Welcome Guest | Log In | Register | Benefits

Search: Site Source Code

Subscribe
Newsletters
Digital Library
RSS

Home Articles News Blogs Source Code Dobb's on DVD Dobb's TV Webinars & Events

Cloud Mobile Parallel .NET JVM Languages C/C++ Tools Design Testing Web Dev Jolt Awards

EMBEDDED SYSTEMS

Tweet 0 Like 0 Share Permalink

Zen for Embedded Systems

By Martin Tracy, January 01, 1990

Post a Comment

Source Code Accompanies This Article. Download It Now.

- tracy.asc
- zen.scr

DDJ's Forth expert presents ZEN, a tiny Forth system written entirely in Forth. Programs written in ZEN are ideal for embedded applications and, says Martin, inherently ROM-able.

This article contains the following executables: ZEN.SCR

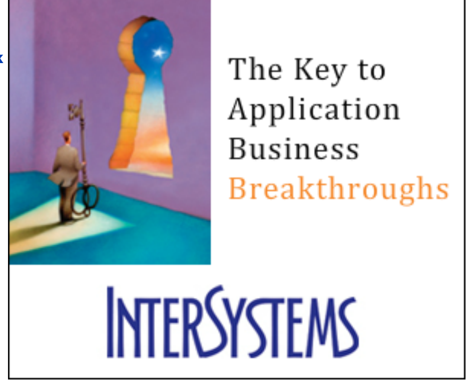
Martin is a DDJ contributing editor and a consultant. He can be reached at 2819 Pinkard Ave., Redondo Beach, CA 90278.

Zen is a tiny Forth with a big purpose: It is a model Forth designed for readability. Have you ever noticed that there are no intermediate Forth programmers? You are either a rank beginner or a black-belt guru. Why? My guess is that after reading Leo Brodie's Starting Forth (or my own Mastering Forth), you are told that the next step to learning Forth is to read the source code to the language itself. This sounds like a good idea; unfortunately, the source code of a professional Forth system is likely to be tightly packed, handcoded, over-optimized, and generally incomprehensible. If you can make it through all of that, you are in a good position to write a book of your own.

What you need is a Forth that is written mostly in ... you guessed it! Dr. Dobb's Journal has a long tradition of publishing personal implementations of computer languages, from Tiny Basic (January 1976 issue) to Small C (May 1980 issue). So to continue the tradition, we are proud to present ZEN, the tiny Forth.

Programs written in ZEN can be quite small, about half the size of programs written in C. In these days of megabyte memory, is small size important? Yes. Small size often means high speed. On-chip memory generally runs at least twice the speed of off-chip memory. A 68HC11 has only 8K of internal masked ROM. The most popular DSP chip, the TMS32010, has only 4K. Even if the density of on-chip memory doubles each year, three years from now microprocessors will support only 64K of internal ROM.

In recognition of the importance of small programs in



Embedded Systems Recent Articles

Understanding Core Data on iOS
Windows Phone:
Surprisingly Easy to Develop For
Using Bluetooth
Low-Cost, Low-Power Servers Begin Their Ascent
The Best of 2011

Most Popular

Stories Blogs

CUDA, Supercomputing for the Masses: Part 1
Writing Lock-Free Code: A Corrected Queue
Understanding Core Data on iOS
Using Bluetooth
A Simple and Efficient FFT Implementation in C++:
Part I

embedded applications, ZEN is inherently ROMable and easily fits in an 8K PROM. Not only that, but ZEN is extensible in ROM. How can a program grow in ROM? Assume that ZEN is resident in a single-board computer (SBC) and is talking to a host computer over an RS-232 serial line. Put a RAM chip in an empty ROM socket, and then send source code to ZEN with instructions to compile it to the address occupied by the RAM. ZEN will write the program there but will assume that all references to RAM point to a separate address space, normally the on-chip RAM.

Test the program interactively and, once it works, tell ZEN to read the RAM image and burn it into a PROM. Finally, remove the RAM chip, plug in the PROM, and you're done. The idea of running the development system in the target hardware may seem novel; what it means is that all testing and debugging take place in the target system. The host computer is used only as a terminal and disk server. The chances of a working program not working when you burn the final PROM are utterly remote.

ZEN Internals

Before you study the ZEN implementation (see Listing One, page 98), you must have a fairly good understanding of Forth. For example, you should know that the body of a CODE definition contains machine code. You may wish to treat the 80 or so CODE words in the ZEN kernel as black boxes, but be sure you know what they're supposed to do. Otherwise, be aware that Forth assemblers differ somewhat from classical assemblers. In most Forth assemblers, operands precede operators.

```
AL 0 [BX] MOV
```

instead of MOV [BX],AL

Furthermore, the ZEN assembler uses numeric local labels

```
         DX AX OR
          1 L# JZ
          AX PUSH
1 L:   BX PUSH
```

instead of using

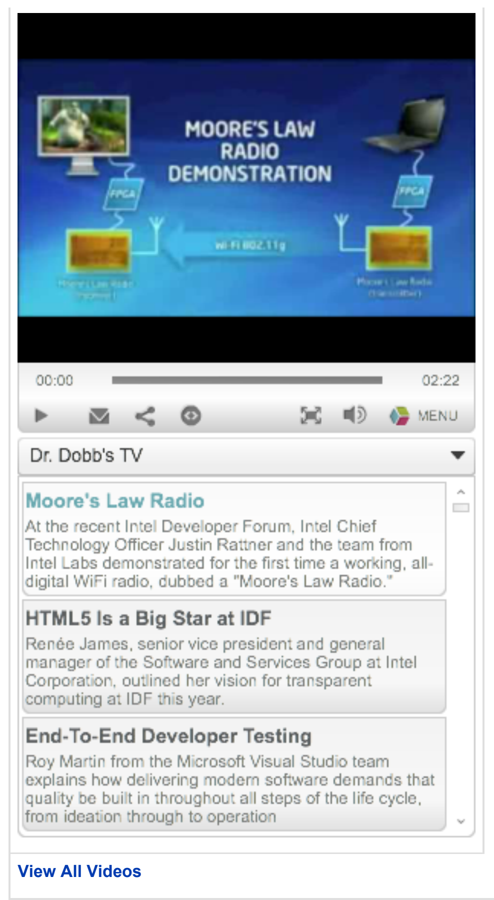
```
           OR AX,DX
           JZ away
           PUSH AX
away   PUSH BX
```

ZEN 1.7 for the 80x86-based IBM PC is a direct-threaded Forth. The SI register points to the next address to execute in a list of compiled addresses. The NEXT interpreter transfers control to the machine code at that address, simultaneously adjusting SI to point to the next address.

```
  NEXT  WORD LODS  AXJMP;
```

The overhead of interpreting a list of ZEN words is therefore two instructions per word. A list of addresses, however, takes far less memory than a series of jumps or calls. In a sense, threaded Forths trade speed for memory. In other words, threaded Forths compile tokens, but the token of a word is simply the address of that word. Microsoft's Compiled Basic and modern "cddr-coded" LISPs have recently copied this method.

Assume that SI points into a Forth definition at a point where it executes DUP. That means that SI points to the compiled address of machine code that duplicates the item on top of the stack.



View All Videos

This month's Dr. Dobb's Journal

This month, in our October 2012 digital issue of Dr. Dobb's Journal, we offer a roundup of JVM languages, an argument for the adoption of test-driven development, a detailed introduction to the Fantom programming language, **and much more!**

Download the latest issue today. >>

Featured Reports

```
   CODE DUP  BX PUSH  NEXT C;
```

The C; command is an abbreviation of END-CODE. This implementation of ZEN keeps the top stack item in the BX register. DUP executes in only three instructions -- one for PUSH and two more for NEXT. This optimization applies to many other machine-coded primitives, such as DROP and @.

```
 CODE DROP        BX POP           NEXT C;
 CODE @              0 [BX] BX MOV     NEXT C;
```

ZEN is a small model (64K) Forth. All four segment registers contain the same segment address. The origins and sizes of the separate RAM and ROM address spaces are specified on block 1.

ZEN Word Set

The ANSI Forth standardization effort is well under way. The current document, ANS X3J14 BASIS 9, is available for $6 from the X3J14 Secretariat (c/o FORTH, Inc., 111 N. Sepulveda Blvd., Manhattan Beach, CA 90266). The BASIS is modified at each meeting until it becomes the draft proposal that, after a period of public review and revision, becomes the ANSI Forth Standard. ZEN 1.7 is an unofficial implementation of BASIS 7 and provides all required words. It supports the double number, file access, and BLOCK extensions.

The stack manipulation words found in ZEN are:

```
   DUP DROP SWAP OVER
   ROT ?DUP
   2DUP 2DROP 2SWAP 2OVER 2ROT
   >R R@ R> 2>R 2R>
   NIP TUCK PICK ROLL
```

ROLL is defined in high-level because it cannot be efficiently mapped into machine code. You will find that 2>R and 2R> obviate almost any need for ROLL. In case you are not familiar with NIP and TUCK, they are equivalent to the phrases SWAP DROP and SWAP OVER, respectively. NIP is especially handy and is a single instruction on the Harris RTX 2000 Forth Processor. It only takes a single instruction (plus NEXT) on the 80x86 as well.

The single-precision arithmetic operators form this set:

```
    + - * / MOD /MOD */
    1+ 1/ 2* 2/
    NEGATE ABS  MAX MIN
```

1+ and 1- increment and decrement, respectively. 2* and 2/ shift once right and left, arithmetically. Notice that 2+ is not available. Use CELL+ instead to move to the next cell in an array. Use CELLS instead of 2* to change cells into bytes, as in:

```
   1024 CELLS ALLOT.
```

Many single-precision operators have double-number equivalents.

```
    D+ D- D2* D2/
    DNEGATE DABS DMAX DMIN
```

Multiplication and division generally mix precisions to avoid calculations with unnecessary accuracy.

```
   UM* UM/MOD  */MOD M* M/MOD
   M+ S>D D>S
```

What's this?

Strategy: Identity and Access Management: An Introduction
Best Practices: Top Mobile Productivity Tools for Finance
Strategy: SaaS, Social and Your ESB
Strategy: The Secret World of Compliance Auditors
Strategy: How Attackers Find and Exploit Database Vulnerabilities

More >>

Featured Whitepapers

What's this?

5 Things You Need to Know About BYOD
Key Attributes Backup Software Needs to Simplify VM Data Protection
VDI-Centric Endpoint Security Can Help Lower Costs and Increase ROI
A Practical Guide to Database Security
IDC Analyst Connection: Helping the Enterprise Address Cloud Strategies Through Business and IT Necessity

More >>

Featured Webcasts

What's this?

Exposing the Money Behind Malware
Strategies for Achieving Application Availability in the Cloud
Smarter Marketing: A new shared agenda for the CMO and CIO
Terms of Transformation: Rethinking Core Policy System Replacement
Dramatically Improve the Way Work Gets Done through Collaboration

More >>

Most Recent Premium Content

Digital Issues

2011 *Dr. Dobb's Journal* Digital Archive

Dr. Dobb's Journal January Digital Issue
Dr. Dobb's Journal February Digital Issue
Dr. Dobb's Journal March Digital Issue
Dr. Dobb's Journal April Digital Issue
Dr. Dobb's Journal May Digital Issue
Dr. Dobb's Journal June Digital Issue
Dr. Dobb's Journal July Digital Issue
Dr. Dobb's Journal August Digital Issue

A full complement of logical and comparison operators exists.

```
   AND OR XOR NOT
   0< 0= 0> < = > U< WITHIN
   D0= D< D= TRUE 0 1
```

WITHIN is especially powerful because it compares within in a circular number space. It can be used to control the duration of an event by comparing the current value of a self-incrementing counter to some future value. All other comparison operators can be based on WITHIN.

These are the memory access primitives:

```
  @ ! C@ C! 2@ 2! D@ D!
  CMOVE CMOVE> MOVE +!
```

D@ and D! are equivalent to 2@ and 2! except that the order of cell storage is not specified. MOVE is a smart CMOVE that moves without overlapping bytes. It cannot be used to fill memory with a character.

Here are the flow-of-control operators:

```
BEGIN WHILE REPEAT UNTIL AGAIN
IF ELSE THEN DO LOOP +LOOP
IJ LEAVE UNDO EXIT
EXECUTE @EXECUTE
```

UNDO removes one level of DO ... LOOP support from the return stack, allowing EXIT to execute within the loop. UNDO EXIT is the preferred method for leaving a word from inside a loop. @EXECUTE is exactly equivalent to @ EXECUTE, but is much faster.

These are the number conversion and formatting commands:

```
  BASE DECIMAL HEX
  CONVERT VAL?  PAD
  <# # #S SIGN HOLD #>
  . U. D. D. R
```

Output conversion and formatting takes place at PAD 1 - and downwards, leaving PAD itself free for applications. VAL? converts strings to numbers, and BASIS 6 defines numeric, string, and character literals.

```
[ASCII] ASCII "ccc" LITERAL
```

ZEN adds DLITERAL, the double-number equivalent of LITERAL. ASCII and [ASCII] make the following character into an ASCII literal inside and outside a definition, respectively. The string literal operator "(quote) works only inside a definition and accepts the following string of characters, up to the next", as a string literal.

```
   COUNT SKIP SCAN /STRING
   FILL BLANK ERASE -TRAILING
   STRING." EVALUATE
```

SKIP SCAN and /STRING are natural factors of WORD. SKIP and SCAN closely map the 80x86 string operator SCAS. /STRING is a generic substring operator; it can be used to select any part of a string. STRING is the fundamental string compiler and is used by the "and." commands.

EVALUATE is the most powerful string operator: It interprets any string. EVALUATE is used for forward referencing, macro definition, and any other instance of late binding. For example, coldstart uses the phrase READY

EVALUATE for device initialization. Suppose you add a device that requires its own initialization:

```
:READY ( new initialization) READY;
```

READY is redefined to include the new initialization. READY EVALUATE executes this new version, which in turn executes the previous version.

The interpreter level uses these words:

```
    BLK >IN      TIB #TIB      SPAN      STATE
    FORTH      CONTEXT      '     FIND      WORD
    CR TYPE       EXPECT KEYS KEY?
    KEY EMIT PAGE MARK TAB
    BL    SPACE SPACES     .((     DEPTH
```

Most Forths have words such as KEY?, which is true if a key is available; PAGE, which clears a screen or page; MARK, which TYPEs in emphasized mode; and TAB, which repositions the cursor to a given X Y coordinate. None of these words have been standardized. In addition, ZEN supports KEYS, which accepts characters without echoing or editing. In general, if the input stream is coming from a human being, use EXPECT; otherwise, use KEYS.

The compiler layer adds these words:

```
    CURRENT DEFINITIONS [] ['], C,
    COMPILE [COMPILE] IMMEDIATE
    RECURSE
```

RECURSE allows a definition to reference itself. Remember, though, that the size of the return stack is likely to be limited. The following defining words are provided:

```
CREATE VARIABLE CONSTANT :;
2VARIABLE 2CONSTANT
VOCABULARY
```

Two other defining words are used to build ZEN: USER and XFER. USER creates a user variable, whose address is based on its offset from the address in the pseudoregister u. A user variable is private to each task in a multitasking environment. XFER creates a transfer command, whose action is specified by its offset in an execution vector pointed to by the user variable x. By changing this execution vector, input and output can be redirected to different devices or files. Because x is a user variable, each device can be controlled by a different task, so a background task can spool text to the printer while a foreground task paints graphics on a screen.

RAM vs. ROM

The RAM and ROM address spaces in ZEN are kept entirely separate. The dictionary, machine code, and tables are built in ROM, and the data fields of variables and arrays are built in RAM. Each word that refers explicitly to RAM has an equivalent that refers instead to ROM.

```
RAM                         ROM
----------------------    -----------------------
CREATE                      VARIABLE
C, ,ALLOT
HERETHERE
DOES>                       GOES>
>BODY                       >DATA
```

Let's say you want to make a table of powers of ten:

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

The TENS table will be built in ROM. Suppose you want an array of seven cells:

```
  VARIABLE DAYS 6 CELLS ALLOT
```

The DAYS array will be built in RAM. Notice that VARIABLE allocates the first cell. It is not possible to initialize DAYS because anything stored in RAM disappears when the power is turned off. You can use GAP to allocate uninitialized cells in ROM, but you will rarely need to do so.

DOES> refers to the body of a defined word in ROM.

```
:COLOR \ set CREG to the value n.
    CREATE ( n), DOES> @ CREG!;

8 COLOR RED 12 COLOR GREEN
```

Executing RED will store 8 into CREG. DOES>DATA to the data field of a defined word in RAM.

```
:KOUNTER \ self-incrementing object.
VARIABLE DOES> DATA 1 SWAP +!;

KOUNTER ALPHA KOUNTER BETA
```

Each execution of ALPHA will increment its value. To read this value, you must be able to recover the data field address.

```
  ' ALPHA >DATA ? (prints value)
```

Use >BODY to recover the parameter field address of an object in

```
  ROM: ' RED >BODY ? (prints 8)
```

Restart and Error Handling

The restart and error-handling mechanism, designed by Wil Baden, was published in FORML 1987. The Forth command-line interpreter is QUIT; it is the default Forth application.

```
  :QUIT RESET
     BEGIN CR QUERY interpret OK?
  AGAIN;
```

When Forth begins executing, it goes to ABORT and from there to QUIT. The function of ABORT will be performed by every error recovery sequence. ABORT is the default Forth error restart sequence.

```
  :ABORT
  BEGIN PRESET QUIT GRIPE AGAIN;
```

In QUIT, the work of clearing the data stack is factored into RESET. The data stack, however, is not completely cleared; one item is left, but this has no impact on existing programs. In ABORT, PRESET clears both stacks all the way to the bottom.

You now have a way to customize the error recovery sequence. In addition, ABORT shows the pattern for dedicating an application: The default application is QUIT, and the default error recovery sequence is GRIPE. GRIPE sounds like TYPE and takes the same parameters, address, and length. It displays the error message provided by ABORT. It may also attempt to display the name of the word being interpreted.

PRESET leaves both stacks empty. Invoking the next word will put the address of the return point on the return stack.

RESET leaves that address alone. A word can get back to that location by RESET EXIT. On the return, the parameters for an error message string are assumed.

The default application and default error recovery sequence can be changed, and ABORT" will still work with them.

```
   :err RESET;

   :ABORT" [COMPILE] IF [COMPILE]"
   COMPILE err [COMPILE] THEN;
   IMMEDIATE
```

err in the definition of ABORT" cannot be replaced with RESET, but you can build a variable error message string and follow it with RESET EXIT or RESET; to emulate ABORT".

The following trivial example changes the default application to indent three spaces for every value on the stack before receiving new input. This process is preventative debugging: If you are surprised by the indentation then you are in trouble. Also, the example demonstrates logical structure as you compile a definition from the keyboard.

The error recovery message is changed as well.

```
:OLD-ABORT ABORT;

:INDENT DEPTH STATE @ -3*SPACES;

:SHELL
     BEGIN CR INDENT
           QUERY interpret OK?
     AGAIN;

:ABORT
    BEGIN PRESET SHELL
          CR." Abort:" GRIPE
    AGAIN;
```

ABORT installs a new application and error handler. OLD-ABORT reinstalls old application and error handler.

With PRESET and RESET you can establish intricate error recovery and restart networks. The following generic QUIT can be used to restart the default application in most implementations. The idea is to return to the position just before you invoke your default application.

```
  :QUIT RESET R> CELL - >R;
```

Mass Storage

ZEN supports both text files and traditional Forth BLOCKs. The text file support includes these words:

```
CREATE-FILE   OPEN-FILE              CLOSE-FILE
DELETE-FILE   RENAME-FILE
READ-FILE                            WRITE-FILE
SEEK-FILE                            FILESIZE                 FILEPOS
READ-LINE                            WRITE-CR
IO-RESULT
```

READ-FILE and WRITE-FILE read a sequence of bytes from the current position in the file to and from a buffer in memory. SEEK-FILE and FILEPOS change the position of a file, and FILE-SIZE returns its size. READ-LINE reads a line of text, and WRITE-CR writes an end-of-line terminator. To write a line of text, use WRITE-FILE followed by WRITE-CR. All file primitives work with a file identifier returned by CREATE-FILE or OPEN-FILE, so multiple files are supported. Error codes, if any, are returned in IO-RESULT.

The BLOCK support includes these words:

```
   BLOCK BUFFER UPDATE
   SAVE-BUFFERS FLUSH LOAD
```

Running ZEN

You can find the source code and executable image of ZEN in the file ZEN190.ARC on CompuServe in the DDJ Forum. ZEN is also available on the GENIE, ECFB, and BIX bulletin boards. Execute the file ZEN.COM and press . You should see the OK message. ZEN has a small set of debugging words you can use to look around:

```
   WORDS .S ? DUMP
```

You can compile and test definitions from the keyboard, or type GO to compile the text file KERNEL.SRC. Use any text editor to edit this file. I use the Sidekick pop-up editor.

Use GUARD to protect all words in the dictionary from FORGET. EMPTY restores the dictionary to the previous GUARDed word set.

I hope you enjoy ZEN and find it useful. Please send all comments and criticisms to me.

```
SCREEN 0
\ ZEN version 1.60-- a simple classical Forth
  ZEN 1.60 is a model implementation of the unofficial
  ANS Forth with Double-Number, File Access, and BLOCK
  Standard Extensions (BASIS6). This model is not endorsed
  by the ANS X3J14 committee. { Comments to go back to the ANS committee look
  like this.} ZEN 1.60 generates an IBM PC 64K small-model ROM-able nucleus.
  BX register is top-of-stack.  DTC with JMP code field.
  Assumes segment registers  CS = DS = ES Thanks to Wil Baden for his
  suggestions. This is a working document.  No guarantees are made to its
  accuracy or fitness.  While this is a working document, it is
  copyrighted 1989 by Martin J. Tracy.  All rights are reserved.

SCREEN 1
\ ZEN nucleus
FORTH DEFINITIONS  8 K-OF-ROM !  " KERNEL.COM" MAKE-OBJECT

32  CONSTANT #Jot     ( number conversion area in bytes)
128 CONSTANT #Safe    ( CREATE safety area--   in bytes)
128 CONSTANT #User    ( total user area size-- in bytes)

HEX
0100 BFFF 2DUP 2CONSTANT #ROM  ROMORG 2!  ( start & end of ROM)
C000 FFFF 2DUP 2CONSTANT #RAM  RAMORG 2!  ( start & end of RAM)
0000 #User - CONSTANT #RP0  ( top of return stack)
#RP0  0080 - CONSTANT #SP0  ( top of data   stack)

START   DECIMAL 2 LOAD   FINIS

SCREEN 2
\ Main LOAD screen
HERE EQU Power  2 CELLS ( power-up) GAP   ," C 1989 by M Tracy"
HERE EQU D0    #ROM , ,   #RAM , ,
HERE EQU H0 ( h) 0 ,     HERE EQU F0  0 , 0 ,  ( forth vlink)
HERE EQU T0 ( r) 0 ,     HERE EQU S0   #SP0 ,

           7 17 THRU  ( Kernel primitives)
          19 27 THRU  ( Numbers and I/O)
          29 41 THRU  ( Interpreter)
          43 69 THRU  ( Compiler)
          71 75 THRU  ( Device dependencies)
          81 86 THRU  ( Mass storage extension)
          77 79 THRU  ( Initialization)
                      ( Application, if any)

HERE H0 !   THERE T0 !
```

```
SCREEN 3
\ Documentation requirements
ZEN 1.60 supports Double-Number, File Access, and BLOCK Standard Extensions.
To compile the BLOCK extension, load the two screens following the File Access
extension. There are two 8-bit bytes per cell. Counted strings may be as long
as 255 bytes. Division is rounded-down. To change to floored division, load the
two screens following the mixed-precision rounded-down operators. The system
dictionary is approximately 7K address units (au's) leaving 56K for the
application. {#RAM and #ROM are currently set for 40K of application dictionary
and 16K of RAM.} {The data stack grows downwards towards the bottom of RAM.}
{The return stack is currently set for 128 au's of RAM.} Only dumb (glass)
terminals are supported. { How are minimum facilities to be specified?}

SCREEN 4
\ Errors and exceptions
If the input stream is inadvertantly exhausted: ABORT" ?"
If a word is not found: ABORT" ?"
If control structures are incorrectly nested: ABORT" Unbalanced"
If insufficient space in the dictionary: ABORT" No Room"
If insufficient number of stack entries: ABORT" Stack?"
If FORGETing within the nucleus: ABORT" Can't"

Division by zero returns a quotient of zero and a remainder
equal to the dividend.
Data and return stack overflows are not detected: the system
may crash or hang, if you are lucky.
Execution of compiler words while interpreting is not prevented;
the result of such execution is undefined.
Invalid and out-of-range arguments are not checked: the result
of using such arguments is very undefined.

SCREEN 5
\ Key to auxiliary commands
Several words used by the metacompiler are described here.

|               make the next word headerless.
," ccc"         compile the characters "ccc."

a ORG           reset HERE to address a.
n EQU <name>    equivalent to a headerless constant with value n.
LABEL <name>    equivalent to  HERE EQU <name> but also activates
                the CODE assembler.
CODE <name>     begins a machine-code definition, usually ended
                by  END-CODE  or  C;

I> and >I       like R> and >R when used to get return addresses.

BASE is returned to DECIMAL after each block is LOADed.

SCREEN 6
 --------------------------------------------------------------
|                                                              |
|  Please direct all comments and inquiries to Martin Tracy    |
|                                                              |
|                                                              |
 --------------------------------------------------------------

SCREEN 7
\ ------ Kernel primitives ------------------------
LABEL colon   BP DEC  BP DEC   SI 0 [BP] MOV   SI POP   NEXT
\ save I register on return stack and set it to new position.
\ This is the action of the code field in all colon definitions.

CODE EXIT   NOP
| CODE semi   0 [BP] SI MOV   BP INC  BP INC
| CODE nope   NEXT C;
\ semi is the action of the semicolon in all colon definitions.
\ EXIT differs from semi as an aid to decompilation.
\ nope is a "no operation" word used for initialization.


SCREEN 8
\ Data objects
LABEL addr  \ the action of all CREATEs.
```

```
   BX PUSH   3 # AX ADD  BX AX XCHG   NEXT

LABEL con   \ the action of all CONSTANTs and VARIABLEs.
   BX PUSH   3 # AX ADD  BX AX XCHG   0 [BX] BX MOV   NEXT C;

VARIABLE u  { Private}  \ USER area pointer.
LABEL uvar  \ the action of all USER variables.
   BX PUSH   3 # AX ADD   BX AX XCHG
   0 [BX] BX MOV   u ) BX ADD   NEXT

LABEL (does)   BP DEC  BP DEC  SI 0 [BP] MOV  \ run-time DOES>
   SI POP  BX PUSH   3 # AX ADD   BX AX XCHG    NEXT C;

SCREEN 9
\ Stack manipulation
CODE DUP  ( w - w w)   BX PUSH   NEXT C;
CODE DROP ( w)         BX POP    NEXT C;

CODE SWAP ( w w2 - w2 w)
   SP DI MOV             BX 0 [DI] XCHG   NEXT C;
CODE OVER ( w w2 - w w2 w)
   SP DI MOV   BX PUSH   0 [DI] BX MOV    NEXT C;

CODE ROT ( w w2 w3 - w2 w3 w)
   DX POP  AX POP   DX PUSH  BX PUSH   AX BX MOV   NEXT C;

CODE PICK ( w[u]...w[1] w[0] u - w[u]...w[1] w[0] w[u])
\ copy kth item to top of stack.
   BX SHL   SP BX ADD   0 [BX] BX MOV   NEXT C;

SCREEN 10
\ Memory access
CODE @ ( a - w)   0 [BX] BX MOV        NEXT C;
CODE ! ( w a)     0 [BX] POP  BX POP   NEXT C;

CODE C@ ( a - b)  0 [BX] BL MOV   BH BH SUB        NEXT C;
CODE C! ( b a)    AX POP  AL 0 [BX] MOV   BX POP   NEXT C;

CODE CMOVE ( a a2 u)
\ move count bytes from from to to, leftmost byte first.
   BX CX MOV   SI BX MOV   DI POP  SI POP
   REP BYTE MOVS   BX SI MOV   BX POP   NEXT C;

SCREEN 11
\ Math operators
| CODE tic   NOP
| CODE lit   WORD LODS   BX PUSH  AX BX MOV    NEXT C;
\ push the following (in-line) number onto the stack.

CODE + ( n n2 - n3)   AX POP  AX BX ADD            NEXT C;
CODE - ( n n2 - n3)   AX POP  AX BX SUB   BX NEG   NEXT C;

CODE NEGATE ( n -  n2)   BX NEG   NEXT C;
CODE ABS    ( n - +n2)
   BX BX OR   1 L# JNS   BX NEG   1 L: NEXT C;

CODE +! ( n a)   AX POP   AX 0 [BX] ADD  BX POP   NEXT C;
\ increment number at address by n.

SCREEN 12
\ Math and logical
CODE 1+ ( n - n2)   BX INC   NEXT C;
CODE 1- ( n - n2)   BX DEC   NEXT C;

CODE 2* ( n - n2)   BX SHL   NEXT C;  { CONTROLLED} { Require?}
CODE 2/ ( n - n2)   BX SAR   NEXT C;  ( arithmetic)

CODE AND ( m m2 - m3)   AX POP  AX BX AND    NEXT C;
CODE OR  ( m m2 - m3)   AX POP  AX BX OR     NEXT C;
CODE XOR ( m m2 - m3)   AX POP  AX BX XOR    NEXT C;

CODE NOT ( w - w2)   BX NOT   NEXT C;  { ( m - m2) ?}

SCREEN 13
```

```
\ Comparisons
CODE 0 ( - n)       BX PUSH   BX BX SUB    NEXT C;   { Feature}
CODE 1 ( - n)       BX PUSH   1 # BX MOV   NEXT C;   { Feature}
CODE TRUE ( - m)    BX PUSH  -1 # BX MOV   NEXT C;   { Control?}

CODE = ( n n2 - f)    AX POP   AX BX CMP
   TRUE # BX MOV   1 L# JZ    BX INC    1 L: NEXT C;

CODE <   ( n n2 - f)    AX POP   BX AX SUB
   TRUE # BX MOV   1 L# JL    BX INC    1 L: NEXT C;
CODE U< ( u u2 - f)    AX POP   BX AX SUB
   TRUE # BX MOV   1 L# JB    BX INC    1 L: NEXT C;

: > ( n n2 - f)    SWAP < ;

SCREEN 14
\ Comparisons against zero and CELL operators
CODE 0= ( n - f)
   BX BX OR   TRUE # BX MOV   1 L# JZ    BX INC    1 L: NEXT C;
CODE 0< ( n - f)
   BX BX OR   TRUE # BX MOV   1 L# JS    BX INC    1 L: NEXT C;

: 0> ( n - f)    0 > ;


2 CONSTANT CELL   { Feature}

CODE CELL+ ( a - a2)    BX INC   BX INC    NEXT C;
CODE CELLS ( a - a2)    BX SHL             NEXT C;

SCREEN 15
\ Branches and loops
| CODE  branch         \ unconditional branch.
   0 [SI] SI MOV    NEXT C;
| CODE ?branch ( f)     \ branch if zero.
   BX BX OR   BX POP     ' branch JZ    2 # SI ADD    NEXT C;

| CODE (do) ( n n2)     \ begin DO...LOOP structure.
   4 # BP SUB       AX POP    HEX 8000 DECIMAL # AX ADD
   AX 2 [BP] MOV    AX BX SUB    BX 0 [BP] MOV    BX POP    NEXT C;

| CODE (loop)          \ terminate DO...LOOP structure.
   WORD 0 [BP] INC   ' branch JNO
LABEL  >loop    2 # SI ADD
| CODE >undo    4 # BP ADD    NEXT
| CODE (+loop) ( n)    \ terminate DO...+LOOP structure.
   BX 0 [BP] ADD   BX POP   ' branch JNO   >loop JO    NEXT C;

SCREEN 16
\ Return stack
CODE >R ( w)     BP DEC   BP DEC   BX 0 [BP] MOV    BX POP   NEXT C;

CODE R@ ( - w)  BX PUSH   0 [BP] BX MOV    NEXT C;
CODE I  ( - n)  BX PUSH   0 [BP] BX MOV   2 [BP] BX ADD    NEXT C;
CODE J  ( - n)  BX PUSH   4 [BP] BX MOV   6 [BP] BX ADD    NEXT C;
CODE R> ( - w)  BX PUSH   0 [BP] BX MOV   BP INC   BP INC   NEXT C;

CODE 2>R ( w w2)
\ push w and w2 to the return stack, w2 on top.
   4 # BP SUB    BX 0 [BP] MOV    2 [BP] POP    BX POP      NEXT C;

CODE 2R> ( - w w2)
\ pop w and w2 from the return stack.
   BX PUSH    2 [BP] PUSH    0 [BP] BX MOV    4 # BP ADD    NEXT C;

SCREEN 17
\ Optimizations and EXECUTE
CODE NIP   ( w w2 - w2)         { CONTROLLED}   AX POP    NEXT C;
CODE TUCK ( w w2 - w2 w w2)   { CONTROLLED}   AX POP
   BX PUSH   AX PUSH    NEXT C;

CODE ?DUP ( w - w w | 0 - 0)
   BX BX OR    1 L# JZ    BX PUSH   1 L: NEXT C;
```

```
CODE EXECUTE ( w)   BX AX XCHG   BX POP   AX JMP C;

CODE @EXECUTE ( w)  { Control?}  { Why w and not a?}
\ @EXECUTE is equivalent to @ EXECUTE but is much faster.
   BX DI MOV   BX POP   0 [DI] AX MOV   AX JMP C;

SCREEN 18



SCREEN 19
\ ------ Input/Output -----------------------------
\ In ZEN, consecutive headerless variables form a category
\ which can be extended but not reduced or reordered.

0 USER entry  2 CELLS + ( skip multitasking hooks)
  USER r       | USER SP0
  USER x       \ XFER vector pointer.
  USER BASE    | USER dpl     | USER hld   EQU #I/0

: THERE ( - a)   r @ ;  { ROM}
: PAD   ( - a)   r @ [ #Jot ] LITERAL + ;  { CONTROLLED}
{ pictured number staging area size undefined?}

: DECIMAL   10 BASE ! ;
: HEX       16 BASE ! ;  { CONTROLLED}

SCREEN 20
\ Double-value data stack operators
CODE 2DUP ( w w2 - w w2 w w2)   SP DI MOV   BX PUSH
   0 [DI] PUSH   NEXT C;

CODE 2DROP ( w w2)   BX POP  BX POP   NEXT C;

CODE 2SWAP ( w w2 w3 w4 - w3 w4 w w2)   AX POP  CX POP  DX POP
   AX PUSH  BX PUSH  DX PUSH  CX BX MOV   NEXT C;

: 2OVER ( d d2 - d d2 d)       2>R 2DUP  2R> 2SWAP ;
: 2ROT  ( d d2 d3 - d2 d3 d)   2R> 2SWAP 2R> 2SWAP ;
{ CONTROLLED}  { Require?}

CODE 2@ ( a - w w2)   2 [BX] PUSH  0 [BX] BX MOV        NEXT C;
CODE 2! ( w w2 a)      0 [BX] POP   2 [BX] POP   BX POP  NEXT C;

SCREEN 21
\ Numeric conversion math support
CODE D+ ( d d2 - d3)   AX POP   DX POP   CX POP
   AX CX ADD   CX PUSH    DX BX ADC   NEXT C;

CODE DNEGATE ( d - d2)   AX POP  AX NEG  AX PUSH
   0 # BX ADC   BX NEG   NEXT C;

: MAX ( n n2 - n3)   2DUP <    IF  SWAP  THEN  DROP ;
: MIN ( n n2 - n3)   2DUP < 0= IF  SWAP  THEN  DROP ;


SCREEN 22
\ Numeric conversion math support
CODE UM* ( u u2 - ud)
   AX POP   BX MUL    AX PUSH   DX BX MOV   NEXT C;

CODE UM/MOD ( ud u - u2 u3)
\ return rem u2 and quot u3 of unsigned ud divided by u.
\ On zero-divide, return quot=0 and rem=low-word-of-ud.
   DX POP  AX AX SUB  BX DX CMP  1 L# JAE
   AX POP  BX DIV  DX PUSH       1 L: AX BX MOV   NEXT C;


SCREEN 23
\ Input number conversion
ASCII A  ASCII 9  1+ - EQU A-10

| : digit ( c base - n t | ? 0)
\ true if the char c is a valid digit in the given base.
```

```
   SWAP [ASCII] 0 -  9 OVER <  DUP
   IF  DROP  A-10 -  10  THEN
   >R  DUP R@ -  ROT R> -  U< ;

: CONVERT ( +d a - +d2 a2)
\ convert the char sequence at a+1 and accumulate it in +d.
\ a2 is the address of the first non-convertable digit.
   BEGIN  1+ DUP >R  C@ BASE @ digit
   WHILE  SWAP  BASE @ UM* DROP  ROT  BASE @ UM*  D+  R>
   REPEAT DROP  R> ;

SCREEN 24
\ Output number conversion
: <#   PAD hld ! ;
: #> ( wd - a u)   2DROP hld @ PAD OVER - ;

: HOLD ( c)   TRUE hld +!  hld @ C! ;
\ add character c to output string.
: SIGN ( n)   0< IF  [ASCII] - HOLD  THEN ;
\ add "-" to output string if w is negative.

: # ( ud - ud2)
\ transfer the next digit of ud to the output string.
   BASE @ >R  0 R@ UM/MOD  R> SWAP >R  UM/MOD  R>
   ROT 9 OVER < IF  A-10 + THEN  [ASCII] 0 + HOLD ;

: #S ( ud - ud2)   BEGIN  #  2DUP OR  0= UNTIL ;
\ convert all remaining digits of ud.  ud2 is 0 0 .

SCREEN 25
\ Transfers
LABEL xvar  \ the action of all transfers.
   u ) DI MOV   x [DI] DI MOV   3 # AX ADD   DI AX XCHG
   0 [DI] DI MOV   AX DI ADD   0 [DI] AX MOV   AX JMP C;

0 XFER TYPE ( a u)              XFER CR
  XFER KEYS ( a u) { Private}   XFER KEY? ( - f) { Extend?}
  XFER MARK ( a u) { Extend?}   XFER PAGE        { Extend?}
  XFER TAB ( n n2) { Extend?} ( Reserved)  DROP

\ KEYS is a simple unfiltered EXPECT which doesn't echo.
\ KEY? is true if a key is available.
\ MARK is like TYPE but highlights if possible.
\ PAGE clears the screen.
\ TAB  moves the cursor to the x (n) and y (n2) coordinates.

SCREEN 26
\ Print spaces
32 CONSTANT BL  { CONTROLLED}   \ ASCII blank

     HERE ( *) BL ,
: SPACE    ( *) LITERAL 1 TYPE ;

HERE ( * )  BL C, BL C, BL C, BL C, BL C, BL C, BL C, BL C,
: SPACES ( +n )  \ output w spaces.   Optimized for TYPE.

   ( * ) LITERAL  OVER 2/ 2/ 2/  ?DUP
   IF  0 DO  DUP 8 TYPE  LOOP  THEN  SWAP 7 AND TYPE ;

SCREEN 27
\ Print numbers
| : (d.) ( d - a u)   \ convert a double number to a string.
   TUCK  DUP 0< IF  DNEGATE  THEN  <#  #S ROT SIGN  #> ;

: D. ( d)   (d.) TYPE SPACE ;
: U. ( u)         0 D. ;
:  . ( n)   DUP 0< D. ;

SCREEN 28

SCREEN 29
\ ------ Interpreter -----------------------------
#I/O ( continued from I/O layer)
  USER BLK  { BLOCK} { Require?}   USER >IN   \ keep together.
```

```
   USER #TIB   CELL+   \ #TIB and TIB's value.
   USER SPAN
   USER STATE  EQU #Used

   VARIABLE last   CELL ALLOT     \ last lfa and cfa.
 | VARIABLE scr    CELL ALLOT     \ last error location.
 | VARIABLE bal  | VARIABLE leaf  \ see compiler.

VARIABLE CONTEXT  { CONTROLLED}
VARIABLE CURRENT  { CONTROLLED}

: TIB ( - a)   #TIB CELL+ @ ;

SCREEN 30
\ Automatic variables
\ These variables are automatically initialized; see COLD.
VARIABLE h   | VARIABLE f   CELL ( ie vlink) ALLOT

   VARIABLE 'pause    \ multitasking hook.
 | VARIABLE 'expect   \ deferred EXPECT
 | VARIABLE 'source   \ deferred input stream.
 | VARIABLE 'warn     \ redefinition warning.
 | VARIABLE 'loc      \ source location field.
 | VARIABLE 'val?     \ string to number conversion.

 | VARIABLE key'  CELL ALLOT   \ one-key look-ahead buffer.

: HERE ( - a)   h @ ;

SCREEN 31
( String operators-- high-level definitions )  EXIT
: COUNT ( a - a2 u)   DUP C@ SWAP 1+ ;
\ transform counted string into text string.
: /STRING ( a u n - a2 u2)  { Control?}  ROT OVER + ROT ROT - ;
\ truncate leftmost n chars of string.  n may be negative.

: SKIP ( a u b - a2 u2)  { Control?}
\ return shorter string from first position unequal to byte.
    >R  BEGIN  DUP
       WHILE  OVER C@ R@ - IF  R> DROP  EXIT  THEN  1 /STRING
       REPEAT   R> DROP ;
: SCAN ( a u b - a2 u2)  { Control?}
\ return shorter string from first position equal to byte.
    >R  BEGIN  DUP
       WHILE  OVER C@ R@ =  IF  R> DROP  EXIT  THEN  1 /STRING
       REPEAT   R> DROP ;

SCREEN 32
( String operators-- low-level definitions )
CODE COUNT ( a - a2 u)   BX AX MOV   AX INC
\ transform counted string into text string.
   0 [BX] BL MOV   BH BH SUB   AX PUSH   NEXT C;
CODE /STRING ( a u n - a2 u2)  { Control?}  CX POP  AX POP
\ truncate leftmost n chars of string.  n may be negative.
   BX AX ADD  BX CX SUB   CX BX MOV   AX PUSH   NEXT C;

CODE SKIP ( a u b - a2 u2) { Contr?}  BX AX MOV  CX POP  DI POP
\ return shorter string from first position unequal to byte.
   1 L# JCXZ   REPE BYTE SCAS    1 L# JZ    CX INC  DI DEC
   1 L: DI PUSH  CX BX MOV   NEXT C;
CODE SCAN ( a 1 b - a2 u2) { Contr?}  BX AX MOV  CX POP  DI POP
\ return shorter string from first position equal to byte.
   1 L# JCXZ   REPNE BYTE SCAS   1 L# JNZ   CX INC  DI DEC
   1 L: DI PUSH  CX BX MOV   NEXT C;

SCREEN 33
\ More string operators
CODE FILL ( a u b)   \ store u b's, starting at addr a.
   BX AX MOV  CX POP  DI POP   REP BYTE STOS   BX POP   NEXT C;

: -TRAILING ( a +n - a2 +n2)   2DUP
\ alter string to suppress trailing blanks.
   BEGIN  2DUP         BL SKIP  DUP
   WHILE  2SWAP 2DROP  BL SCAN  REPEAT  2DROP  NIP - ;
```

```
EXIT
: FILL ( a u b)   \ store u b's, starting at addr a.
   SWAP ?DUP 0= IF  2DROP EXIT  THEN
   >R OVER C!  DUP 1+ R> 1- CMOVE ;

SCREEN 34
\ Input stream operators
| : source ( - a u)   #TIB 2@ ;   \ input stream source.
:  /source ( - a u)   'source @EXECUTE  >IN @ /STRING ;

| : accept ( n f)   IF  1+  THEN  >IN +! ;
\ accept characters by incrementing >IN.

: parse ( c - a u)   \ parse a character-delimited string.
   >R  /source  OVER SWAP  R> SCAN  >R  OVER -  DUP R> accept ;

: WORD ( c - a)      \ parse a character-delimited string;
\ leading delimiters are accepted and skipped;
\ the string is counted and followed by a blank (not counted).
   >R  /source  OVER R> 2>R  R@ SKIP  OVER SWAP  R> SCAN
   OVER R> -  SWAP accept  OVER -  31 MIN  THERE  DUP >R
   2DUP C!  1+ SWAP CMOVE  BL R@ COUNT + C!  R> ;

SCREEN 35
\ Dictionary search
CODE thread ( a w - a 0 , cfa -1 , cfa 1)
\ search vocabulary for a match with the packed name at  a .
   DX POP  SI PUSH
 1 L: 0 [BX] BX MOV                ( chain thru dictionary )
   BX BX OR  5 L# JZ               ( jump if end of thread )
   DX DI MOV  ( 'string)   BX SI MOV  2 # SI ADD   ( SI=nfa)
   0 [SI] CL MOV     31 # CX AND   0 [DI] CL CMP   ( count = ?)
   1 L# JNZ ( lengths <>) DI INC  SI INC  ( to body of 'string)
   REPE BYTE CMPS ( names =?)  1 L# JNZ   ( jump not matched)
   CX POP   SI PUSH  ( cfa )
   CX SI MOV   BYTE 32 # 2 [BX] TEST  ( immediate bit )
   TRUE # BX MOV  4 L# JZ   BX NEG     4 L: NEXT
 5 L: SI POP   DX PUSH ( 'str)  ( BX = 0)   NEXT C;

SCREEN 36
\ FIND  [ and ]
: FIND ( a - a 0 | a - w -1 | a - w 1)
\ search dictionary for a match with the packed name at  a .
\ Return execution address and -1 or 1 ( IMMEDIATE) if found;
\ ['] EXIT 1 if  a  has zero length;  a 0  if not found.
   DUP C@ ( a l) DUP
   IF  31 MIN OVER C! ( a) CONTEXT @ thread ( a -1/0/1) DUP
      IF  EXIT  THEN       CONTEXT @ f -
      IF  DROP  f thread  THEN   EXIT
   THEN ( a 0) 2DROP  ['] EXIT 1 ;

: ]  TRUE STATE ! ;   \ stop interpreting; start compiling.
: [      0 STATE ! ;   \ stop compiling; start interpreting.
  IMMEDIATE

SCREEN 37
\ Data and return stack
\ Set data and return stack pointers, respectively:
| CODE sp! ( a)   BX SP MOV  BX POP   NEXT C;
| CODE rp! ( a)   BX BP MOV  BX POP   NEXT C;

: RESET  { Feature}  \ reset return stack for error recovery.
   I>  entry CELL - rp!  >I ;
: PRESET { Feature}  \ empty both stacks and prepare system.
   SP0 @ sp!  I> entry rp! >I  SP0 @ 0 #TIB 2!  0 STATE ! ;

| : err   RESET ;

CODE DEPTH ( - n)   \ # items on stack before DEPTH is executed.
   BX PUSH   u ) BX MOV   SP0 [BX] BX MOV   SP BX SUB   BX SAR
   NEXT C;

SCREEN 38
```

```
( Memory management-- high-level definitions)  EXIT
: ALLOT ( n)   r +! ;   \ allocate n RAM data bytes.
: GAP   ( n)   h +! ;   \ allocate n dictionary bytes.  { ROM}

: C, ( w)   h @ C!  1 h +! ;     \ ie  HERE C!  1 GAP ;
\ append low byte of w onto the dictionary.
: ,  ( w)   h @ !  CELL h +! ;   \ ie  HERE !  CELL GAP ;
\ append w onto the dictionary.

EXIT  { In an all-RAM system:}
: GAP   ALLOT ;   : THERE   HERE ;     : >DATA   >BODY ;
: GOES>  [COMPILE] DOES> ;  IMMEDIATE

SCREEN 39
( Memory management-- low-level definitions)
CODE ALLOT ( n)   \ allocate n RAM data bytes.
   r # DI MOV   u ) DI ADD   BX 0 [DI] ADD  BX POP   NEXT C;
CODE GAP   ( n)   \ allocate n dictionary bytes.  { ROM}
   h # DI MOV                BX 0 [DI] ADD  BX POP   NEXT C;

CODE C, ( w)   h # DI MOV   0 [DI] DI MOV
\ append low byte of w onto the dictionary.
   BL 0 [DI] MOV   1 # BX MOV   ' GAP JU
CODE ,  ( w)   h # DI MOV   0 [DI] DI MOV
\ append w onto the dictionary.
   BX 0 [DI] MOV   2 # BX MOV   ' GAP JU  FORTH

SCREEN 40
\ Code and data fields
: >BODY ( w - a)   3 + ;
: >DATA ( w - a)   3 + @ ;  { ROM}

: >code ( cfa - 'code)  1+  DUP @ CELL+ + ;
\ finds code address associated with cfa.
| : alter ( 'code cfa)   1+  TUCK CELL+ -  SWAP ! ;
\ point the cf to the given code addr.  Skip the CALL byte.

| : nest, ( 'code )   HERE  232 ( CALL) C,  CELL GAP  alter ;
\ create the code field for colon words, DOES> and GOES>
| : code, ( 'code )   HERE  233 ( JMP ) C,  CELL GAP  alter ;
\ create the code field for data words.

: patch ( 'code cfa)   233 ( JMP ) OVER C!  alter ;
\ make 'code the new action of the cf.  Used by (;code).

SCREEN 41
\ Alignment, string and error primitives
\ : ALIGN   HERE 1 AND GAP ;            { ALIGN}
\ force dictionary to the next even address.
\ : REALIGN ( a - a2)   DUP 1 AND + ;  { ALIGN}
\ force address to the next even address.

| : (") ( - a l)   I> COUNT  2DUP +  ( REALIGN) >I ;
\ leave the address and length of an in-line string.

| : huh? ( w)   0= ABORT" ?" ;
\ error action of several words.

: ' ( - w)   BL WORD  DUP C@ huh?  FIND huh? ;

\ : I>   [COMPILE] R> ;  IMMEDIATE  { ALIGN}
\ : >I   [COMPILE] >R ;  IMMEDIATE  { ALIGN}

SCREEN 42


SCREEN 43
\ ------ Compiler ---------------------------------
: COMPILE   I>  DUP CELL+ >I  @ , ;
\ compile the word that follows in the definition.

: header   \ create link and name fields.
   ( ALIGN)              'loc  @EXECUTE  ( extra fields )
   BL WORD  DUP C@ huh?  'warn @EXECUTE  ( redefinition?)
```

```
   HERE last !  HERE CURRENT @ DUP @ , ! ( link field)
   HERE OVER C@ 1+ CMOVE                 ( name field)
   HERE C@ DUP 128 OR C, GAP   HERE last CELL+ ! ;

SCREEN 44
\ Defining words
: CREATE ( - a)
   header  [ addr ] LITERAL code, ;

: VARIABLE ( - a)
   header  [ con ] LITERAL code,  THERE ,
   0 THERE ! ( courtesy )  CELL ALLOT ;

: CONSTANT ( - w)
   header  [ con ] LITERAL code,  , ;

SCREEN 45
\ DOES> and GOES>
| : (;code)   I> last CELL+ @ patch ;
\ the code field of (;code) is at  ' DOES> >BODY CELL+


: DOES>   COMPILE (;code)  [ (does) ] LITERAL nest, ; IMMEDIATE
\ eg  : KONST   CREATE  ,  DOES> @ ;

: GOES>  { ROM}  [COMPILE] DOES>  COMPILE @ ;  IMMEDIATE
\ eg  : VALUE   VARIABLE   GOES> @ ;

SCREEN 46
\ Literals
: LITERAL ( - w)     COMPILE lit  , ;  IMMEDIATE
\ compile w as a literal.
: [']      ( - w)  '  COMPILE tic  , ;  IMMEDIATE
\ compile-form of ' ("tick").

:  ASCII  ( - c)   BL WORD 1+ C@ ;  \ return value of next char.
: [ASCII] ( - c)                   \ compile value of next char.
   ASCII  [COMPILE] LITERAL ;  IMMEDIATE

: STRING ( c)  { Feature}  \ string compiler, eg 32 STRING ABC
   parse  DUP C,  HERE OVER GAP  SWAP CMOVE  ( ALIGN) ;

: " ( - a u)     \ string literal, eg " cccc"
   COMPILE (")  [ASCII] " STRING ;  IMMEDIATE
: ."   [COMPILE] "  COMPILE TYPE ;  IMMEDIATE

SCREEN 47
\ Flow of control
| : ?bal   DUP bal @ < huh?  PICK @ 0= huh? ;
| : -bal   bal @ huh?  TRUE bal +!  DUP @ huh? ;

: BEGIN  HERE  1 bal +! ;                          IMMEDIATE

: IF     COMPILE ?branch  [COMPILE] BEGIN   0 , ;  IMMEDIATE
: THEN   0 ?bal  TRUE bal +!  HERE SWAP ! ;        IMMEDIATE
: ELSE   0 ?bal  COMPILE  branch  [COMPILE] BEGIN  0 ,
         SWAP   [COMPILE] THEN ;                   IMMEDIATE

: UNTIL  -bal  COMPILE ?branch  , ;                IMMEDIATE
: AGAIN  -bal  COMPILE  branch  , ;  { Control?}   IMMEDIATE
: WHILE   bal @ huh?  [COMPILE] IF  SWAP ;         IMMEDIATE
: REPEAT  1 ?bal  [COMPILE] AGAIN [COMPILE] THEN ; IMMEDIATE

SCREEN 48
\ Definite loops
: DO   COMPILE (do)  [COMPILE] BEGIN ;   IMMEDIATE

: LEAVE  COMPILE >undo  COMPILE branch
   HERE  leaf @ , leaf ! ;               IMMEDIATE

| : rake,   \ gathers leaf's.  Courtesy of Wil Baden.
   DUP ,  leaf @
   BEGIN  2DUP U< WHILE  DUP @ HERE ROT !  REPEAT
   leaf ! DROP ;
```

```
:  LOOP   -bal  COMPILE  (loop)  rake, ;  IMMEDIATE
: +LOOP   -bal  COMPILE (+loop)  rake, ;  IMMEDIATE

: UNDO   COMPILE >undo ;  IMMEDIATE

SCREEN 49
\ Colon definitions
: :   \ create a word and enter the compiling loop.
   CURRENT @ CONTEXT !
   header  [ colon ] LITERAL nest,
   last @ @ CONTEXT @ !  0 0 bal 2!  ] ;

: ;   \ terminate a definition.
   bal 2@ OR ABORT" Unbalanced"
   last @ CURRENT @ !
   COMPILE semi [COMPILE] [ ;  IMMEDIATE

SCREEN 50
\ Vocabularies
: FORTH   f CONTEXT ! ;

: DEFINITIONS   CONTEXT @ CURRENT ! ;
\ new definitions will be into the CURRENT vocabulary.

: VOCABULARY
\ when executed, a vocabulary becomes first in the search order.
   VARIABLE   HERE  f CELL+ ( ie vlink) DUP @ , !
   CELL GAP ( value for automatic initialization)
   GOES> CONTEXT ! ;

SCREEN 51
\ Misc. compiler support
: IMMEDIATE   last @ CELL+  DUP C@ BL ( ie 32) OR SWAP C! ;

: [COMPILE]     ' , ;  IMMEDIATE
\ force compilation of an otherwise immediate word.

:  (    [ASCII] ) parse  2DROP ;  IMMEDIATE   ( comments)
: .(    [ASCII] ) parse   TYPE ;  IMMEDIATE   \ messages.

: RECURSE   last CELL+ @ , ;  IMMEDIATE   \ self-reference.

SCREEN 52
( Hall of fame--  high-level)  EXIT
: M+ ( d n - d2)  { Control?}  S>D D+ ;   \ add n to d.

: >< ( u - u2)  { Control?}  DUP 255 AND  SWAP 256 * OR ;
\ reverse the bytes within a cell.

: WITHIN ( u n n2 - f)  { Control?}  OVER - >R - R> U< ;
\ true if n <= u < n2  given circular comparison.

: ERASE ( a u)     0 FILL ;  { CONTROLLED}
: BLANK ( a u)    BL FILL ;  { CONTROLLED}

SCREEN 53
( Hall of fame--  low-level)
CODE M+ ( d n - d2)           { Control?}  \ add n to d.
   BX AX XCHG  CWD   BX POP  CX POP  AX CX ADD  CX PUSH
   DX BX ADC   NEXT C;

CODE >< ( u - u2)  { Control?}  BL BH XCHG   NEXT C;
\ reverse the bytes within a word.

: WITHIN ( u n n2 - f)  { Control?}  OVER - >R - R> U< ;
\ true if n <= u < n2  given circular comparison.

: ERASE ( a u)     0 FILL ;  { CONTROLLED}
: BLANK ( a u)    BL FILL ;  { CONTROLLED}

SCREEN 54
\ Byte move operators
: CMOVE> ( a a2 u)  { CONTROLLED}
```

```
\ move u bytes from a to a2, rightmost byte first.
   DUP DUP  >R  D+ R>  ?DUP
   IF  0 DO  1- SWAP 1-  TUCK C@ OVER C!  LOOP THEN 2DROP ;

: MOVE ( a a2 u)   \ move u bytes from a to a2 without overlap.
   >R  2DUP U< IF  R> CMOVE>  ELSE  R> CMOVE  THEN ;

: ROLL ( w[u] w[u-1]...w[0] u - w[u-1]...w[0] w[u])
\ rotate kth item to top of stack.  { Delete?}
   DUP  BEGIN  ?DUP WHILE  ROT >R      1- REPEAT
        BEGIN  ?DUP WHILE  R> ROT ROT  1- REPEAT ;

SCREEN 55
( Double-number math-- high-level)  EXIT
: S>D ( n - d)   DUP 0< ;           \ extend n to d.
: D>S ( d - n)   DROP ;  { DOUBLE}  \ truncate d to n.
{ Require?}

: D- ( d d2 - d')   DNEGATE D+ ;  { DOUBLE}

: D2* ( d - d*2)   2DUP D+ ;
: D2/ ( d - d/2)   SWAP 2/ 32767 AND         { DOUBLE}
   OVER 1 AND IF  32768 OR  THEN  SWAP 2/ ;  { Require?}

SCREEN 56
( Double-number math-- low-level)
CODE S>D ( n - d)   \ extend n to d.
   BX AX XCHG  CWD  AX PUSH  BX DX XCHG   NEXT C;
CODE D>S ( d - n)   BX POP  NEXT C;  { Req?}  \ truncate d to n.

CODE D- ( d d2 - d3)   BX DX MOV  AX POP  BX POP  CX POP
   AX CX SUB  CX PUSH  DX BX SBB   NEXT C;   { DOUBLE}

CODE D2* ( d - d2)
   AX POP   AX SHL  BX RCL   AX PUSH   NEXT C;
CODE D2/ ( d - d2)  { DOUBLE}  { Require?}
   AX POP   BX SAR  AX RCR   AX PUSH   NEXT C;

SCREEN 57
\ More Double-number math
: D<  ( d d2 - f)
   ROT 2DUP = IF  2DROP U< EXIT THEN  2SWAP 2DROP > ;

: D0= ( d - f)        OR 0= ;     { DOUBLE}
: D=  ( d d2 - f)   D- OR 0= ;    { DOUBLE}

: DABS ( d - ud)   DUP 0< IF  DNEGATE  THEN ;  { Double?}

: DMAX ( d d2 - dmax)  { DOUBLE}
   2OVER 2OVER D<     IF 2SWAP THEN  2DROP ;
: DMIN ( d d2 - dmin)  { DOUBLE}
   2OVER 2OVER D< NOT IF 2SWAP THEN  2DROP ;

SCREEN 58
\ Double-number operators
: 2CONSTANT ( - w)   CREATE  , ,  DOES> 2@ ;
\ create a double constant.  { DOUBLE}
: 2VARIABLE ( - a)   VARIABLE  0 THERE ! CELL ALLOT ;
\ create a double variable.  { DOUBLE}

: D@ ( a - d)   2@ ;  { DOUBLE}
: D! ( d a )    2! ;  { DOUBLE}

: DLITERAL ( d ) ( - d)  { Double?}  \ compile d as a literal.
   SWAP [COMPILE] LITERAL [COMPILE] LITERAL ; IMMEDIATE

: D.R ( d n)  { DOUBLE}
\ print d right-justified in field of width n.
   >R  TUCK  DABS  <#  #S ROT SIGN  #>
   R>  OVER - 0 MAX SPACES  TYPE ;

SCREEN 59
( Mixed-precision multiply and divide-- high-level)  EXIT
: M* ( n n2 - d)  { Control?}
```

```
\ signed mixed-precision multiply.
   2DUP XOR >R  ABS SWAP ABS UM*  R> 0< IF NEGATE THEN ;

: M/MOD ( d n - rem quot)  { Control?}
\ signed rounded-down mixed-precision divide.
   2DUP XOR >R  OVER >R  ABS >R DABS R> UM/MOD
   SWAP R> 0< IF  NEGATE  THEN
   SWAP R> 0< IF  NEGATE  THEN ;

SCREEN 60
( Mixed-precision multiply and divide-- low-level)
CODE M* ( n n2 - d)  { Control?}
\ signed mixed-precision multiply.
   BX AX XCHG  DX POP   DX IMUL   AX PUSH  DX BX MOV   NEXT C;

CODE M/MOD ( d n - rem quot)  { Control?}  DX POP  AX POP
\ signed rounded-down mixed-precision divide.
   BX BX OR  5 L# JZ  ( divide by zero?)
   BX IDIV        AX BX MOV  DX PUSH   NEXT
 5 L: AX DX MOV  0 # BX MOV  DX PUSH   NEXT C;

SCREEN 61
( Mixed-precision multiply and divide-- floored)  EXIT
CODE M* ( n n2 - d)  { Control?}
\ signed mixed-precision multiply.
   BX AX XCHG  DX POP   DX IMUL   AX PUSH  DX BX MOV   NEXT C;

: M/MOD ( d n - rem quot)  { Control?}
\ signed floored mixed-precision divide.
   DUP >R  2DUP XOR >R  DUP >R  ABS >R DABS R> UM/MOD
   SWAP R> 0< IF  NEGATE  THEN
   SWAP R> 0< IF  NEGATE  OVER IF  R@ ROT -  SWAP 1-  THEN THEN
   R> DROP ;

SCREEN 62
\ Multiply and divide
: /MOD ( n n2 - n3 n4)   >R DUP 0< R> M/MOD ;

: /   ( n n2 - n3)   /MOD  NIP ;
: MOD ( n n2 - n3)   /MOD  DROP ;

\ Intermediate product is 32 bits:
: */MOD ( n n2 n3 - n4 n5)   >R M* R> M/MOD ;
: */    ( n n2 n3 - n4)      >R M* R> M/MOD  NIP ;

CODE * ( n n2 - n3)   AX POP  BX IMUL   AX BX MOV   NEXT C;

EXIT
: * ( n n2 - n3)   UM* DROP ;

SCREEN 63
\ Number conversion operator
| : val? ( a u - d 2 , n 1 , 0)
\ string to number conversion primitive.  True if d is valid.
\ Returns d if number ends in final '.' and sets dpl = 0
\ Returns n if no punctuation present   and sets dpl = 0<
   [ #Jot 1- ] LITERAL MIN  PAD 1- OVER -  TUCK >R  CMOVE
   BL PAD 1-  DUP dpl ! C!  0 0 R>
   DUP C@ [ASCII] - = DUP >R - 1-
   BEGIN  CONVERT  DUP C@  DUP [ASCII] : =
     SWAP [ASCII] , [ASCII] / 1+ WITHIN  OR
   WHILE  DUP dpl !  REPEAT  R> SWAP >R IF  DNEGATE  THEN
   PAD 1- dpl @ - 1- dpl !   R> PAD 1- = ( valid?)
   IF  dpl @ 0< IF DROP 1 ELSE 2 THEN  ELSE  2DROP 0  THEN ;

: VAL? ( a u - d 2 , n 1 , 0)  { Feature}  'val? @EXECUTE ;

SCREEN 64
\ Interpreter proper
| : val, ( ... w )
\ compiles the top w stack items as numeric literals.
   DUP BEGIN  ROT >R                1- ?DUP 0= UNTIL
       BEGIN  R> [COMPILE] LITERAL  1- ?DUP 0= UNTIL ;
```

```
: interpret  { Feature}  \ the text compiler loop.
   BEGIN  BL WORD  FIND  ?DUP
     IF    STATE @ =  ( Imm?) IF  ,  ELSE EXECUTE  THEN
     ELSE  COUNT VAL?  DUP huh?
           STATE @ IF  val,  ELSE  DROP  THEN
     THEN
   AGAIN ;

SCREEN 65
\ QUIT support
: EVALUATE ( a u)   \ evaluate a string.
   #TIB 2@ 2>R  #TIB 2!  BLK 2@ 2>R  0 0 BLK 2!  interpret
   2R> BLK 2!  2R> #TIB 2! ;

: EXPECT ( a +n)   'expect @EXECUTE ;

: QUERY  { CONTROLLED}
\ fill TIB from next line of input stream.
   0 0 BLK 2!  TIB 80 EXPECT  SPAN @ #TIB ! ;

: ok?   \ status check.
   D0 @ [ #Safe ] LITERAL -  HERE U< ABORT" No Room"
   DEPTH 0< ABORT" Stack?" ;

: OK?  { Feature}  ok?  STATE @ 0= IF ." ok" THEN ;

SCREEN 66
\ QUIT and ABORT
: QUIT   \ default main program.
   RESET  BEGIN  CR QUERY  SPACE interpret  OK?  AGAIN ;

: GRIPE ( a u)  { Feature}  \ default error handler.
   BLK @ IF  BLK 2@ scr 2!  THEN
   THERE COUNT TYPE SPACE  ( msg ) TYPE ;

: ABORT   BEGIN  PRESET QUIT  GRIPE  AGAIN ;
\ default main program and error handler, courtesy Wil Baden.

: ABORT"   \ compile error handler and message.
   [COMPILE] IF  [COMPILE] "  COMPILE err  [COMPILE] THEN ;
  IMMEDIATE

SCREEN 67
( Debug-- EXIT when done)
: .S  { Control?}  \ display the data stack.
   DEPTH 0 MAX  ?DUP
   CR  IF 0 DO  DEPTH I - 1- PICK  .  LOOP  THEN  ." <-Top " ;

: DUMP ( a u)  { RESERVED}  \ simple dump.
   SPACE 0 DO  DUP 7 AND 0= IF  SPACE  THEN  DUP C@ .  1+ LOOP
   DROP ;

: ? ( a)   @ . ;  { Control?}

: WORDS  { Control?}  \ simple word list.
   CONTEXT @
   BEGIN @ ?DUP
   WHILE  DUP CELL+ COUNT 31 AND TYPE SPACE  REPEAT ;

SCREEN 68
\ FORGET support
| : clip ( a 'lfa)   \ unlink words below the given address.
   BEGIN   DUP @
   WHILE  2DUP @  SWAP U< NOT ( ie U<= )
     IF    DUP @ @ OVER ! ( unlinks it )  ELSE  @  THEN
   REPEAT  2DROP ;

: crop ( lfa)
\ crop dictionary to the given link address.
   f CELL+ ( ie vlink)  2DUP clip
   BEGIN  @ ?DUP WHILE  2DUP CELL - @ ( ie >RAM) clip
   REPEAT  FORTH DEFINITIONS  DUP CURRENT @ clip  h ! ;

SCREEN 69
```

```
\ FORGET and variations
: GUARD    h H0 3 CELLS CMOVE   THERE T0 ! ;   { Feature}
: EMPTY    H0 h 3 CELLS CMOVE   T0 @   r  ! ;   { Feature}

: >link ( cfa - lfa)
   BEGIN  1- DUP C@ 128 AND UNTIL  CELL - ;

: FORGET   \ forget words from the following <name>.
   CURRENT @ CONTEXT !    ' >link
   DUP HERE H0 @ WITHIN ABORT" Can't"  crop ;
{ FORGET cannot recover RAM and so is not ROMable.}
{ Delete?}


SCREEN 70


SCREEN 71
\ ------ Device drivers ---------------------------
HEX
| CODE (type) ( a u)    BX CX MOV  DX POP    1 # BX MOV
   40 # AH MOV   21 INT   BX POP   'pause ) JMP C;

| CODE KDOS ( - key -1 , ? 0)
\ check for key pressed.
\ Special keys are returned in high byte with low byte zeroed.
   BX PUSH   FF # DL MOV    6 # AH MOV    21 INT
   0 # BX MOV   2 L# JE       AH AH SUB    ( special key?)
   AL AL OR     1 L# JNZ     7 # AH MOV    21 INT
   AH AH SUB    AL AH XCHG
 1 L: TRUE # BX MOV    2 L: AX PUSH    'pause ) JMP C;

SCREEN 72
\ KEY and EMIT actions
  13 EQU #EOL ( end-of-line)    10 EQU #LF   ( line-feed)
HERE EQU $Eol   #EOL C, #LF C,    2 EQU #Eol

| : (cr)    $Eol #Eol (type) ;

| : (key?) ( - f)    \ true if key pressed since last KEY.
   key' @ 0= IF  KDOS  key' 2!  THEN  key' @ ;

: KEY ( - n)    BEGIN  (key?) UNTIL  key' CELL+ @   0 key' ! ;
: EMIT ( b)    hld C!  hld 1 TYPE ;

SCREEN 73
\ EXPECT action
08   EQU #BSP ( backspace)     127 EQU #DEL ( delete)
27   EQU #ESC ( escape)
HERE EQU $Bsp ( * ) 3 C, #BSP C, BL C, #BSP C,

| : expect ( a +n)    >R   0   ( a o)
\ read upto +n chars into address; stop at #EOL or #ESC
   BEGIN  DUP R@ <
   WHILE  KEY 127 ( 7-bit ASCII) AND
     DUP #BSP =   OVER #DEL = OR
     IF    DROP  DUP IF  1-  $Bsp COUNT TYPE  THEN
     ELSE  DUP #EOL = OVER #ESC = OR
       IF  DROP  SPAN !  R> 2DROP  EXIT  THEN
       ( otherwise) BL MAX >R  2DUP +  R> OVER C!  1 TYPE  1+
     THEN
   REPEAT  SPAN !  R> 2DROP ;

SCREEN 74
\ Dumb terminal actions
| : (keys) ( a +n)    >R  0   ( a o)
\ read upto +n chars into address without echo; stop at #EOL
   BEGIN  DUP R@ <
   WHILE  KEY  DUP #EOL =
     IF  R> 2DROP  DUP >R  ( early out)
     ELSE  BL MAX >R  2DUP +  R> SWAP C!  1+  THEN
   REPEAT  SPAN !  R> 2DROP ;

| : (mark) ( a n)     ." ^"  TYPE ;
```

```
| : (page)    25 0 DO   CR   LOOP ;
| : (tab)   ( n n2)     CR   DROP SPACES ;

SCREEN 75
\ Initialize automatic variables
HERE EQU RAMs
] nope expect source nope nope  val? [
( key' ) 0 , 0 ,
HERE RAMs - EQU #RAMs


SCREEN 76


SCREEN 77
\ ------ Initialization ----------------------------
D0 CONSTANT parms    \ System parameter table.

CREATE glass         \ Simple transfer table.
] (type) (cr)  (keys) (key?)  (mark) (page)  (tab) nope [

: READY   ." Ready" ;  { Feature}  \ Initialize application.
: BYE     0 EXECUTE ;  { Feature}  \ Shut down  application.


SCREEN 78
\ Initialization-- high-level
160 CONSTANT VERSION  { Feature}  \ ZEN 1.60

| : vocabs    \ initialize vocabularies.
   f CELL+ ( ie vlink)
   BEGIN  @ ?DUP
   WHILE  DUP CELL+ @ OVER CELL - @ ( ie >RAM) !  REPEAT ;

| : cold    \ high-level coldstart initialization.
   TRUE ( wake) entry entry 2!  TO 2@ r 2!   glass x !
   RAMs 'pause #RAMs CMOVE
   EMPTY  vocabs  PRESET  FORTH DEFINITIONS  DECIMAL
   " READY" EVALUATE   ABORT ;

\ If all definitions are headerless, substitute:  READY  ABORT ;

SCREEN 79
\ Initialization-- low-level
HEX  HERE ( *) ," No Room $"

| CODE Coldstart    \ low-level initialization.
   1000 # BX MOV    4A # AH MOV    21 INT  ( enough room?)
 1 L# JNC   ( No:)
   ( *) 1+ # DX MOV    9 # AH MOV    21 INT    0 # JMP   ( Bye)
 1 L: #SP0 # SP MOV    #RP0 # BP MOV    BP u ) MOV
   ' cold >BODY # SI MOV   ( I register)    NEXT C;

HERE ( * ) Power ORG    ASSEMBLER   ' Coldstart # JMP C;
     ( * )        ORG


SCREEN 80

SCREEN 81
\ ------ FILE extension ----------------------------
#Used USER IO-RESULT  DROP

26  EQU #EOF   \ control-Z marks the end of older text files.

128 EQU buff   \ MS-DOS command tail and default fcb buffer.
192 EQU name   \ RENAME-FILE takes two names.

256  buff - EQU #buff    \ size of buffer in bytes.
name buff - EQU #name    \ size of name in bytes plus zero.

| : >fname ( a u - a2)    \ convert string to ASCIIZ file name.
   buff  2DUP 2>R  SWAP MOVE  R@  0 2R> + C! ;
```

```
SCREEN 82
\ MS-DOS interface
HEX
CODE fdos ( DX CX handle function# - AX)
\ generic call to MS-DOS
   BX AX MOV  BX POP  CX POP  DX POP   21 INT
LABEL return   AX BX MOV   1 L# JB   AX AX SUB   2 L# JZ
 1 L: BX BX SUB ( non-zero retcode forces zero result)
 2 L: u ) DI MOV   AX IO-RESULT entry - [DI] MOV   NEXT C;

| CODE rename ( a a2 function# - AX)
   BX AX MOV  DI POP  DX POP   21 INT   return JU C;

| CODE seek ( DX CX handle function# - AX DX)
   BX AX MOV  BX POP  CX POP  DX POP   21 INT
   DX PUSH   return JU C;

SCREEN 83
\ 5 file primitives
HEX
: OPEN-FILE   ( a u - w)   >fname  0 0 3D02 fdos ;
: CREATE-FILE ( a u - w)   >fname  0 0 3C00 fdos ;

: DELETE-FILE ( a u)   >fname  0 0 4100 fdos  DROP ;
: CLOSE-FILE  ( w)      0 0 ROT     3E00 fdos  DROP ;

: RENAME-FILE ( a u a2 u2)
   >fname  name #name CMOVE>
   >fname  name  5600 rename  DROP ;

SCREEN 84
\ Read, write and seek bytes
HEX
\ Read or write u bytes to or from address a to file w.
: READ-FILE  ( a u w - u2)   3F00 fdos ;
: WRITE-FILE ( a u w - u2)   4000 fdos ;

: SEEK-FILE ( doff n w - dpos)   \ add an offset to file w.
\ n neg: to start; n pos: to end; n zero: to current.
   SWAP DUP IF  0< CELLS 1+  THEN  4201 + seek ;

\ Return file position or size.
: FILEPOS  ( w - d)   >R  0 0 0 R> SEEK-FILE ;
: FILESIZE ( w - d)   >R  0 0 1 R> SEEK-FILE ;

SCREEN 85
\ Read and write lines of text
: WRITE-CR ( w)   $Eol #Eol ROT WRITE-FILE  DROP ;

: READ-LINE ( a u w - 0 0 | u2 t)
{ Greater performance will result if the end-of-line sequence  }
{ is read into the address and the size u adjusted accordingly.}
   >R  buff OVER 1+ #buff MIN R@ READ-FILE  ( a u u2)
   DUP 0= IF  R> 2DROP 2DROP  0 0 EXIT THEN ( end of file)
   buff OVER #EOL SCAN  NIP ( a u u2 u3)
   ?DUP IF   #Eol OVER - >R -
        ELSE  2DUP U< >R  THEN  MIN R> ( a u4 #seek)
   ?DUP IF  S>D 0 R@ SEEK-FILE 2DROP  THEN
   buff OVER #EOF SCAN  NIP -   ( remove if no control-Zs)
   R> DROP ( a u4) >R  buff SWAP R@ CMOVE>  R> TRUE ;

SCREEN 86
\ Load and save files
: GO ( a u)  { Feature}  \ evaluate the KERNEL.SRC file.
   " KERNEL.SRC" OPEN-FILE DUP huh?  ( w) >R
   BEGIN  buff  DUP 64 R@ READ-LINE
   WHILE  EVALUATE  REPEAT  2DROP   R> CLOSE-FILE ;

: SAVE-FILE ( a u)  { Feature}  \ save the dictionary by name.
   CREATE-FILE DUP huh?  ( w) >R
   'pause RAMs #RAMs CMOVE   GUARD   f CELL+ ( ie vlink)
   BEGIN  @ ?DUP ( save vocabularies)
   WHILE  DUP CELL - @ ( ie >RAM) @ OVER CELL+ !  REPEAT
   256  HERE OVER - R@ WRITE-FILE  DROP   R> CLOSE-FILE ;
```

```
SCREEN 87
\ BLOCK word set
VARIABLE system    VARIABLE block#    VARIABLE update
VARIABLE buffer  1024 CELL - ALLOT

: seek-block ( u w - a n w)
   >R  1024 UM* TRUE  R@ SEEK-FILE 2DROP  buffer 1024 R> ;

: SAVE-BUFFERS  { BLOCK}  system @ 0= ABORT" No File"
   system 2@ seek-block WRITE-FILE DROP ;

: BUFFER ( u - a)  { BLOCK}  >R  block# 2@  R@ - AND
   IF  SAVE-BUFFERS  THEN   0 R> block# 2!  buffer ;

: BLOCK  ( u - a)  { BLOCK}
   DUP block# @ = IF  DROP buffer  EXIT THEN
   BUFFER >R  system 2@ seek-block READ-FILE DROP  R> ;

SCREEN 88
\ BLOCK support
: EMPTY-BUFFERS  { CONTROLLED}  0 TRUE block# 2! ;

HEX
: UPDATE  { BLOCK}  TRUE update ! ;
: FLUSH   { BLOCK}
   SAVE-BUFFERS  0 0 system @ 4500 fdos  CLOSE-FILE ;

: LOAD ( u)  { BLOCK}
   BLK 2@ 2>R  0 SWAP BLK 2!  interpret  2R> BLK 2! ;

: block   BLK @ ?DUP IF  BLOCK 1024  ELSE  #TIB 2@  THEN ;

\ Use this definition if the BLOCK word set is compiled:
: READY   ." Ready!"    ['] block 'source !
   " NEW.SCR" OPEN-FILE  DUP huh?  system !  EMPTY-BUFFERS ;
```

Related Reading

- News
- Commentary

- **Kaazing WebSocket Gateway Pushes Offline iPhone Data**
- **Altova Conjures Up XML Alchemy**
- **Symfony in PHP Major Components**
- **Visual Studio 2012 Arrives, Windows 8 Development Commences**
  More News»

- Most Popular
- On the Web

- **Hacking for Fun: Programming a Wearable Android Device**
- **Using SQLite on Android**
- **A Gentle Introduction to OpenCL**
- **Easy DOM Parsing in Java**
  More Popular»

- Slideshow
- Video

- **Developer's Reading List**
- **Developer's Reading List**
- **Developer's Reading List**
- **Developer's Reading List**
  More Slideshows»

More Insights
White Papers

- 5 Things You Need to Know About BYOD
- Take Advantage of a Unified Workspace

More >>

Reports

- Strategy: SaaS, Social and Your ESB
- IT Pro Ranking: SIEM

More >>

Webcasts

- New approaches to Systems Engineering and Embedded Software Development
- Dramatically Improve the Way Work Gets Done through Collaboration

More >>

INFO-LINK

World's Fastest O
Database.
Get This Month's
Dobb's Journal - D
Now!
News, Commenta
Features - Get Dr
Update Newslette
Dr. Dobb's Is Now
iPad! Download T
Today.

Like

Login or Register to Comme

Showing 0 comments

Sort by oldest first

THE WEB

:w iPad and iOS 5.1 Mean for

; gorgeous. But local storage for HMTL5 is
n the new iPad and performance of some apps is
eep dive into the issues, including benchmarks

ring as A Concurrency

a way of passing data between a producer and a
at different rates. It ensures that the consumer
e data with minimal lag.

GDB Breakpoints in C Source

nted to embed GDB breakpoints in C source
like this:

```
>,\n");
)INT;
d!\n");
```

nel Exploits

rnel? Because it has a huge attack surface with
nteresting bugs. This presentation (pdf) takes a
o recently reported Linux-kernel exploits.

: Web" >>

Enabling People and Organizations to Harness the Transformative Power of Technology

CIOs & IT Professionals

- Black Hat
- BYTE
- Cloud Connect
- Dark Reading
- Enterprise 2.0
- Enterprise Connect
- Enterprise Efficiency
- HDI
- InformationWeek
- InformationWeek 500
- InformationWeek 500 Conference
- InformationWeek Events
- InformationWeek Global CIO
- InformationWeek Healthcare
- InformationWeek India
- InformationWeek Reports
- InformationWeek SMB

Software Developers

- Dr. Dobb's
- Dr. Dobb's M-Dev
- Dr. Dobb's Journal
- Dr. Dobb's Update
- TechWeb.com

Web & Digital Professionals

- Internet Evolution
- Online Marketing Summit
- TechWeb.com

Government Officials

- GTEC Ottawa
- InformationWeek Government
- TechWeb.com

Vertical Markets

- Advanced Trading
- Bank Systems & Technology
- CreateYourNextCustomer
- InformationWeek Government
- InformationWeek Healthcare
- Insurance & Technology
- Light Reading / Telecom
- The CMO Site
- Wall Street & Technology

Game Industry Professionals

- Gamasutra.com
- Game Developers Conference (GDC)
- Independent Games Festival
- Game Developer Magazine
- GDC Europe
- GDC China
- Game Career Guide

Global Communications Service Providers

- 4G World
- Heavy Reading
- Heavy Reading Insiders
- Pyramid Research
- Light Reading
- Light Reading India
- Light Reading Mobile
- Light Reading Cable
- Light Reading Europe
- Light Reading Asia
- Ethernet Expo
- TelcoTV
- Tower Summit
- Light Reading Live & Virtual Events
- Webinars

Most Popular

- Cable Catchup
- Cloud Connect Blog
- Digital Life
- Evil Bytes
- InformationWeek Reports
- Interop Blog
- Monkey Bidness
- Over the Air
- Personal Tech
- The Philter
- Valley Wonk

Interop
Mobile Connect
Network Computing
No Jitter
TechWeb.com
The BrainYard
Game Advertising Online

UBM TechWeb Reader Services

About UBM TechWeb | Advertising Contacts | Technology Marketing Solutions | Contact Us | Feedback

Reprints | TechWeb Digital Library / White Papers | TechWeb Events Calendar | TechWeb.com

Terms of Service | Privacy Statement | Copyright © 2012 UBM TechWeb, All rights reserved.

Powered by Zend/PHP

Dr. Dobb's Home | Articles | News | Blogs | Source Code | Dobb's on DVD | Dobb's TV | Webinars & Events

About Us | Contact Us | Site Map | Editorial Calendar