' PAGE 'PAGE !

  : +PAGE ( -- )
      'PAGE @ EXECUTE  1 P# +!  +L  ." Page "  P# ?
      SPACE  DATE  SPACE  'APP @ COUNT TYPE
      +L  RPT @  CELL - @ ?DUP IF EXECUTE THEN ;
```

+L Issues a **CR** and used to increment the line count in **L#**, but that was removed in favor of shorter reports that do not have extra page headers. Also resets the column pointers using **0COL**.

81b   ⟨*line* 81b⟩≡   (80)
```
  : +L ( -- )   -spin  CR  0COL ;
```

REFERENCES

0COL, 2.8.3

2.8.4   The Page Banner

At the top of each page of the report appears the "page banner" which includes:

Page nn <date> <optional text>

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 **'APP** variable. The definition **APP" <optional text>"** is available to make this easier.

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.

82   ⟨*heading* 82⟩≡   (73)

```
: HEADING ( a -- )    DUP RPT !
   CELL+  DUP >R  CELL+ COUNT TYPE
   R@ 3 CELLS - @ ?DUP IF EXECUTE THEN  +L
   R> @  COUNT TYPE +L ;
```

2.8.5   How the Columns Table Works

The format of the columns table is:

Byte Contents

address - 8 Address of the optional routine executed at the end of the header.

- 4 Address of the page heading vectored routine.

+ 0 Address of column widths table.

+ 4 Address of column headings.

+ 8 Counted string of header.

+ 8 + (h) Counted string of columns.

+ 8 + (h) + (c) Column widths table.

+ 8 + (h) + (c) + n -1 marks end of column table.

A -1 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 looking for additional backslashes in a loop. Each time it encounters a backslash; it replaces it with a space and 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 -1) 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):

```
01234567890123456789012345678901234567890123456789
```

`\ Account#\Name \Balance]`

The backslash 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 backslash after the "Name" heading occurs at relative position 22; the difference, 10, is compiled as the width of the second column. The delimiting **]** occurs at 30, and the difference of 8 is compiled as the width of the third column. Finally, a -1 is compiled to indicate the end of the table.

The finished column table, as constructed by **`[R`**, contains:

`12 10 8 -1`

The total width of all columns equals the position number of the last non-blank character.

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 one character.

2.8.6   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 2.8.4;

2. Performs a **+L**;

3. Executes a word called **TITLE**. **TITLE** is defined as:

84   ⟨*title* 84⟩≡   (73)
```
: 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 (2.8.5).

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 second 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 are some examples:

```
( Non-standard Report Headings )

APP" Acme Manufacturing Co."

VARIABLE WHICH

[R \Col1 \Col2 \Col3] CONSTANT SHOW-TITLE1

:   .ITEM WHICH ? ;
:   ."ITEM" ." Report on Item No.  " .ITEM ;
:   'ITEM' ."ITEM" TITLE ;

:   SHOW1 ( n) WHICH !
['] 'ITEM' SHOW-TITLE1 CELL- !
SHOW-TITLE1 LAYOUT ;

[R+ ."ITEM" \Col1 \Col2 \Col3] CONSTANT SHOW-TITLE2

:   SHOW2 ( n) WHICH !
SHOW-TITLE2 LAYOUT ;

[R+ .ITEM Report on Item No.  \Col1 \Col2 \Col3] CONSTANT SHOW-TITLE3

:   SHOW3 ( n) WHICH !
SHOW-TITLE3 LAYOUT ;
```

This example shows a report for some particular item that is selected numerically, like this:

2500 SHOW1 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:

```
Page 1 05/18/2005 Acme Manufacturing Co.
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 examples to vector the user-defined code into the new-page routine:

1. Create a title/column heading pair as usual (cases 1 and 2, leave the "title" blank).

2. Define words, 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. Finally it invokes **TITLE**, which displays the title/column-heading pair.

3. Define the report-generating word (the word **SHOW1**) in the usual way, using the address of the title to set the vectored location (1 cell back).

4. Alternatively, use **[R+** to create titles that have custom execution (the words **SHOW2** and **SHOW3**).

All of these reports have the same output, and other reports may be co-resident; since each has its own title/ column-heading table, each has its own new-page execution behavior.

2.8.7   Totals and Subtotals

The Data Base Support package 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 (2.1.5). The value of *n* is calculated by this Forth phrase:
   (# of registers needed) 3 * 2 * CELLS 2 CELLS + 16 +
   See the source comment associated with the word **REGISTER** on your system.

86a   ⟨*register* 86a⟩≡   (73)
```
: REGISTER ( -- a )   WORKING 16 + ;
```

1. 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 **TOTALS** (see below).

2. As the fields are being displayed, accumulate the values in the subtotal registers by using either **SUM** or **FOOT**.

3. When you wish to display the subtotals (if at all), use the word **SUB**, followed by an appropriate numeric output command.

4. When you wish to display the totals, invoke **TOTAL**. This copies the totals to the subtotals registers and adds them to the grand totals. Then use **SUB** as in Step 3.

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

TOTALS (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 TOTALS
creates three subtotal accumulators and three grand-total accumulators, and sets all to zero. **TOTALS** must be used at the beginning of a report if any of the following words are used.

86b   ⟨*totals* 86b⟩≡   (73)
```
: TOTALS ( n -- )   2* CELLS REGISTER  OVER 3 * 2 CELLS + ERASE  REGISTER ! ;
```

SUM (d n) Adds *d* to the subtotal accumulator for the nth relative field.

86c   ⟨*sum* 86c⟩≡   (73)
```
: SUM ( d n -- )   2* CELLS REGISTER + 2+! ;
```

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.

87a   ⟨*foot* 87a⟩≡   (73)
```
: REG ( -- a )   REGISTER 2@ MOD  2 CELLS + DUP
   REGISTER CELL+ !  REGISTER + ;

: FOOT ( d -- d )   2DUP REG 2+! ;
```

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.

87b   ⟨*sub* 87b⟩≡   (73)
```
: SUB ( -- d )   REG  DUP >R  2@  2DUP REGISTER @ R@ + 2+!  0 0 R> 2! ;
```

TOTAL ( ) Adds the totals to the grand totals and copies the totals to the subtotal accumulators. It leaves the totals in a state such that the display of the subtotals will set the totals to 0.

87c   ⟨*total* 87c⟩≡   (73)
```
: TOTAL ( -- )    REGISTER DUP @ DUP 0 DO  >R
          2 CELLS + DUP DUP R@ + 2@  DUP 2OVER ROT ROT 2!
          DUP 2OVER ROT ROT R@ 2* + 2+!  DNEGATE ROT R@ + 2!  R>
       2 CELLS +LOOP  2DROP ;
```

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.

87d   ⟨*grand* 87d⟩≡   (73)
```
: GRAND ( -- )   REGISTER DUP @ >R  2 CELLS + DUP R@ 2* +  SWAP R> MOVE ;
```

The following example shows how subtotals and grand totals can be easily computed and displayed:

```
Page 1   05/17/2005
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:

88   ⟨*wines* 88⟩≡   (14)

```
  FILE Wines.dbf

  : /Wines ( -- ) \ Runtime file setup
     Wines.dbf  S" Wines.dbf" >FILE
     TRUE REVERSE !  LITTLE-ENDIAN ;

  28 500 0 BLOCK-DATA WINES

  0  16 BYTES Location   NUMERIC Chablis   NUMERIC Rose
      NUMERIC Champagne   DROP

  : .amounts   Location ?B  Chablis N@ S>D FOOT .D
     Rose N@ S>D FOOT .D  Champagne N@ S>D FOOT .D ;

  : .subs   SUB .D  SUB .D  SUB .D  +L ;

  [R Wine Inventory by Store\Location        \Chablis\  Rose\ Champagne]
     CONSTANT WINES-TITLE

  : INVENTORY   WINES-TITLE LAYOUT  3 TOTALS  +L
     ." Northern California"  +L
     WINES RECORDS DO  I READ  .amounts  I 4 = IF  +L
        SKIP-COL .subs +L  ." Southern California " +L
     THEN LOOP  +L
     SKIP-COL  .subs  ." Grand Total:     " COLS DROP
     TOTAL .subs ;
```

```
: enter-wine ( Cablis Rose Champagne -- )   WINES   SLOT READ
   Champagne N!  Rose N!  Chablis N!  Location PUT ;
```

The phrase **3 TOTALS** appears in the definition of **INVENTORY**. 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**:

Chablis N@ S>D 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; **S>D** 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@ S>D 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**. 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. 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

displays the subtotals for Southern California, and issues a **+L**.

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, **TOTAL** copies the total accumulators to the subtotal registers, and **.subs** displays these.

2.9   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 database needs.

In general, we can formulate two simple rules for planning your database:

1. Look at the kinds of information you have.

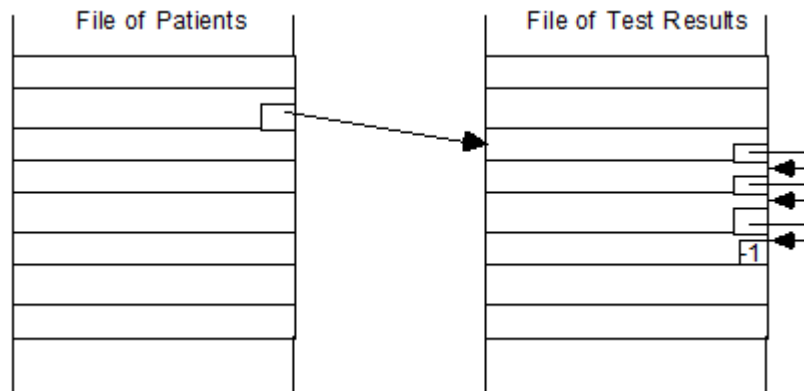2. Arrange like kinds of information into files.

Figure 14:

2.9.1   A Hospital Patient Management Data Base

Our goal in this example is to create a database 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 are 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 database designers would opt for variable-length records and fields. But variable-length records are complex and slow, as we saw in 2.1. With nearly the same convenience we can achieve the same results by using a fixed header plus a variable number of subordinate records.

This is where chains come in. Fig. Figure 14 on page 90 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. Figure 15 on page 91).

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. Figure 16 on page 91).

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.
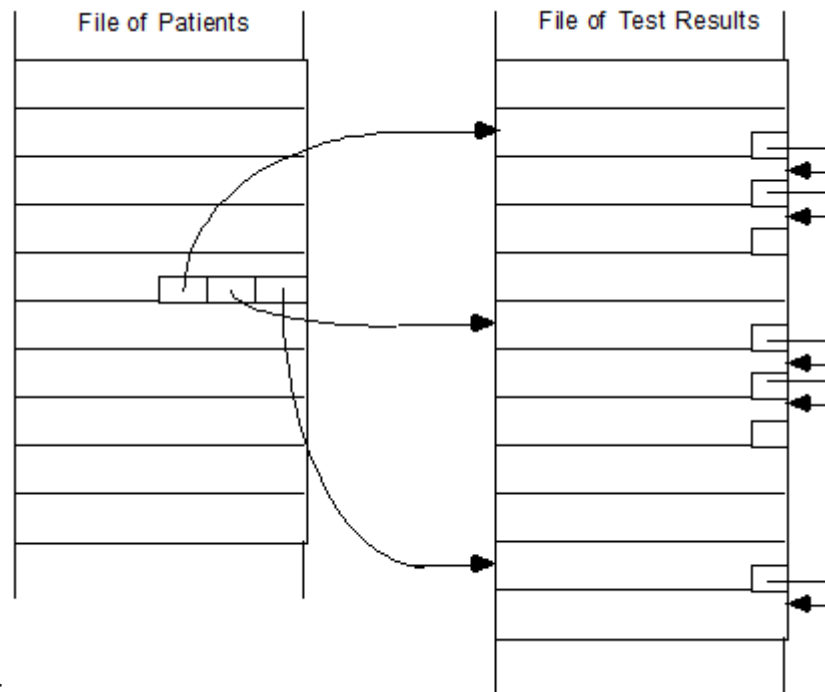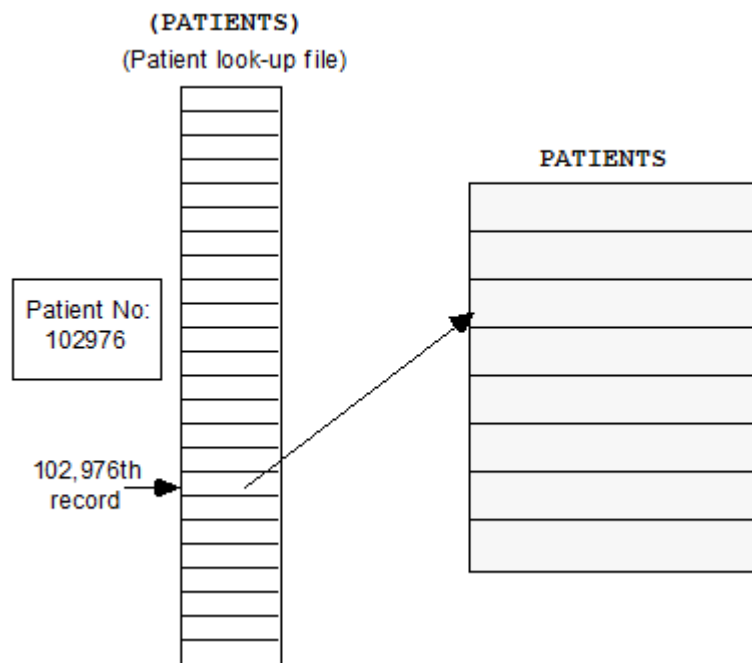
Figure 15:

Figure 16:

In general, direct access is much faster than searching, and should be used whenever appropriate.

2.9.2   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 database? 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 suggests 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
3343   47   626    29 NOV  1210  2100   10.74    0.00    10.74
3205        2270   10 OCT  2100  1030  779.74  779.74     0.00
2773   61   930    30 SEP  1210  2100   59.04   59.04     0.00
```

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 2.8.7.

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 this.

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. 2.16. As we saw in Section 2.7, the polyFORTH Data Base Support package 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 this balance sheet, 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
```

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 package 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**,
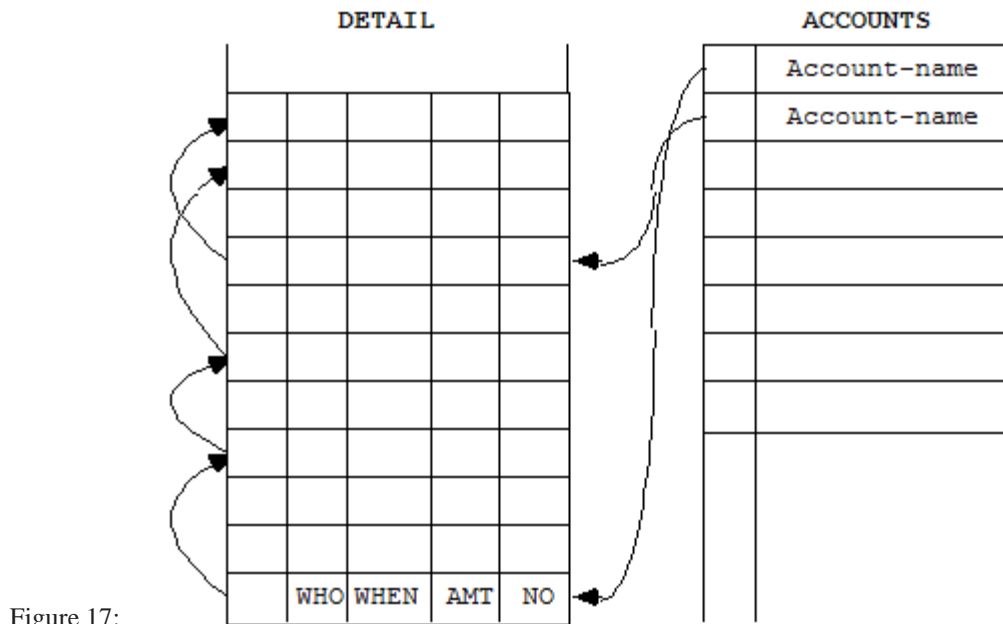
Figure 17:

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.

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, and 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. Figure 17 on page 95).

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 sub classify 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 database? 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. 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, and 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 database 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.)

2.9.3   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 database, a necessary strategy when dealing with thousands of anything. Another example in which coded numbers can reveal hierarchy is with inventory or parts numbers.

Diagram of the database 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 containing 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.
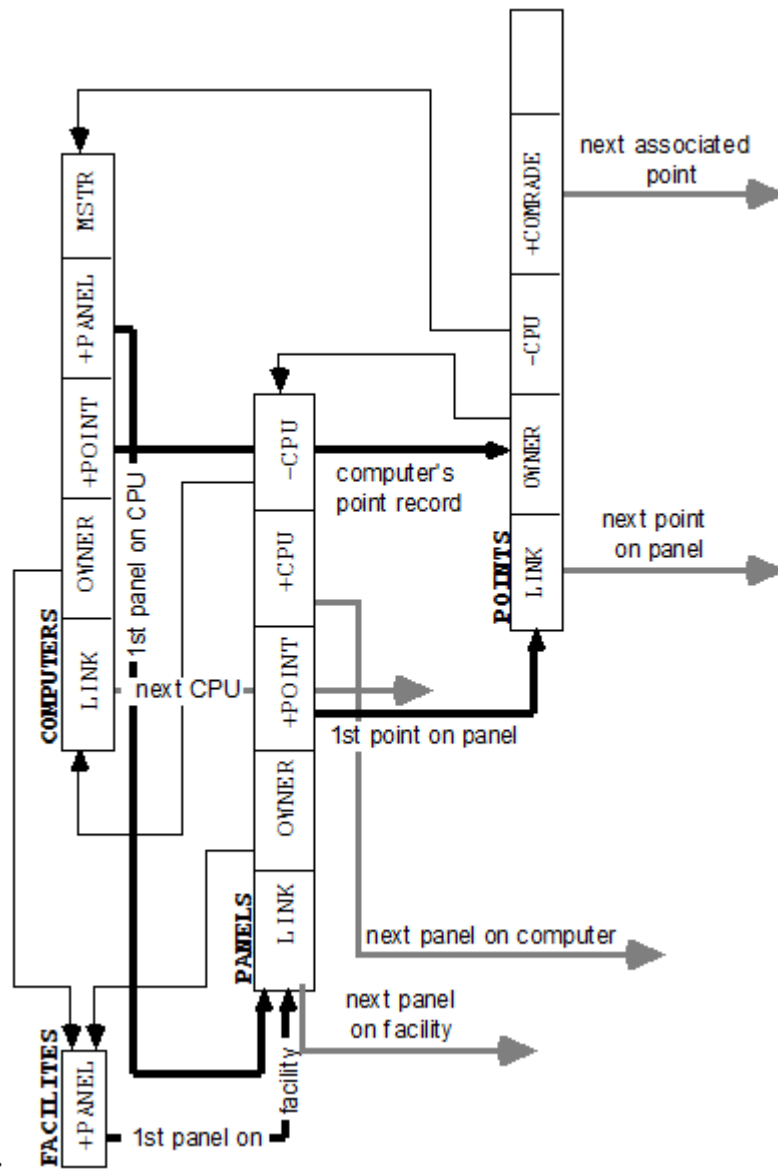
Figure 18:

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 database 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.

2.9.4   A Filing Scheme for Image Processing Applications

Our final example illustrates the flexibility the Data Base Support package 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 package 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.

2.10   Glossary Utility

The **Glossary** 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 **Glossary** utility provides commands to maintain this file and to print reports that include either selected glossary vocabularies or the complete file of words.

100   ⟨*glossary* 100⟩≡   (14)

```
 : HELP ( S: -- ) ( G: This is the  GLOSSARY  help screen )
  CR PAGE ." HELP               To display this  GLOSSARY  application"
      CR ."    This utility uses a disk file named Glossary.dbf"
      CR ." SUMMARY             Displays all words in the file."
      CR ." /VOCABULARY _____   Displays all words in vocabulary _____ ."
      CR ." VOCAB _____         Specifies vocabulary for entries & searches"
      CR ." n SOURCE 2 nC!       Specifies source block  n  for entries."
      CR ." NEW _____           Enters  ____  making it current."
      CR ."    Thereafter use  U  to enter descriptive text."
      CR ." FIND _____          Locates and shows _____ making it current."
```

```
    CR ." F                      Re-displays the current word."
    CR ." n AT                   Changes the current word's source to  n ."
    CR ." STACKS                 Enter 'input' and 'output' stack arguments."
    CR ." l T  ( or  t)          Types line  l (0-n)  making it current."
    CR ." P  ( or  p ) _____     Puts  _____  on the current line."
    CR ." U  ( or  u ) _____     Inserts  _____   under the current line."
    CR ." The date used for today's entries will be "  DATE ;
```

⟨*glossary-file* 102⟩
⟨*vocab* 103⟩
⟨*f* 106a⟩
⟨*stacks* 107b⟩
⟨*new* 104⟩
⟨*delete* 108a⟩
⟨*at* 107c⟩
⟨*t* 105b⟩
⟨*p* 105c⟩
⟨*u* 105a⟩
⟨*x* 105d⟩
⟨*find* 107a⟩
⟨*moved* 107d⟩
⟨*summary* 106b⟩
⟨*vocabulary* 106c⟩

2.10.1   File Structure

The **Glossary** 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 glossary.

You must pre-allocate 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 28-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.

102    $\langle$*glossary-file* 102$\rangle\equiv$ (100)

```
  FILE Glossary.dbf

  : /Glossary ( -- ) \ Runtime file setup
     Glossary.dbf  S" Glossary.dbf" >FILE
     TRUE REVERSE !  LITTLE-ENDIAN ;

  ( Bytes   records  origin              name )
     28       432    0        BLOCK-DATA (GLOSSARY)
     64      2300    +ORIGIN  BLOCK-DATA  GLOSSARY

  4 ( LINK )  24 BYTES NAME+VOC  DROP   ( Index )

  4 ( LINK )  12 BYTES WORD-NAME
              12 BYTES VOC
              16 BYTES BEFORE
              16 BYTES AFTER
               NUMERIC SOURCE
               NUMERIC ENTERED  DROP

  4 ( LINK )  60 BYTES PHRASE  DROP
```

2.10.2   Loading Instructions

As with the other examples, the **Glossary** is loaded automatically with the pFDatabase.fth

The procedure for entering a word into a **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.

2.10.3   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 2 nC!

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, 2.10.8

2.10.4   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

103   ⟨*vocab* 103⟩≡   (100)
```
  : VOCAB ( -- )   BL TEXT  VOC S! ;
```

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, 2.10.7
Finding Previously Entered Words, 2.10.8

2.10.5   Glossary Entries

Words are entered into the glossary through the **NEW** command. This command sets the basic entry into the file. It has the following format:

NEW word-name (*e.g.,* NEW 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 **Glossary** is twelve characters. If a longer word name is entered, its length will be truncated to twelve characters.

104   ⟨*new* 104⟩≡   (100)

```
 : (FIND) ( -- n a ) \ Usage: (FIND) <name>
     BL TEXT WORD-NAME S!  (GLOSSARY) NAME+VOC ;


 : (ENTER) ( -- ) \ Usage: (ENTER) <name>
     (FIND) -BINARY IF
         SAVE  GLOSSARY SLOT  DUP HEAD !  LINK !  RESTORE
         NAME+VOC S@  +ORDERED  NAME+VOC S!
         GLOSSARY FIRST  NAME+VOC B!
         SOURCE 2 nC@  SOURCE N!
     ELSE  ORDERED RELEASE  LINK L@  GLOSSARY READ
     THEN  @DATE ENTERED N! ;


 : NEW ( -- ) \ Usage: NEW <name>
     (ENTER)  STACKS  CR  64 SPACES  0 #L ! ;
```

Following **NEW**, 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, 2.10.8
T and **P**, 2.10.6

2.10.6   Text Specification

The **Glossary** provides commands that 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.

105a   ⟨*u* 105a⟩≡   (100)

```
: U ( -- ) \ Usage: U text
    #L @  CHAIN P  1 #L +! ;
```

Following the use of **NEW** 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 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)

105b   ⟨*t* 105b⟩≡   (100)

```
: T ( n -- )    1+ DUP -LOCATE ABORT" Not there"  #L !  CR
    2 SPACES  PHRASE B? ;
```

After a line of text has been displayed, you can modify it by using the following command:

P replacement-text-line

105c   ⟨*p* 105c⟩≡   (100)

```
: P ( -- ) \ Usage: P text
    1 TEXT  PHRASE B! ;
```

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

105d   ⟨*x* 105d⟩≡   (100)

```
: X ( -- )    #L @ 1- UNCHAIN ;
```

You cannot delete the last remaining descriptive line, but you can change it. All others may be deleted.

REFERENCES

NEW, 2.10.5
FIND, 2.10.8

2.10.7   Definition Display

To display the current entry, type:

F

106a   ⟨*f* 106a⟩≡   (100)

```
: (SHOW) ( -- )    FIRST  WORD-NAME ?B  VOC ?B  BEFORE ?B  AFTER ?B
    SOURCE ?N  ENTERED NU@  .M/D/Y  BEGIN
        +L  -NEXT 0= WHILE
            10 SPACES  PHRASE B?
    REPEAT ;


[R FORTH GLOSSARY\ NAME        \VOCABULARY  \INPUT            \OUTPUT        \BLK\      EN
    CONSTANT GLOSSARY-TITLE


VARIABLE #L


: F ( -- )    GLOSSARY-TITLE HEADING  FIRST (SHOW)  1 #L ! ;
```

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.

106b   ⟨*summary* 106b⟩≡   (100)

```
: SUMMARY ( -- )    GLOSSARY-TITLE LAYOUT
    (GLOSSARY) RECORDS DO
        I GLOSS IF
            +L (SHOW)
    THEN  LOOP  SPACE ;
```

The command:

/VOCABULARY

will print definitions as for **SUMMARY** but only in the glossary vocabulary whose name is specified.

106c   ⟨*vocabulary* 106c⟩≡   (100)

```
: /VOCABULARY ( -- ) \ Usage: \VOCABULARY <name>
    VOCAB  GLOSSARY-TITLE LAYOUT  (GLOSSARY) RECORDS DO
        I GLOSS IF
            VOC  SWAP OVER ADDRESS -TEXT 0= IF
                +L (SHOW)
    THEN  THEN  LOOP  SPACE ;
```

REFERENCES

Glossary Vocabularies, 2.10.4
Making an Entry Current, 2.10.8

2.10.8   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 **NEW** or **FIND**, the current line number is initialized to zero.

107a ⟨*find* 107a⟩≡ (100)
```
: FIND ( -- ) \ Usage: FIND <name>
   (FIND) BINARY HEAD !  GLOSSARY F ;
```

The current word's stack entries may be changed by typing:

STACKS

107b ⟨*stacks* 107b⟩≡ (100)
```
: STACKS ( -- )   FIRST
   CR  ." Before: "  BEFORE ASK
   CR ." After: "  AFTER ASK  F ;
```

The current word's source-block# may be changed by typing:

new-blk# AT

107c ⟨*at* 107c⟩≡ (100)
```
: AT ( n -- )   FIRST  SOURCE N! ;
```

All of the words defined on a source block# can be moved by typing:

old-blk# new-mlk# MOVED

107d ⟨*moved* 107d⟩≡ (100)
```
: GLOSS ( n -- t )    (GLOSSARY) READ  LINK L@ DUP 0> IF
      HEAD !  GLOSSARY FIRST TRUE
   ELSE  DROP 0  THEN ;

: MOVED ( s d -- )    SWAP  (GLOSSARY) RECORDS DO
      I GLOSS DROP  SOURCE N@ OVER = IF
         OVER SOURCE N!
   THEN  LOOP  2DROP ;
```

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, 2.10.6
Making a Vocabulary Current, 2.10.4

2.10.9 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

108a ⟨*delete* 108a⟩≡ (100)

```
: DELETE ( -- )    (FIND)  BINARY
    ORDERED GRAB  -ORDERED
    GLOSSARY  BEGIN
        READ  LINK 0 SNATCH  DUP 0<
    UNTIL  DROP ;
```

3   Testing

Part of the motivation for converting this tool into a Literate Programming document was the discovery that there was some problem using it under a 64-bit Forth. That led to the discovery of problems with the existing samples even in 32-bit Forths. I have loaded all of the sames in the **pFDatabase.fth** source file (see 2.1.5), so it is a simple matter to run their display routines. I am only using **gforth** right now, and it has a built-in **assert(...   )** function. So I will use that right now, until I need to worry about doing something that is more portable.

108b ⟨*test* 108b⟩≡ (14)

```
⟨fits 109a⟩
⟨text= 109b⟩
⟨-linked 109c⟩
⟨double-stacked 109d⟩

: test ( -- )
    /People assert( PEOPLE PHONE fits ) display-everyone
    /Personnel assert( PERSONNEL RECORDS - 4 = ) all
        2 READ assert( S" Greg" NICKNAME text= )
    /Customers FALSE ALT ! assert( SERIALS 2 -linked ) 1 show 2 show
        TRUE ALT ! assert( SERIALS 3 -linked ) 1 show 2 show
    /Accounts assert( ACCOUNTS BALANCE 1 double-stacked ) balances
    /Wines INVENTORY
    /Glossary SUMMARY
;
```

The 1st problem I discovered was that the **people-fields** used 2 more bytes than the **PEOPLE** database was defined to use. While this is a common proplem with database definitions, it is not easily solved in a generic manner. Therefore, a specific test will have to suffice:

109a   ⟨*fits* 109a⟩≡   (108b)
```
  : fits ( n a -- flag )   WORKING - + B/R @ > 0= ;
```

The problem with the **PERSONNEL** data was that it had been created with **FALSE REVERSE !** and **BIG-ENDIAN**, which isn't how it was described in this document. The latter is easy to test against a know file size. The former requires a way to test text fields:

109b   ⟨*text=* 109b⟩≡   (108b)
```
  : text= ( a1 n1 n2 a2 -- flag )   2DUP B@ (S.) COMPARE 0= ;
```

The **CUSTOMERS** example provided the 1st case of a 64-bit problem. This ended up being a problem that **L@** returned unsigned 32-bit values and we need an **LU@** to do that and sign extend the value in **L@**, just as we had to do in **N@** when porting this code from 16-bit to 32-bit. A simple test for a negative link is needed here:

109c   ⟨*-linked* 109c⟩≡   (108b)
```
  : -linked ( n -- flag )   READ LINK L@ 0< ;
```

The **ACCOUNTS** example has a similar problem with **D@**, but it is not so easily solved. When this code was converted from 16 to 32-bits, 16-bit compatibility was abandonded. The **LONG** fields were added and the **DOUBLE** fields were extended to cover 64-bits. However, this time the desire is to preserve the compatibility and not add another field type. Therefore, another solution must be developed, which can actually bring back the support for the incompatible, 16-bit systems. To test this, we need to make sure that stack effects don't affect the test, like this:

109d   ⟨*double-stacked* 109d⟩≡   (108b)
```
  : double-stacked ( a n -- flag )   READ >R DEPTH R> SWAP >R D@ 2DROP DEPTH R> - 0= ;
```

The WINES example works fine at this point, as does the GLOSSARY, so we are done for now. However, it's is worth noting, in summary, that I did not test everything, which is appropriate for legacy applications, such as this. Tests can be added as needed, when problems are found.