```
 %if 0 ; docs{

  A BOOTABLE FORTH STYLE "OS"

  This file represents a bootable readonly forth-like system.  It uses a
  byte-code approach to creating a simple forth dictionary.  Its overall aim
  is to create a portable and minimal booting interactive system. All
  commands entered interactively (in the SHELL loop) are ’compiled’ into
  bytecode either into an anonymous code buffer or to the dictionary and then
  executed.

  Also, the general aim of the code is to get to a source interpreter
  and compiler in the minimum code size possible.

  This is by no means a standard forth! It is an experiment.

  MOTIVATION

    "our civilization will collapse under the weight of its own complexity"
       charles moore.

    "simplicity is more fertile than complexity"
       anon

    This code is an attempt to answer a question:
      "what is the smallest bootable, interactive, portable system
       that can add to itself from source code, in a structured,
       orderly and understandable way?"

    To elaborate...
      "Smallest"
        The aim of any developer should be to reduce the size and
        complexity of the core system (not increase it) while
        satisfying the conditions below. Currently (2018) about 7K core
        system.
      "Portable"
        from microcontroller to mainframe using a rigorous practical
        virtual machine and bytecodes.
      "interactive": actually this condition is not so important
        because a system that can add to itself from source code
        can easily compile and execute an interactive REPL interpreter
        or shell. But it is important that the user can choose what
        bits to add to the system, and what not to add.
      "Add to itself"
        via disk, serial connection, or tcp, using globally unique and
        locatable object names (words). So each word has a domain name
        which, when combined with the word itself, forms a globallly
        unique name.
      "from source code"
        This condition implies that a compiler must be present
      "Bootable" doesnt require any operating system on any architecture.
      "Structured"
        must add to itself in manageable, contained units of code
        that can be debugged individually.
      "Understandable"
        must strive to be simple enough for an average coder to
        understand after reading for a few days.

    Actually forth contains partial but powerful solutions to some of these
    criteria.

    The question above is important because it has implications for our
    consumption of resources, impact on the environment, beholden-ness
    to obsolescence and relationship with technology. In particular
    power consumption is related to software complexity. Simple software
    can run on small low-power consumption computers (such as a coin-cell
    powered bicycle computer).

    Another important aspect is digital security
    and the "knowability" of code. Opaque massive compiled code is
    inherently insecure because it is unknown and unknowable. At some
    point governments will need to acknowledge this, because the
    trojan horses they implant in various systems will be compromised
    by the trojan horses that others place in their systems. It is
    a futile arms race of insecurity.

    This question also has implications for how human beings interact with
    technology and whether they are in control of those interactions or are
    victims of them. The divide between "programmer" and "user" is artificial
    and serves commercial purposes not human ones. Just as the divide between
    operating system and software is also artificial.

    To expand a little, the code uses bytecode and a simple 2 stack virtual
    machine to hopefully allow portability between microcontrollers and
    modern laptops/desktops. By attempting to  port to very disparate chip
    architectures (avr micros, x86, z80, arm), the designer is encouraged to
    think about what is essential for usability and what is just fluff. This
    may seem to contradict one of charles moore’s principles which is "dont
    code for what you may need in the future, dont try to generalise your
    code". But it is more an attempt to harness the power of the commonality
    of the internet and the power of virtual machines to cheat obsolescence.
    Tinyness and minimalism will hopefully not be sacrificed.

    The system tries to make the core as tiny as possible while not
    sacrificing "understandability".  So the aim is to get to a source code
    compiler in the minimum amount of code. Forth ideas facilitate this.

    Another important idea is that of universal naming. All objects (in this
    case forth words) should have a resolvable, locatable, unambiguous
    universal name eg 4th.core.dup Since almost all code is source, then each
    object (forth word) consists of a minimal syntax (space delimited words)
    and a series of named-objects (other forth words). This has the simple
    but powerful consequence that, given the name of a word, all
    "dependencies" can can be located and obtained, and the given word can be
    compiled and run. This also applies to data structures.

    On top of this system we could code a parsing machine
    see bumble.sf.net/books/gh/gh.c (nearly but incomplete) which
    would allow the forth machine to recognise and assemble more
    expressive syntaxes for code and data structures.

  JOURNEY

    copying machine code or peeks and pokes into a vic20 (?)
    qbasic as a teenager. Nice help screens on an IBM XT?
    First had to learn x86 assembly which I had been meaning
    to do since the age of 12. Learn about
    x86 bios calls. Then had to think about forth for a number of years.
    First heard of forth from D. McBain. Then had to think about parsing and
    compiling etc.  Coding php scripts, in Wagga Wagga NSW. html-form code
    editor in php, writing text converters in #!/usr/bin/sed and thinking
    about the sed "virtual machine" in Almetlla de Mar, Catalunya. learning
    [TCL] and its simple bracket syntax (almost as simple as forth).
    Learning a little Java in Bogota, but never achieving anything useful.
    Linux shell scripting and bash. c coding. Writing the pattern
    parser unsuccessfully in many languages. Some avr assembly.
    Markdown and code that produces code. Learning x86 assembly booting from
    MikeOs template.

  FEATURES

    Case sensitive, and all core words are lower case, but
    in documentation I may make them upper case so that it is
    obvious that I am referring to the forth WORD and not the
    English word.

    Unlike some forths, this version compiles commands entered
    interactively to an anonymous buffer. The advantage of this is that
  flow control words like if, fi, else, begin, until... can be
  used interactively, not just in colon : definitions.

  Bytecode. Not a standard forth: eg, THEN is FI
  "execute" is "pcall" to match with "fcall" opcode

Even "proceedures" (everything after the no-op procedure) are
written in pseudo forth. That means that they are just a series
of calls - to opcodes and other proceedures. The idea
is to make porting to another architecture simpler.

DIFFERENCES FROM (94?) STANDARD FORTH

   I have attempted to keep the idea of a "compiling" forth.
   This means that there is really no such thing as "interpreting"
   in this forth. The only interpreting opcode is PCALL and I
   may remove it. My motivation for this is to stick as closely to
   the operation of real machines. Chips general can only execute
   machine code that exists at some location in memory. I want to
   adhere to this principle to make the idea of a "virtual machine"
   more coherent.

   * All defining words will probably have to be immediate.
   * All standard words are lower case.

   postpone is post
   RECURSE is just the name of the word. For example
     : gcd ( a b -- gcd) ?dup if tuck mod gcd fi  ;
     just works and finds the greatest common divisor.
     This may have the implication that two words cannot have
     the same name (which is also required by unique naming
     ideas and namespaces)
   IF/THEN is IF/FI
     because I find THEN just too different to every other
     computer language ever invented.
   EXECUTE is PCALL
   WORD is WPARSE
     because the word word is used for way too many things in
     Forth. eg function in dictionary, 2 bytes of data, space delimited
     text ...Also wparse works for various whitespace
     characters (tab, newline, space ) not just the space
     (asci 32) character.
   VARIABLE is VAR
     because I dont need to type any more than I already do.
   IMMEDIATE is IMM
   COMPILE, is ITEM,
     and works slightly differently from COMPILE, (it takes a flag
     that indicates if the argument is an opcode, literal or procedure)
     But that may change.
   I is ii
   J is jj
   CHAR
     is an immediate word. To use CHAR
     in a colon definition, do "char * literal" or just use [char]
   accept does not take a character limit, yet.
   COMPARE is ( a A n -- flag ) rather than ( a n A N -- a' )
     so this compare assumes that the 2 strings are of equal
     length.

 STATUS

 The system is edging towards a usable forth-style system. It
 can read and write to usb from an x86 system in real mode, but
 there is no easy way to edit buffers yet (apart from 'accept')
 so is probably not usable as a standalone system yet.

 PROBLEMS

    defining words do not update the dictionary code pointer
```

```
   but 'does>' does, so we can use does at the end of
   a defining word to make sure the code pointer is updated

PERFORMANCE

  A virtual machine is normally considered to run slower than
  native machine code, but some optimisation techiques are
  possible. We can replace x86 'calls' to opcodes with jumps
  and we can keep the top of the stack in ax. Also it should not
  be too difficult to compile bytecode to native machine code, if
  necessary.

* 16 june 2018 performance testing:

  With exec.x using calls to opcodes and pop/push return pointer
  in the opcodes, performance is as follows using the word
  'timeloop' to measure looping speed

  With qemu on asus e402m machine
    500 000 loops took 495 milliseconds
    1000 000 loops took 935 milliseconds
  Booting from usb
    1 million loops took 110 ms
    2 million loops took 220 ms
    10 million loops took 1100 ms
  So as expected, much faster on the actually machine rather
  than in qemu. But is this fast enough to play chess?

* 17 june 2018
  On Ibm thinkpad r52, using exec/call version, real mode
  using word 'timeloop'. booting from usb. Strangely this seems
  twice as fast as when I converted to using jmps and having
  top of stack held in dx.

    1 million loops took 55 milliseconds
    10 million loops took 275 ms
    40 million loops took 1155 ms

  So surprisingly the old r52 thinkpad was 4 times faster than
  the more modern asus machine in real mode.

* 18 june 2018

  system converted from call/ret (in exec.x) to jmp ...
  With qemu on asus e402m machine. Using word 'timeloop'
    500 000 loops took 275 milliseconds
    1000 000 loops took 605 milliseconds
  So by converting from call/ret to jmp there is about a 30%
  improvement in qemu.

  On asus eeepc (seashell), using jmp in exec/opcode, real mode
  booting from usb.
    1 million loops took 55 milliseconds
    10 million loops took 605 ms
    16 million loops took 1045 ms

  asus e402m machine Booting from usb, jmp (not call/ret)
    10 million loops took 990 ms
    So not much improvement on modern machine. Perhaps
    modern chips do realmode very badly.

  With qemu on asus e402m machine, jmps and top of stack in DX
  real mode.
    1 million loops took 385 milliseconds
    2 million loops took 825 milliseconds
  So there seems to be a significant improvement with top of stack
  in DX.

VIRTUAL MACHINE AND OPCODES
```

```
    This forth system tries to emphasize portability, knowability
    and interoperability. The idea is that each opcode should be
    standard and have a defined semantic operation on the machine.
    Also, periferals are considered part of the machine. But
    periferals (sensors, actuator, transducers etc) are myriad
    and pluggable and unpluggable. So how do we handle this
    situation: The answer is that the set of available opcodes
    defines the machine- actually forms the machines "signature".
    Since possible periferals and devices attached to the core
    2-stack machine are infinite, we need at least 2 bytes to
    describe them. Or a utf8 type of variable length encoding.

    So: "1234567" may be the opcode to read a 3 axis accelerometer.
    The number 1234567 must be unique, universal and defined in
    some public document.
    But in the actual machine this opcode will be probably mapped to
    a much smaller number (say 66) so that it can be encoded in one
    byte or less (depending on how many periferals/sensors the actual
    machine has). So the signature of the stack machine includes
    all standard opcodes expressed as a list of ranges 1-4, 7-9, 11, 13
    etc as well as any mappings eg 1-4,7,9,66:1234567

    Also there are further subtleties here, which need attention.
    Eg: Absent opcodes can be "emulated" or in some cases just
    keyed to NOP. For example an opcode which moves a robotic
    arm could be emulated as a video monitor display of the robotic
    arm moving.

    An opcode which sets the text colour for a monitor may not be essential
    for the operation of the software and may just be keyed to NOP.

    It would be nice to be able to create missing opcodes using
    "high level" forth (source code) as well as assembler appropriate
    to the current chip architecture.

    Some opcodes, even if absent, can always be constructed from
    core stack machine operations. For example double precision
    arithmetic (eg D+ D- ) can be constructed from 16 bit forth
    opcodes (+ - dup swap etc). But if speed and efficiency is
    important for the given application, then these operations
    should be coded as opcodes written in assembler.

    The general aim is for the structure of the target machine
    to be knowable and analysable from within code.

  STRUCTURE OF THE DICTIONARY

    One of the key ideas of a forth-like system is the use of a
    linked list with each item in the list being a data structure
    with the name and code for a particular function. This linked list
    is called the dictionary. The current code uses the following
    structure for the dictionary:

      [link to previous word] 2 bytes
      [name of word]
      [count of name] 1 byte || IMMEDIATE FLAG
      [code]
      [data or "parameters"]
      ... next word structure

  This uses "reverse" name counts. So the byte containing the
  length of the name is after the name, not before as in the majority
  of forths. This allows us to decompile
  byte code by getting a list of pointers to code and
  then looking up the name. But it may have other unknown disadvantages.
  eg
   db 'minus', 5
   dw exec    ; link to previous

  Another refinement is to hold the top element of the
  stack in ax, which simplifies a lot of stack manipulation.
  eg 1+ becomes "inc ax" etc.

  OBJECT ORIENTATION

   The code below may contain a hint as to how to emulated object
   orientation with forth. Each of the function pointers
   could be like a method on an object.

   create buttons ' ring , ' open , ' laugh , ' cry ,
     : button ( nth --) 0 max  3 min
       cells buttons + @ execute ;

  IDIOMS

    This section contains recipes for doing simple or common programming
    tasks with this forth

    * use a counter to print something once every 8 loops
    -------

     var nn
     : ++ dup @ 1+ swap ! ;

     : something
        ; initialise counter to 0
        0 nn !
        begin
          ; print 8 words to a line
          nn ++ nn 8 mod 0 = if cr fi
        again ;
    ,,,

  CULTURE

    Forth has its own culture. It uses a set of words and concepts
    which are completely different from the main stream coding
    world. It has a set of naming conventions which are not related
    to normal coding naming conventions.

    These days forth is a forgotten backwater with little or no mainstream
    relevance apart from embedded systems, but its ideas remain powerful.

  NAMING

     Since Forth words can contain any punctuation, these little
     characters are (over?) used extensively to try to convey the
     meaning and purpose of words. Below are some notes about
     traditional ways that this has been done

     TOS top of stack, displayed as rightmost element.
     NOS next element on stack.

      / divide by something eg 2/ divide by 2
      / word divider eg: r/w "read/write"  s/o "search order"
      / search for something (convention from unix)
        eg ss/ search for words in source code blocks

      . display something (print to screen and remove from stack)
      , compile something (store executable code or data in memory)
      : define a new word or data buffer
      @ fetch something from memory (and push to data stack)
      ! store something in memory.
      ' get the execution address of something.
      ' modify something (in stack comments) eg x' "x modified"
      c character operation eg: c@ fetch one character (8 bits)
      (something) is the runtime behaviour of "something"
      [something] is an immediate version of the word "something"

       That means that it executes "immediately" or at compile-time.
      XT means "execution token" which is just the address of
      code that can be run by the virtual machine. (either byte code
      or machine-code).

   INSPIRATION

     The code was original inspired by the helpful and simple instructions
     from MikeOS on how to get a minimal booting x86 "operating system"
     working. The code was also inspired by the incredible simplicity
     of forth-like systems, and also, the ease of implementing a bytecode
     system using indirect jumps.

     In general, simplicity appears to be much more fertile than
     complexity. Linus Torvalds was inspired by the simplicity of
     Tannenbaum’s minix system to start the Linux journey. Tannenbaum
     may have been inspired by the simplicity of early unix systems.

     Donald Knuths "literate programming" where the source code
     includes the documentation and also self-documents. The idea
     is that the source code is a document which describes itself,
     both at compile-time and at run-time

   IMMEDIATE WORDS

     One of the main semantic problems of forth systems is the idea of
     immediate vs non-immediate words and "run-time" versus "compile-time".
     Immediate words execute at compile-time and non-immediate words execute
     at run-time. These ideas occur because forth is a compiling system.

       eg char "
           puts the character " (integer 34) on the stack at run-time
       but [char] "  or [ char ] "
           does the same thing at compile time, no not correct...

   FORTH BOOKS

     "An invitation to forth" ...
     "Threaded Interpretive Languages" by RG Loeliger
        Apparently some info on how to build a forth system.
     "A Complete Forth" (1983)
        A good general overview of early forth
     "Thinking Forth": Brodie
        An introduction to forth. Available free online.
     "Forth Programming Handbook" 3rd Edition, Conklin, Rather (2007)
        A good reference and explainer for the 1994 standard
        language. Also reasonable recent.
     "Forth: The next step" by Ron Geere
        This is a simple and useful book which defines some handy
        new words in forth (such as squareroot etc).
     "Forth Application", S.D.Roberts
        Strange and unreadable, at least to me. But no doubt with
        some good ideas.
     Scientific Forth: Julian V. Noble
        A vey well regarded forth book but out of print.
     Finite State Machines in Forth, an article by J.V.Noble
        http://galileo.phys.virginia.edu/classes/551.jvn.fall01/fsm.html
     Good articles by Noble
        http://galileo.phys.virginia.edu/classes/551.jvn.fall01/primer.htm
        http://galileo.phys.virginia.edu/classes/551.jvn.fall01/primer.htm
     Realtime forth

   PEOPLE
     Anton Ertl
     Mitch Bradley
```

```
   FORTH SYSTEMS

     This is a cursory overview of various active and historical forth
     systems.

     Dr dobbs: the commando forth compiler
     GFOS
       A booting forth system for x86, some sound and image support. HDE ide
     https://el-tramo.be/blog/waforth
     spf.sf.net
     Figforth, forth 79 standard, forth 94 standard
     LMI-Forth
       for the IBM pc
     MVP-Forth
       not sure where it ran
     Open Firmware (Mitch Bradley)
       Still used at OLPC for
       booting laptops. A major project.
     cforth
       based on mitch bradley olpc forth
     gforth
       a big complete forth written in c. (non-booting)
     amforth
       for avr chips
     flashforth
       for microcontrollers, eg avr (arduino) using flash memory.
       Runs on Harvard architecture microcontrollers and uses
       a "memory map" technique to separate data memory from
       program memory.

     Multos - stack based vm os
     Colour vision systems, bacchus marsh, victoria, au
     Keycorp, chatswood, KYC www.keycorp.net
     www.forthos.org
     http://www.vsta.org/contact/andy.html
        A bootable x86 forth in protected mode with oo extension
        Example of forthos oo syntax
          Set -> new constant mySet
          1 mySet -> add 123 mySet -> add

      * Add a method to a class
        Set -> :method addRange ( high low self -- )
          -rot do i over -> add loop drop method;

   DONE TASKS

    mar 2019
     * ".rs" to display the return stack, using "rpick" opcode
     * wrote de, as source
     * made ss/ page output
     * made all flow control words use long jumps. Preserved
       short jump words as If/Until/Again etc.. capital case.
     * made last use current
     * wrote a very simple stack underflow check
       in assembly code within "exec". the check is bypassable on
       compilation (for speed of execution).
     * wrote a return stack underflow check.
     * wrote very basic abort word that clears stacks and restarts "shell"
     * made namespaces using interleaved linked lists (wordlists)
       but still need to get "defs" to work, and start off a new list
     * wrote "word." as source code.
     * dont stop de, when an unknown opcode is encountered.
     * made ls print words from all lists in s/o, and
       print the name of the wordlist before each group
       of words
     * made a basic accept.span which takes a character limit
     * speed up find.so for searching all wordlists in searchorder
     * made "missing" do an abort.
```

```
 TASKS

   * rewrite "does>" and "(does>)" as source code. This may
     help to think about what they do, as well.
   * dont add duplicate wordlists to the search order.
   * make "accept" take a character limit.
     This will change the code for all
     "accept" variants.
   * only load the minimum number of blocks before "shell"
   * fix the splash screen reporting of bytecode words (too many)
   * make "anon" like "pad", a transient buffer just after
     the last word in the dictionary. By using a "defining" state
     know when to erase/overwrite the anon buffer, and when
     to execute it. "create" sets state=def, but what sets it
     back state=anon (anonymous compilation). Can we get rid
     of "immediate" state all together?

   * write .code as source (used by de,)
   * write "sliteral" as source code (scompile is used by "create
     so probably cant be written as source code).
   * preserve the old short jump "if" as the word "If"
     to use for compressed code.
   * maybe (clean) and (check) can be combined into one
     function, if limit is reached then drop rstack params
     See commented code way below in (loop). Yes, but
     "next" will still use the (clean) code but not the
     (check) code.
   * fix "ls" so that word and line count is cumulative
     for all word lists. If a wordlist is empty, then print
     "<empty>" etc.
   * decompile compiled strings in words.
   * make word. use the size of the word when decompiling,
     or else just stop when the next xt is reached (check
     with xt- or xt+).
   * make "ops" and xt>op scan through the opcode table
   * improve the history buffer handling for "accept.history".
     If a duplicate command is added to the history buffer, remove
     the old instance and add the new instance.
   * Add some kind of a time stamp for the history buffer
   * make a command to write the history buffer to disk
     or else just store the buffer in on disk, and access it
     with "buffer" etc.
   * make a way of stepping through code (ambitious).
   * try to show the state of x86 registers as code is
     executing, eg es:di (return stack pointer), ss:sp
     (data stack pointer), dx top of stack, etc.

   * comment out the data stack underflow check and time
     the speed difference for some hard task.
   * rewrite "all" to load all blocks until first block with
     no data.
   * write "reload" which deletes all words from the dictionary
     back to "last"? or back to "thru"
     and then loads all blocks sequencially until first empty block
   * opcode to clear data stack, another to clear return stack ?
   * improve .rscall to display the return stack with named fcalls
     using Un, etc. The complication are "pcalls"

   * dont! create a new "state"==defining which is set to true by
     create.
   * work out how to rewrite <# # #> for double precision numbers.
   * write a word like umd/mod (unsigned mixed double mod division)
     which standard # seems to need
     ie umd/mod ( ud u -- ud u / remainder and quotient for /mod *)
   * make nfind look in all the word lists in the s/o
     search order buffer.
   * Write a word in bytecode that searches
     dict from particular xt, then make nfind use that word
   * make Ls print all words in all wordlists, even vocabs
     that are not in the search order.
   * make "missing" or miss.new print the current search order
     stacks. Eventually make "missing" look up words on
     a www index
   * make a word "where" which will search all vocabs
     for a word.
   * make "words" list all words in the first vocab in s/o
     that is "context".
   * make Vocabs and Forth the 2 minimum wordlists in the
     search order. So "only" will reset the s/o buffer to
     these two lists.
   * use EOL instead of 13/10 numbers etc. This is more portable

   * write in' ( A -- ) which shows the name of the
     word containing address A.
     just walk the dictionary(ies) and when xt < A then stop.
     This is complicated by interleaved dictionaries.

   * asci animals. using "line, lineto" etc
   * fix create bug for defining words maybe by: writing (create) and create
     as an immediate word, and adding code to semicolon to see if
     def bit is set and compiling semicolon code there.
     So we check the def bit and either do FCALL (;) or LIT, FCALL
     C, LITW, (;), , etc depending on whether def is set. Bug has
     been fixed for does> and (does>)
   * allow simple editing of blocks to test writing blocks
   * change 'decomp' so that it displays the address as well
     as the name of the fcall functions.
   * use wsize in see see, etc to decompile whole word.
   * write DEPS ( xt -- ) create a list of xts which are
     dependencies of word represented by xt.
   * write a 'dependency' compiler. Find a word on disk,
     start compiling it, leaving a start marker. recurse down
   * a "search bit" in the name count field (next to the
     immediate bit, perhaps. This makes the word invisible
     in the dictionary)... not sure if necessary.
   * make READ do a number of attempts to read disk blocks (1K)
   * rewrite some words as "source" now that the : compiler
     more or less works.
   * find out what words are really necessary for the : word
   * it would be good to be able to add new "opcodes" dynamically
     to allow for the situation where new hardware becomes or is
     available. For example, if a gyroscope is available, then
     extended opcodes should read data from it. Yes the machine
     should be able to build apon itself/ add new opcodes to itself.
   * make a "document" dictionary, or shadow comments to other diskblocks
   * "dictionary full" check, ie. is there any more memory in which
     to place new words? 'unused' word or 'free'
   * implement buffer. Buffer writes to disk and frees
     a buffer and block reads from disk. block calls buffer
   * all defining words need to be immediate, so 'does>' could
     just do that automatically.


 GOTCHAS

   * If you have multiple byte codes on a jump line like this:
     db FETCH, JUMPNZ, .notempty-$
     then the target address .notempty-$ will not be correct

   * There is no way to list all wordlists, unless we maintain
     a list of wordlists and add to it every time a vocab is created.

   * In old forths u* and u/ leave a double precision result.
   so u/ is "um/mod" and u* is "um*" I think. So that code needs
   to be translated.

   * If one of the word counts in bytecode is wrong you get an
     error: "emit ?? @last "
 SOLVED BUGS

   mar 2019
   * "pad char z" interactively actually puts asci code for 'z' on the stack
     before pad! This is because "char z" is getting executed immediately
     and "pad" isnt. A practical solution is to just do
     "pad [char] z" interactively, which works because all those
     words get compiled (to "anon") anyway.

   * do/loop does not have the +/-128 byte limit anymore
   * removed 128 jump limit from all flow control words,
     I think, and put them in source.

   * feb 2019: u. stopped working because I changed /mod to be
       a signed division. Change u. to use u/mod instead

   * mod /mod and / did not work for negative numbers !!!
     Needed to use a signed division instruction in x86
   * a bug in bytecode Forth vocab, where .notempty-$
     is 1 byte too many. Because db fetch, jumpnz, <target> on
     one line alters the target calculation !!
   * using find.so to search all wordlists really slowed down
     searching and compilation... This was because wfind was
     too slow. I recoded using the same code as "ffind" and
     it sped up again.

 BUGS

   * bracket comments dont seem to work interactively ( eg this )
   * accept.arrow has a similar bug to accept.hist
   * accept.hist does not allow entering long commands
     interactively. It corrupts the dictionary after >in so
     it is overflowing the "term" buffer. accept.zero seems
     ok.

   * the following corrupts the dict when entered interactively
     : xx begin key? if ekey u. space u. fi again ;
    the error is: " : missing @<name> "
    the dictionary is corrupted before the word ">in" (doesnt
    list with "ls" )

   * If I make searchorder 2, forth.d, vlist.d
     Then "order" doesnt work properly but if its the other
     way round then it is ok.

   * 2 "also" words crash the system, but one is ok!
      also Asm
      also Edit
     If we put these in different blocks, its also ok...

   * "word." doesnt work well for last word in the dictionary
     (tries to decompile code from the name, not from xt). But
     seems to work for all other words.
   * buff count ' codepage wfind
      returns 0 because ' is immediate and executes before
      "buff count". (buff needs to contain the name of a word).
      One solution is to make ' the same as ['], which creates
      an litw opcode.

   * I can do 49 load in the repl but not in "thru" or in
     block 33. This is odd.
   * "wsize" on the last word in the dictionary returns
      a very very big number.
   * ." " and s" " cause a meltdown because the leading space
     is skipped with in++ then parse ignores the next " and
     tried to find the next " with not good results. A solution
     is to make parse stop if the first char is ". This may be
     standard "parse" behaviour.

   * in the snippet below the defined words .green etc
     have to be in immediate brackets.
     : colour create c, does> @ fg ; imm
     [ 2 colour .green 10 .lgreen 4 colour .red 6 .brown
       13 .pink 14 .yellow ]

   * "clear" seems to underflow the stack to 4098
   * The following doesnt work because 128 is converted
     into a signed integer. It actually subtracts 128 from dp @ !!!

       db FCALL
       dw dp.p
       db FETCH, LIT, 128, PLUS

   * I think having a signed c@ etc is a bug. We need
     an extend sign opcode for relative byte jumps
     but not automatic signing of c@

   * Bug!
    The phrase "10 10 pad pad 4 + showbuffer" is causing a strange bug...
    But the word "..sb" works ok! So interactively showbuffer doesnt
    work but it does within another word. An unusual bug, and its
    too late tonight to work it out...

   * The old dp here problem arises with the def below
    : Defer create reset [op:] exit 0 , 0 c, ; imm
    If we do "defer t1 defer t2" then t2 is compiled
    right after the count field of t1 ! I dont think the
    "does>" trick will work here.
    But adding a "code>here" cludge does work. Eg
     : Defer create reset [op:] exit 0 , 0 c, code>here ; imm
    The problem can be seen by using the first "Defer" (with
    no "code>here") and inspecting dp
    For example: Defer t1 dp @

   * The following definition is causing a spectacular meltdown.
     The "3 allot" is causing the problem, but I dont know why.
       : Defer create [op:] exit 3 allot ; imm

   * nop should be "noop" to be more standardish

   * circle and fcircle have a zero bug at the moment.

   * "deci 256 hex U." prints 256. This is because the interpreter
     converts the string 256 to a number at compile time which is
     before "hex" has executed... Then a litw is compiled but
     it is not affected by hex. So we could make hex immediate?
     Or have a "state smart" hex?

   * doing "post >r .s" in the interactive interpreter
     crashes the system.

   * semicolons with no preceding : def will crash the system. Eg
      nop ;

   * The code below causes problems. It executes correctly,
     header is created and the data "abc" is written to
     the dictionary, but the ">code" variable
     is not updated properly

     [ create test char a c, char b c, char c c,  ]

     We can fix the problem with:

     [ create how char a c, char b c, char z c, code>here ]
     or with
     [ create how char a c, char b c, char z c, sync ]
     But or some reason, if I put sync, or code>here in c, then
```

```
      the system crashes.

    * need to do
      [ create table 1 , 2 , 3 , 4 , ]
      in the interactive interpreter to make a list of values

    * If you write -1 instead of 1- there will be no error
      but the word will not do what you want. 1- is an opcode
      the decrements the top stack item. -1 pushes -1 onto
      the stack. This is an easy mistake to make.

    * doing do/loop in the interpreter is crashing with
      showglyph and "ms" etc. but seems to work in source code

    * running on asus e402M the cursor flashes even when
      I try to hide it. Need to set bit 5 ? Also, green cursor
      leaving green lines on screen.

    * watch for any defining word that is not immediate! eg
      var, con, 2con etc.

    * watch for nested comments ( ( ... ) ) !

    * see the xyset: code to see how to create a data array
      in this dodgy and buggy forth (need a superfluous "does>")

    * to define a new word in a source block, eg "xyset: test"
      we have to put that phrase after all other definitions in
      the block, otherwise it causes and error. This comes back
      to the old dictionary/anon switch bug.

    * [ create data 0 , 1 , 2 , ]
      this works in the interactive interpreter but not in a source
      block. In source we have to do
      : data [ 0 , 1 , 2 , ] ;
      which is odd. Also, in standard forth you dont need [...] around
      the words.

    * When a defining word uses create but is not immediate then
      it crashes the whole system, which seems a little overly dramatic.

    * "see" doesnt display defined words very well. This is because
      see cant, at the moment, display data fields (strings etc) within
      words.

    * how to return the parameter (data field) for a defined word?

    * c@ and c@+ convert bytes to signed words, but do
      we really need signed bytes?

    * ss/ only prints one match per block

    * Extra semi-colons give strange errors, like ": ?? within"
      even though the extra semi-colon was well before the
      "within" word.

    * the "ss/" is missing some searches because it seems
      to add a space at the end of the search word.

    * "thru" or one of its words is eating 1 stack item.

    * when running on x86, not qemu, some video modes dont
      clear text on the screen after backspace space characters
      are emitted.

    * "ii" doesnt work within a loop if there is anything
      extra on the return stack (put there with ">r" for example
      within the loop). This may just be a feature of forth
      since there is no way for ii to know what is on the
      return stack

    * ./ not working interactively well. That is
      : ./ 10 parse drop drop ; imm
      : ./ 10 parse type ; imm
      truncates one character.

      This is because the input buffer has no newline ending (just a counted
      string). And when "parse" doesnt find a character it is truncating one
      character. One work-around is to put an extra space at the end of the
      line.

      Changed the behaviour of parse so that it returns the rest of the input
      stream when it doesnt find the character.

   see mike gonta: "all you wanted to know about usb booting but
     were afraid to ask" for important disk geometry info, fat12 info
     and read/write usb memory. complete source example

MEMORY MAP

    The way that the code and data is laid out in memory is
    important. The bootloading segment (512 bytes) loads more
    code into memory (a few K) and then jumps to the entry point.
    When the code dictionary grows (by the use of new colon :
    definitions) the new dictionary entries should be placed in
    memory after the end of the dictionary.

CHALLENGES

    How to write new code words back to disk? We can either write
    compiled code to disk as part of the dictionary, or just write
    source code to disk in forth-style "blocks". This is potentially
    dangerous, if we accidentally write to the computer hard disk
    we may corrupt the file system or even the operating system!

    Some kind of fat12 file system would be convenient, so that
    I could edit source code on some other operating system. Or at least
    copy it.

    Harvard architecture is not going to be easy.

HOW TO BUILD AND RUN THIS CODE

    We can either run the code in a virtual machine like qemu
    or else actually write it to a usb or cd and boot it!
    The simulator is good for testing, but the usb or cd boot
    shows you how it works on real hardware!

tools:
     nasm, qemu, dd, bash shell (or any other shell for compiling etc)
     linux (makes things easier), sed, date, vim,
     asciidoctor, enscript - for formatting and printing source code

* create a printable pdf of the code in 2 columns landscape format
      enscript -o - -2r -f Courier7 os.asm | ps2pdf - os.pdf

* compile with nasm into a bootable executable
      nasm -fbin -o os.bin os.asm;
* make a 1.4Meg floppy image and insert the executable into it.
      sudo rm os.flp; sudo mkdosfs -C os.flp 1440;
      sudo dd status=noxfer conv=notrunc if=os.bin of=os.flp’
* run the executable floppy image with qemu simulator (VM)
      sudo qemu-system-i386 -fda os.flp’

To "burn" to usb or cd try:
* use dmesg to see what your usb flash drive is called
* unmount the usb flash drive
     umount /dev/sdc
   * write the bootable floppy image to flash drive: WARNING!
      sudo dd if=os.flp of=/dev/sdc
   !! The code above deletes all other files on the memory stick
      Be very very careful that you dont do this to your hard-disk
      or you will end up with an unbootable computer. !!

   The light should flash on the usb stick indicating that data
   is being written. Then just reboot the computer and choose the
   boot device. I think I had to "enable csm" or "choose floppy" mode
   in my bios to get the usb stick to boot.

   Some bash aliases for compiling and running the code.

     # make qemu open maximised but not full screen
     alias os='sudo qemu-system-i386 -fda os.flp & sleep 1; wmctrl -r QEMU -b add,m
aximized_vert,maximized_horz'

    Below is a bash function which includes preprocessing of forth
    source code in %if 0; %endif; blocks. The preprocessing code is in
    os.sed. This is convenient because
    NASM doesnt seem to have any kind of multiline DB or DW syntax
    which means for defining forth source code in a nasm file we have to
    put "db ' " around every line. The sed line deletes the marker lines
    (lines ending in "code{" and "}code" )
    as well as comment and empty lines and put the "db '" syntax
    around each line of text.

     # run the forth os
     os() {
        if [ -z "$1" ]; then
          name="os"
        else
          name=$1;
        fi
        echo " running ${name}.flp "
        # get rid of other qemu windows for clarity
        sudo pkill qemu
        sudo qemu-system-i386 -fda ${name}.flp & sleep 1;
        wmctrl -r QEMU -b add,maximized_vert,maximized_horz
     }
     # compile the forth style byte code system with some sed preprocessing
     # of the nasm file (to allow multiline text source code)
     ccos() {
        if [ -z "$1" ]; then
          name="os"
        else
          name=$1;
        fi

        echo "now compiling ${name}.asm "
        echo $(date +%d%b%Y-%I%P)
        sed -f os.sed ${name}.asm > ${name}.pre.asm
        # The line below adds the time and date to the machine name
        sed -i "/\.sig-machine\.name/s/' *$/.$(date +%d%b%Y-%I%P)'/" ${name}.pre.as
m
        nasm -fbin -o ${name}.bin ${name}.pre.asm;
        sudo rm ${name}.flp;
        sudo mkdosfs -C ${name}.flp 1440;
        sudo dd status=noxfer conv=notrunc if=${name}.bin of=${name}.flp
     }
     cos() {
        cat os.asm | sed -f os.sed > os.pre.asm
        nasm -fbin -o os.bin os.pre.asm;
        sudo rm os.flp;
        sudo mkdosfs -C os.flp 1440;
        sudo dd status=noxfer conv=notrunc if=os.bin of=os.flp
     }
     # compile and run the forth system.
     alias oss="cos;os"

 * The sed script used to preprocess the nasm source file

 #!/bin/sed
 /code{ *$/,/}code *$/ {
   # ignore lines starting with "times" in code blocks
   # 't' jumps to end of script.
   s/^ *times/times/;t;
   /^ *$/d;
   /^ *;/d;
   /code{/d;
   /}code/d;
   # need to 'escape' double quotes
   s/"/",'"',"/g;
   s/^ */db " /;
   # put newline on the end of each line
   #s/ *$/ "/
   #dos line endings
   #s/ *$/ ",13,10/
   # unix new lines are better
   s/ *$/ ",10/
 }

HISTORY

 27 march 2021
   working on the : ed block editor. screen scrolling
   etc.

 19 march 2021

   reexamining this after some years, some things are apparent:
   using 'here' to switch between the dictionary and anon buffer
   is a bad idea. A 16 bit forth is limiting, 32 is better. Eg
   trying to find the distance between two points on screen might involve
   doing 300 dup * which will overflow a 16 bit forth.
   Also: it is hard to avoid cryptic stack juggling for some words.
   Sometimes factoring doesnt help. But there is a lot of good and
   interesting code here.

 4 april 2019
   Improved the block editor "ed". It can navigate with the
   arrow keys, pageup pagedown etc.

   Made a basic character insert. Made cr/nl insert work (change 13 to
   10), also delete and backspace. Made insert stop when buffer full. I
   can add a "mode" variable like vim, and display information when in
   command mode (like editor state). The command mode will only affect
   the "normal" keys, not special keys.

 3 april 2019

   Added some functions to the block editor, such as pagedown
   pageup. linedown/lineup. Some debugging.

 1 april 2019

   Some work on : ed the new block editor.

 29 mar 2019

   moving towards a more traditional style of forth programming
   which is much more highly "factored" than other languages.
   So words are short. This seems to increase readability
   if one doesnt use local variables.

 28 mar 2019

   Will rewrite a factored block editor.
```

```
    Slightly improved "asc" by printing asci values every
    8 characters. Made a singlebox and doublebox word, to use
    the "box" word.

    Wrote what I think is a more readable version of the ascibox
    word. Factored out top/sides/bottom.

    rethinking "ed". A block editor does not need to have
    a count byte/word.

  24 mar 2019

    starting to write "mtype" and "ed" to edit and display
    multiline text.

  21 mar 2019

    Wrote boxtype and .b" which prints a line of text in an
    asci box. Near the bottom of the screen the bottom of the
    box is not printed properly.

    Vlist exists now, but is not yet working as a list of
    wordlists.

    wrote >s/o which is the action performed by vocabs.
    Put >s/o in bytecode so that Forth and Vlist can use it.
    >s/o is analogous to >r. If there are < 3 wordlists,
    >s/o will append to the end of the s/o buffer. Other
    wise it replaces the last (top) item. This is so that
    Forth and Vlist will always be the minimum wordlists.

    I want the "Vlist" vocab (which is a list of vocabs) to
    always be available because I want universal names to be
    important and useful in this forth. The idea of this is
    to allow the sharing of code between different developers
    and even between different forths.

  20 mar 2019

    wrote utype and .u" which prints underlined text. Could
    easily adapt to print boxed text (asci box chars).

    If "does>" just calls a word, then that word should be
    coded straight after the data field (instead of having
    a double FCALL).

    Had the idea to animate an object along a track, eg a
    car, or several cars and make them alway avoid each other.

  19 mar 2019

    Made bytecode words (not opcodes) colour green in .xt

    Speed up wfind by copying exactly the bytecode of
    "ffind". They now execute at the same speed. Saved
    the old slow version of wfind as wfind.slow
    Named shortjump if as IF/FI

  18 mar 2019

    Booting on asus e402m in real mode: t/wfind=110ms and
    t/ffind=55ms. These values are only approximate because
    the clock only ticks 17 times per second.

    Trying to use shortjump Until to get a bit of extra speed,
    but its not working.

    Realised that the opcodes "0=, jumpz xx" can be optimised to
    "jumpnz xx". Just wrote Until0 which does "jumpnz"

    * times with "buff" containing "neg", on qemu emulator.

    Originally at 300 iterations, t/wfind took about 1155 ms
     and t/ffind takes 440 ms (on qemu, with nothing in "buff" buffer)
    removing calls to xt+ and xt>name, didnt seem to make any
    difference. Inlining s= also doesnt make much difference.

     recoding 2drop as an opcode, reduces t/wfind to 990 ms
     Making 0= an opcode reduces t/wfind to 935 ms
     Using a short-jump if/fi (+/-128 bytes) reduces t/wfind to 880 ms
     recoding "rot" as an opcode reduces to 825 ms

    Made 2drop an opcode, because I think stack ops should
    be fast. Also, trying to speed up wfind, which is twice
    as slow as ffind at the moment. Wrote "time" to time
    the operation of a word.

  16 mar 2019

    Started to write a foreach pointer iterator loop, so as
    not to have to use ii (loop parameter) in loops.

    r@ and 2r@ and 2r> 2>r should be opcodes, since that should
    speed up looping. In a stack machine, the stack operations
    should be fast.

    doing ’ find.so is nfind  noticeably slows down compiling.
    Maybe "ii s/o dup @ 2* + " etc is slowing down find.so
    It would be nice to have "for.each/next" which iterates
    a pointer over an array, with the pointer stored on the
    return stack and decremented by "next". One version for
    characters another for 2 byte cells, etc.

  15 mar 2019
    Adding words to empty wordlists just seems to work.

    Made nfind a deferred word. can now do ’ find.so is nfind
    but first we need to load vocabs for it to work.

    Wrote find.s/o which seems to be working, searches for
    a word in any list in the search-order buffer s/o.
    Now just make it deferred, and assign it, so that tick.p will
    look through the whole s/o buffer.

    Simplified looping through the s/o buffer using for/next and
    ii. This can be applied to "order" as well.

    How to make find work for the whole search order
    Modify "wfind" to be ( A n xt -- xt’ ) search from
    string at A length n, starting from xt. Then write
    a word "afind" which loops through the s/o and calls "wfind"
    Could make "nfind" a deferred word and sub in "afind" when
    vocabs are created, but this will degrade compiling performance
    since nfind is called when compiling every word...

  10 mar 2019
    Worked a little on vocabs. I will define Forth and
    Vlist in bytecode so that they are available right
    from the start.

    wrote xt? to check if an address is an execution
    token. But needs to search all wordlists.

  9 mar 2019

    wrote Un, which decompiles n instructions, and displays
    them paged, in colour. Also configured "see" and "word."
```

```
   to use the new decompiler. Still need to decompile strings
   eg litw:15429
      litw:9
      jump:11
      ... string data ...

 8 mar 2019

   wrote rclear to clear the return stack and put it
   in "abort" which worked.
   Changed bytecode de, to pause after 18 lines, and
   to display, roughly, the number of bytes in the word
   being decompiled. But de, still shows too many bytes.
   Put word. and head. in source code.

   Made in++ to skip one character in the input stream. Made
   s" and ." ignore the leading space (using in++). But
   this makes ." " an error ! which will crash the system

   Made a switch seeanon to show the anonymous buffer in
   "shell". Wrote c> to print a char

 7 mar 2019

   wrote .r and a .rcall that nearly works. Just need a
   xt? ( A -- F ) word to check if an address is an xt or
   not. Need to handle pcalls!! on rstack!

   How to create colours
   : colour create c, does> @ fg ;
   7 colour .white

 6 mar 2019

   Trying to reduce the number of source blocks until
   "shell" can work.

   System size:
      Opcodes: 1610 bytes ( 69 opcodes )
   Byte codes: 2845 bytes
   Total size: 21084 bytes ( 456 total words )

   cmforth introduced the idea of a separate word list for
   compiling words, to avoid the need for the immediate/compilation
   state switch. But I would like to avoid "immediate" altogether
   (because real machines dont really "interpret", as far as I
   can see). So this cmforth idea (in pygmy too) doesnt appeal.

   Wrote elementary abort and abort" but we need to separate "shell" from
   resetting the stacks etc.  So no startup message when "abort"
   executes...

   wrote opcode "rpick" which picks element n from
   the return stack. I cant think how else to write ".rs"
   to display the return stack. The alternative
   is to write "n>r" and "nr>" and "ndup" but "n>r" and "nr>"
   can only be tested together (they cancel each other out).

   Wrote a rudimentary stack underflow check using sp<=4096
   although "clear" seems to make it go to 4098
   Also wrote an rstack underflow check, but this seems less
   useful since the return stack always has stuff on it
   (the call to "shell" etc)

 4 mar 2019

   Changed the order of the loop parameters for do/loop on the return
   stack. The counter is on top of the return stack. This makes it
   compatible with "next" and rloop and allows "ii" to work for both.

 3 mar 2019

  Wrote helper functions for loop/Loop/next called (check) and (clean).
  These compress the source code but will make the loops run slightly
  slower. The original compiling loops are in block 63 Added an extra
  parameter (initial value) to for/next so that ii 'ii and "unloop" should
  all work with for/next.

  Wrote loop in source (with a long jump) and also Loop which
  uses a short jump. Wrote a "for/next" loop. If I put the
  loop parameters on the return-stack in the opposite order,
  then I could use the same "ii" word for "for/next". Then I
  could use for/next for thru and get loop out of bytecode...
  etc.

 1 mar 2019

  Made "ls" loop through all vocabs in search order.
  But dogged by the +/-128 byte loop bug (need to convert
  loop to ljump etc). ls is looping in reverse order.

 28 feb 2019

  Still looking at nfind.p and how to loop through each
  vocab in the s/o search order buffer. Once we can do that,
  then vocabs should start to work.

  put pad in source. Also changed its def to use space
  just after the last dictionary entry.

 27 feb 2019

  Put s/o and "context" into bytecode. "ls" uses
  context @ @. But nfind needs to iterate through
  all wordlists in s/o not just context. I still like the
  idea of having a wordlist wordlist which, with "forth"
  will always be available. So "vocab" will do something like:
    : vocab also Vocabs defs create 0 , prev
        does> etc... ; imm
  That is, it will activate the Vocabs wordlist and add the
  new vocab there. Then it should restore the previous "current"
  wordlist.

 26 feb 2019

  Made steps towards using "context" for ls and nfind
  but they use the bytecode context, and not the s/o context.
  We may have to make a deferred word context, or else
  make "context" a pointer to a variable. Yes a deferred word.

  Made an error handler word when a word is not found
  in the dictionary. Also, made this a "deferred" word.
  That is, a word that can be revectored later.
  This is important because it would be nice to look up
  missing words in source code or on the net and compile
  them. This is the start of a "dependency compiler"

  Changed last to use "current" vocab. Which works for
  creating new words. But "nfind" is still only searching
  with "last", so only searches the current wordlist. nfind
  needs to search all wordlists in the search order.
  And "ls" needs to do the same.

  Renamed >code to "dp" dictionary pointer.
  renamed /2 to 2/ which is more standardish.

 25 feb 2019
```

```
  Looking at openfirmware. Also uses anonymous colon definitions

 24 feb 2019

  Changed base to be a 16bit cell, because that is what all
  other forths do.

  Progress to make: fix "last" so that is does "current @"
  Fix prev, only, also, so that forth is always in the search order.
  Make optable a word in the system so we can add new opcodes.
  Make "ops" etc scan through the opcode table (0 terminated, or
  with length at opcode 0).

  Look at Pygmy forths x86 assembler, which is reputed to be
  well implemented

 21 feb 2019

  Ideas: To implement D+ well, we need "add with carry" and "add with
  overflow" etc like all machines actually have. But we can
  just implement it by using the FLAGS opcode and examining
  the individual ov/ca flags. But implementing "flags" on the
  machine is tricky, because every opcode needs to update the
  flags properly.

  Recoded "defer/is" to actually compile an opcode/fcall into
  the buffer. This allows opcodes to work with defer, which
  is handy. So we can do:
     Defer steep ’ swap is steep

  wrote a "reset" word which sets dp and here to point
  just after the wordname count field. So colon can be defined
  like " : : create reset ; imm " Although that is a circular
  definition. Or try this...
    [ create : ] create reset ;
  I found this necessary while trying to write a Defer word
  which is just a buffer to contain code...

 20 feb 2019

  Realised that dont need "zerolink" word, because create gets the
  backlink from last/latest. When a vocab is empty it will have a zero
  backlink anyway.  We will just make latest do "current @". And when
  create sets last, we will set current instead. So current will have to
  go into bytecode, and it will point to a "dummy" vocab (the core words)
  until we define real vocabs in source code.

 19 feb 2019

  Another interesting word would display the search order
  of wordlists and the name of the last word in each.

  Trying to respect the forth standards and traditions more
  by renaming my non-standard words to something different
  but similar to the Ans forth word.

  Made alot of progress implementing standardish wordlists.
  Eg: words.. order, also, defs, context, current, vocab etc
  Now I need to change the way find gets its search start.

  Another idea, just use blank lines in blocks to index
  end/start of words. Can still use an indexing word like
  ## , maybe at the end of the block. If at end, then
  have a reverse count (length) at end of block

  moving away from the idea of having a "wordlist" wordlist.
  That is a list of wordlists. So, all the "vocabs"
  such as forth, editor, assembler are words in the
  core dictionary. But this means there is no simple way to
  display all existing wordlists.

 18 feb 2019

  Renamed l.addu etc to li/addu which seems more forthish,
  but wordlists may obviate such things.

  Started to write a set of words to implement wordlists
  (which I like to call namespaces or vocabularies ).

  Copied useful words vector and cvector from the Ron Geere
  book. Dont know if they will work. Good test for interactive
  and block compiling. Seem to be working when compiled from
  source.

  Wrote nrect, which makes a rectangle with line of
  thickness n. todo... ncircle etc

  try to display coloured circles in a grid (like an
  iphone calculator.

  should write "where" word which shows in which blocks a
  word is defined. eg where Var will list block numbers, and free space
  in block.

 17 feb 2019

  Wrote "ccpoints". A pseudo circle with midpoint (200,200) radius
  80 can be drawn with
    "200 200 80 35 ccpoints 35 ctrace"

  Wrote circle and fcircle but they seem quite slow. Also
  changed "pong" to use fcircle to show the ball.

  Wrote "circle" with x^2+y^2=r^2

  Tried to write um/mod but it doesnt seem to be working
  Just behaving like u/mod

  um/mod ( ud u1 -- r q )
    This can be used as the basis for double precision math
    see "math.txt" at flashforth. If quotient is too big for
    16bits then its an error

  From math.txt
   ( n1 n2 -- d)
  : m* 2dup xor >r abs swap abs um* r> ?dnegate

16 feb 2019

  wrote um* as an opcode. To write D. and UD. might
  need uD/ and D/  double division. Fixed u. bug caused by
  change in /mod (signed). Looking at pforth source. Should change
  JUMPS to branches? and pix to plot. Half wrote u. in source.
  This might be useful for D. Also it is probably better to write
  words like <# # #> for number conversion and use a temporary
  string rather than just outputting the numbers directly.

14 feb 2019

  A solution to block limitations... have an index word
  at the top of each block. eg ## with binary indexes following
  indicating the start of each word in the block.
  Also, at the end of the block, a continuation word such as
  ... which indicates that the word continues in the next
  block.

12 feb 2019
```

```
    todo
      write a : circle word using y=R*sin(x) and then
      or just use x2+y2=r2
      print 4 points for each x (ie x,y x,-y -x,y -x,-y)
      Adapt ron geeres trig functions.
      write : ccpoints ( n a b -- / which returns n points on
      the circumference of a circle on the data stack. Then
      use those points to draw lots of radiating lines.

    Wrote "lineto", sspoints, and polar to give polar co-ordinates
    Turtle graphics are pretty close.

     A new solution: dont have an "anon" compile buffer. Just compile
     non-defining words straight after the last word in the dictionary,
     keeping a marker of the end of the last word. This has the advantage
     of allowing anonymous compilations of any length (not limited by
     the "anon" buffer.

     A problem: "defer test ’ nop is test " is going to fail
     because "nop" is an opcode. So defer will only work with procedures
     (words). This is part of a bigger problem... that opcodes and
     procedures always need to be treated differently.

     One solution might be to include, eg: NOP, EXIT just after the
     opcode header for "nop" and just before the machine code for nop.
     This would be the execution token for nop and would allow the
     opcode to be called with FCALL, nop.p . It would also allow words
     like "is" to work just as in standard forths.

     Also PCALL would just work with opcodes as well and we could
     execute all opcodes in "immediate" mode without having to compile
     anything. (Currently they are being compiled to obuff.d with
     and EXIT and then PCALLed)

     The disadvantage is and extra 2 bytes per opcode.


   10 feb 2019

     Working on n degree rotation matrices using scaled trig
     values (sin*10K, cos*10K). Had the idea for tab complete
     for long words. Should be able to use the s/ word to
     implement tab complete.

   9 feb 2019

     Wrote the scale */ operator ( n1 n2 n3 -- n1*n2/n3) which
     seems to be working for signed numbers. But still hangs
     for eg:  1234 1234 23 */

     Apparently fixed the signed /mod problem by using "idiv"
     signed division, with the cdw to extend sign from ax to dx:ax
     (which is the register pair used by idiv as dividend). But should
     I used cdw with u/mod as well?

   8 feb 2019

     Made a key? opcode using int16h ah=1. Made a sign wave graph
     but discovered that my /mod / mod dont work with negative numbers!!
     which is a big handicap. Need to look at "idiv" instruction.

     Wrestling once more with the code>here bug.
     [ create aword 1 , 2 , 3 , ]
     fails in source and interactively because the dictionary
     code point ">code" is not updated by , or c,

     [ create aword 1 , 2 , 3 , code>here ]
     Appears to fix the problem but is very inelegant.
     Also wrote a word "sync" in bytecode that sets dp=here if
     dp < here

     All these problems could be avoided by compiling everything
     into the dictionary (even anonymous definitions). But that means
     that after every new word definition we would have to build
     an anonymous header which would slow down source compilation
     quite a lot.

   6 feb 2019

     Current size of the system is:
       Opcodes: 1493 bytes ( 63 opcodes)
       Byte codes: 2920 bytes ( 74 bytecode words )
       Total Size: 15550 bytes ( 328 total words )
     The bytecode space has reduced slightly as we move words
     into source code (do/loop etc)

     Would like to put "loop" into source, but I dont have enough
     room in block 0 ( used by thru also there). Also need to
     change jumpz in loop to jumpnz/ljump.

     Wrote [call:] <name> which compiles a call to the procedure
     at runtime. This is also like a double postpone word.

     Working on an archane word [op:] <opcode> which compiles
     the named opcode at runtime, I think. This word is useful
     for writing words like if/do/loop/again etc. But there may
     be a simpler way. For example, write a (loop) word that
     performs the runtime behaviour of "loop". This avoids having
     to compile code when loop is compiled. This is hard to think
     about because the compile time for one word is the runtime
     for another.

   5 feb 2019

     Maybe "colour pong" would be a good challenge now, for this
     forth. And then try to compile it to native code.

     Would like to make a little "glyphmaker" word to create
     xysets which can be displayed as glyphs (either with asci blocks
     or with pixels). When displaying with asci blocks the aspect
     ration is >50% because asci glyphs are 8x16 not 8x8

     Made rotations of glyphs work using rot90setn which rotates
     90 degress about point n (0,1,2...) of an xyset (point set).
     A small animation "arot" demonstrates. Because the coordinate
     system of the screen is  +down, -up the rotation actually appears
     to be happening counter-clockwise, but the matrix rotation is
     actually positive.

     A useful paradigm for paging and exiting out of a begin/again or
     begin/until loop if "q" is pressed...

     ( nn counts words, ll counts lines typed )
     0 nn ! 0 ll !
     begin
       ( print 8 words to a line )
       nn ++ 8 mod 0 = if
         cr
         ( page and prompt every 22 lines )
         ll ++ 22 mod 0 = if
           1 fg [char] > emit key
           ( quit if ’q’ pressed )
           [char] q = if exit fi
           0 bg cr
         fi
       fi
     again
```

```
  4 feb 2019

    Could write word u.s to display the stack as unsigned values.

    Might be useful to have a postpost word for situations as
    below. Made asci box displayer "abox"
    Wrote rainbow for colourful text.
    : rb" ( print rainbow coloured text )
      post s" ’ rainbow literal post call, ; imm

    Started to write some matrix rotation code.
    Wrote a simple "ms" based on a loop. Could do better
    Thinking about asci tetris, xysets, xyglyphs, matrix rotation
    glyphmaker etc. Interesting maths in all this and seems quite
    easy to implement in forth.

  2 feb 2019

    Started to write "edit" based on accept.arrow. With some
    success. However the stack params should be changed
    to include a text length, because not all buffers are
    counted... The length could be stored in a variable.
    Also, showbuffer can be eliminated.
    So, need to change various words.. eg prevc, nextc, ctype


    Looking at the wordlist words such as context, current,
    vocab, also, only, etc. I may modify some of these words
    because I want the idea of namespaces to be quite strict.

    Started to move to unix line endings \n=asci 10
    not dos endings. Ignoring asci 13=\r

  1 feb 2019

    Now get rid of accept.byte in bytecode.

    Accept.hist is useful because it has some command history.
    But it needs a lot more history. Also, it would be nice
    to be able to write this history to disk. Thus we would
    have a system that can be developed on itself.

    Also, need an "editor" which is like accept.hist but displays
    multiline buffers and can navigate with all arrow keys. Same
    strategy as with accept.hist (use showbuffer/ctype after every
    keystroke to show the buffer) and same stack structure
    ( x y A A+i ) .

    Made accept.hist save the buffer after /enter/ is pressed.
    Rewrote if0 and ifnot to reflect new longjump if version.
    Made "defer" and "is". These words allow
    installing new versions of "accept" into the shell for example.
    We can use is like this: ’ accept.hist is accept
    This makes the shell (which is already compiled to use the "accept"
    deferred word) use "accept.hist".

  31 jan 2019

    Rewrote if/else/fi using jumpz/ljump combination which should
    allow long text in if clauses.

    Acceptx is almost working, only bugged by the if/fi +/- 128 byte
    bug. But it is working with command history (2 buffers)
    Removed fi.p bytecode and replaced with source.

    The machine needs a sign extend instruction, eg signed extend
    byte on stack to word.

    Also, should c@ and c@+ convert bytes to signed words (16 bits)?
    What is the utility of this? Bytes are used as relative +/- jump
    addresses, but not on the stack...

    Trying to improve dump, but should dump display signed or
    unsigned hex bytes, or should it display words? Wrote xytype
    and xyemit.

  30 jan 2019

    Made PIX display pixel with the forground colour. Made
    CURSOR opcode which sets the shape, size of the opcode.
    This is useful for making the text cursor disappear which
    is often desirable.

    New ideas for anonymous words: What is now the anonymous
    buffer will become part of the dictionary. When create
    executes it will check if anon exists and overwrite it, if so.
    When does> or semi-colon executes, it will re-establish anon

  29 jan 2019

    Made "acceptx" work, except for the distracting blinking cursor.
    Also, it only really works in text mode 3 because as far as
    I can see, that is the only mode that displays background colours.
    In other modes we can use different cursor (eg forground colour)
    by modifying the "ctype" word.

    wrote "prevc" and "nextc" to advance and regress the insert (cursor)
    position.

  27 jan 2019

    wrote delchar and insert. Thought about what happens when
    "move" tries to move 0 chars. It seems to work.

  26 jan 2019

    Modified showbuffer to use green cursor. Trying to change
    acceptx to do a proper backspace. need to debug.

    Made background colours work in video mode 3 by modifying emit
    to use ah=0x09 int 0x10 function. Modified "ls" to print 8 words
    to a line.

    Strange things with qemu and video modes and background colours...
    In video mode 3, function ah=0x09 prints forground and background
    colours, but in mode 16, it only prints forground colours.

    Also, using ah=0x09 int 0x10 then everything gets printed on one
    line (if there are no explicit line breaks) but with ah=0x0E int 0x10
    then text is displayed on successive lines of the screen.
    In video mode 3, ah=0x09 cannot print any colours.

    In conclusion: to get background colours, we need to use function
    ah=0x09 int 0x10 with video mode 3, or similar.

    It doesnt seem possible to get background colours with
    function 0x0E int10 but it is with Ox09. But 0x09 is a
    bit more work because we have to advance the cursor.
    Also 0x09 print 13,10 as visible characters (not carriage
    return)

  25 jan 2019

    wrote cmove which is just "move" but can handle overlapping
    memory areas where A > B and A < B. Appears to be working.

  24 jan 2019
  Now ready to move on to acceptx using x y A A+i and "showbuffer"

  If words are immediate and should not be, strange errors occur.
  An vice versa.

  wrote "append" which appends characters to a counted string.
  rewrote "all" to load from last loaded block to n.
  wrote "defblock" which puts on the stack block number of
  the first def of the word. wrote "lo/" which tries
  to load all blocks up to a particular word. This is a kind
  of ersatz dependency loader.

  We can create a string constant in the dictionary with
   : message s" hello" ;
  but this contains an unnecessary jump opcode at the
  start of the definition.

22 jan 2019

  The conversion of acceptx to showbuffer should be quite
  straight-forward once showbuffer is working well.

  Bug!
  The phrase "10 10 pad pad 4 + showbuffer" is causing a strange bug...
  But the word "..sb" works ok! So interactively showbuffer doesnt
  work but it does within another word. An unusual bug, and its
  too late tonight to work it out...

  Working on "showbuffer" word to display a buffer at x,y on
  the screen with a cursor. Will change acceptx to use it.

  Realised that editing with arrow keys requires a different
  approach in "acceptx". Need to reposition cursor and display
  buffer on each key stroke. But will there be flicker?

  Fixed a getxy bug which made getxy eat top stack item.

  It would be good to have a word eg "showdef" that shows the source code
  for a word from the source blocks. Could just use a dodgy parse for next
  ';' to get the end of the word.

  Made "ss/" print the line with "typeline". But typeline could
  just use parse, no?

  Made "s/" and "ss/" work with the new (almost standard) "search"
  word.

  The idiom to print 3 things to line is
   20 0 do
     ./ print something
     ii 1+ 3 mod 0 = if cr fi
   loop

  Removed the bytecode "again" version.

  Apparently fixed ljump.x bug (si-3 not si-2 to realign) and
  wrote a source "again" using ljump which seems to work.

  Thoughts for words: write a "para" word which is like
  "accept" but accepts a whole paragraph, and hopefully
  allows key editing. Also extend acceptx to allow arrow
  key movement and insertion. These are steps towards writing
  an editor.


21 jan 2019

  Made a source "again" and a long jump version.
  The begin/again 128 byte bug continues to vex: The solution
  is to compile something like
    jumpz, 4; ljump -200,
  That is compile a short jump over the long jump. In fact
  we can check if the target is > +/- 128 bytes and then
  compile the short or long jump depending.

  Made "search" more standard. But ..search is freezing because
  of the above bug. Also need to integrate the new search in
  ss/

20 jan 2019

  Thoughts on the linked list dictionary. If other link
  fields are included then each word can belong to
  one or more "categories" not just namespaces. This increases
  the complexity of the dictionary but also the classificatory
  power. link field format: n:link, m:link0 etc where
  n,m are ordinal references to the category list.

  wrote "cmove" to copy data from source to destination with
  dest correct if areas overlap.

  Made wsize a bit more accurate. (mostly correct). Wrote
  "size" which makes wsize easier to use. Tried to change
  "search" to leave the found address but gave up for now.
  Copied some double number definitions from the Ron Geere
  book.

19 jan 2019

  Wrote a square, octagon drawing words, from a "line"
  word.

18 jan 2019

   words: "ns" provides a pointer to the namespace list.
   "name <name>" create a new namespace (pointer) in the
   namespace list.

17 jan 2019

   Thoughts on source structure: Keep the block structure because
   it is simple, but have an index which somehow give the location
   and lenght of each word, without naming each word...

   Almost fixed the behaviour of parse so that it returns rest of input
   when the char is not found. But one char is being truncated.

   Made a word ss/ which searches source blocks for a word.
   But whole line search is truncating one character. See BUGS

   Made substr which checks if a string is a substring of another.

16 jan 2019

   Need to write a "copy" or "cmove" (standard)
   word for copying strings/data etc. The word is called cmove
   because when the data areas overlap only the destination is
   correct. Otherwise it behaves just like a copy command.

   This will be used by acceptx to recall history With the
   arrow keys. Need to improve "dump" so that it shows asci characters
   too and put 8 or 16 on a line.

   coded "if0" based on the source version of "if". Worked on
   acceptx.
   Made an alias for if0 with the idiom ": ifnot post if0 ; imm"
   This seems a bit counter intuitive since if0 is itself an
```

```
       immediate word. But it appears to work

       When an error occurs, made "inputcompile" print the last word
       compiled in the dictionary. Handy for debugging.
       Also, add a "block number" to the block buffer, so we know
       what block is currently in the buffer, also handy for debugging.
       Also, add a "updated" flag in the block buffer, so that we
       know when the buffer needs to be written to disk.

    15 jan 2019

       Bug! begin/again and begin/until compile to relative jumps
       which have a limit of 128 bytes!! This is not enough!.

       Need a "break" word that will jump just after the next "again"
       or "until".

       Its not necessary to use unloop in a begin/again or begin/until loop.

       Wrote word 's/ <prefix>' which displays all words starting
       with the given prefix. Also wrote "prefix" and a test loop
       for it.

       fixed a bug, bfree was eating the stack, causing .s to crash
       after 'df'

    14 jan 2019

       Bug??! unbalanced >r r> will crash the shell because
       pcall anon will not return properly. There should be a
       way to "reset the stacks" (both data and return). Eg when
       an error occurs

       Also: make "anon" a namespace (so a "sub" linked-list) which
       will contain command history (like the bash history file)
       Allow an up arrow in "accept" (use ekey)

       refining ideas for namespaces: We can create a new word such as
         :: <namespace> <word>
       This will search for <namespace> in the namespace list. If not
       found it will create a new namespace variable in the list (with
       a null pointer since there are no words in this namespace, yet).
       Then it will create the <word> in the new namespace. Creating the
       new word involves creating a word with a "zero" backlink (since
       this is the top word in this dictionary) and then updating the
       namespace variable to point to this new word (since it is the
       last word in the list).
         ::var <namespace> <variable>
       Will work in a similar fashion

    11 jan 2019

       An interesting idea!!!: All anonymous commands should be compiled
       to the dictionary!! That is, words entered interactively should
       be compiled to the "anon" buffer, (as is done already), and then
       copied to the end of the dictionary with the name "anon0".
       The only major change is that the anon buffer should also have
       a back link to the last word in the dictionary.

       Also, new defined words are first created in the anon buffer
       and then copied to the dictionary. This may even resolve the whole
       here>anon and here>code problems.

       (Also anon2 is renamed anon3, anon1 is renamed anon2 and anon0 is
       renamed anon1... this allows for a simple "command history", at least 3
       previous commands. Also, this idea is a step towards "dependency
       compiling", which involves trying to compile a word even even if its
       dependencies (constituent words) are not in the dictionary.
```

```
       Code is compiling.

       There is a "creation" namespace: All new words created will
       be put into that namespace (linked list) until further notice.
       Then there is also the "search" namespace(s) which are an
       ordered list of namespaces to search for the given word.
       The first one found is executed/compiled. And finally there
       is a unique-name namespace which only applies to the word
       immediately following it.
         For example: "forth.math pi"
         will only search for and exec/compile the word "pi" in the
         forth.math namespace

       While fiddling with the code yesterday, about 17 words mysteriously
       disappeared from the loaded dictionary.

       The words source, load and inputcompile should return a
       flag indicating success or failure. This should allow error
       trapping and easier debugging.

       Thoughts: when loading blocks (or namespaces) a better
       message should be displayed (we could put that in "thru")
       Namespaces should be a namespace! This cryptic phrase means
       that a word which is a namespace (eg: forth.core) should
       be in the namespace "namespace". Now, the word "dup" for example
       is in the namespace "forth.core". Clear as mud. But by putting
       namespace words in their own namespace we can dealwith them
       differently. Maybe opcodes should also be in their own namespace
       but I havent thought that through.

    24 dec 2018

       Need to solve several problems before writing more forth
       code. Dependency problems. So a word should be able to look up
       obtain and compile its dependencies. This is not so hard: the
       process is, look in the forth dictionary, if the word is not there
       look on the disk, if the word is not there, look on the net
       using fullname.

       Namespaces will be handled by interleaving the lists within the
       same dictionary. Each word will only link back to the next word
       in its own namespace (or domain). So there will be several linked
       lists interleaved in the forth dictionary. At the top of the dict
       there will be a "start-link" table, associating namespaces with
       where to start searching in the dictionary.

       This system should be much faster for look up and compiling that
       having a domain name prefix field in every word.

       The other problem in being able to search the code on disk for
       a word without compiling all words. This may be handled with a
       simple character lenght field at the start of each word. This
       allows the coder to skip over that word if it is not the searched
       for word.

       Compiling a word with dependencies: Start to compile a word to
       a "compile buffer" (not to the dictionary). If the word compiles fine
       then copy to the dict, if not leave a space, search for the dependency
       word on disk/net and start compiling the dependency word to the
       compile buffer after the incomplete word. Then copy to the dict.
       This is recursive.

    24 august 2018

       A few more move vectors for chess pieces. Thought about
       words ,{{ } and {{ } which compile byte data to a given address.
       eg c,{{ 1 2 3 4 5 } or 1234 c{{ 1 2 3 4 }

    22 august 2018
```

```
      Created the chess vectors for knight and bishop on a 12x8 board
      See the arduino book for a sketch of how to code chess in this
      forth. Source code on disk needs to be stored in a structured fashion.
      Or maybe have shadow information about start and end of words.
   late july 2018
      Started a port of this machine to the atmega328p avr chip
      (the usual "arduino" chip). Then got distracted coding timer/counters
      in avr asm.

   19 july 2018
      A big challenge is going to be accomodating Harvard architecture
      microcontrollers within this machine. Where code and data space
      are separate. Flashforth may have some answers for this.

   17 july 2018

      Should use asciidoctor and asciidoc format to create a printable
      version of this source code.  Wrote a palindrome word, which uses
      recursion, and is possibly the simplest recursive descent parser...
      Did some work on the 'deps' word, still not recursive.  still grappling
      with code>here problems. hard to visualise.  The simplest way seems to
      be to update the dp point if code < here in the here! word. But this
      is not working.

        Machine: 4th.x86.topdx.17july2018-11pm
        Opcodes: 1410 bytes ( 62 opcodes)
        Byte codes: 3078 bytes ( 82 bytecode words )
        Total Size: 11335 bytes ( 250 total words )

   16 july 2018
      might be good to have a begin/until which has a counter
      with no limit. wrote "page" which prints text one screen
      at a time.
      wrote 'println' which is like type but only prints one line.
      Hopefully fixed a bug in li/0 which sets a list to zero
      probably will put code in c, and , to update dp if
      necessary.
   11 july 2018
      made some progress on : deps. fixed a bug in list.addu
      more 'create' bugs... The following line doesnt work
      [ create pawn 1 c, 2 c, 3 c, ]
      This is because the dp pointer is not updated after the c,
      words. So, 'here' is updated, I think I need to change the
      way 'here' and dp work.
   6 july 2018
      wrote 'within'. ppm image format easiest to display, or pgm as a glyph
      a random number series. an editable buffer,
      test.tonumber and test.type removed from bytecode
   4 july 2018
      also accept can be put in source.
      wrote a source 'accept' with a maximum number of chars. used
      a helper variable. wrote a 'continue' which jumps out of an
      if clause and back to begin. But will fail for 2 nested ifs
      tried to write an accept with arrow keys but got lost.
      simplest random seed using the clock, but need a series.
      eg var R : rnd clock drop 16 mod ; : r' R @ getnext R ! ;

   2 july 2018
      Would like to write 'accept' as source with some command
      history.
      cleaning up. removed old some bytecode words like 'seecomp'
      and 'one' which were used to debug the system. Left dotstack.p
      as bytecode because it could be useful to debug bytecode words.

   1 july 2018
      Moved [ and ] into source. wrote a new shell in source.
        opcodes: 1410 bytes ( 62 opcodes)
        bytecode: 3689 bytes ( 89 bytecode words )
        total: 9427 bytes ( 221 total words )

   30 june 2018
      wrote a source .S version of .s
      should add some command line history and write that history
      to blocks, so that work is not lost. write accept as source
      and allow arrow keys to edit. Multiline accept. Make an
      asci sperm whale

   29 june 2018

      asci animals: could do sperm whale, python.

   28 june 2018

      Writing a little chess: code. Also fixed a cbw sign extension
      problem for c@ c@+ probably caused in the transition to
      the top dx engine.

      Tidying up lib.p  block0 now loads all the other blocks.
      Made numbers and opcodes work in immediate state. The ops
      get compiled to a short buffer and executed immediately.

      working on the buffer editor ed: so as to try to make this
      forth self-supporting. That is, to be able to edit and develop
      the system with no OS. Need lots more 'line arithmetic' to make
      it work. Also, how to edit over rs323 connection. eg vi etc
        opcodes: 1404 bytes ( 62 opcodes)
        bytecode: 3852 bytes ( 212 words )
        total: 9059 bytes

   27 june 2018
      An x86 assembler can work like this
        code: doit ax inc, ax bx sub, ;code

      So the operand comes first and is pushed onto the stack, then
      the inc, word compiles the mnemonic with its operand. This raises
      the questions of wordlists and vocabs. ax is a word in the
      assembler word list.

      Recursive factorial, and arithmetic, calculate e exponent.
      Put domain fields in some words. The recursive factorial
      is very succint, like gcd.

   26 june 2018
      All this needs reconsideration. But the domain table idea
      seems sound at the moment. (See later notes about an
      interleaved dictionary for namespaces)

      This is claytons object orientation. Domain table is like this
      domain.table:
         db 1, 0, 3, '4th'
         db 2, 1, 1, 4, 'set'
         ; extra space here for new domains

      the 1st number is the domain symbolic constant, used in the domain
      field of the compiled word. The second is a count
      of how many prefix domains (none for 4th) then comes a list of prefix
      domains as symbolic constants, then the counted string name.  This is
      compact for compilation but allows all names in the dictionary (and
      outside of it) to be unique.  Claytons oo is like this

        * define the object and just return a reference to the obj
          at runtime
        : set create allot blah blah does> blah, not much ; imm :dom

        the word ':dom' compiles a new domain for the last compiled
        word. The new domain 'set' with possible prefix domains
        will be appended to the domain.table if there is sufficient
        room. So defining words should automatically set the domain
        of defined words to the same as themselves?

        * create an instance of the object
        20 set ss
        When set create 'ss' it probably should sets its domain
        automatically to 'set'?

        * create some words which operate on a ref to the instance eg
        : additem ( address -- ) blah blah ;
        * set the domain of the new word to set
        dom set. So additem has a full name of '4th.set.additem' but
        it could still be used by a different object.

        * use the new method on the instance
        ss m> additem
        'm>' can use the reference to ss (left on the stack by ss) to
        get its xt and therefore its domain. Then use that domain to
        do a special find only searching for that domain.

      m> could change the domain used by 'find' when searching
      for a word (eg 'additem') in the dictionary.

      So find will only find 'methods' of 'set' because they are the only
      words with that domain.  But normally these words/methods will be
      invisible to find because the domain will be '4th' or somesome.

      Write a word that prints fully qualified names using the
      domain field in the compiled word eg
         4th.set.additem etc

      Apart from using this as a claytons OO system, it is a useful
      step towards universal unique locatable names in source code.

      Thinking about domain fields and how to use them. Have got
      this schema so far: domain field is 1 byte in each word.
      the 'find' word has an implicit or explict domain during its
      search. Or a list of domains. So when we do a normal 'find'
      we wont find words with different domains unless we explicitly
      search for them. This may be parallel to forths 'wordlist'
      concept, same same but different. Also, we can take one step
      towards an object system by hiding object methods in their
      own domain.

   25 june 2018
      Working on the ed: edit buffer object. This seems promising.
      Once we can edit a buffer then the system becomes almost
      self sustaining. Started to put in domain fields. (revise to namespace)
      Wrote a list: data type that has an addu "method" for adding
      unique elements as well as a normal add method. I think the
      code is more readable. Wrote allot0, dump and dumpw
      wrote !1+ but not !1- That is, decrement the value
      at the address and store new value there.
      Modified list: so that object reference is no longer left on
      stack. Wrote 'thru' to load a series of blocks.

      .' prints in what word an address is found, and the offset from
      the start of the word.

      Thinking about how to code data structures. Put one reference
      on the stack and make 'methods' that access and manipulate
      part of the structure. Also, make the methods use a
      simple reference to the object. Eg an editing buffer has
       capacity, insertion point and pointer to buffer.

   24 june 2018
      Found a bug in r@. Needed to dig under the word return pointer
      (of r@ itself).
      One extra ';' causes fatal errors.
      made the 'unloop' word which is needed for exiting from
      loops before the counter has finished. wrote erase and fill.
      Wrote set.ls and set.add and set: which appear to create a
      set that only contains unique items (16bit elements).
        opcodes: 1302 bytes
        bytecode: 3847 bytes
        total: 8274 bytes

   23 june 2018

      made atxy and getxy as opcodes to obtain and set the current cursor
      position. wrote 'arrow' to move cursor around. Now the trick is
      to move the insertion point around in a text buffer. This is a map.

      Cant do an exit from a do/loop, need an unloop or 'leave'
      word which will get the loop parameters of the return stack.

      The create bug still exists but can be avoided by putting does>
      at the end of defining words. Will try to write ed which
      allows basic editing of a buffer (1024bytes) which can then be
      saved to disk with 'save'.
        opcodes: 1270 bytes
        bytecode: 3839 bytes
        total: 7770 bytes

   22 june 2018
      made write.x appear to work. Wrote 'save' which saves the first
      buffer to disk block n. This maybe enough to use the os to
      develop itself. Only tested on qemu. Need to test on real hardware.
      But need some kind of editor to edit buffers/blocks.
        opcodes: 1245 bytes
        bytecode: 3829 bytes
        total: 7322 bytes

     included a "sanity-check" in the block writing routine which
     checks that the number 31415 is the 8th byte of the boot sector

   21 june 2018
      Since the aim is to achieve a minimum size and simplicity while
      maintaining portability, it is useful to record the system size.
        opcodes: 1231 bytes
        bytecode: 3829 bytes
        total: 7280 bytes

      Regarding the create runtime bug (dictionary code pointer not
      advanced etc):

      Wrote 'def' to set the defining control bit in a word.

      After consideration, we see that defining words, when they
      exit at run-time need to do essentially what ; semicolon does
      at run-time. However, when a defining word runs, there is
      no semicolon. This arises because the system compiles even code
      which is entered interatively. When a defining word stops
      defining, the compile point needs to be switched back to
      the 'anon' buffer and the dp point needs to be updated.

      One solution, is to make CREATE set a 'def' control bit in
      the count byte of the name. So create becomes an immediate
      word and compiles a call to (create) which is just the
      same as the current create. Then when semicolon executes
      it checks def control bit, and if set, compiles a context switch.

      This bug also effects does>. The does> bug seems fixed by
      essentially putting ; semicolon code into (does>)
      We could write (;) a runtime semicolon.

   20 june 2018
      Struggling with this code>here bug in defining words. One hack
      is to define var like
      : var create 0 , code>here ; imm
   but that is inelegant. When I put code>here into c, I seem to
   get a complete meltdown, maybe a stack overflow, or infinite
   loop. Not sure why yet.

   Discovered a bug in the way that code>here is called at the
   end of a new defining word, so that space is not allocated
   for the data field of the new word. So when we execute a defining
   word the defined word doesnt get any data space. This is
   tricky, maybe c, , and allot etc also need to update dp or
   call code>here. Or in here>anon call code>here first so that
   the code compile point is set correctly

   Wrote 'mname' which shows the machine name and
   signature, and maybe opcodes. Edited the bash script cos.sh
   (and .bashrc) so that todays date is inserted in the machine name
   when the code is compiled. This helps to know what version we
   are running.

19 june 2018
   in' or in.xt is a useful word. Finds the word in which a memory
   address occurs (this is not its standard name) and returns
   its execution token.
   Wrote a new bash function ccos so that I can compile different
   versions of the machine easily.
18 june 2018

   Realised that a unique machine name is useful for installing or
   compiling new opcodes. The name indicates the underlying hardware
   and the 2 stack machine implementation strategy.

   Converted to holding TOS (top-of-stack) in the dx register.
   This should improve performance, especially for calculation
   intensive operations. But the performance difference is not
   very noticable at the moment.

   Need better error reporting in IN, or item, so we
   can work out where something is going wrong.

   If exec.x was an opcode and could call itself, then using call/ret
   for each opcode may have some useful and interesting consequences.
   But real machines dont seem to do this (an opcode which can call
   itself), and I cant think of any immediate benefit.

   Converted call/ret in exec.x/opcodes to
   jmp [] etc. This appears to be working.
   Thinking about using jit just-in-time opcodes which would allow
   the compilation to machine code of the most used words. thus
   improving performance. Conversion appears initially successful.

17 june 2018
   will try to convert 'calls' in exec.x to jumps (done). which should
   save code-space and time. Old version saved to os.call.asm
16 june 2018
   fixed do/loop to make it standard. fixed sed preprocessing
   for 'times' lines. wrote write.x but havent tested.
   wrote timeloop to do simple performance testing. appeared to fix
   read.x by using int 13h ah=8 function to get disk parameters.
   On the asus computer, sectors=63 and sides=32 drive=0
15 june 2018
   Could write a 'set' type, with a capacity and length, with
   an 'add' method, which would only add a 16bit value if it
   wasnt already in the set. This would be useful for building
   a set of unique xts (execution tokens) when finding the
   compiled dependencies for a given word. Also, could adapt this
   technique to a 'point set' where each element in the set is
   2 16bit values, which could be used for making a list of
   block number + offsets for a dependency compile from source
   (only the required word and its dependencies are compiled).
```

```
   Realised that the exec.x function could just jump to the
   opcodes, rather than calling them, which would simplify
   the coding of each opcode.
   made 'splash' show opcode and byte codes sizes. changed ' to
   make it more standard. rewrote ." to simplify
   trying to write a new read.x to calculate chs for floppy
   emulation, but need to print off and analyse. Thought of
   a dependency compiler creating a list of block numbers
   and offsets.
13 june 2018
   realised that if fi begin again etc can all be coded in
   source.
12 june 2018
   made a splash screen 'splash'. Saw how the scale operator
   */ is useful for multiplying by fractions without losing
   precision eg: 100 2 3 */ gives 2/3rds of 100. The intermediate
   result is a double number.
   could make 'state' have 3 states, not 2. ie immediate/delegate/compile
   in immediate, everything is compiled, in delegate, the word
   itself decides whether to compile or execute, and 'compile' where
   everything gets compiled even immediate words.

11 june 2018

   can define [char] like this.
   : [char] postpone char postpone literal ;
   defined ."
   Also postpone could have a
   syntax like [: ... :] which basically would force even immediate
   words to be compiled. This would just be a branch in the
   item,/compile, word.

   fixed parse so that it doesnt include final delimiter
   started a sed preprocessor in a separate file 'os.sed' Need to stop
   os.sed from mangling labels like 'block1:' etc. Realised that postpone
   is an important word. It just compiles an FCALL to the word even if
   the word is immediate. Can be defined in source. Thought about how to
   emulate opcodes in high level forth. The opcode calls a word similar to
   FCALL that pushes address of emulating word on call stack. But need to
   do some very simple assembling to get the write address into the
   machine code. Does 'call,' need to be immediate?

10 june 2018

   Made 'postpone' which compiles an fcall to immediate words like
   sliteral (otherwise they would execute immediatly). This is because we
   want sliteral to execute when parse has finished (at run time) not at
   compile time. Made s" using postpone.

   Did sliteral and (does>), the runtime behaviour of "does>" and
   wrote does> . Made call, ( xt -- ) to compile an FCALL to xt
   realised the importance of (does>) and other (...) words.
   Otherwise we would have to compile lots of code while
   executing 'does>'.
9 june 2018
   Made the preprocessor put a 13,10 at the end of
   each line because plain text source will have them. This allows
   a comment-to-end of line syntax which is handy.
   Fixed a problem in HERE>CODE. This word executes
   the ANON buffer but was not resetting it, so
   words like IMM were executing multiple times.
   FIB stopped working today, not sure why
   Made a LITERAL word to use with CHAR in colon defs
   more or less fixed PARSE. so comments like ( a b --) are
   easy now. eg: : ( [char] ) parse ; imm
8 june 2018
   Wrote a very basic sed preprocessor for "os.asm" and a bash function to
   use it (see comments at top of file about how to compile and run).
```

```
       But need a Forth comment syntax to make it worthwhile, and to
       get that I need PARSE to work, either in source or bytecode.
       rewrote INWORDS and FINDWORDS as source.
       These words are only really useful as a testing mechanism.
    7 june 2018
       work on this forth!! Copied a recursive FIB fibonacci word.  And I
       didnt even think about them when I was coding! Rewrote "keycode" as
       source.  When loading source code into a buffer, it might be good to
       get rid of extra white space...  Defined char but realised that there
       is a problem with the use of char in a colon def, because the character
       code is pushed onto the stack at compile time, instead of compiling a
       literal to push onto stack at run time
    6 june 2018
       Thought that PCALL could be modified to execute opcodes
       as well. Not so easy.
       ACCEPT uses the rstack to save the buffer address, but
       2DUP makes this unnecessary. Also "accept" needs a char limit.
       In this system, it seems all defining words must be immediate???
       So we must do
         : variable create 0 , ; immediate
         : var create 0 , ; imm
       Need to fix other parse now. using CREATE in : not so good ?
       Thinking about how to implement immediate execution of opcodes
       and literal numbers (to push onto the stack). Could compile
       to a small anonymous immediate buffer and execute immediately.
       Wrote CREATE. Appears to be working. Made : COLON and ; use
       CREATE  which appears to be working.
    5 june 2018
       fixed WPARSE to use the >IN stream and got : COLON to use it.
       Thought of the idea of a machine "signature". Which is
       just a list of opcodes plus mappings.
       eg 1-5,7-33,34:2101
         This means that the given machine has standard opcodes
         1-5 and 7-33 and the standard opcode 2101 which is
         mapped to 34.

       Made a byte code version of [ and ] for testing. Started
       CREATE. wrote WPARSE for parsing whitespace delimited
       words, skipping initial whitespace. opcodes cant
       execute immediately! nor numbers! fix? WPARSE
       advances the input stream with IN+

       dp is a pointer but HERE is a value and >IN is 2 values
       This is a bit confusing and not consistent. Could make dp
       a value, and CODE a pointer.

       Wrote a HERE>CODE word which sets the here var to the next available
       location in dictionary. It is called by CREATE and thus all defining
       words. It also writes an EXIT to the end of ANON and executes anon with
       FCALL This ensures that what is in the anonymous compile buffer ANON
       will always get executed before new code is written to the dictionary.
       We can call this a "context switch".  The ANON compile buffer is where
       non-defining interactive commands are compiled to.

    4 june 2018
       Wrote a whitespace word WSPACE that returns a
       flag indicating if a char is tab=9 cr=13 space=32 or 0 etc
       This can be used in WPARSE

       Wrote LOAD which just loads the first block. A better LOAD
       would be : load 2* block1 + 2 first read first 1 K source ;
       which should load any block number eg 3 load. But we should
       also implement block & buffer.

       And started to move code into that source code block from lib.p
       rewrote >IN and IN0 so that they would use a length
       variable for input streams. Realised that SOURCE
       and IN, are different, because SOURCE establishes a new
       input stream. We dont seem to have to save the current one
       because, by the time it executes the current stream has been
       compiled... but, if we execute SOURCE immediately then we
       may get serious problems.

    3 june 2018
       Made an word IMM which sets the last word in the dictionary to
       immediate.  Thinking about the exec function which executes bytecode.
       should this be able to call itself? I cant think of any analogy in a
       real machine, so I dont think so.  Could be good to have a word which
       pushes the address of the opcode table onto the stack to allow
       manipulation of opcodes from within forth

    2 june 2018
       Changed LAST to be a pointer. Made a STATE variable
       Put state test into ITEM, WORD.
       But [ and ] words must be immediate themselves! or they dont work.

       Made the READ opcode sort of working (up to sector 18 ?) but
       need to grapple with the idea of emulated disk geometries
       and convert sector number to head+cylinder+sector number etc
       Want to make this opaque, so that the opcode handles this
       mess and the code can just treat the disk as an enormous
       array of sectors/blocks.

       After that can code buffer and block words. Maybe use last byte
       of block to store "update" bit and block number. This info
       is used by buffer and block words to determine if a read and/or
       write to disk is necessary. Will just have one source buffer
       initially for simplicity.

       Think I am close to achieving a robust, compact, and powerful
       system. Once LOAD/BUFFER/BLOCK are working well, we can
       rewrite many words as source code, thus reducing the core even
       further. Still need to fix DO and LOOP
       And write VAR/VARIABLE, CON/CONSTANT etc. Also could try to wrap core
       and source blocks in a super basic fat12 file system so
       that source code can be edited on another operating system?
       But even FAT12 looks complicated!

    29 may 2018
       writing out some standard forth words in %if0 block
       Will need to preprocess %if0 block below  to put "db ’ " etc
       in front of every line of some forth source. squareroot word
    28 may 2018
       thinking about CREATE and DOES>. realised that the implementation
       of these words is not that difficult. DOES> needs to compile an
       FCALL in new defined words to the code immediately after it, as well as
       an EXIT for the defining word. Probably will do this by jumping over
       parameter field.
    23 may 2018
       Made EKEY work on x86 architecture. Made a LOOP immediate word.
       but its not like the standard forth LOOP at the moment.
    15 May 2018
       Finally got a : compiler working, so new words can be added
       to the dictionary. Wrote inputcompile (IN,) to compile
       the input buffer to the HERE pointer.
       Made a basic foreground
       colour changer just for fun FG opcode and a video mode changer VID
       for colour vga style monitors
    11 May 2018
       separated this forth-like system into a new file. Up until
       now, have been developing as part of the osdev booklet.
    10 june 2017
       made a return stack with es:di and made fcall.x and exit.x
       use the return stack, apparently successfully which allows
       nested procedures.

%endif ; }docs
```

```
 BITS 16
 [ORG 0]

  jmp 07C0h:bootload    ; Goto segment 07C0
    db 'pi:'
    dw 31415         ; magic number
    drive: db 0      ; a variable to hold boot drive number
    db 'boot:now'
  bootload:
    mov ax, cs     ; the code segment is already correct (?!)
    mov ds, ax     ; set up data and extended segments
    mov es, ax
    mov [drive], dl ; save the boot drive number
    mov ax, 07C0h   ; Set up 4K stack space after this bootloader
    add ax, 288     ; (4096 + 512) / 16 bytes per paragraph
    mov ss, ax      ; with a 4K gap between stack and code
    mov sp, 4096

     ; save the DL register or else dont modify it
     ; it contains the number of the boot medium (hard disk,
     ; usb memory stick etc)
     ; The 'floppy' Drive is NOT necesarily 0!!!

   reset:            ; Reset the virtual floppy drive (usb)
     mov ax, 0       ;
     mov dl, [drive] ; the boot drive number (eg for usb 128)
     int 13h         ;
     jc reset        ; ERROR => reset again
   readdisk:
     mov ax, 1000h       ; ES:BX = 1000:0000
     mov es, ax          ; es:bx determines where data loaded to
     mov bx, 0           ;
     mov ah, 2           ; Load disk data to ES:BX
     ;mov al, 8           ; Load 8 sectors 512 bytes * 8 == 4K
     mov al, 16           ; Load 16 sectors 512 bytes * 16 == 8K
     mov ch, 0           ; Cylinder=0
     mov cl, 2           ; start sector=2 (sector 1 is the boot sector)
     mov dh, 0           ; Head=0
     mov dl, [drive]     ;
     int 13h             ; Read!
     jc readdisk         ; ERROR => Try again or exit

   jmp 1000h:0000      ; Jump to the loaded code

   times 510-($-$$) db 0   ; pad out the boot sector
                           ; (512 bytes)
   dw 0AA55h               ; end with standard boot signature

; ****
; this below is the magic line to make the new memory offsets
; work. Or compile the 2 files separately
; https://forum.nasm.us/index.php?topic=2160.0

   section stage2 vstart=0

   jmp start

  ; aliases for each bytecode, these aliases need to be in the
  ; same order as the pointer table below
  ; The nasm code below gives values of 1,2,3,4,5 etc to each
  ; bytecode alias. A new opcode can be inserted without having
  ; to update all the following opcodes.

  DUP equ 1
  DROP equ DUP+1
  SWAP equ DROP+1
  OVER equ SWAP+1
  ROT equ OVER+1
  TWODUP equ ROT+1
  TWODROP equ TWODUP+1
  FLAGS equ TWODROP+1
  DEPTH equ FLAGS+1
  RDEPTH equ DEPTH+1
  RON equ RDEPTH+1
  ROFF equ RON+1
  RFETCH equ ROFF+1
  TWORON equ RFETCH+1
  TWOROFF equ TWORON+1
  STORE equ TWOROFF+1
  STOREPLUS equ STORE+1
  FETCH equ STOREPLUS+1
  FETCHPLUS equ FETCH+1
  CSTORE equ FETCHPLUS+1
  CSTOREPLUS equ CSTORE+1
  CFETCH equ CSTOREPLUS+1
  CFETCHPLUS equ CFETCH+1
  COUNT equ CFETCHPLUS    ; count is an alias for c@+
  ZEROEQUALS equ CFETCHPLUS+1
  EQUALS equ ZEROEQUALS+1
  NOTEQUALS equ EQUALS+1
  LESSTHAN equ NOTEQUALS+1
  ULESSTHAN equ LESSTHAN+1
  LIT equ ULESSTHAN+1
  LITW equ LIT+1
  EMIT equ LITW+1
  AKEY equ EMIT+1
  KEY equ AKEY+1
  EKEY equ KEY+1
  GETXY equ EKEY+1
  ATXY equ GETXY+1
  PLUS equ ATXY+1
  MINUS equ PLUS+1
  INCR equ MINUS+1
  DECR equ INCR+1
  NEGATE equ DECR+1
  DPLUS equ NEGATE+1
  LOGOR equ DPLUS+1       ; logical or
  LOGXOR equ LOGOR+1
  LOGAND equ LOGXOR+1
  SCALE equ LOGAND+1
  DIVMOD equ SCALE+1
  UDIVMOD equ DIVMOD+1
  UMIXDIVMOD equ UDIVMOD+1
  TIMESTWO equ UMIXDIVMOD+1
  MULT equ TIMESTWO+1
  UMULT equ MULT+1
  UMIXMULT equ UMULT+1
  RPICK equ UMIXMULT+1
  FCALL equ RPICK+1
  PCALL equ FCALL+1
  EXIT equ PCALL+1
  LJUMP equ EXIT+1
  JUMP equ LJUMP+1
  JUMPZ equ JUMP+1
  JUMPF equ JUMPZ     ; jumpf (false) is an aliase for jumpz
  JUMPNZ equ JUMPZ+1
  JUMPT equ JUMPNZ    ; jump-true alias for jump-not-zero
  RLOOP equ JUMPNZ+1
  DIVTWO equ RLOOP+1
  READ equ DIVTWO+1  ; loads sectors from disk
  WRITE equ READ+1   ; writes sectors to disk (usb etc)
  FG equ WRITE+1   ; foreground colour for text and pixels
  BG equ FG+1      ; background colour
  CURSOR equ BG+1  ; set the screen cursor shape (0=hidden)
  CLS equ CURSOR+1 ; clear screen
  VID equ CLS+1    ; video mode
                   ; changed PIX to PLOT
  PLOT equ VID+1   ; plot one pixel on screen at xy, with colour FG
```

```
GLYPH equ PLOT+1 ; display glyph on screen
RTC equ GLYPH+1  ; real time clock
CLOCK equ RTC+1  ; number of clock ticks since midnight
NOOP equ CLOCK+1 ; no operation & end marker

;*** control bits and mask
IMMEDIATE equ 0b10000000
MASK equ 0b00011111

; Machine signature goes here
; machine name: eg x86.v1 opcodes and mappings
; the machine name can be used to install or compile new
; opcodes for this particular architecture
; but should this be an opcode or a forth word?
machine.doc:
  ; db ' allows access to the machine name, and signature '
  ; dw $-machine.doc
machine:
  dw 0
  db 'machine', 7
machine.p:
  db LITW
  dw machine.name
  db EXIT
machine.name:
; the name of the machine, including on what chip it is
; implemented and a mnemonic about how it is implemented
; maybe include compile date in this machine name.
  db machine.sig-machine.name-1, '4th.x86.topdx'
; maybe a machine description here
; the machine signature which is like an opcode map
machine.sig:
  db 4, '1-55'

optable.doc:
  ; db ' allows access to the machine name, and signature '
  ; dw $-machine.doc
optable:
  dw machine.p
  db 'optable', 7
optable.p:
  db LITW
  dw op.table
  db EXIT
; a table of code pointers. The pointers have the
; same offset in the table as the value of the opcode
; Zero is the false flag in Forth, so it is handy not to
; have any opcode defined for forth
op.table:
  dw 0, dup.x, drop.x, swap.x, over.x, rot.x
  dw twodup.x, twodrop.x
  ; db twoswap.x
  dw flags.x, depth.x, rdepth.x
  dw ron.x, roff.x, rfetch.x, tworon.x, tworoff.x
  dw store.x, storeplus.x, fetch.x, fetchplus.x
  dw cstore.x, cstoreplus.x
  dw cfetch.x, cfetchplus.x
  dw zeroequals.x, equals.x, notequals.x,
  dw lessthan.x, ulessthan.x, lit.x, litw.x
  dw emit.x, akey.x, key.x, ekey.x, getxy.x, atxy.x
  dw plus.x, minus.x, incr.x, decr.x, neg.x
  dw dplus.x
  dw logor.x, logxor.x, logand.x
  dw scale.x, divmod.x, udivmod.x, umixdivmod.x, timestwo.x
  dw mult.x, umult.x, umixmult.x
  dw rpick.x, fcall.x, pcall.x, exit.x
  dw ljump.x, jump.x, jumpz.x, jumpnz.x, rloop.x
  dw divtwo.x
  dw read.x, write.x
```

```
  dw fg.x, bg.x, cls.x, cursor.x, vid.x, plot.x
  dw glyph.x
  dw rtc.x, clock.x
  ; could put just-in-time opcodes here.
  ; I think zero is a better teminator no? Yes because
  ; -1 is a valid address although unlikely address!!!
  dw noop.x, -1, 0, 0, 0, 0
  ; fill the table with zeros, so we can add more
  ; in code...

; this is the function which executes the byte codes
; takes a pointer to the code. Jumps are relative to the first
; byte of the jump instruction. This is not an opcode because
; it represents the machine itself...
exec:
  dw optable.p
  db 'exec', 4
exec.x:
  ; get exec.x return address out of the way.
  ; Actually exec.x never returns! (since SHELL is an infinite loop)
  pop word [returnexec]     ; save return ip
  pop si      ; get pointer to code
exec.nextopcode:
  ; stack underflow tests. But this magic number should
  ; be an "equ". It should be 4096 but something is going on
  ; comment out these 2 lines for more speed
  cmp sp, 4098
  ja .underflow
.loadopcode:
  xor ax, ax      ; set ax := 0
  lodsb           ; al := [si]++
  cmp al, 0       ; zero marks end of code
  je .exit

.opcode:
  mov bx, ax      ; get opcode (1...nop etc) into bx
  shl bx, 1       ; double bx because its a word pointer

  ; each opcode jumps back to exec.nextopcode. This improves speed
  ; compared to doing call/ret.
  ; Can hold top of stack in dx "sub dx, bx " etc
  jmp [op.table+bx] ; use opcode as offset into opcode table

  ; check for stack underflow here ???
  ; Actually this is never executed because each opcode
  ; jumps straight back to exec.nextopcode: directly
  jmp exec.nextopcode
.exit:
  ;This never exits. shell is infinite loop, but it could exit...
  push word [returnexec]     ; restore fn return ip
  ret
.underflow:
  mov sp, 4096
  mov al, 'S'
  mov ah, 0x0E
  int 10H
  mov al, '!'
  mov ah, 0x0E
  int 10H
  mov ah, 0    ; wait for keypress bios function
  int 16h      ; ah := asci code and al := scan code
  jmp .loadopcode

returnexec dw 0

plus.doc:
  ; db 'add the top 2 elements of the stack.'
  ; db ' ( n1 n2 -- n1+n2 ) '
  ; db ' This opcode is agnostic about whether the two 16 bit '
```

```
    ; db ' numbers are signed or unsigned. The flags opcode can be '
    ; db ' checked to see if there is an overflow or carry. '
    ; dw $-plus.doc
  plus:
    dw exec.x
    db '+', 1
  plus.x:
    pop bx
    add dx, bx
    jmp exec.nextopcode

  ; better to use add with carry on 16bits
  ; and add with overflow. Then we can implement dplus easily
  dplus.doc:
    ; db 'add the 2 double numbers '
    ; db ' ( d D -- d+D ) '
    ; db ' This opcode is agnostic about whether the two 16 bit '
    ; db ' numbers are signed or unsigned. The flags opcode can be '
    ; db ' checked to see if there is an overflow or carry. '
    ; dw $-dplus.doc
  dplus:
    dw plus.x
    db 'd+', 2
  dplus.x:
    ; TOS (dx) has high word of first double
    mov ax,  dx
    shl eax, 16
    pop ax    ; lower word here
    pop dx
    shl edx, 16
    pop dx
    add edx, eax
    push dx   ; lsw become NOS
    shr edx, 16   ; msw becomes TOS (dx)

    ; a stack method to convert singles to doubles
    ; push word [wHigh]
    ; push word [wLow]
    ; pop dword [dwResult]
    jmp exec.nextopcode

  minus.doc:
    ; db 'subtract the top element of stack from next top'
    ; db ' ( n1 n2 -- n1-n2 ) '
    ; dw $-minus.doc
  minus:
    dw dplus.x
    db '-', 1
  minus.x:
    pop ax
    sub ax, dx
    mov dx, ax    ; tos=dx
    jmp exec.nextopcode

  incr.doc:
    ; db ' ( n -- n+1 )
    ; db 'Increment the top element of the data
    ; db 'stack by one. '
    ; dw $-incr.doc
  incr:
    dw minus.x
    db '1+', 2
  incr.x:
    inc dx
    jmp exec.nextopcode

  decr.doc:
    ; db ' ( n -- n-1 )
    ; db 'Decrement top element of the data stack by one. '
    ; dw $-decr.doc
  decr:
    dw incr.x
    db '1-', 2
  decr.x:
    dec dx
    jmp exec.nextopcode

  ; Called "negate" in standard forths. Called "minus" in older
  ; forths (figforth?).
  neg.doc:
    ; db ' ( n -- -n )
    ; db 'Negates the top item of the stack'
    ; dw $-neg.doc
  neg:
    dw decr.x
    db 'neg', 3
  neg.x:
    neg dx
    jmp exec.nextopcode

  logor.doc:
    ; db ' ( n1 n2 -- n1 V n2 )
    ; db 'the logical OR of n1 and n2'
    ; dw $-logor.doc
  logor:
    dw neg.x
    db 'or', 2
  logor.x:
    pop ax
    or dx, ax
    jmp exec.nextopcode

  logxor.doc:
    ; db ' ( n1 n2 -- n1 V n2 )
    ; db 'the logical or of n1 and n2'
    ; dw $-logxor.doc
  logxor:
    dw logor.x
    db 'xor', 3
  logxor.x:
    pop ax
    xor dx, ax     ; top of stack == dx
    jmp exec.nextopcode

  logand.doc:
    ; db ' ( n1 n2 -- n1 && n2 )
    ; db 'the logical and of n1 and n2'
    ; dw $-logand.doc
  logand:
    dw logxor.x
    db 'and', 3
  logand.x:
    pop ax
    and dx, ax
    jmp exec.nextopcode

  divtwo.doc:
    ; db '(n1  - n1/2) '
    ; db ' divide n1 by 2 '
    ; dw $-divtwo.doc
  divtwo:
    dw logand.x
    db '2/', 2
  divtwo.x:
    shr dx, 1   ; do dx := (dx+1)/2   /tos=dx
    jmp exec.nextopcode

  ; this is unsigned division, which is not usually what we want
```

```
udivmod.doc:
  ; db '(n1 n2 - remainder quotient) '
  ; db ' divide n1 by n2 and provide remainder and quotient. '
  ; db ' n2 is the top item on the stack '
  ; dw $-divmod.doc
udivmod:
  dw divtwo.x
  db 'u/mod', 5
udivmod.x:
  ; maybe could optimise by doing div dx ??
  mov bx, dx  ; divisor is top of stack. tos=dx,
  xor dx, dx  ; set dx := 0
  pop ax      ; dividend is next element
  div bx      ; does dx:ax / bx remainder->dx; quotient->ax
  push dx     ; put remainder on stack
  mov dx, ax  ; put quotient on top of stack
  jmp exec.nextopcode

; this is signed division
divmod.doc:
  ; db '(n1 n2 - remainder quotient) '
  ; db ' divide n1 by n2 and leave remainder and quotient. '
  ; dw $-divmod.doc
divmod:
  dw udivmod.x
  db '/mod', 4
divmod.x:
  ; maybe could optimise by doing div dx ??
  mov bx, dx  ; divisor is top of stack. tos=dx,
  xor dx, dx  ; set dx := 0
  pop ax      ; dividend is next element
  cwd         ; sign extend ax -> dx:ax
  idiv bx     ; does dx:ax / bx remainder->dx; quotient->ax
  push dx     ; put remainder on stack
  mov dx, ax  ; put quotient on top of stack
  jmp exec.nextopcode

scale.doc:
  ; db '(n1 n2 n3 - n1 M* n2 / n3) '
  ; db ' do n1*n2 (2 cell result) and divide by n3 (1 cell result) '
  ; dw $-scale.doc
scale:
  dw divmod.x
  db '*/', 2
scale.x:
  mov bx, dx  ; save t-o-s (n3) for later
  pop ax      ; n2
  pop dx      ; n1
  imul dx     ; do dx:ax := ax*dx
  idiv bx     ; does dx:ax / bx remainder->dx; quotient->ax
  mov dx, ax  ; put quotient on top of stack
  jmp exec.nextopcode

; unsigned. Is this working? seems to be.
umixdivmod.doc:
  ; db '(ud n - rem quo) '
  ; db ' divide unsigned double by n and leave remainder and quotient. '
umixdivmod:
  dw scale.x
  db 'um/mod', 6
umixdivmod.x:
  xor ebx, ebx ; ebx=0
  mov bx, dx   ; divisor is top of stack. tos=dx,
  ;pop dx      ; high cell of dividend is higher on stack
  xor edx, edx ; set dx := 0
  xor eax, eax ;
  pop ax
  shl eax, 16  ; mov high cell of dividend
  ; or try edx:eax
  pop ax      ; low cell of dividend
  div ebx     ; does edx:eax  ? / bx remainder->edx; quotient->eax
  push dx     ; put remainder on stack
  mov dx, ax  ; put quotient on top of stack
  jmp exec.nextopcode

timestwo.doc:
  ; db '(n1 -- n1*2 ) '
  ; db ' double n1. This basically performs a '
  ; db ' left shift on the bits in n1 '
  ; dw $-timestwo.doc
timestwo:
  dw umixdivmod.x
  db '2*', 2
timestwo.x:
  shl dx, 1   ; tos=dx
  jmp exec.nextopcode

mult.doc:
  ; db '(n1 n2 -- n1*n2 ) '
  ; db ' signed multiplication '
  ; dw $-mult.doc
mult:
  dw timestwo.x
  db '*', 1
mult.x:
  pop ax
  imul dx       ; do dx:ax := ax*dx
  ; need to set some overflow flag here if dx<>0
  mov dx, ax    ; result in top of stack, tos=dx
  jmp exec.nextopcode

umult.doc:
  ; db '(n1 n2 -- n1*n2 ) '
  ; db ' unsigned multiplication '
  ; dw $-umult.doc
umult:
  dw mult.x
  db 'u*', 2
umult.x:
  ;mov bx, dx  ; tos=dx
  ; we dont use the result in dx here so could not do this
  ;xor dx, dx  ; set dx := 0
  pop ax
  mul dx        ; do dx:ax := ax*dx
  mov dx, ax    ; result in top of stack, tos=dx
  jmp exec.nextopcode

; The high order 16 bits of a double number is on top of the
; stack.
umixmult.doc:
  ; db '(a b -- d / do a*b and leave double result d, unsiged ) '
  ; db ' unsigned mixed multiplication '
umixmult:
  dw umult.x
  db 'um*', 3
umixmult.x:
  pop ax        ; next-on-stack into AX
  mul dx        ; do dx:ax := ax*dx
  push ax       ; result in top 2 stack items, high item is TOS
  jmp exec.nextopcode

rpick.doc:
  ; dw $-fcall.doc
rpick:
  dw umixmult.x
  db 'rpick', 5
rpick.x:
  ; get nth item from return-stack onto the data-stack
```

```
    ; so 1 rpick gets the top item on the r-stack
    mov bx, dx    ; bx:=dx*2
    add bx, dx
    neg bx        ; negate bx because rstack grows to high memory
                  ; (unlike data stack)
    mov dx, [es:di+bx]  ;
    jmp exec.nextopcode

  fcall.doc:
    ; db 'Call a virtual proceedure on the bytecode stack machine'
    ; db 'The current code pointer (in the SI register)'
    ; db 'is saved - pushed onto the return stack [es:di] and the address
    ; db 'of the virtual proc to execute is loaded into SI. '
    ; dw $-fcall.doc
  fcall:
    dw rpick.x
    db 'fcall', 5
  fcall.x:
    lodsw
    mov [es:di], si
    add di, 2
    mov si, ax      ; adjust the si code pointer
    jmp exec.nextopcode

  ; We can also write "execute" as well which will work
  ; with opcodes by using a temporary compile buffer
  ; in the dictionary or just doing pcall. see word "exec"

  pcall.doc:
    ; db ' ( xt -- ) '
    ; db ' Call a procedure using the top element on '
    ; db ' the data stack as the execution address. This '
    ; db ' allows the implementation of function pointers'
    ; db ' In standard forths this is called "execute". But '
    ; standard forth does not distinguish between opcodes and procs
    ; so execute will do both.
    ; dw $-pcall.doc
  pcall:
    dw fcall.x
    db 'pcall', 5    ; or call it exec/ex/execute
  pcall.x:
    mov [es:di], si ; save ip to return stack
    add di, 2
    mov si, dx      ; get proc exec address from stack
    pop dx
    jmp exec.nextopcode

  exit.doc:
    ; db 'exit a virtual procedure by restoring si '
    ; db 'code pointer'
    ; dw $-exit.doc
  exit:
    dw pcall.x
    db 'exit', 4
  exit.x:
    sub di, 2
    mov si, [es:di]  ; restore si from rstack
    jmp exec.nextopcode

  dup.doc:
    ; db 'Duplicates the top item on the stack.'
    ; dw $-dup.doc
  dup:
    dw exit.x       ; link to previous word
    db 'dup', 3     ; strings are 'counted'
  dup.x:
    push dx    ; dup TOS
    jmp exec.nextopcode
```

```
  drop.doc:
    ; db 'removes the top item on the stack.'
    ; dw $-drop.doc
  drop:
    dw dup.x       ; link to previous word
    db 'drop', 4   ; name with reverse count
  drop.x:
    pop dx         ; remove top element of stack
    jmp exec.nextopcode

  twodrop.doc:
    ; db 'removes the top item on the stack.'
    ; dw $-drop.doc
  twodrop:
    dw drop.x       ; link to previous word
    db '2drop', 5   ; name with reverse count
  twodrop.x:
    pop dx         ; remove top 2 elements of stack
    pop dx
    jmp exec.nextopcode

  swap.doc:
    ; db 'swaps the top 2 items on the stack.'
    ; dw $-swap.doc
  swap:
    dw twodrop.x         ; link to previous word
    db 'swap', 4
  swap.x:
    pop ax      ; get top stack item
    push dx     ; put them back on in reverse order
    mov dx, ax  ; tos=dx
    jmp exec.nextopcode

  ; opcode version of 2swap
  twoswap.doc:
    ; db 'swaps the top 2 items on the stack with the next 2'
    ; ( a b c d -- c d a b )
  twoswap:
    dw swap.x         ; link to previous word
    db '2Swap', 5
  twoswap.x:
    pop ax      ; 2nd-on-stack
    pop bx      ; 3rd-on-stack
    pop cx      ; 4th-on-stack
    push ax
    push dx     ;
    push cx
    mov dx, bx  ; tos := 3rd-onstack
    jmp exec.nextopcode

  over.doc:
    ; db ' ( n1 n2 -- n1 n2 n1 ) '
    ; db ' Puts a copy of 2nd stack item on top of stack. '
    ; dw $-over.doc
  over:
    dw twoswap.x        ; link to previous word
    db 'over', 4
  over.x:
    pop ax      ; get NOS (next-on-stack) stack item
    push ax     ;
    push dx     ;
    mov dx, ax  ; add copy of 2nd item on top of stack
    jmp exec.nextopcode

  ; trying to speed the machine up with these opcodes
  rot.doc:
    ; db ' ( a b c -- b c a ) '
  rot:
    dw over.x        ; link to previous word
    db 'rot', 3
  rot.x:
    pop ax      ; get n-o-s (b)
    pop bx      ; get nn-o-s (a)
    push ax     ; b
    push dx     ; c
    mov dx, bx  ; a
    jmp exec.nextopcode

  twodup.doc:
    ; db ' ( n1 n2 -- n1 n2 n1 n2 ) '
    ; db ' copies 2 stack items onto stack '
    ; dw $-twodup.doc
  twodup:
    dw rot.x
    db '2dup', 4
  twodup.x:
    pop ax      ; get NOS next stack item
    push ax     ;
    push dx     ;
    push ax     ; n1 n2 n1 n2
    jmp exec.nextopcode

  flags.doc:
    ; db ' ( -- n ) '
    ; db ' push flag register onto the stack  '
    ; db ' execution flags such as carry overflow negate etc '
    ; db ' are pushed onto the stack '
    ; dw $-flags.doc
  flags:
    dw twodup.x
    db 'flags', 5
  flags.x:
    push dx   ; save NOS on stack
    pushf
    pop dx    ; get flags into TOS (dx)
    jmp exec.nextopcode

  depth.doc:
    ; db ' ( -- n ) '
    ; db ' Puts on the stack the number of stack items '
    ; db ' before this word was executed '
    ; dw $-depth.doc
  depth:
    dw flags.x
    db 'depth', 5
  depth.x:
    push dx
    mov bx, sp
    mov dx, 4096  ; 4K stack (but could change!)
    sub dx, bx    ;
    shr dx, 1     ; div by 2 (2 byte stack cell)
    ;causing problems ???
    ;dec ax        ; the exec.x call doesnt count
    jmp exec.nextopcode

  rdepth.doc:
    ; db ' ( -- n ) '
    ; db ' Puts on the stack the number of stack items '
    ; db ' on the return stack before this word '
    ; db ' was executed '
    ; dw $-rdepth.doc
  rdepth:
    dw depth.x
    db 'rdepth', 6
  rdepth.x:
    push dx
    mov dx, di
    shr dx, 1
    jmp exec.nextopcode

  ron.doc:
    ; db '( S: n -- )( R: -- n ) '
    ; db ' put the top item of the data stack onto the return stack.'
    ; dw $-ron.doc
  ron:
    dw rdepth.x
    db '>r', 2
  ron.x:
    mov ax, dx     ; value to store at address (tos=dx)
    stosw          ; [es:di] := ax, di+2
    pop dx
    jmp exec.nextopcode

  roff.doc:
    ; db '( S: -- n)( R: n -- ) '
    ; db ' put the top item of the return stack onto the data stack.'
    ; dw $-roff.doc
  roff:
    dw ron.x
    db 'r>', 2
  roff.x:
    ; These underflow checks may not be necessary for working
    ; code, but should make debugging easier.
    cmp di, 0
    je .underflow
    push dx         ; push new nos on stack
    sub di, 2       ;
    mov dx, [es:di] ; get top item off return stack
    jmp exec.nextopcode
  .underflow:
    mov al, 'R'
    mov ah, 0x0E
    int 10H
    mov al, '!'
    mov ah, 0x0E
    int 10H
    mov ah, 0    ; wait for keypress bios function
    int 16h      ; ah := asci code and al := scan code
    jmp exec.nextopcode

  rfetch.doc:
    ; db '( S: -- n )( R: n -- n ) '
    ; db ' fetch top rstack item onto data stack '
    ; dw $-rfetch.doc
  rfetch:
    dw roff.x
    db 'r@', 2
  rfetch.x:
    push dx         ; push new nos on stack
    ;sub di, 2       ;
    mov dx, [es:di-2] ; get top item off return stack
    jmp exec.nextopcode

  ; this can be an opcode
  tworon.doc:
    ; db '( S: n -- )( R: -- n ) '
    ; db ' put the top item of the data stack onto the return stack.'
    ; dw $-ron.doc
  tworon:
    dw rfetch.x
    db '2>r', 3
  tworon.x:
    mov ax, dx     ; value to store at address (tos=dx)
    stosw          ; [es:di] := ax, di+2
    pop dx
    mov ax, dx     ; value to store at address (tos=dx)
    stosw          ; [es:di] := ax, di+2
    pop dx
    jmp exec.nextopcode

  tworoff.doc:
  tworoff:
    dw tworon.x
    db '2r>', 3
  tworoff.x:
    push dx         ; push new nos on stack
    sub di, 2       ;
    mov dx, [es:di] ; get top item off return stack
    push dx         ; push new nos on stack
    sub di, 2       ;
    mov dx, [es:di] ; get top item off return stack
    jmp exec.nextopcode

  store.doc:
    ; db '( w adr -- ) '
    ; db ' place 2 byte value w at address "adr" '
    ; dw $-store.doc
  store:
    dw tworoff.x
    db '!', 1
  store.x:
    mov bx, dx     ; pointer to address
    pop ax         ; value to store at address
    mov [bx], ax   ; 2 byte is stored
    pop dx
    jmp exec.nextopcode

  storeplus.doc:
    ; db '( w adr -- adr+2 ) '
    ; dw $-storeplus.doc
  storeplus:
    dw store.x         ; link to previous word
    db '!+', 2
  storeplus.x:
    mov bx, dx     ; pointer to address
    pop ax         ; value to store at address
    mov [bx], ax   ; 2 byte value is stored
    inc dx         ; advance address and put on stack
    inc dx         ; advance address and put on stack
    jmp exec.nextopcode

  fetch.doc:
    ; db '( adr -- n ) '
    ; db ' Replace the top element of the stack with the '
    ; db ' value of the 16bites at the given memory address '
    ; dw $-fetch.doc
  fetch:
    dw storeplus.x         ; link to previous word
    db '@', 1
  fetch.x:
    mov bx, dx     ; address in tos
    mov dx, word [bx]
    jmp exec.nextopcode

  fetchplus.doc:
    ; db '( adr -- adr+2 n ) '
    ; db ' Replace the top element of the stack with the '
    ; db ' value of the 16bites at the given memory address '
    ; db ' and increment the address by 2 bytes. '
    ; dw $-fetchplus.doc
  fetchplus:
    dw fetch.x         ; link to previous word
    db '@+', 2
  fetchplus.x:
    mov bx, dx
    mov dx, word [bx]  ; value on top of stack
    add bx, 2   ; increment address by 1 word (2 bytes)
    push bx     ; save address on stack
    jmp exec.nextopcode

  cstore.doc:
    ; db '( n adr -- ) store the byte value n at address adr.'
    ; db ' eg: 10 myvar ! '
    ; db '    puts the value 10 at the address specified by "myvar" '
    ; db ' The address is the top value on the stack. '
    ; dw $-cstore.doc
  cstore:
    dw fetchplus.x     ; link to previous word
    db 'c!', 2
  cstore.x:
    mov bx, dx     ; pointer to address
    pop ax         ; value to store at address
    mov [bx], al   ; only the low value byte is stored
    pop dx
    jmp exec.nextopcode

  cstoreplus.doc:
    ; db '( n adr -- adr+1 ) store the byte value n at address adr.'
    ; db ' And increment the address '
    ; dw $-cstoreplus.doc
  cstoreplus:
    dw cstore.x         ; link to previous word
    db 'c!+', 3
  cstoreplus.x:
    mov bx, dx     ; pointer to address
    pop ax         ; value to store at address
    mov [bx], al   ; only the low value byte is stored
    inc dx         ; advance address and put on stack (tos=dx)
    jmp exec.nextopcode

  cfetch.doc:
    ; db '( adr -- n ) Replace the top element of the stack with the value '
    ; db ' of the byte at the given memory address.'
    ; db ' eg: myvar @ . '
    ; db '  displays the value at the address given by "myvar" '
    ; dw $-cfetch.doc
  cfetch:
    dw cstoreplus.x      ; link to previous word
    db 'c@', 2
  cfetch.x:
    mov bx, dx  ; tos=dx
    xor dx, dx  ; set dx := 0
    mov al, byte [bx]
    cbw         ; convert signed byte to signed word ax
    mov dx, ax  ; put literal value on stack (tos=dx)
    jmp exec.nextopcode

  ; need to sign extend !!
  cfetchplus.doc:
    ; db '( adr -- adr+1 n ) '
    ; db ' Replace the top element of the stack with the value '
    ; db ' of the byte at the given memory address and increment the '
    ; db ' address . This is exactly the same as "count"'
    ; dw $-fetchplus.doc
  cfetchplus:
    dw cfetch.x         ; link to previous word
    db 'c@+', 3
  cfetchplus.x:
    mov bx, dx
    xor dx, dx  ; set dx := 0
    mov al, byte [bx]
    cbw         ; convert signed byte to signed word ax
    mov dx, ax  ; put literal value on stack (tos=dx)
    inc bx      ; increment address by 1
    push bx     ; save new address on stack
    jmp exec.nextopcode

  zeroequals.doc:
    ; db ' ( n1 n2 -- flag ) '
    ; db 'Puts -1 (true) on the stack if n1==n2 '
    ; db 'otherwise puts zero (false) on the stack. '
    ; dw $-equals.doc
  zeroequals:
    dw cfetchplus.x     ; link to previous word
    db '0=', 2
  zeroequals.x:
    ; faster?
    neg dx      ; ZF=1/0 for zero/non-zero, CF=not(ZF)
    sbb dx,dx   ; dx = 0/-1 for CF=0/1
    inc dx      ; 1 when dx was 0 at start, 0 otherwise
    jmp exec.nextopcode

    ; slower?
    cmp dx, 0      ; if tos==nos
    je .true
  .false:
    mov dx, 0
    jmp exec.nextopcode
  .true:
    mov dx, -1
  .exit:
    jmp exec.nextopcode

  equals.doc:
    ; db ' ( n1 n2 -- flag ) '
    ; db 'Puts -1 (true) on the stack if n1==n2 '
    ; db 'otherwise puts zero (false) on the stack. '
    ; dw $-equals.doc
  equals:
    dw zeroequals.x     ; link to previous word
    db '=', 1
  equals.x:
    pop bx         ; 2nd stack item
    cmp dx, bx     ; if tos==nos
    je .true
  .false:
    mov dx, 0
    jmp exec.nextopcode
  .true:
    mov dx, -1
  .exit:
    jmp exec.nextopcode

  notequals.doc:
    ; db ' ( n1 n2 -- flag ) '
    ; db 'Puts 0 (false) on the stack if n1==n2 '
    ; db 'otherwise puts -1 (true) on the data stack'
    ; dw $-notequals.doc
  notequals:
    dw equals.x     ; link to previous word
    db '<>', 2
  notequals.x:
    pop bx         ; 2nd stack item
    cmp dx, bx
    jne .true
  .false:
    mov dx, 0
    jmp exec.nextopcode
  .true:
    mov dx, -1
  .exit:
    jmp exec.nextopcode

  lessthan.doc:
    ; db ' ( n1 n2 -- flag ) '
    ; db 'Puts 0 (false) on the stack if n1<n2 '
    ; db 'otherwise puts -1 (true) on the data stack'
    ; db 'this is a signed comparison'
    ; dw $-lessthan.doc
  lessthan:
    dw notequals.x
    db '<', 1
  lessthan.x:
    pop bx         ; 2nd stack item
    cmp dx, bx
    jg .true       ; jg is a signed comparison
  .false:
    mov dx, 0
    jmp exec.nextopcode
  .true:
    mov dx, -1
  .exit:
    jmp exec.nextopcode

  ulessthan.doc:
    ; db ' ( n1 n2 -- flag ) '
    ; db 'Puts true -1/1 on the stack if n1<n2 '
    ; db 'otherwise puts false=0 on the data stack'
    ; db 'this is an unsigned comparison'
    ; dw $-ulessthan.doc
  ulessthan:
    dw lessthan.x
    db 'u<', 2
  ulessthan.x:
    pop bx         ; 2nd stack item
    cmp dx, bx
    ja .true       ; jg is an unsigned comparison
  .false:
    mov dx, 0
    ;jmp .exit
    jmp exec.nextopcode
  .true:
    mov dx, -1
  .exit:
    jmp exec.nextopcode

  lit.doc:
    ; db 'Pushes an 8 bit literal value onto the stack'
    ; dw $-lit.doc
  lit:
    dw ulessthan.x     ; link to previous word
    db 'lit', 3
  lit.x:
    push dx        ; current tos onto stack
    xor ax, ax     ; set ax := 0
    lodsb          ; al := [si]++ get literal char into AL
    cbw            ; convert signed byte to signed word ax
    mov dx, ax     ; put literal value on stack (tos=dx)
    jmp exec.nextopcode

  litw.doc:
    ; db 'Pushes an 16 bit literal value onto the stack'
    ; dw $-litw.doc
  litw:
    dw lit.x     ; link to previous word
    db 'litw', 4
  litw.x:
    push dx       ; current tos onto stack
    lodsw         ; ax := [si]++ get literal char into AX
    mov dx, ax    ; put lit onto stack (tos=dx)
    jmp exec.nextopcode

  ; New version using function ah=0x09 int 0x10. This version can
```

```
; show background colours in video mode 3 (but not in graphic video
; modes like 16). Background colours are nice for text editors (for
; example to display a cursor). The disadvantage is that everything
; appears on one line on the screen unless explicit cr/nl 13,10
; characters are used.
emit.doc:
  ; db 'removes and displays top item on stack as an ascii character.'
  ; db 'Or else moves the cursor about the screen (eg tab/ff/nl/cr/space)'
  ; db 'The character is in the low byte of the stack item...'
  ; dw $-emit.doc
emit:
  dw litw.x
  db 'emit', 4
emit.x:
  mov ax, dx     ; char in dl (dx is TOS) -> al
  ; if the character is printable (not cr/nl/tab/ff etc)
  ; then print it with ah=0x09, otherwise print it with
  ; ah=0x0E
  cmp al, 7
  jb .printable  ; jb (jump if below) is an unsigned comparison
  cmp al, 13
  ja .printable  ; ja (jump if above) is an unsigned comparison
  mov ah, 0x0E   ; bios teletype function
  int 10h
  pop dx         ; tos in dx
  jmp exec.nextopcode

.printable:

  mov bl, [bg.d] ; high 4 bits are background colour
  shl bl, 4      ; put the number (0-15) in high bits
  mov dl, [fg.d] ; low 4 bits are foreground colour
  or bl, dl      ;

  ;mov bl, 0x2F   ; just for debug.
  ; I dont understand this page number parameter...
  mov bh, 0     ; assume we are working in the first page
  mov ah, 0x09   ; the 'function' number for colour print
  mov cx, 1     ; print the character once
  int 10h
  ; increment the cursor position
  mov ah, 03h   ; get cursor position into dx
  int 10h
  mov ah, 02h   ; set cursor position function
  inc dl        ; increment the column position
  int 10h

  pop dx         ; tos in dx
  jmp exec.nextopcode

; The old version of emit using function ah=0x0E int 0x10
; The main disadvantage is that background colours dont seem to
; work with this teletype function.
emito.doc:
  ; db 'removes and displays top item on stack as an ascii character.'
  ; db 'I suppose the character is in the low byte of the stack item...'
  ; dw $-emit.doc
emito:
  dw litw.x
  db 'emit', 4
emito.x:
  mov ax, dx     ; char in dl (dx is TOS) -> al
  ; these background colours arent working in qemu (?)
  mov bl, [bg.d] ; high 4 bits are background colour
  shl bl, 4      ; put the number (0-15) in high bits
  mov dl, [fg.d] ; low 4 bits are foreground colour
  or bl, dl      ;
  mov ah, 0x0E   ; bios teletype function
  int 10h
  pop dx         ; tos in dx
  jmp exec.nextopcode

; This code uses function ah=0x09 instead of ah=0x0E
; to print a character. The cursor must also be advanced.
; Another option is to write directly to video memory to
; print characters
%if 0
  mov bh, 0     ; assume we are working in the first page

  mov ah, 09h   ; the 'function' number for colour print
  mov cx, 1     ; print the character once
  int 10h       ; do it with a bios interrupt

  mov ah, 03h   ; get cursor position into dx
  int 10h
  mov ah, 02h   ; set cursor position function
  inc dl        ; increment the column position
  int 10h
%endif

; key? can also use 16h ah=01h
akey.doc:
  ; db 'Flag if a key is available'
  ; dw $-key.doc
akey:
  dw emit.x    ; link to prev
  db 'key?', 4  ; reverse counted string
akey.x:
  push dx      ; make tos into nos
  mov ah, 1    ; check if key available
  int 16h      ; al := ...
  jz .nokey
  mov dx, 1    ; put flag on stack
  jmp exec.nextopcode
.nokey:
  mov dx, 0
  jmp exec.nextopcode

; key? can also use 16h ah=01h
key.doc:
  ; db 'Get one keystroke from user and place on stack'
  ; db 'The key is represented as an ascii code in the low byte '
  ; db 'of the stack item.'
  ; dw $-key.doc
key:
  dw akey.x    ; link to prev
  db 'key', 3  ; reverse counted string
key.x:
  push dx      ; make tos into nos
  mov ah, 0    ; wait for keypress bios function
  int 16h      ; ah := asci code and al := scan code
  mov ah, 0    ; set ah = 0
  mov dx, ax   ; save asci code onto stack (tos=dx), high byte zero
  jmp exec.nextopcode

ekey.doc:
  ; db ' ( -- event flag ) '
  ; db 'Get one keyboard event and place on stack'
  ; db 'This includes arrow keys etc. The flag indicates if the '
  ; db 'code represents an extended character (eg arrow key) '
  ; db 'or just an ordinary asci character '
  ; dw $-ekey.doc
ekey:
  dw key.x      ; link to prev
  db 'ekey', 4  ; reverse counted string
ekey.x:
  push dx
```

```
    mov ah, 0    ; wait for keypress bios function
    int 16h      ; ah := asci code and al := scan code
    cmp al, 0    ; is extended char (AL == 0)  ?
    je .extended  ; wait for next key if not ->
    mov ah, 0    ; set ah = 0
    push ax      ; save asci code onto stack, high byte zero
    mov dx, 0       ; false flag
    jmp exec.nextopcode

  .extended:
    mov al, ah   ; set ah = al
    xor ah, ah   ; set high byte to zero
    push ax      ; save event code onto stack, high byte zero
    mov dx, 1       ; true flag (is extended char)
    jmp exec.nextopcode

  getxy.doc:
    ; db ' ( -- x y )
    ; db ' Return the x and y position of the cursor. '
    ; dw $-getxy.doc
  getxy:
    dw ekey.x       ; link to previous
    db 'getxy', 5   ; reverse counted string
  getxy.x:
    ; dl=x/col dh=y/row
    push dx      ; make tos into nos
    mov ah, 03h  ; bios function: get cursor position into dx
    int 10h      ; invoke bios
    mov bx, dx   ; dl=x, dh=y
    mov dl, dh
    and dx, 0x00FF
    and bx, 0x00FF
    push bx
    jmp exec.nextopcode

  atxy.doc:
    ; db ' ( x y -- )
    ; db ' set the x and y position of the cursor. '
    ; dw $-setxy.doc
  atxy:
    dw getxy.x
    db 'atxy', 4
  atxy.x:
    ; dl=col dh=row
    mov dh, dl
    pop bx
    mov dl, bl
    mov ah, 02h  ; bios function: set cursor position specified in dx
    int 10h      ; invoke bios
    pop dx
    jmp exec.nextopcode

  ljump.doc:
    ; db 'jumps to a relative virtual instruction.'
    ; db ' The jump is given in the next 2 bytes'
    ; dw $-ljump.doc
  ljump:
    dw atxy.x       ; link to prev
    db 'ljump', 5   ; reverse count
  ljump.x:
    xor ax, ax      ; set ax := 0
    lodsw           ; al := [si]; si=si+2 get relative jump target into AL
    sub si, 3       ; realign si to LJUMP instruction (target is 2 bytes)
    add si, ax      ; adjust the si code pointer by offset
    jmp exec.nextopcode

  jump.doc:
    ; db 'jumps to a relative virtual instruction.'
    ; db ' The relative jump is given in the next byte.'
    ; db ' eg: JUMP, -2, jumps back 2 instructions in the bytecode'
    ; db ' eg: LIT, '*', EMIT, JUMP, -3, '
    ; db '  prints a never-ending list of asterixes '
    ; dw $-jump.doc
  jump:
    dw ljump.x       ; link to prev
    db 'jump', 4   ; reverse count
  jump.x:
    ; jumps can be handled in the exec routine
    ; handle jumps by modifying virtual ip (in this case SI)
    xor ax, ax      ; set ax := 0
    lodsb           ; al := [si]++ get relative jump target into AL
    cbw             ; convert signed byte al to signed word ax (neg offset)
    sub si, 2       ; realign si to JUMP instruction,
    add si, ax      ; adjust the si code pointer by jump offset
                    ; do we need to decrement si ?? yes, more logical
    jmp exec.nextopcode

  jumpz.doc:
    ; db ' ( n -- )
    ; db 'jumps to a relative virtual instruction if top '
    ; db 'stack element is zero. The flag value is removed '
    ; db 'from the stack
    ; db ' The relative jump is given in the next byte.'
    ; db ' eg: JUMPZ, -2, jumps back 2 instructions in the bytecode'
    ; db ' eg: KEY, DUP, EMIT, LIT, '0', MINUS, JUMPNZ, -6 '
    ; db '  allows the user to type until zero is pressed. '
    ; dw $-jump.doc
    ; handle jumps by modifying virtual ip (in this case SI)
  jumpz:
    dw jump.x       ; link to prev
    db 'jumpz', 5   ; reverse count
  jumpz.x:
    xor ax, ax      ; set ax := 0
    lodsb           ; al := [si]++ get relative jump target into AL
    ; check stack for zero, if not continue with next instruction
    cmp dx, 0       ; if dx != 0 continue (tos==dx)
    jne .exit
    cbw             ; convert signed byte al to signed word ax (neg offset)
    sub si, 2       ; realign si to JUMP instruction,
    add si, ax      ; adjust the si code pointer by jump offset
  .exit:
    pop dx
    jmp exec.nextopcode

  ; should jumps take top stack element off ? yes
  jumpnz.doc:
    ; db 'jumps to a relative virtual instruction if top stack element '
    ; db ' is not zero.
    ; db ' The relative jump is given in the next byte.'
    ; db ' eg: JUMPNZ, -2, jumps back 2 instructions in the bytecode'
    ; db ' eg: KEY, DUP, EMIT, LIT, 'q', MINUS, JUMPNZ, -6 '
    ; db '  allows the user to type until "q" is pressed. '
    ; dw $-jumpnz.doc
  jumpnz:
    dw jumpz.x       ; link to prev
    db 'jumpnz', 6   ; reverse count
  jumpnz.x:
    ; handle jumps by modifying virtual ip (in this case SI)
    xor ax, ax      ; set ax := 0
    lodsb           ; al := [si]++ get relative jump target into AL
    ; check stack for zero, if so continue with next
    ; instruction (dont jump)
    cmp dx, 0       ; if bx != 0 continue
    je .exit        ; the only difference with jumpz !
    cbw             ; convert signed byte al to signed word ax (neg offset)
    sub si, 2       ; realign si to JUMP instruction,
    add si, ax      ; adjust the si code pointer by jump offset
  .exit:
    pop dx
    jmp exec.nextopcode

  rloop.doc:
    ; db ' ( R: n -- n-1 ) '
    ; db ' Decrements loop counter on return stack and jumps to '
    ; db ' target if counter > 0 '
    ; db ' like the x86 loop instruction this is a pre-decrement '
    ; db ' so a loop counter of 2 will loop twice. The disadvantage '
    ; db ' is that a loop counter of 0 will loop 2^16 times. '
    ; dw $-rloop.doc
  rloop:
    dw jumpnz.x       ; link to prev
    db 'rloop', 5     ; reverse count
  rloop.x:
    ; handle loops by modifying virtual ip (in this case SI)
    xor ax, ax      ; set ax := 0
    lodsb           ; al := [si]++ get relative loop target into AL
    ; check return stack for zero, if so continue with next
    ; instruction (dont jump/loop)
    mov bx, [es:di-2] ; get top return stack item into bx
    dec bx          ; decrement the loop counter on the return stack
    cmp bx, 0       ; if bx != 0 continue
    mov [es:di-2], bx ; update the counter
    je .exit        ; the only difference with jumpz !
    cbw             ; convert signed byte al to signed word ax (neg offset)
    sub si, 2       ; realign si to JUMP instruction,
    add si, ax      ; adjust the si code pointer by jump offset
  .exit:
    jmp exec.nextopcode

  ; sectors/track and sides are set by bootup code int 13h, ah=8
  ; SectorsPerTrack: dw 18  ; standard floppy config
  ; Sides: dw 2             ; floppies only have one 'platter'

  read.doc:
    ; db ' ( 1st-sector n addr -- flag=T/F ) 
    ; db ' reads n sectors from disk starting at sector to memory addr '
    ; db ' returns 0 on failure, 1 on success. '
    ; dw $-read.doc
  read:
    dw rloop.x
    db 'read', 4
  read.x:

                   ; 1st-sect n address
     mov fs, dx      ; destination memory address in this segment
   .reset:           ; Reset the virtual floppy drive (usb)
     mov ax, 0       ;
     mov dl, byte [drive.d] ;boot drive number (eg for usb 128)
     int 13h         ;
     jc .reset       ; read error => reset again
   .read:

     ; need to save es since rstack points with it!!
     mov [saven.es], es
     mov ax, ds      ; set es:=ds because we read into this segment
     mov es, ax      ; es:bx determines where data loaded to

     pop gs          ; how many sectors ( -> AL)
     pop ax          ; start sector

     ; -- thanks to mike gonta for this code
  .logical:
     ; Will there be issues with multi-track reads?
     ; Also, sectors start at one not zero !!
     ; So sector 1 is the boot sector on disk.
     ; Calculate track (cylinder), head and sector (chs) settings
     ; for the int 13h (ah=2) read disk function
     ; IN: logical sector in AX, OUT: correct registers for int 13h
     ;    cl := start sector
     ;    ch := track (or cylinder)
     ;    dh := head (or side)
     ;    dl := drive or boot device
     ;    ah := 2 (Int 13h read sectors from disk function)
     ;    al := how many sectors to read
     ;    es:bx := destination address in memory
     ;
     ;push bx
     ;push ax
     ; ax has how many sectors to read
     mov bx, ax                 ; Save logical sector
     mov dx, 0
     div word [sectorspertrack.d] ; First the sector
     add dl, 01h                ; Physical sectors start at 1
     mov cl, dl                 ; Sectors belong in CL for int 13h
     mov ax, bx

     mov dx, 0                  ; Now calculate the head
     div word [sectorspertrack.d]
     mov dx, 0
     div word [sides.d]
     mov dh, dl                 ; Head/side
     mov ch, al                 ; Track
     ;pop ax
     ;pop bx
     mov dl, byte [drive.d]     ; Set correct device

     mov bx, fs      ; es:bx is destination address in ram for read
     mov ax, gs      ; ax (al) := how many sectors to read
     mov ah, 2       ; int 13H function ah:2 "Load disk data to ES:BX"
     int 13h         ; Read!
     jc .readerror   ; ERROR => Try again

     ;*** restore es since rstack points with it!!
     mov es, [saven.es]
     mov dx, 1       ; return true flag for success
     jmp exec.nextopcode

   .readerror:
     ;*** restore es since rstack points with it!!
     mov es, [saven.es]
     mov dx, 0        ; return false flag for read error
     jmp exec.nextopcode

  saven.es: dw 0

  ; this has the potential to 'brick' a computer so we need to
  ; get it right before doing it.
  write.doc:
    ; db ' ( first-sector n addr -- flag=T/F/2 )
    ; db ' write sectors to disk '
    ; dw $-write.doc
  write:
    dw read.x
    db 'write', 5
  write.x:

    ; to do: test, very, very, carefully
                     ; 1st-sect n address
     mov fs, dx      ; source memory address in this segment
   .reset:           ; Reset the virtual floppy drive (usb)
     mov ax, 0       ;
     mov dl, byte [drive.d] ;boot drive number (eg for usb 128)
     int 13h         ;
     jc .reset       ; read error => reset again
   .testread:
     ; below is a sanity check read to make sure we are going
    ; to write to the correct disk.
    ;    cl := (1) start sector, in this case boot sector
    ;    ch := (0) track (or cylinder) number
    ;    dh := (0) head (or side)
    ;    dl := drive or boot device 0-3=diskette; 80H-81H=hard disk
    ;    ah := 3 (Int 13h write sectors to disk function)
    ;    al := (1) how many sectors to read
    ;    es:bx := where to read data. In this case a testbuffer
    ;             which is 'first' + 512 bytes

    mov [saven.es], es
    mov ax, ds      ; at the moment 64K segment
    mov es, ax      ; es:bx determines where data is read from
    mov dh, 0       ; first side
    mov dl, byte [drive.d]     ; Set correct device
    mov bx, fs      ; es:bx is destination address in ram for read
    add bx, 1024    ; read to next block
    mov ch, 0       ; first track/cylinder
    mov cl, 1       ; read boot sector (1)
    mov ax, 1       ; ax (al) := how many sectors to read
    mov ah, 2       ; int 13H function ah:2 "read data from disk from ES:BX"
    int 13h         ; read it!
    jc .readerror   ; if error should try again, but we dont...
    mov dx, word [es:bx+8]    ; address of magic number
    ; if the magic number (pi*10000) is not there then this
    ; maybe the wrong disk, so dont write to it!
    cmp dx, 31415
    jne .notpi
    ;mov es, [saven.es]
    ;jmp exec.nextopcode

  .write:
    ;*** need to save es since rstack points with it!!
    ;mov [saven.es], es
    mov ax, ds      ; at the moment 64K segment
    mov es, ax      ; es:bx determines where data is written from
    pop gs          ; how many sectors ( will go in AL)
    pop ax          ; logical start sector
 .logical:
    ; Will there be issues with multi-track write? probably
    ; So sector 1 is the boot sector on disk.
    ; Calculate track (cylinder), head and sector (chs) settings
    ; for the int 13h (ah=3) write to disk function
    ; IN: logical sector in AX, OUT: correct registers for int 13h, ah=3
    ;    cl := start sector (1-n)
    ;    ch := track (or cylinder) number (0-n)
    ;    dh := head (or side)
    ;    dl := drive or boot device 0-3=diskette; 80H-81H=hard disk
    ;    ah := 3 (Int 13h write sectors to disk function)
    ;    al := how many sectors to read
    ;    es:bx := source address in memory of data to write
    ;

    ; ax has how many sectors to write
    mov bx, ax                 ; Save logical sector
    mov dx, 0
    div word [sectorspertrack.d] ; First the sector
    add dl, 01h                ; Physical sectors start at 1
    mov cl, dl                 ; Sectors belong in CL for int 13h
    mov ax, bx

    mov dx, 0                  ; Now calculate the head
    div word [sectorspertrack.d]
    mov dx, 0
    div word [sides.d]
    mov dh, dl                 ; Head/side
    mov ch, al                 ; Track

    ; setting the drive correctly is pretty
    ; important, see sanity check above

    mov dl, byte [drive.d]     ; Set correct device
    mov bx, fs      ; es:bx is destination address in ram for write
    mov ax, gs      ; ax (al) := how many sectors to read

    mov ah, 3       ; int 13H function ah:3 "write data to disk from ES:BX"
    int 13h         ; Write it!
    jc .writeerror  ; if error should try again, but we dont...

    ;*** restore es since rstack points with it!!
    mov es, [saven.es]
    mov dx, 1          ; return true flag for success
    jmp exec.nextopcode

  .readerror:
  .writeerror:
    ;*** restore es since rstack points with it!!
    mov es, [saven.es]
    mov dx, 0          ; return false flag for read error
    jmp exec.nextopcode

  .notpi:
    mov es, [saven.es]
    mov dx, -1         ; notpi flag
    jmp exec.nextopcode

; This opcode and vid.x are obviously not going to be
; available on all hardware. So we need to think about
; how to configure plugable opcodes. Eg: what if a device
; has a gyroscope. We want opcodes to read from that gyroscope
;
fg.doc:
  ; db ' ( n -- ) '
  ; db ' set foreground colour for emit'
  ; dw $-fg.doc
fg:
  dw write.x
  db 'fg', 2
fg.x:
  mov [fg.d], dl  ;
  pop dx
  jmp exec.nextopcode
fg.d: db 5   ; colour cyan

; not working
bg.doc:
  ; db ' ( n -- ) '
  ; db ' set the background colour for emit'
  ; dw $-bg.doc
bg:
  dw fg.x
  db 'bg', 2
bg.x:
  mov [bg.d], dl  ;
  pop dx
  jmp exec.nextopcode
  ;ret
bg.d: db 0    ;

cls.doc:
  ; db ' ( -- ) '
  ; db ' clear screen'
  ; dw $-cls.doc
cls:
  dw bg.x
  db 'cls', 3
cls.x:
  push dx
```

```
    mov ah, 06h   ; ah=function number for int10 (06)
    mov al, 00h   ; al=number of lines to scroll (00=clear screen)
    ;mov bx, 700h  ; bh=color attribute for new lines (grey)
    mov bx, 00h   ; bh=color attribute for new lines
    xor cx, cx    ; ch=upper left hand line number of window (dec)
    ; cl=upper left hand column number of window (dec)
    mov dx, 184fh ; dh=low right hand line number of window (dec)
    ; dl=low right hand column number of window (dec)
    int 10h
    pop dx
    jmp exec.nextopcode

  cursor.doc:
    ; db ' ( n m -- / display cursor block from row m to row n, n>m *) '
    ; db ' sets the text screen cursor shape, with 0 0=hidden'
    ; dw $-cursor.doc
  cursor:
    dw cls.x
    db 'cursor', 6
  cursor.x:
    ; CH = Scan Row Start, CL = Scan Row End
    ;  CX=0607h normal underline cursor (start at row 6 end row 7)
    ;  CX=0007h full-block
    ;  CX=0706h start=7>end=6 This may hide the cursor!
    ;  Or set bit 5 to hide cursor
    ;  CX=2607h invisible cursor (bit 5 set)
    ; push dx
    mov ah, 01h   ; int10 set cursor shape function
    xor cx, cx    ;
    pop cx        ; cl <- row end (NOS)
    mov ch, dl    ; top of stack into ch
    int 10h
    pop dx        ; 3rd-on-stack into DX (new tos)
    jmp exec.nextopcode

  vid.doc:
    ; db ' ( n -- ) '
    ; db ' set video mode to n'
    ; db ' on x86 try 13h mode'
    ; dw $-vid.doc
  vid:
    dw cursor.x
    db 'vid', 3
  vid.x:
    mov ax, dx  ; get video mode
    mov ah, 0   ; ah=0 set video mode function, al=mode
    int 10h
    pop dx
    jmp exec.nextopcode

  ; pix is called "plot" in some old forths.
  plot.doc:
    ; db ' ( x y -- / show 1 pixel at [x,y] with colour from FG *) '
    ; dw $-plot.doc
  plot:
    dw vid.x
    db 'plot', 4
  plot.x:
    ; using interrupts to draw pixels is very slow but ok
    ; for playing around
    pop cx      ; get x-coordinate
                ; y-coordinate in dx
    ; mov al, 15      ; white
    mov al, [fg.d]  ; foreground colour set by FG
    mov ah, 0ch     ; put pixel
    int 10h         ; draw pixel
    pop dx
    jmp exec.nextopcode
```

```
glyph.doc:
  ; db ' ( a -- ) '
  ; db ' display a colour glyph at pixel position xy'
  ; dw $-glyph.doc
glyph:
  dw plot.x
  db 'glyph', 5
glyph.x:
  jmp exec.nextopcode

; this may be useful for timing code
; clock updates at 1193180/65536 (about 18.2) ticks per second.
; counts per second  18
; counts per minute  1092
; counts per hour    65543
; counts per day     1573040
; clock incremented approx every 55ms
clock.doc:
  ; db ' ( -- D ) '
  ; db ' number of clock ticks since midnight '
  ; dw $-clock.doc
clock:
  dw glyph.x
  db 'clock', 5
clock.x:
  push dx
  mov ah, 00h  ; interrupt to get clock ticks since midnight
  int 1Ah      ; cx:dx now holds number of clock ticks since midnight
  push dx      ; low byte on lower stack position
  mov dx, cx   ; high byte on higher stack position
  jmp exec.nextopcode

; There are "issues" here. It is not always possible to
; set format for rtc time, so need to check format from status
; register B and convert if necessary. Also, should check for
; 2 values the same in a loop, so overcome updating problems
rtc.doc:
  ; db ' ( -- secs mins hours days months years ) '
  ; db 'return 6 values on stack representing real time and date. '
  ; db 'called time&date in standard forths. '
  ; dw $-rtc.doc
rtc:
  dw clock.x
  db 'rtc', 3
rtc.x:
  push dx       ; save top of stack
  xor ax, ax    ; set ah, al == 0

  ; status register B controls format of rtc values but
  ; cant always be set

  cli
  mov al, 0x0B       ; try to set date format
  out 0x70, al       ; address reg
  ; set bit 1=24hour, bit 2=binary/bcd
  mov al, 6          ; set 2 low bits on date register B
  out 0x71, al       ; set register
  sti

.updating:

  ; not used at the moment
  mov al, 0x0A       ; check if an update in progress
  out 0x70, al       ; address reg
  in al, 0x71        ; get data from cmos data reg
  test al, 0x80      ; is high bit set?

  ; in theory we are not supposed to read the real time clock
  ; data if an update is in progress, but we wont worry at the
```

```
    ; moment about this

    cli                ; disable interrupts
    mov al, 0x00       ; select seconds
    out 0x70, al       ; address reg
    in al, 0x71        ; get seconds data from data reg
    sti
    push ax

    cli
    mov al, 0x02       ; select minutes
    out 0x70, al       ; address rtc minute register
    in al, 0x71        ; get data from cmos data reg
    sti
    push ax

    cli
    mov al, 0x04       ; select Hour
    out 0x70, al       ; address rtc minute register
    in al, 0x71        ; get hour data from cmos data reg
    sti
    push ax

    mov al, 0x07        ; Day of month
    out 0x70, al        ; cmos select reg
    in al, 0x71         ; cmos data reg
    push ax

    mov al, 0x08         ; Month
    out 0x70, al        ; cmos select reg
    in al, 0x71         ; cmos data reg
    push ax

    mov al, 0x09        ; Year
    out 0x70, al        ; cmos select reg
    in al, 0x71         ; cmos data reg
    mov dx, ax          ; tos=dx

    jmp exec.nextopcode

  noop.doc:
    ; db 'Does nothing. For some reason most machines '
    ; db 'include this instruction. Also it is a good '
    ; db 'end marker for the opcodes '
    ; dw $-noop.doc
  noop:
    dw rtc.x
    db 'nop', 3
  noop.x:
    jmp exec.nextopcode

  ; ********************************
  ; end of opcodes
  ; ********************************

  ; ************
  ; some immediate words
  ; ************

  ; namespace table here??

  hello.doc:
    ; db ' ( -- )
    ; db ' Just testing immediate procs '
  hello:
    dw noop.x
    db 'hello', IMMEDIATE | 5
  hello.p:
    db LIT, '!', EMIT
    db LIT, 'h', EMIT
    db LIT, 'i', EMIT
    db EXIT

  literal.doc:
    ; db ' (comp: n -- )(run: -- n)
    ; db ' Push the TOS onto the data stack at run-time '
    ; db ' This advances compile point by 3 bytes '
  literal:
    dw noop.x
    db 'literal', IMMEDIATE | 7
  literal.p:
    db LIT, LITW     ; n op=LITW
    db FCALL
    dw ccomma.p      ; n    /compile LITW opcode at HERE
    db FCALL         ; tos is 2bytes so use , not c,
    dw comma.p       ; tos will be pushed onto stack at runtime
    db EXIT

  ; create hangs on no input, but wparse doesnt.
  ; important word!! probably used by all defining words
  ; including colon : and VAR & CONSTANT
  ; create is not "immediate" funnily enough but colon is.
  ; there is a bug with defining words. actually we need a (create)
  ; which is this code below, and then an immediate create which
  ; sets a defining bit.
  create.doc:
    ; db ' ( -- )
    ; db ' create a new word header in the dictionary '
    ; db ' CREATE also compiles code to push the address of '
    ; db ' the parameter field onto the stack at runtime. '
    ; db ' As often pointed out, CREATE does not allocate any space '
    ; db ' for the parameter field. The coder can do that with ALLOT or '
    ; db ' C, or , etc .'
    ; db ' In this x86 implementation, the start of the parameter field '
    ; db ' is just the next available byte in the dictionary. '
  create:
    dw literal.p
    db 'create', 6
  create.p:
    ; here>code does lots of important stuff, see above.
    ; such as executing the anon buffer
    db FCALL
    dw heretocode.p
    ; now compile the name to the dictionary. Use code in colon
    ; to see how.

    ; before any new words have been compiled with colon :
    ; then "last" points to "last" ! After new words have been
    ; added then "last" will point to the last word added
    ; (in the current namespace ...)
    db FCALL
    dw last.p       ; /get pointer to last word in dictionary
                    ; or last word in "current" definition word
                    ; list
    db FETCH
    ;*** insert link to previous last word in dict at HERE
    db FCALL
    dw comma.p

    ;*** get the name & length of the new word
    db FCALL
    dw wparse.p          ; a n
    ;*** if wparse returns 0 then no name, only whitespace
    db DUP               ; a n n
    db JUMPZ, .noname-$  ; a n
    db DUP, RON          ; a n       r: n
    ; compile string to current compile position
    db FCALL
    dw scompile.p       ;          r: n
    db ROFF             ; n
    ; compile count|control byte after name
    db FCALL
    dw ccomma.p
    ; set last pointer to new word execution address
    db FCALL
    dw here.p          ; adr   /xt for new word
    db FCALL
    ; change this to FCALL current.p (current definition wordlist)
    ; no just change last.p def
    dw last.p          ; adr last  /pointer to last word
    db STORE
    ; now compile some code that puts the parameter field address
    ; on the stack
    db FCALL
    dw here.p        ; adr  / address of next byte in dict (current xt)
    db LIT, 4, PLUS  ; adr+3 / adjust for LITW, <adr>, EXIT == 4 bytes
    db LIT, LITW     ; adr+3 op=LITW
    db FCALL
    dw ccomma.p      ; adr+3  / compile LITW opcode
    db FCALL         ; address is 2bytes so use , not c,
    dw comma.p       ; /compile address of next byte in dictionary
    db LIT, EXIT     ; op=EXIT
    db FCALL
    dw ccomma.p      ; / compile EXIT opcode
    db FCALL
    dw codetohere.p  ; set dp to here
    db EXIT

  .noname:
    db DROP, DROP       ; clear data stack
    db LIT, '?', EMIT    ;
    db LIT, '?', EMIT    ;
    db EXIT

  callcomma.doc:
    ; db ' ( xt -- )
    ; db ' compile an fcall to xt '
  callcomma:
    dw create.p
    db 'call,', IMMEDIATE | 5
  callcomma.p:
    db LIT, FCALL      ; xt op=fcall
    db FCALL
    dw ccomma.p        ; xt
    db FCALL
    dw comma.p
    db EXIT

  ; This can be tested with : d drop ; : var go ; find d (does>)'
  ; should reset the code field of "go" to behaviour of "d"
  ; all defining words need to be immediate, so 'create' could
  ; just do that automatically
  rundoes.doc:
    ; db ' ( -- )
    ; db ' perform run-time behaviour of does> '
  rundoes:
    dw callcomma.p
    db '(does>)', 7
  rundoes.p:
                      ; xt  /xt -> code after does> in defining word
    ;db EXIT
    ; when (does>) runs then 'here' should be pointing just after
    ; the parameter field of the new word. That is, just after any
    ; data space that has been allotted with 'allot' or ',' etc
    db FCALL
    dw here.p         ; xt here
    db DUP            ; xt H H
```

```
    db FCALL
    dw last.p         ; xt H H L*
    db FETCH          ; xt H H last
    db DUP            ; xt H H L L
    ; set new compile point to last xt (word just created)
    db FCALL
    dw ishere.p       ; xt H H L
    ; compile jump opcode
    db LIT, JUMP      ; xt H H L op=jump
    db FCALL
    dw ccomma.p       ; xt H H L
    ; calculate jump offset. The jump must jump over the
    ; parameter field of the defined word (this field is
    ; allocated with 'allot' or , or c, etc)
    db MINUS          ; xt H (H-L)
    ; compile offset
    db FCALL
    dw ccomma.p       ; xt H
    ; reset "here" to end of new word (after parameter field)
    db FCALL
    dw ishere.p       ; xt
    ; compile code to push parameter field of new word onto stack
    ; at the run-time of the new word its param field goes onto
    ; the stack, so that the code after does> can use it.
    db FCALL
    dw last.p         ; xt last*
    db FETCH          ; xt last
    db LIT, 4, PLUS   ; xt last+4
    db FCALL
    dw literal.p      ; xt
    ; compile call to xt (code just after does> in defining word)
    db FCALL
    dw callcomma.p    ;
    ; the code below is just ; semicolon
    ; could write
    ; db FCALL
    ; dw semicolon.p

    ; compile an exit for the new word being defined
    db LIT, EXIT      ; op=exit
    db FCALL
    dw ccomma.p
    ; update the dictionary compile point and reset compile
    ; point to anon buffer (these are tasks that semicolon normally
    ; does)
    db FCALL
    dw codetohere.p    ; do code>here
    db FCALL
    dw heretoanon.p    ;
    db EXIT

  ; there are 3 "times" here. Compile-time for the defining word
  ; run-time for defining word (which is also compile-time for
  ; the defined word). And run-time for the defined word.
  ; These three "times" can make thinking about defining words
  ; tricky
  does.doc:
    ; db ' ( -- )
    ;
  does:
    dw rundoes.p
    db 'does>', IMMEDIATE | 5
  does.p:
    ; source:
    ;  : does>
    ;    here 7 + post literal [op:] fcall [call:] (does>)
    ;    [op:] exit ; imm

    ; compile current 'here' (compile point) so that it will
    ; be pushed onto stack at run-time of defining word
    ; (which is compile-time of defined word)
    db FCALL
    dw here.p       ; H
    ; adjust the literal to account for the code following
    ; which will be compiled
    db LIT, 7, PLUS ; H+7
    db FCALL
    dw literal.p    ;
    ; compile "fcall (does>)"
    ; but this is the same as call,
    db LIT, FCALL   ; H op=fcall
    db FCALL
    dw ccomma.p     ; H
    db LITW
    dw rundoes.p    ; xt
    db FCALL
    dw comma.p
    ; compile an exit (for does> at run-time)
    db LIT, EXIT    ; op=exit
    db FCALL
    dw ccomma.p
    db EXIT

  resetp.doc:
    ; db ' ( -- )
    ; db ' Resets dp and here to just after the wordname count byte'
  resetp:
    dw does.p
    db 'reset', 5
  resetp.p:
    ; After create HERE and dp @ should both be pointing just after
    ; code to push parameter field onto stack, so we need to
    ; repoint these back to the xt (just after the name count byte)
    db FCALL
    dw dp.p       ; dp
    db DUP            ; dp dp
    db FETCH, LIT, 4  ; dp code* 4
    db MINUS          ; dp code*-4
    db SWAP           ; code*-4 dp
    db STORE          ;
    db FCALL
    dw heretocode.p   ; set here to point to dp
    db EXIT

 ; things to do in CREATE (called by : COLON)
 ;   Set HERE to next code space with HERE>CODE
 ;   create a header. first link back using LAST
 ;   and set LAST to new word
 ;   then compile name, with S,
 ;   compile count, then just exit and let IN, handle the rest
 ;   (which is actually calling : COLON anyway).
 ; In semicolon:
 ;   when ; compile "exit" set dp to here
 ;
  colon.doc:
    ; db ' ( -- )
    ; db ' Creates a new word in the dictionary.'
    ; db ' The only difference with the CREATE word is that '
    ; db ' no code is appended after the word name and count '
  colon:
    dw resetp.p
    db ':', IMMEDIATE | 1
  colon.p:

    ; HERE>CODE in CREATE will execute stuff in the ANON buffer
    ; (as with all defining words which use create.)
    db FCALL
    dw create.p
  db FCALL
  dw resetp.p
  db EXIT

;.noname:
;  db DROP, DROP        ; clear data stack
;  db LIT, '?', EMIT    ;
;  db LIT, '?', EMIT    ;
;  db EXIT

semicolon.doc:
  ; db ' ( -- )
semicolon:
  dw colon.p
  db ';', IMMEDIATE | 1
semicolon.p:
 ; compile an exit
 ; set dp to here
 ; set ishere to anon buffer
  db LIT, EXIT
  db FCALL
  dw ccomma.p        ; compile opcode exit
  db FCALL
  dw codetohere.p    ; do code>here
  db FCALL
  dw heretoanon.p    ; compile all to anon, not dictionary
  db EXIT

%if 0
 ; in source
%endif

%if 0
; in source, can delete
%endif

; this can be written as source but is useful for debugging
; bytecode words and so will be left.
dotstack.doc:
  ; db ' ( -- )
  ; db ' display the items on the data stack without '
  ; db ' altering it. The top (or most recent) item '
  ; db ' is printed rightmost '
dotstack:
  dw semicolon.p
  db '.s', 2
dotstack.p:
  ; below we need a copy of the stack depth
  ; because it gets decremented by the rloop
  ; need to sieve stack items onto rstack
  ; with >r, swap, r>, swap, >r etc
  db DEPTH, DUP       ;  n n
  db JUMPNZ, 4        ;  n
  db DROP
  db EXIT
  ;*** put all items on return stack
  db DUP              ; n n
  db RON              ; n      r: n
  db SWAP, ROFF       ; n a n-x
  db SWAP, RON        ; n n-x  r: a
  db RON              ; n      r: a n-x
  db RLOOP, -5        ; n      r: a n-x-1
  db ROFF             ; n 0    r: a b c ...
  db DROP             ; n      r: a b c ...
  ;*** print all stack items
  db RON              ;        r: a b c ... n
  db ROFF             ; n      r: a b c ...
  db ROFF             ; n c    r: a b
```

```
    db DUP              ; n c c  r: a b
    db FCALL
    ; this should be dw dot.p because stack is normally shown
    ; as signed numbers. See source version.
    dw udot.p           ; n c    r: a b
    db LIT, ' ', EMIT   ; n c    r: a b
    db SWAP, RON        ; c      r: a b n
    db RLOOP, -11       ; c      r: a b n-1
    db ROFF             ; a b c ... 0
    db DROP             ; a b c ...
    db EXIT

  %if 0
  ; in source, can delete
  %endif

  rcount.doc:
    ; db ' ( adr -- adr-n n ) '
    ; db ' count a reverse counted string, ignoring control bit(s). '
    ; db ' Given a pointer to the count byte of a  '
    ; db ' reverse counted string, return the address of the 1st byte '
    ; db ' of the string and its length. This proceedure '
    ; db ' may also handle the anding out of the immediate '
    ; db ' control bit which is stored in the msb of the '
    ; db ' count for execution tokens '
    ; dw $-rcount.doc
  rcount:
    dw dotstack.p
    db 'rcount', 6
  rcount.p:
                    ; adr
    db DUP, CFETCH  ; a n  / get the count
    db LIT, 0b00011111 ; a n mask
    db LOGAND       ; a n&mask
    db DUP          ; a n n
    db RON, MINUS   ; adr-n
    db ROFF         ; adr-n n
    db EXIT

 ; this doesnt seem to be used. can remove
 ; maybe have a "defining" state instead of this control bit?.
  def.doc:
    ; db ' ( xt -- ) '
    ; db ' Set the "defining" control bit in the count byte '
    ; dw $-def.doc
  def:
    dw rcount.p
    db 'def', 3
  def.p:
                    ; adr
    db DECR         ; adr-1
    db DUP, CFETCH  ; a n  / get the count
    ; the def bit is the second bit in count byte
    db LIT, 0b01000000 ; a n mask
    db LOGOR        ; a n.v.mask
    db SWAP         ; m a
    db CSTORE       ;
    db EXIT

  ; used by other bytecode so cant really rewrite as source
  dotxt.doc:
    ; db ' ( adr - ) '
    ; db ' given a valid execution token on the stack '
    ; db ' print the name of the procedure'
    ; db ' The source definition might be
    ; db ' : .xt -1 rcount type ; '
    ; dw $-dotxt.doc
  dotxt:
    dw def.p
```

```
    db '.xt', 3
  dotxt.p:
                  ; xt
    db DECR       ; adr-1
    db FCALL
    dw rcount.p
    db FCALL
    dw type.p
    db EXIT

  xttoop.doc:
    ; db ' ( adr -- n ) '
    ; db ' Given the address of an execution token '
    ; db ' or procedure on the stack provides the numeric '
    ; db ' opcode for that procedure or else 0 for '
    ; db ' an address which does not correspond to a bytecode.'
    ; db ' This is used to compile text to bytecode '
    ; db ' if not an opcode, then compile FCALL etc '
  xttoop:
    dw dotxt.p
    db 'xt>op', 5
  xttoop.p:
                  ; adr
    db DUP        ; adr adr
    db LITW       ;
    dw op.table   ; a a T
    db FETCHPLUS  ; a a T+2 [T]
    db SWAP, RON  ; a a [T]      r: T+2
    db EQUALS     ; a flag       r: T+2
    db JUMPT, 19   ; a           r: T+2
    db DUP, ROFF  ; a a T+2
    ;**** check if end of table
    db DUP        ; a a T+2 T+2
    db FETCH      ; a a T+2 [T+2]
    db LIT, -1    ; a a T+2 [T+2] -1
    db EQUALS     ; a a T+2 flag
    db JUMPF, .moreops-$  ; a a T+2
    db DROP, DROP, DROP   ;
    db LIT, 0     ; 0  / return false if not opcode
    db EXIT
    ;
  .moreops:
    db JUMP, -21  ; a a T+2
    ;** opcode found, calculate offset
    db DROP, ROFF  ; T+2
    db DECR, DECR  ; T
    db LITW
    dw op.table
    db MINUS       ; T - optable
    db DIVTWO      ; opcode
    db EXIT

  %if 0
  ; rewrite as source
  %endif

  udot.doc:
    ; db ' ( n -- ) '
    ; db ' display top stack element as unsigned '
    ; db ' number in the current base. '
    ; dw $-udot.doc
  udot:
    ;dw dotcode.p
    dw xttoop.p
    db 'u.', 2
  udot.p:
    ; as source
    ; : u. ( n -- )
    ;  0 >r
    ;  begin
    ;    base @
    ;    \ n b   r: count
    ;    /mod r> 1+ >r
    ;    ; rem quot
    ;    dup
    ;  until
    ;
    db LIT, 0, RON     ; n            r: 0
  .divide:
    db LITW
    dw base.d          ; n adr        r: 0
    db CFETCH          ; n base
    db UDIVMOD         ; rem quotient
    db ROFF, INCR, RON ; rem quotient r: 0+1
    db DUP
    db JUMPNZ, .divide-$
                      ; rem rem rem ... 0
    db DROP           ; rem rem ...   r: x
    ; as source
    ; drop r> 0
    ; do
    ;   digits ii + c@ emit
    ; loop drop ;
    ;
  .printc:
    db LITW            ; use digit lookup table
    dw digits.d        ; r r r ... adr  r: x
    db PLUS            ; r r ... adr+r  r: x
    db CFETCH          ; r r ... d      r: x
    db EMIT            ; rem ...  print asci digit
    db RLOOP, .printc-$   ; rem ...        r: x-1
    db ROFF, DROP         ; clear rstack
    db EXIT

  ; But this is also the way to print a double number, name clash
  ddot.doc:
    ; db ' ( n -- ) '
    ; db ' display top stack element as unsigned '
    ; db ' number in decimal '
    ; dw $-ddot.doc
  ddot:
    dw udot.p
    db 'd.', 2
  ddot.p:
                       ; n
    db LIT, 0, RON     ; n            r: 0
    db LIT, 10         ; n 10         r: 0
    db DIVMOD          ; rem quotient
    db ROFF, INCR, RON ; rem quotient r: 0+1
    db DUP, JUMPNZ, -7
                       ; rem rem rem ... 0     r: x
    db DROP            ; rem rem ...           r: x
    db LIT, '0', PLUS, EMIT   ; rem ...  print remainder
    db RLOOP, -4       ; rem ...        r: x-1
    db ROFF, DROP      ; clear rstack
    db EXIT

  dot.doc:
    ; db ' ( n -- ) '
    ; db ' display top stack element as a signed number '
    ; db ' in the current base (if u. does so) '
    ; dw $-dot.doc
  dot:
    dw ddot.p
    db '.', 1
  dot.p:
                        ; n
    db DUP              ; n n
    db LIT, 0           ; n n 0
    db LESSTHAN         ; n flag   / n<0 ?
    db JUMPF, 6         ; n
    db NEGATE           ; -n
    db LIT, '-', EMIT   ; -n   /print negative sign
    db FCALL
    dw udot.p
    db EXIT

  cdot.doc:
    ; db ' ( n -- ) '
    ; db ' display top stack element as a signed 8 bit '
    ; db ' number in the current base (if u. does so) '
    ; db ' This is useful for displaying relative jumps '
    ; db ' which are 1 byte signed numbers. '
    ; db ' eg: 255 = -1, 254 = -2, 128 = -127
    ; dw $-cdot.doc
  cdot:
    dw dot.p
    db 'c.', 2
  cdot.p:
                        ; n
    db DUP              ; n n
    db LITW
    dw 128              ; n n 128
    db LESSTHAN         ; n flag  / n < 128
    db JUMPT, 6         ; n
    db LITW
    dw 256
    db MINUS            ; n-256
    db FCALL
    dw dot.p
    db EXIT

  ; see source code for a simpler way to do this
  ; without a lookup table of digits. eg : extract
  todigit.doc:
    ; db ' ( c -- n flag ) '
    ; db ' converts the ascii character of a digit '
    ; db ' on the stack to its corresponding integer '
    ; db ' using the base (1<base<37) '
    ; db ' If the character c is not  '
    ; db ' a valid digit in the current base, then zero '
    ; db ' is put onto the stack as a false flag. Otherwise'
    ; db ' -1 is put onto the stack '
    ; dw $-todigit.doc
  todigit:
    dw cdot.p
    db '>digit', 6
  todigit.p:
                     ; c
    db DUP           ; c c
    db LITW
    dw digits.d      ; c c adr
    db LITW
    dw base.d        ; c c adr adr
    db CFETCH        ; c c a n
    db RON           ; c c a      r: n

    db CFETCHPLUS    ; c c a+1 C  r: n
    db SWAP, RON     ; c c C      r: n a+1
    db EQUALS        ; c flag     r: n a+1
    db JUMPT, 10     ; c          r: n a+1
    db DUP           ; c c        r: n a+1
    db ROFF          ; c c a+1    r: n
    db RLOOP, -8     ; c c a+1    r: n-1
    ;*** not a valid digit
    db DROP, DROP    ; c          r: 0
    db ROFF          ; c 0
    db EXIT
    ;*** valid asci digit, convert to number
    db DROP          ;             r: n-x a+1
    db ROFF, DROP    ;             r: n-x
    db ROFF          ; n-x
    db LITW
    dw base.d        ; n-x adr
    db CFETCH        ; n-x n
    db SWAP          ; n n-x
    db MINUS         ; x
    db LIT, -1       ; x -1   / -1 is true flag
    db EXIT

  digits.doc:
    ; db ' ( -- A ) '
  digits:
    dw todigit.p
    db 'digits', 6
  digits.p:
    db LITW
    dw digits.d
    db EXIT
  digits.d: db '0123456789ABCDEFGHijklmnopqrstuvwxyz'

  ; base is 16bits in all forths. so we can do: 10 base !
  base.doc:
    ; db ' ( -- adr ) '
    ; db ' puts on the stack the address of the current '
    ; db ' numeric base '
    ; db ' which is used for displaying and parsing '
    ; db ' numbers. The base should be 1 < base < 37 '
    ; db ' since these are the digits which can be '
    ; db ' displayed using numerals and letters '
    ; db ' eg: base c@ .   '
    ; db '   displays the current base '
    ; dw $-base.doc
  base:
    dw digits.p
    db 'base', 4
  base.p:
    db LITW
    dw base.d
    db EXIT
  base.d: dw 10

  %if 0
  %endif

  tonumber.doc:
    ; db ' ( adr n -- adr/n flag ) '
    ; db ' Given a pointer to string adr with length "n" '
    ; db ' attempt to convert the string to '
    ; db ' a number. If successful put number and true flag'
    ; db ' on the stack, if not put pointer to incorrect digit'
    ; dw $-tonumber.doc
  tonumber:
    dw base.p
    db '>number', 7
  tonumber.p:

                     ; a n
    db LIT, 0, RON   ; a n         r: 0  /false neg flag
    db RON           ; a           r: 0 n
    ;*** check for +/- at first char
    db DUP, CFETCH  ; a c        r: 0 n
    db LIT, '+'     ; a c '+'    ...
    db EQUALS       ; a flag     ...
    db JUMPF, 8     ; a          ...
    db ROFF, DECR, RON ; a       r: n-1
    db INCR         ; a+1        r: 0 n-1
    db JUMP, 16     ; a+1        r: 0 n-1
    db DUP, CFETCH  ; a c        r: 0 n
    db LIT, '-'     ; a c '-'    r: 0 n
    db EQUALS       ; a flag     r: 0 n
    db JUMPF, 9     ; a          r: 0 n

    ;*** set a +/- flag on the rstack
    db ROFF, DECR   ; a n-1      r: 0
    db ROFF, DECR   ; a n-1 -1
    db RON, RON     ; a          r: -1 n-1
    db INCR         ; a+1        r: -1 n-1

    ;*** exit if zero length string or just +/-
    db ROFF          ; a n       r: -1
    db DUP           ; a n n     r: -1
    db JUMPNZ, 5     ; a n       r: -1
    db ROFF, DROP    ; a 0
    db EXIT
    db RON           ; a         r: -1 n

    db LIT, 0        ; a 0        r: n  /initial sum
    db SWAP          ; 0 a        r: n
    db CFETCHPLUS    ; 0 a+1 d      r: n
    ;*** check if digit
    db FCALL
    dw todigit.p     ; 0 a+1 D flag    r: n
    db JUMPF, 24     ; 0 a+1 D         r: n
    ;***
    db RON          ; 0 a+1        r: n 0-9
    db SWAP         ; a+1 s        r: n digit  /s is sum

    db LITW
    dw base.d
    db CFETCH       ; a+1 s base   r: n digit
    db UMULT        ; a+1 s*base   r: n digit

    db ROFF         ; a+1 s*base digit  r: n
    db PLUS         ; a+1 s        r: -flag n
    db SWAP         ; s a+1        r: -flag n
    db RLOOP, -16   ; s a+1        r: -flag n-1  /back to c@+
    db ROFF, DROP   ; s a+1        r: -flag
    db DROP         ; s            r: -flag
    db ROFF         ; s -flag
    db JUMPF, 3     ; s      / skip if +
    db NEGATE       ; -s     / negate if flag set

    db LIT, -1      ; s -1    /value and true flag
    db EXIT
    ;*** non digit  ; sum a+1 d
    db DROP, SWAP   ; a+1 sum      r: 0 n
    db DROP         ; a+1          r: 0 n
    db LIT, 0       ; a+1 0
    db ROFF, DROP   ; clear return stack
    db ROFF, DROP   ; clear return stack
    db EXIT

  toword.doc:
    ; db ' ( -- adr n ) '
    ; db ' put on the stack a pointer to the current '
    ; db ' word and its length. '
  toword:
    dw tonumber.p
    db '>word', 5
  toword.p:
    ; handle zero case (no word found)
    db LITW
    dw toword.d       ; adr
    db FETCH          ; aw
    db DUP           ; aw aw
    db LITW
    dw inp.d        ; aw aw a
    db FETCH         ; aw aw ap
    db SWAP          ; aw ap aw
    db MINUS         ; aw n
    db EXIT
  toword.d dw 0  ; pointer to start of current word

  wspace.doc:
    ; db ' ( c -- flag ) '
    ; db ' return true if character c is whitepace (tab, space, 0 etc) '
  wspace:
    dw toword.p
    db 'wspace', 6
  wspace.p:
    ; stack diagram no good
             ; c
    ; space
    db DUP       ; c c
    db LIT, ' '  ; c c space
    db EQUALS    ; c flag
    db SWAP      ; flag c
    ; carriage return
    db DUP       ; c c
    db LIT, 10   ; c c space
    db EQUALS    ; c flag
    db SWAP      ; flag c
    ; newline
    db DUP       ; flag c c
    db LIT, 13   ; flag c c cr
    db EQUALS    ; flag c flag
    db SWAP      ; f f c
    ; zero
    db DUP       ; f f c c
    db LIT, 0    ; f f c c 0
    db EQUALS    ; f f c f
    db SWAP      ; f f f c
    ; tab
    db DUP       ; f f f c c
    db LIT, 9    ; f f f c c tab
    db EQUALS    ; f f f c f
    db SWAP      ; f f f f c
    ; combine flags
    db DROP      ; f f f f
    db LOGOR     ; f f f
    db LOGOR     ; f f
    db LOGOR     ; f
    db LOGOR     ; f
    db EXIT

  wparse.doc:
    ; db ' ( -- adr n ) '
    ; db ' return next word and length in the current input stream.'
    ; db ' the input stream is parsed for whitespace delimited words. '
    ; db ' WPARSE skips initial whitespace. It is an important word '
    ; db ' because it is the standard way to get the next word in '
    ; db ' the current input stream, so things like CHAR and CREATE '
    ; db ' rely on it.
    ; db ' For a more traditional forth "parse" see "parse.p" '
    ; dw $-parse.doc
  wparse:
    dw wspace.p
    db 'wparse', 6
  wparse.p:
    ; >in
    db FCALL
    dw toin.p       ; adr n
    db DUP          ; a n n
```

```
    ;*** no more text so nothing to parse
    db JUMPZ, .notext-$  ; a n
    ;*** get the name of the new word
    db RON          ; adr            r: n
  .nextspace:
    db CFETCHPLUS   ; a+1 [a]        r: n
    ; check for other whitespace, eg 0 tab, cr etc
    ; write a "whitespace" word
    db FCALL
    dw wspace.p         ; a+1 flag      r: n   /true if whitespace
    ;db LIT, ' '    ; a+1 c space    r: n
    ;db EQUALS      ; a+1 flag       ...
    db JUMPF, .nonspace-$     ; a+1            ...
    db RLOOP, .nextspace-$    ; a+1            r: n-1
    ;*** no char found, so return 0
    db DECR         ; a              r: 0
    db ROFF         ; a 0
    db EXIT         ;
    ;*** char found
  .nonspace:
    db DECR         ; a              r: n-m
    ;*** for debug
    ; db DUP, CFETCH, EMIT
    db DUP          ; a a               ...
    ;*** check rstack==0 and exit if so

    ;*** scan for next whitespace
  .nextchar:
    db CFETCHPLUS   ; a a+1 [a]        r: n-m
    ;*** check for whitespace character.
    db FCALL
    dw wspace.p     ; a a+1 flag
    db JUMPT, .space-$       ; a a+1            ...
    db RLOOP, .nextchar-$    ; a a+1            r: n-m-1
    db INCR         ; a a+2 ... /balance decr
    ;***
  .space:
    db DECR         ; a a+p-1          r: 0
    db ROFF, DROP   ; a a+p-1      /clear rstack
    ;*** now calculate length of word
    db RON, DUP, ROFF  ; a a a+p-1
    db SWAP, MINUS  ; a p-1
    ; do 2dup, in+ at this point to advance the parse point
    ; in the input stream.
    db TWODUP       ; a n a n
    db FCALL
    dw inplus.p     ; a n
    db EXIT
 .notext:           ; a 0
    db EXIT

  ; the length returned may be +1 (?)
  ; this allows comments eg
  ; : ( 41 parse ; imm

  ; Can write this as source
  parse.doc:
    ; db ' ( char -- adr n ) '
    ; db ' scan input stream for char and return length and address.'
    ; db ' The current input stream (IN) is scanned for the next char'
    ; db ' matching char. The parse position of the stream is advanced '
    ; db ' after this call. If not found, then parse position and 0'
    ; db ' is returned.'
    ; dw $-parse.doc
  parse:
    dw wparse.p
    db 'parse', 5
  parse.p:
    ; as source
    ; : parse >r >in dup
    ;    ; a n n   r: char
    ;    if >r dup
    ;      ; a a   r: char n
    ;      begin c@+ r> r>
    ;        ; a a+1 [a] n char
    ;        2dup >r >r
    ;        ; a a+1 [a] n char r: char n
    ;        swap drop =
    ;        ; a a+1 F       r: char n
    ; ...

    ;        if r> r> 2drop
    ;          ; a a+p-1              /clear rstack
    ;          over - 2dup in+  1- ;
    ;
    db RON          ;  r: char
    db FCALL
    dw toin.p       ; adr n      r: char
    db DUP          ; a n n      r: char
    ;* no more text so nothing to parse
    db JUMPZ, .notext-$  ; a n   r: char
    ;* make the input stream length the counter
    db RON          ; adr        r: char n
    db DUP          ; a a                ...
    ;*** scan for next char
  .nextchar:
    db CFETCHPLUS   ; a a+1 [a]  r: char n
    ;*** check for scan character.
    db ROFF, ROFF   ; a a+1 [a] n char
    db TWODUP       ; a a+1 [a] n char n char
    db RON, RON     ; a a+1 [a] n char r: char n
    db SWAP, DROP   ; a a+1 [a] char   r: char n
    db EQUALS       ; a a+1 T/F        r: char n

    ;db FCALL
    ;dw wspace.p     ; a a+1 flag
    db JUMPT, .found-$       ; a a+1   r: char n
    db RLOOP, .nextchar-$    ; a a+1   r: char n
  .notfound:
                    ; a a+n   r: char 0


   ; change the behaviour of parse so that it
   ; advances to the end of the stream if not found.

   ; db DROP         ; a       r: char 0
   ; db ROFF, ROFF   ; a 0 char
   ; db DROP         ; a 0
   ; db EXIT

   ;db INCR         ; a a+2 ... /balance the next decr
   ;***

  .found:
    ;db DECR         ; a a+p-1    r: char 0
    db ROFF, DROP   ; a a+p-1     r: char  /clear rstack
    db ROFF, DROP   ; a a+p-1              /clear rstack
    ; calculate the length until the character
    db OVER, MINUS  ; a p-1
    ; Advance the parse point in the input stream.
    db TWODUP       ; a n a n
    db FCALL
    dw inplus.p     ; a n
    ; decrement text length to ignore final delimiter character
    db DECR         ; a n-1
    db EXIT
 .notext:              ; a 0       r: char
    db ROFF, DROP      ; a 0
```

```
  db EXIT

; A simple error message when a word is not found.
; The "missing" word should be overridden with, for example
;   ’ miss is missing
; where blah is a word which gives a good error message
; (shows the search order, stacks, etc)
miss.doc:
miss:
  dw parse.p
  db ’miss’, 4
miss.p:
  ; ( n/xt/op 0 )
  db LIT, 13, EMIT
  db LIT, 10, EMIT
  db DROP, DROP       ;
  db FCALL
  dw toword.p         ; ad n
  db LIT, 5, FG       ; set word colour to cyan
  db FCALL
  dw type.p
  db LIT, 4, FG       ; set word colour to red
  db LIT, ’ ’, EMIT
  db LIT, ’?’, EMIT
  db LIT, ’?’, EMIT
  db LIT, ’ ’, EMIT
  db LIT, 2, FG        ; colour green
  db LIT, ’@’, EMIT
  db LIT, 14, FG       ; colour yellow
  db FCALL        ; pointer to last created word in dictionary
  dw last.p       ; A
  db FETCH        ; [A]
  db FCALL
  dw dotxt.p          ; print out last word compiled
  db LIT, 7, FG       ; set word colour to cyan
  db LIT, 13, EMIT
  db LIT, 10, EMIT
  ;*** compile an exit even when an error occurs
  ;
  db LIT, EXIT
  db FCALL
  dw ccomma.p        ; compile opcode EXIT
  db EXIT

; do something when a word is not found
; This is a deferred word. In the current implementation
; (feb 2019) deferred words are just 4 bytes of code-space
; (no "does>" element). Either an opcode+exit is compiled there
; or an fcall+<address>+exit
; do:
;  ’ newmiss is missing
; where newmiss provides more information about the missing
; word.
; Eventually "missing" could look up words in source code
; and compile them. Or find similar words in different
; word lists, or look for the words on the net.
missing.doc:
; ( n/xt/op flag=0 )
missing:
  dw miss.p
  db ’missing’, 7
missing.p:
  db FCALL
  dw miss.p
  db EXIT
  db 0

; this is almost identical to "source"
inputcompile.doc:
    ; db ' ( -- ) '
    ; db ' compiles the input stream starting from the current '
    ; db ' input parse position until the end of the stream '
    ; dw $-inputcompile.doc
  inputcompile:
    dw missing.p
    db 'in,', 3
  inputcompile.p:
  .nextword:
    db FCALL
    dw wparse.p           ; a+x n  /address+length of next word
    ;*** if parse returns 0 then no more words (all space)
    db DUP               ; a+x n n
    db JUMPZ, .endofstream-$     ; a+x n
    ;*** convert name to number/opcode/fpointer + flag
    db FCALL
    dw tick.p            ; n/op/xt flag
    ;db FCALL
    ;dw dotstack.p
    ;*** if flag==0 abort, unknown word/number
    db DUP              ; m flag flag
    db JUMPZ, .error-$  ; n/op/xt flag

    ;*** immediate words like if/fi begin will leave
    ;    parameters on the data stack
    db FCALL
    dw itemcompile.p        ;
    db JUMP, .nextword-$
  .endofstream:
    ;*** no more words (>in returned zero)
    ;*** compile a final exit even if no semi-colon was
    ; given
    db DROP, DROP       ; clear stack
    db LIT, EXIT
    db FCALL
    dw ccomma.p         ; compile opcode EXIT
    db EXIT
  .error:
    ;db FCALL
    ;dw dotstack.p
    db FCALL
    dw missing.p      ; handle error with deferred word "missing"
    db EXIT

  ; now, should adr be a counted string? should we compile only one
  ; block (1K), or have another paramter to determine the length?
  ; this should be (adr length -- ). And source should be just the
  ; same word as inputcompile.p
  ; One problem: words outside of : defs are compiled to the anon
  ; buffer, and then need to be executed. Eg the immediate word
  ;
  source.doc:
    ; db ' ( adr n -- ) '
    ; db ' compile forth source code from address adr '
    ; db ' This is called "load" in many forths.'
    ; dw $-source.doc
  source:
    dw inputcompile.p
    db 'source', 6
  source.p:

    ; need to save the current input stream and position
    ; eg in.d in.length inp.d and toword.d
    ; but where to save ?

    ;*** set "in" var to adr
    db FCALL
    dw resetin.p         ; set in = adr = >in = >word
```

```
    db LITW
    dw anon.d
    db FCALL
    dw ishere.p

    ;*** rub out anon
    db LIT, EXIT
    db FCALL           ; compile opcode exit
    dw ccomma.p

    db LITW
    dw anon.d
    db FCALL
    dw ishere.p

    ;*** compile input buffer
    db FCALL
    dw inputcompile.p

  .run:
    ; now run the compiled stuff
    ; but only run anon if it has something in it...
    db LITW
    dw anon.d
    db PCALL
    db FCALL
    dw ok.p
    db EXIT

  ok.doc:
    ; db ' ( -- ) '
    ; db ' just prints ok '
  ok:
    dw source.p
    db 'ok', 2
  ok.p:
    ; db LIT, 13, EMIT
    ; db LIT, 10, EMIT
    db LIT, 2, FG     ; set to green
    db LIT, ' ', EMIT
    db LIT, 'o', EMIT
    db LIT, 'k', EMIT
    db LIT, 7, FG     ; set to white
    db LIT, 13, EMIT
    db LIT, 10, EMIT
    db EXIT

  %if 0
  %endif


  ; this is a deferred word so that we can write a word
  ; to search all wordlists (find.so) and use it later. But
  ; find.so is making compilation very slow.
  nfind.doc:
  ; ( n/xt/op flag=0 )
  nfind:
    dw ok.p
    db 'nfind', 5
  nfind.p:
    db FCALL
    dw ffind.p
    db EXIT
    db 0

  ; There also needs to be a list of all wordlists, not just
  ; the ones in the search order buffer.

  ; with wordlists/namespaces, nfind will search namespaces in
  ; the search order s/o buffer.
  ; Ans forth find is ( A -- xt ) where A is a counted string.

  ; This is probably the most important word for speed of compilation
  ffind.doc:
    ; db ' ( adr n -- xt ) '
    ; db ' return the execution address for the given word function. '
    ; db ' Given a pointer to a string "adr" with length n '
    ; db ' return the execution token (address) for the word'
    ; db ' or else zero if the word was not found. '
    ; dw $-nfind.doc
  ffind:
    dw nfind.p
    db 'ffind', 5
  ffind.p:
                    ; a n
                    ; see find.so for a word that
                    ; searches all wordlists in
                    ; the s/o search order buffer.
    db FCALL        ; pointer to last created word in dictionary
    dw last.p       ; a n A
    ; using context nearly works for nfind, but it fails
    ; in block 49 when the context is changed.
    ;dw context.p
    db FETCH        ; a n last

  .again:
    db DECR         ; point to count|control byte of name
    db FCALL
    dw rcount.p     ; a n A N

    ; db RON, OVER, ROFF  ; a n A n N
    ;*** now compare the 2 string lengths n & N
    db SWAP         ; a n N A
    db RON, RON     ; a n         r: A N
    db DUP, ROFF    ; a n n N     r: A
    db EQUALS       ; a n flag    r: A
    db JUMPF, .lengthsnotequal-$    ; a n         r: A
    db ROFF         ; a n A
    db SWAP         ; a A n

    ;*** save values on rstack, clumsy?
    db RON, TWODUP   ; a a A A     r: n

    db RFETCH        ; a a A A n   r: n
    db SWAP, RON     ; a a A n     r: n A
    ;*** compare two strings
    db FCALL         ; are strings equal?
    dw ncompare.p    ; a flag      r: n A
    ;
    db JUMPF, .notequal-$     ; a           r: n A
    ; if strings same clear stacks
    db DROP, TWOROFF   ; A n
    db PLUS, INCR      ; A+n+1
    ; A+n+1 is the execution address. found, so exit now
    db EXIT
    ; if false, balance stacks and jump down
  .notequal:
                    ; a           r: n A
    db TWOROFF      ; a A n
    db SWAP         ; a n A
    db JUMP, 3      ; a n A  /get next pointer
  .lengthsnotequal:
    db ROFF         ; a n A
    db DECR, DECR   ; a n A-2    /A-2 points to previous
    db FETCH        ; a n [A-2]
    db DUP          ; a n [A-2] [A-2]
    db JUMPNZ, .again-$  ; a n [A-2]
```

```
    db TWODROP, DROP  ; clear stack
    db LIT, 0         ; zero means not found
    db EXIT

  immediate.doc:
    ; db ' ( -- ) '
    ; db ' make the last word immediate. '
    ; db ' Set the immediate control bit'
  immediate:
    dw ffind.p
    db 'imm', 3
  immediate.p:
    db FCALL
    dw last.p        ; addr
    db FETCH         ; xt
    db DECR          ; xt-1
    db DUP           ; xt-1 xt-1
    db CFETCH        ; xt-1 [xt-1] /get the count/control byte
    db LIT, IMMEDIATE  ; xt-1 count|control 0b10000000
    db LOGOR         ; xt-1 nVimm / zero or non zero
    db SWAP          ; m xt-1
    db CSTORE        ; trap! the count|control is only one byte
    db EXIT

  isimmediate.doc:
    ; db ' ( xt -- flag ) '
    ; db ' returns 0 if word is not immediate. <>0 otherwise '
    ; db ' given the execution address for a procedure '
    ; db ' return a flag indicating if the procedure '
    ; db ' is immediate or not. Immediate procedures are '
    ; db ' executed at compile time, not compiled '
    ; db ' so, essentially, they compile themselves. '
    ; db ' this allows the compiler to be extended by '
    ; db ' procedures. The immediate control bit is the '
    ; db ' most significant bit of the count byte in the '
    ; db ' name. Immediate words have both a compile-time and '
    ; db ' a run-time behaviour, whereas non-immediate words '
    ; db ' have only a run-time behaviour.
  isimmediate:
    dw immediate.p
    db 'imm?', 4
  isimmediate.p:
                     ; xt
    db DECR          ; xt-1
    db CFETCH        ; [xt-1] /get the count byte
    db LIT, IMMEDIATE  ; n 0b10000000
    db LOGAND        ; n & imm / zero or non zero
    db EXIT

  tick.doc:
    ; db ' ( A n -- n flag) '
    ; db ' given a pointer "A" to a string of length n '
    ; db ' attempt to convert the name to either '
    ; db ' an opcode, procedure execution code, or number/integer '
    ; db ' the flag indicates the type of token returned '
    ; db ' a flag of zero means that the name is neither '
    ; db ' number nor opcode nor xt'
    ; db ' flag=0 not number nor word '
    ; db ' flag=1 if number, 2 if opcode, 3 if procedure '
    ; dw $-tick.doc
  tick:
    dw isimmediate.p
    db 'tick', 4
  tick.p:
                       ; a n
    db TWODUP          ; a n a n
    db FCALL
    dw nfind.p            ; a n xt/0
    db DUP               ; a n xt xt
    db JUMPNZ, .opcode-$ ; a n xt

    ;*** not found, try to convert to number
    db DROP              ; a n
    db FCALL
    dw tonumber.p         ; adr/n flag
    db JUMPNZ, .number-$  ; adr/n
    ;*** not a number, push false flag and exit
    db LIT, 0             ; adr 0
    db EXIT

    ;*** is a number
  .number:
                       ; n
    db LIT, 1          ; n 1   /flag=1 means number
    db EXIT

  .opcode:
    ;*** check if opcode
                  ; a n xt
    db SWAP, DROP   ; a xt
    db SWAP, DROP   ; xt
    db DUP          ; xt xt
    db FCALL        ; does this address correspond to an opcode
    dw xttoop.p     ; xt op|0
    db DUP          ; xt op|0 op|0
    db JUMPZ, .proceedure-$   ; xt op
    ;*** is an opcode
    db SWAP, DROP   ; op      /drop execution token address
    db LIT, 2       ; op 2    /flag=2 means opcode
    db EXIT
  .proceedure:
    ;*** must be procedure,
                  ; xt 0=op
    db DROP         ; xt
    db LIT, 3       ; xt 3   /flag=3 means procedure
    db EXIT

  ; This is only used by decomp.p
  args.doc:
    ; db ' ( op -- flag ) '
    ; db ' given a valid opcode return flag=1 if the '
    ; db ' the opcode requires a one byte '
    ; db ' argument or return flag=2 if the opcode requires '
    ; db ' a 2byte argument or flag=0 if no arguments required. '
    ; dw $-args.doc
  args:
    dw tick.p
    db 'args', 4
  args.p:
    ; lit, jump, jumpz, jumpnz, rloop
    db DUP            ; op op
    db LIT, LIT       ; op op op2
    db EQUALS         ; op flag
    db JUMPT, .one-$  ; op

    db DUP         ; op op
    db LIT, JUMP   ; op op op2
    db EQUALS      ; op flag
    db JUMPT, .one-$  ; op

    db DUP         ; op op
    db LIT, JUMPZ  ; op op op2
    db EQUALS      ; op flag
    db JUMPT, .one-$  ; op

    db DUP         ; op op
    db LIT, JUMPNZ ; op op op2
    db EQUALS      ; op flag
    db JUMPT, .one-$  ; op

    db DUP         ; op op
    db LIT, RLOOP  ; op op op2
    db EQUALS      ; op flag
    db JUMPT, .one-$  ; op

    ; litw, fcall, ljump,

    db DUP         ; op op
    db LIT, LITW   ; op op op2
    db EQUALS      ; op flag
    db JUMPT, .two-$  ; op

    db DUP         ; op op
    db LIT, FCALL  ; op op op2
    db EQUALS      ; op flag
    db JUMPT, .two-$  ; op

    db DUP         ; op op
    db LIT, LJUMP  ; op op op2
    db EQUALS      ; op flag
    db JUMPT, .two-$  ; op

  .zero:
    db DROP
    db LIT, 0     ; 0
    db EXIT
  .one:
    db DROP
    db LIT, 1     ; 1
    db EXIT
  .two:
    db DROP
    db LIT, 2     ; 2
    db EXIT

  debug.doc:
    ; db ' shows values for dp and here '
  debug:
    dw args.p
    db 'debug', 5
  debug.p:
    ; trying to see what is going on with dp and here
    db FCALL        ; get dict compile point
    dw dp.p     ; code*
    db FETCH        ; dp
    ; show the code point
    db LIT, 'c', EMIT
    db LIT, ':', EMIT
    db FCALL
    dw udot.p
    db LIT, ' ', EMIT
    db FCALL        ; get here compile point
    dw here.p       ;
    db LIT, 'h', EMIT
    db LIT, ':', EMIT
    db FCALL
    dw udot.p
    db EXIT

  sync.doc:
    ; db ' if dp is < here then make dp=here '
  sync:
    dw debug.p
    db 'sync', 4
  sync.p:
    db FCALL        ; get dict compile point
    dw dp.p     ; code*
    db FETCH        ; dp
    db FCALL        ; get here compile point
    dw here.p       ; dp here
    db ULESSTHAN      ; flag=T/F
    db JUMPF, .end-$  ;
    ;db LIT, '*', EMIT
    ;db LIT, '<', EMIT
    ;db LIT, '*', EMIT
    ;db LIT, ' ', EMIT
    db FCALL        ; get here compile point
    dw here.p       ; dp here
    db FCALL        ; get dict compile point
    dw dp.p     ; code*
    db STORE        ; do: here dp !
    ;db FCALL        ; get dict compile point
    ;dw debug.p     ; code*
    db EXIT
  .end:
    db EXIT

  ccomma.doc:
    ; db ' ( n -- ) '
    ; db ' compile byte value n at next available byte '
    ; b  ' as given by here.p '
    ; dw $-ccomp.doc
  ccomma:
    dw sync.p
    db 'c,', 2
  ccomma.p:
    ;*** compile single byte
    db FCALL        ; /get compile point
    dw here.p       ; n adr
    db CSTOREPLUS   ; a+1
    db FCALL
    dw ishere.p     ; /update compile point

    ; This crashes the system, not sure why...
    ;db FCALL
    ;dw sync.p     ; /update compile point

    ; need to update dp if here is > than dp
    ; because custom defining words allocate data space in the
    ; dictionary.

  .end:
    db EXIT

  comma.doc:
    ; db ' ( n -- ) '
    ; db ' compile 2 byte value at next available space '
    ; db ' as given by the "here" variable '
    ; db ' code: : , here !+ ishere ; '
    ; dw $-comma.doc
  comma:
    dw ccomma.p
    db ',', 1
  comma.p:
    db FCALL        ; /get compile point
    dw here.p       ; n adr
    db STOREPLUS    ; a+2
    db FCALL
    dw ishere.p     ; /set compile point to new address (a+2)
    db EXIT

    ; trying to get dictionary compile point to advance
    db FCALL        ; /get dict compile point
    dw dp.p     ; code*
    db FETCH        ; dp
    db FCALL        ; /get here compile point
```

```
    dw here.p       ; dp here
    db INCR         ; dp here+1
    ; check if dp < here+1
    db ULESSTHAN      ; flag=T/F
    db JUMPF, .end-$  ;
    ;db FCALL
    ;dw dotstack.p
    ;db KEY
    ;db EMIT
    ;db FCALL
    ;dw codetohere.p
  .end:

  scompile.doc:
    ; db ' ( adr n -- ) '
    ; db ' compile the string at adr with length n to the '
    ; b  ' current compile position as given by here '
    ; dw $-wordcompile.doc
  scompile:
    dw comma.p
    db 's,', 2
  scompile.p:
                            ; ad n
    ; if text length is 0 then do nothing.
    db DUP                  ; ad n n
    db JUMPNZ, .notzero-$ ; ad n
    db DROP, DROP   ;
    db EXIT
  .notzero:
    ;*** n > 0
    db RON          ; ad        r: n   /n is loop counter
  .nextchar:
    db CFETCHPLUS           ; ad+1 c    r: n
    db FCALL
    dw ccomma.p             ; ad+1      r: n
    db RLOOP, .nextchar-$   ; ad+1      r: n-1
    db DROP                 ;           r: 0
    db ROFF, DROP           ; /get rid of rloop counter
    db EXIT

  ; standard core forth word
  ; probably could write this as source
  sliteral.doc:
    ; db ' (comp: adr n -- ) (run: -- A n ) '
    ; db ' compile the string at adr with length n to the '
    ; db  ' current definition. At run-time push the address '
    ; db ' and length of the compiled string onto the stack '
    ; dw $-sliteral.doc
  sliteral:
    dw scompile.p
    db 'sliteral', IMMEDIATE | 8
  sliteral.p:

                            ; ad n
    ; if text length is 0 then do nothing.
    db DUP                  ; ad n n
    db JUMPNZ, .notzero-$ ; ad n
    db DROP, DROP
    db EXIT
  .notzero:
    ; do other stuff here like compile
    ; code to push new A and n on stack at runtime
    ; ... compile jump
    db FCALL
    dw here.p               ; ad n here
    ; adjust address by length of code about to be compiled
    db LIT, 8, PLUS         ; ad n H+5
    db FCALL
    dw literal.p            ; ad n
    db DUP                ; ad n n
    db FCALL
    dw literal.p          ; ad n
    ; compile a jump over the string
    db LIT, JUMP          ; ad n op=jump
    db FCALL
    dw ccomma.p           ; ad n
    ; calculate and compile jump offset (jump over string)
    db DUP                ; ad n n
    db LIT, 2, PLUS       ; ad n n+2
    db FCALL
    dw ccomma.p           ; ad n
    ; copy the string to the current compile position
    db FCALL
    dw scompile.p
    db EXIT

  ; this doesnt always actually compile something (if it is
  ; an immediate word etc) so should be called something else
  ; like doword.p doitem.p
  itemcompile.doc:
    ; db ' ( #/opcode/xt flag -- ) '
    ; db ' compile number/opcode/xt at next available position '
    ; b  ' as given by "here" variable where flag indicates '
    ; db ' the type of item. '
    ; db ' flag=1 literal number, 2 opcode, 3 procedure'
    ; dw $-compile.doc
  itemcompile:
    dw sliteral.p
    db 'item,', 5
  itemcompile.p:
    ;*** 1=literal number eg: 1357, -10, 0
                       ; n flag
    db DUP             ; n flag flag
    db LIT, 1, EQUALS  ; n flag 0/-1
    db JUMPZ, .notnumber-$    ; n flag

    ;*** check if state is immediate
    db FCALL
    dw state.p      ; n flag adr
    db FETCH        ; n flag state
    db JUMPT, .immediateNumber-$  ; n flag

    ;*** compile number
    db DROP         ; n
    db LIT, LITW    ; n op
    db FCALL        ; /get compile point
    dw here.p       ; n op adr
    db CSTOREPLUS   ; n a+1
    db STOREPLUS    ; a+3
    db FCALL
    dw ishere.p     ; /update compile point
    db EXIT

  .immediateNumber: ; n flag
    ; do almost nothing because the number is already
    ; on the stack.
    db DROP         ; n
    db EXIT

  .notnumber:
    ;*** check if is opcode
                       ; n flag
    db DUP             ; n flag flag
    db LIT, 2, EQUALS  ; n flag 0/-1
    db JUMPZ, .notopcode-$   ; n flag

    ;*** check if state is immediate
    db FCALL
    dw state.p      ; op flag adr
    db FETCH        ; op flag state
    db JUMPT, .immediateOp-$  ; n flag

    ;*** 2 is opcode
    ;*** compile the bytecode to given address
    db DROP         ; op
    ; why not just use ccomma.p ??
    db FCALL        ; /get compile point
    dw here.p       ; op adr
    db CSTOREPLUS   ; adr+1 '
    db FCALL        ; /set compile point
    dw ishere.p     ;
    db EXIT

  .immediateOp:     ; op flag
    db DROP         ; op
    db FCALL         ;
    dw obuff.p      ; op adr
    db CSTORE       ;
    ; The second byte of obuff will always be an EXIT, so there
    ; is no need to write one (defined in bytecode)
    ; now actually execute the opcode
    db FCALL         ;
    dw obuff.p      ; adr
    db PCALL
    db EXIT

  .notopcode:
    ;*** check if procedure
                       ; n flag
    db DUP             ; n flag flag
    db LIT, 3, EQUALS  ; n flag 0/-1
    db JUMPF, .error-$ ; n flag

    ;*** 3 procedure
                    ; xt flag
    db DROP         ; xt

    ;*** check if word is immediate
    db DUP          ; xt xt
    db FCALL
    dw isimmediate.p  ; xt flag
    db JUMPT, .immediate-$  ; xt

    ; bug! bug! This only applies to procedures, not
    ; opcodes, so even if state is immediate, opcodes and
    ; numbers will not be executed immediately. I think this
    ; is a problem.

    ;*** check if state is immediate
                    ; xt
    db FCALL
    dw state.p      ; xt adr
    db FETCH        ; xt state
    db JUMPT, .immediate-$  ; xt

    ;*** neither word nor state is immediate, so compile
    db LIT, FCALL   ; xt op
    db FCALL        ; /get compile point
    dw here.p       ; xt op adr
    db CSTOREPLUS   ; xt adr+1
    db STOREPLUS    ; adr+3
    db FCALL        ; /set compile point
    dw ishere.p     ;
    db EXIT

    ;*** immediate proc, execute, dont compile
  .immediate:
    db PCALL
    db EXIT

  .error:
                    ; n flag
    db DROP, DROP
    db EXIT

  %if 0
  ; in source, so can delete this bytecode.
  %endif

  ; if n mod 20 = 0 wait for a key press. This
  ; is to provide basic paging. Could make this a
  ; deferred word. This is not used now
  waitkey.doc:
    ; db ' ( n -- ) '
  waitkey:
    dw itemcompile.p
    db 'wk', 2
  waitkey.p:
    db LIT, 18, DIVMOD, DROP ; remainder
    db JUMPNZ, .continue-$   ; r
    db LIT, 13, EMIT, LIT, 10, EMIT
    db LIT, '~', EMIT
    ; wait for keypress
    db KEY, DROP             ;
  .continue:
    db EXIT

  %if 0

  %endif

  ; The dictionary pointer is called "dp" in pforth and gforth
  dp.doc:
    ; db ' ( -- adr ) '
    ; db ' puts on the stack a pointer to the next available byte'
    ; db ' in the dictionary. '
    ; dw $-dp.doc
  dp:
    dw waitkey.p
    db 'dp', 2
  dp.p:
    db LITW
    dw dp.d
    db EXIT
  dp.d: dw dictionary

  codetohere.doc:
    ; db ' ( -- ) '
    ; db ' sets the dict compile point to the here var'
    ; dw $-codetohere.doc
  codetohere:
    dw dp.p
    db 'code>here', 9
  codetohere.p:
    db FCALL
    dw here.p    ; a
    db LITW
    dw dp.d      ; a A
    db STORE     ;
    db EXIT

  ; this is an important word because it is called by
  ; create, and therefore all defining words. In this implementation
  ; of forth it writes an exit to end of anon (here) and
  ; then executes the anon buffer. This is because this forth
  ; is a "compiling" forth, rather than an "interpreted" forth,
  ; so the default behavior is to compile even interactive commands
  heretocode.doc:
    ; db ' ( -- ) '
    ; db ' sets the here var to the dict compile point'
    ; dw $-codetohere.doc
  heretocode:
    dw codetohere.p
    db 'here>code', 9
  heretocode.p:
    db LIT, EXIT ; op=exit
    db FCALL
    dw ccomma.p  ; compile exit op to end of anon (at "here")
    db FCALL     ; execute everything in the anon buffer
    dw anon.d
    ; need to "empty" the anon buffer after executing
    ; to avoid multiple unwanted calls
    db FCALL     ; set here to start
    dw heretoanon.p
    db LIT, EXIT ; op=exit
    db FCALL
    dw ccomma.p  ; compile EXIT op to start of anon (at "here")
    db LITW
    dw dp.d  ; dp*
    db FETCH     ; [>code]
    db LITW
    dw here.d    ; [>code] a
    db STORE     ;
    db EXIT

  ; Perhaps we should compile everything first to the "anon" buffer!!
  ; even new dictionary entries, and then copy them across to
  ; the dict. This seems to simplify the semantics of all this.

  ; No, the anon buffer will just be a pointer, not a buffer

  heretoanon.doc:
    ; db ' ( -- ) '
    ; db ' sets the compile point to the start of the anon buffer'
    ; dw $-heretoanon.doc
  heretoanon:
    dw heretocode.p
    db 'here>anon', 9
  heretoanon.p:
    db LITW
    dw anon.d    ; anon*
    db LITW
    dw here.d    ; anon* here*
    db STORE     ;
    db EXIT

  here.doc:
    ; db ' ( -- adr ) '
    ; db ' puts on the stack the current compile position'
    ; db ' in the code data space '
    ; dw $-here.doc
  here:
    dw heretoanon.p
    db 'here', 4
  here.p:
    db LITW
    dw here.d
    db FETCH
    db EXIT
  here.d: dw 0   ; pointer to next byte

  ishere.doc:
    ;db ' ( adr --  ) '
    ;db ' set position of next available byte'
    ;db ' for compilation ("here" variable) to address adr'
  ;dw $-ishere.doc
ishere:
  dw here.p
  db 'here!', 5
ishere.p:
                ; adr
  db DUP        ; A A
  db LITW
  dw here.d     ; A A b
  db STORE      ; A

  ; this seems the right place to update the dictionary
  ; compile point, if necessary, but it is not working.
  db FCALL      ;
  dw dp.p   ; A C*
  db FETCH      ; A C
  ; debug
  ;db TWODUP
  ;db FCALL
  ;dw udot.p
  ;db LIT, ' ', EMIT ;
  ;db FCALL
  ;dw udot.p
  ;db LIT, ' ', EMIT ;

  db ULESSTHAN  ; t/f
  db JUMPT, .end-$
  ;db FCALL
  ;dw codetohere.p
.end:
  db EXIT

; according to standard forth docs, this should not be modified by
; forth words except [ ] etc.
; [ and ] are be defined in source as
; : [ 1 state ! ; immediate
; : ] 0 state ! ; immediate
state.doc:
  ; db ' ( -- adr ) '
  ; db ' pushes a pointer to current state (immediate or compile)'
  ; db ' modified by [ and ] only (not by colon) '
  ; db ' state 0=normal/compile state 1=immediate '
  ; dw $-state.doc
state:
  dw ishere.p
  db 'state', 5
state.p:
  db LITW
  dw state.d
  db EXIT
state.d: dw 0

first.doc:
  ; db ' ( -- adr ) '
  ; db ' address of first buffer (for loading source from disk)'
  ; dw $-first.doc
first:
  dw state.p
  db 'first', 5
first.p:
  db LITW
  dw first.d
  db FETCH
  db EXIT
; near the end of this segment, just for testing.
; in reality should be just after the dictionary or stack
; remember distinction between disk map and ram memory map
first.d: dw 50*1024
```

```
blockone.doc:
  ; db ' ( -- adr ) '
  ; db ' sector number of first source block (1K) on disk'
  ; db ' each sector is 512 bytes in length. '
  ; dw $-blockone.doc
blockone:
  dw first.p
  db 'block1', 6
blockone.p:
  db LITW
  ; the 1+ is a hack because this is 1 short for some
  ; reason
  dw 1+(diskcode-$$)/512
  db EXIT

; "a" is top of stack. ie rightmost is last-on first-off
copy.doc:
  ; db ' ( A n a -- ) '
  ; db ' copy n bytes from address A to address a '
  ; db ' this cannot deal with overlapping memory areas. '
  ; dw $-copy.doc
copy:
  dw blockone.p
  db 'copy', 4
copy.p:
                   ; A n a
  db SWAP          ; A a n
  db RON           ; A a    r: n /n loop counter
  db SWAP          ; a A
.again:
  db CFETCHPLUS      ; a A+1 [A]
  ; db DUP, EMIT     ; debug
  db SWAP            ; a [A] A+1
  db RON             ; a [A]       r: n A+1
  db SWAP            ; [A] a       r: n A+1
  db CSTOREPLUS      ; a+1         r: n A+1
  db ROFF            ; a+1 A+1     r: n
  db RLOOP, .again-$ ; a+1 A+1     r: n-1
  db ROFF            ; a+n A+n 0
  db DROP, DROP, DROP ;
  db EXIT

; Ans forth compare is ( cA n cB m -- 0/-1/1 )
; zero means strings are equal -1 1st is "smaller" 1 opposite
; I should rename this ncompare to get out of the way of
; the standard forth word. Or this should be called nstr=
; because Ans compare returns more than a flag.
ncompare.doc:
  ; db ' ( a A n -- flag) '
  ; db ' given 2 pointers to strings a and A '
  ; db ' compare the 2 strings for n bytes '
  ; db ' and put -1=true on stack as flag if the strings are '
  ; db ' the same or false=0 on stack if the strings '
  ; db ' are different. '
  ; dw $-ncompare.doc
ncompare:
  dw copy.p
  db 'ncompare', 8
ncompare.p:
                   ; a A n
  db RON           ; a A   r: n /n loop counter
  db CFETCHPLUS    ; a A+1 [A]
  ; db DUP, EMIT     ; debug
  db SWAP          ; a [A] A+1
  db RON, RON      ; a           r: n A+1 [A]
  db CFETCHPLUS    ; a+1 [a]     r: n A+1 [A]
  ; db DUP, EMIT     ; debug
  db ROFF          ; a+1 [a] [A]  r: n A+1
  db EQUALS        ; a+1 flag     r: n A+1
```

```
    db JUMPT, 10      ; a+1          r: n A+1
    db ROFF, ROFF     ; a+1 A+1 n
    db DROP, DROP, DROP    ; clear stacks
    db LIT, 0         ; flag=0 (false)
    db EXIT
    db ROFF           ; a+1 A+1      r: n
    db RLOOP, -18     ; a+1 A+1      r: n-1
    db ROFF           ; a+n A+n 0
    db DROP, DROP, DROP    ; clear stacks
    db LIT, -1
    db EXIT

  type.doc:
    ; db ' ( adr n -- ) '
    ; db ' Prints out n number of characters starting '
    ; db ' at address adr. '
    ; dw $-type.doc
  type:
    dw ncompare.p
    db 'type', 4
  type.p:
                    ; adr n
    ;*** if count zero, do nothing
    db DUP          ; adr n n
    db JUMPNZ, .sometext-$  ; adr n
    db DROP, DROP   ; clear data stack
    db EXIT
  .sometext:
    db RON          ; adr        r: n
  .nextchar:
    db CFETCHPLUS   ; adr+1 c    r: n
    db EMIT         ; adr+1      r: n
    db RLOOP, .nextchar-$ ; adr+1      r: n-1
    db ROFF, DROP, DROP   ; clear stacks
    db EXIT

  %if 0
  ; in source, so can delete this bytecode.
  %endif

  in.doc:
    ; db ' ( -- adr )
    ; db 'Puts on the stack the address of the '
    ; db 'current input source/ buffer '
    ; dw $-in.doc
  in:
    dw type.p
    db 'in', 2
  in.p:
    db LITW
    dw in.d
    db EXIT
  in.d: dw 0       ;  need to initialize
  in.length: dw 0  ;  need to initialize

  term.doc:
    ; db ' ( -- adr )
    ; db 'Puts on the stack the address of the '
    ; db 'user input buffer (terminal buffer). This'
    ; db 'is a common source for interpreting and '
    ; db ' compiling '
    ; dw $-term.doc
  term:
    dw in.p
    db 'term', 4
  term.p:
    db LITW
    dw term.d
    db EXIT
  term.d: times 128 db 0   ; counted buffer for user input

  dotin.doc:
    ; db ' ( -- )
    ; db 'display the contents of the current input stream '
    ; db 'or buffer.  '
    ; dw $-dotin.doc
  dotin:
    dw term.p
    db '.in', 3
  dotin.p:
    ; change this because the input stream is not always a
    ; counted string
    db LITW
    dw in.d          ; adr
    db COUNT         ; a+1 n
    db DUP           ; a+1 n n
    db FCALL
    dw udot.p         ; a+1 n
    db LIT, ':', EMIT ; a+1 n
    db FCALL
    dw type.p
    db EXIT
  ; : in.. in count dup u. sp type ;


  toin.doc:
    ; db ' ( -- adr n ) '
    ; db ' put on stack parse position in input stream'
    ; db ' and number of characters remaining in stream. '
    ; db ' This is used with parse etc'
    ; dw $-toin.doc
    ; was a strange bug with this link
  toin:
    dw dotin.p
    db '>in', 3
  toin.p:
    ;** need to remember that in.length, inp.d and in.d are
    ; pointers and need to be fetched before use... *p
    ;
    db LITW
    dw inp.d          ; adr
    db FETCH           ; >in
    db DUP             ; >in >in
    db LITW
    dw in.d            ; >in >in in.d
    db FETCH           ; >in >in in
    db MINUS           ; >in offset
    db LITW
    dw in.length       ; >in offset adr
    db FETCH           ; >in offset [in.length]
    db SWAP            ; >in length offset
    db MINUS           ; >in remainder
    ;db FCALL
    ;dw dotstack.p
    db EXIT

  ; a pointer to the current parse position in the input stream
  inp:
    dw toin.p
    db 'inp', 3
  inp.p:
    db LITW
    dw inp.d          ; A
    db EXIT
  inp.d: dw 0

  ; maybe call this setin or in! instore or in+
  ; why not just get adr and n from >word?
```

```
inplus.doc:
  ; db ' ( adr n -- ) '
  ; db ' update word and parse position in input'
  ; db ' where the start of the word is given by pointer'
  ; db ' adr and the length of the word is n. The parse '
  ; db ' position will be adr+n after this call '
  ; db ' eg: pad resetin pad accept >in parse 2dup type '
  ; db '    atin >in .s  etc'
  ; dw $-inplus.doc
inplus:
  dw inp.p
  db 'in+', 3
inplus.p:
  ; probably should check here that the new
  ; parse position is not beyond the end
  ; of the input stream (length)
                  ; adr n
  db SWAP, DUP      ; n adr adr
  db LITW
  dw toword.d       ; n adr adr a2
  db STORE          ; n adr
  db PLUS           ; n+adr
  db LITW
  dw inp.d         ; n+adr ap
  db STORE          ;
  db EXIT

; we dont actually use the toword.p proceedure.
resetin.doc:
  ; db ' ( adr n -- ) '
  ; db ' set word and parse position to 0 in input'
  ; db ' and set the input buffer to point to address '
  ; db ' adr and the stream length to n. '
  ; db ' set vars in.d in.length inp.d toword.d '
  ; dw $-resetin.doc
resetin:
  dw inplus.p
  db 'in0', 3
resetin.p:
                ; adr n
  db LITW
  dw in.length    ; adr n in.length
  db STORE        ; adr
  db DUP          ; adr adr
  db LITW
  dw in.d         ; adr adr in.d
  db STORE        ; adr
  db DUP          ; adr adr
  db LITW
  dw inp.d       ; adr adr inp.d
  db STORE        ; adr
  db LITW
  dw toword.d     ; adr word.d
  db STORE        ;
  db EXIT

anon.doc:
  ; db ' ( -- adr )
  ; db 'Puts on the stack the address of the '
  ; db 'buffer to hold anonymous definitions. This '
  ; db 'contains compiled byte code for user input '
  ; dw $-anon.doc
anon:
  dw resetin.p
  db 'anon', 4
anon.p:
  db LITW
  dw anon.d
  db EXIT
anon.d: times 128 db 0    ; compiled byte code

; Another idea instead of obuff is just to use memory after the
; last word in the dictionary. Eg:
; : obuff here 128 + ;
; But then, we would have to write an EXIT after every opcode
obuff.doc:
  ; db ' ( -- adr )
  ; db ' A buffer used to execute opcodes which have been '
  ; db ' called in immediate state.
  ; dw $-obuff.doc
obuff:
  dw anon.p
  db 'obuff', 5
obuff.p:
  db LITW
  dw obuff.d
  db EXIT
; this will contain one opcode and EXIT, so only 2 bytes
obuff.d:
  db NOOP, EXIT
; times 4 db 0  ;

buff.doc:
  ; db ' ( -- adr )
  ; db ' a testing buffer'
  ; dw $-buff.doc
buff:
  dw obuff.p
  db 'buff', 4
buff.p:
  db LITW
  dw buff.d
  db EXIT
buff.d: times 64 db 0

drive.doc:
  ; db ' ( -- adr )
  ; db ' a variable to hold virtual drive number '
  dw buff.p
  db 'drive', 5
drive.p:
  db LITW
  dw drive.d     ; ad
  db EXIT
drive.d: db -1

sides.doc:
  ; db ' ( -- adr )
  ; db ' how many sides or platters a disk has. '
  ; db ' This information may be needed for disk reads. '
  dw drive.p
  db 'sides', 5
sides.p:
  db LITW
  dw sides.d
  db EXIT
sides.d: dw -1

sectorspertrack.doc:
  ; db ' ( -- adr )
  ; db ' How many sectors each track has. eg 18 for floppy '
  ; db ' This information may be needed for disk reads. '
  dw sides.p
  db 'sectors', 7
sectorspertrack.p:
  db LITW
  dw sectorspertrack.d
  db EXIT
```

```
sectorspertrack.d: dw -1

lib.doc:
  ; db ' ( -- adr )
  ; db ' some source code to load'
  ; dw $-lib.doc
lib:
  dw sectorspertrack.p
  db 'lib', 3
lib.p:
  db LITW
  dw lib.d
  db EXIT
lib.d:
   db lib.end-$-1   ; number of characters (used by source and type)
   ; block1 is the sector (512 byte block) of the first
   ; source code block (forth block=1K bytes) on disk
   ; actually parse and compile this source code
   ; use clock to time compile time

   db ' 52 emit 116 emit 104 emit 33 emit 13 emit 10 emit'  ; message "4th!"
   db ' clock '
   ; load the 0 block, which loads all the others
   db ' block1 2 first read drop first 1024 source '
   db ' clock '
   ;db ' timer cr .s '

lib.end:

  ; A

onorder.doc:
  ; db ' ( A -- )
  ; puts the vocabulary at A on top of the search order
  ; A points to the data field of the vocabulary. If there are
  ; 2 or less vocabs in s/o then the new one is appended, otherwise
  ; it replaces the top vocab in s/o
onorder:
  dw lib.p
  db '>s/o', 4
onorder.p:
  ; A
  db FCALL
  dw searchorder.p
  db DUP, FETCH
  ; check if |s/o| < 3 ( Forth and Vlist are minimum wordlists)
  db DUP, LIT, 3, LESSTHAN
  ; A s/o |s/o| Flag
  db JUMPF, .morethantwo-$
  db INCR, OVER, STORE
  ; A s/o
  db DUP, FETCH
  ; A s/o |s/o|
.morethantwo:
  ; A s/o |s/o|
  db TIMESTWO, PLUS
  ; A S'
  db STORE
  db EXIT
  ; or maybe if search order is less than 2 add, not replace

; A vocabulary. It may not even need a "head" namefield etc.
; when "create" runs it should update this link instead of
; last.d  So, initially, before create has run, the last
; word in the dictionary (bytecode) is "last.p". To get a pointer
; we must do current @ @ or else FCALL, last.p
;
; This is the "Forth" vocabulary or wordlist. It is the most important
; along with "Vlist" (which is a list of wordlists) and is always
; available to be searched and executed with nfind.p (via wfind and find.so)
; But it only becomes available when
;   ' find.so is nfind
; is executed. This word ("Forth") will probably be in the Vlist
; dictionary/vocab/wordlist/namespace.

forth.doc:
  ; db ' ( -- )
  ; db 'A vocabulary. Makes Forth 1st in search order'
forth:
  ;This will be zero when Forth and Vlist are the only
  ;initial wordlists in Vlist
  dw 0
  ;dw onorder.p
  db 'Forth', 5
forth.p:
  db JUMP, 6  ; jump to continue
  ; these nops are here because thats how vocabs will
  ; be compiled in source code. (When does> resets the
  ; code field, it leaves a gap)
  db NOOP, NOOP
forth.d: dw last.p
  ; here we set forth.d as the value
  ; in searchorder.d+2.
  db LITW
  dw forth.d
  ; A

  db FCALL
  dw onorder.p
  db EXIT

vlist.doc:
  ; db ' ( -- )
  ; db 'A wordlist which is a list of wordlists !!'
vlist:
  dw forth.p
  db 'Vlist', 5
vlist.p:
  db JUMP, 6  ; jump past datafield
  ; these nops are here because thats how vocabs will
  ; be compiled in source code. (When does> resets the
  ; code field, it leaves a gap)
  db NOOP, NOOP
vlist.d: dw vlist.p
  db LITW
  dw vlist.d
  ; A
  db FCALL
  dw onorder.p
  db EXIT

%if 0
%endif

current.doc:
  ; db ' ( -- adr )
  ; db 'Puts on the stack a pointer to current wordlist '
  ; db 'for new definitions. The word "last" will just
current:
  ;dw vlist.p
  ; vlist and forth vocabs are in the vlist dictionary/vocab
  dw onorder.p
  db 'current', 7
current.p:
  db LITW
  dw current.d      ; ad
  db EXIT
; actually current.d needs to point to a wordlist datafield
```

```
   ; not to the actually last word in that list.. This is
   ; so "order" can display what is the current wordlist
   ; So last can do "current @ @" to actually get a pointer ??
   ; to the last word in the list, but we cant do "A last !"
   ; anymore?. This is confusing, but pointers to pointers always
   ; are.
   current.d: dw forth.d

   ; Ans forth says this buffer must be >= 8 cells
   searchorder.doc:
     ; db ' ( -- A / search order buffer *)'
   searchorder:
     dw current.p
     db 's/o', 3
   searchorder.p:
     db LITW
     dw searchorder.d
     db EXIT
   searchorder.d:
     ; initial s/o is just "Forth" vocab ( in bytecode )
     dw 2, vlist.d, forth.d
     times 8 dw 0

   context.doc:
     ; db ' ( -- A )
     ; db 'Puts on the stack a pointer to the 1st wordlist in the '
     ; db 'search order.'
   context:
     dw searchorder.p
     db 'context', 7
   context.p:
     db FCALL
     dw searchorder.p
     db DUP, FETCH, TIMESTWO, PLUS
     db EXIT
   ; context.d: dw forth.d

   ; Last points to the last word of the "current" definition namespace
   ; (wordlist/vocabulary)
   ;
   ; What happens if wordlist is empty?
   ; Often called "latest"?
   last.doc:
     ; db ' ( -- adr )
     ; db 'Puts on the stack a pointer to (xt) address of the last word '
     ; db 'in the dictionary. This changes when new words are  '
     ; db 'added via colon : definitions or other defining words '
   last:
     dw context.p
     db 'last', 4
   last.p:
     db LITW
     ;dw last.d      ; ad
     dw current.d
     db FETCH
     db EXIT
   ;last.d: dw last.p

   code:

     db JUMP, .system-$

  .system:
     ; lib.p just loads the 1st block (block 0) which loads the
     ; other disk source code blocks and then compiles and runs
     ; 'shell' which is the repl interpreter (normally called 'quit'
     ; in forth)
     db FCALL
     dw lib.p
     db COUNT  ; source takes address + length
     db FCALL
     dw source.p
     db 0

start:

   mov ax, cs      ; cs is already correct (?!)
   mov ds, ax      ; data segment

   ;*** save the (virtual) drive we have loaded
   ;    code from. Handy for disk writes later

   ; have to set data segment DS first
   ; get the disk geometry parameters and save for use by
   ; read.x
   ; code from mike gonta. This fixed read problems where there
   ; are more than 18 sectors on a track etc.
   mov [drive.d], byte dl ;
   mov ah, 8
   int 13h
   ; jc .diskerror
   and cx, 3Fh
   mov [sectorspertrack.d], cx
   movzx dx, dh
   add dx, 1
   mov [sides.d], dx

   ; point es:di directly after the code and data segment
   ; i.e. after the 8 sectors (8 * 512 bytes)
   ; which contain code and data. We will use es:di
   ; as the return stack pointer. When
   ; a value is pushed on the return stack, the value
   ; is written to [es:di] and di is incremented by 2

   ; add ax, 256      ; 256 * 16 = 4096, 4K (8 sectors)
   ; put a gap of 4K between code and stacks for
   ; dictionary entries etc
   ;*** 8K code + 8K gap then rstack. But are the data and
   ; return stacks growing towards each other???

   ;add ax, 512     ; 512 * 16 = 8K
   add ax, 1024     ; 1024 * 16 = 16K
   mov es, ax       ; using es:di as return stack pointer
   mov di, 0

   ; the calculations are as follows
   ; we have loaded 16 sectors = 16 * 512 bytes = 9162 bytes == 8K
   ; we want a data stack of size 4K
   ; (which is big) = 4094 bytes
   ; also we want a return stack of size 4K/8K
   ; for hefty recursive functions, although these
   ; huge sizes are not necessary.
   ; x86 hardware stack grows up or down? ...
   ; divide by 16 because that is how segment
   ; addressing works
   ; That is: if we multiply the number in ss or es
   ; or ds by 16, we get a absolute memory address

   ;*** ax is pointing to start of the rstack so
   ;*** add 4K more for data stack
   add ax, 256      ; 256*16=4K
   mov ss, ax       ; a 4K stack here
   mov sp, 4096     ; set up the stack pointer

   push code
   call exec.x

stayhere:  jmp stayhere
```

```
  ;*** new words can be compiled here
  ;
  dictionary: dw 0

  ; Pad remainder of n( = n/2 K) sectors with 0s
  ; The number below (6,7,8 etc) only has to be as big
  ; as the dictionary.
  times (6*1024)-($-$$) db 0

  ; MEMORY MAP (may 2018)
  ;  To avoid confusion, remember the clear distinction between
  ;  the disk map (code and data on disk) and the ram memory map.
  ;
  ;  need to clarify this memory map.
  ;  This may change as code grows, but the idea is to keep code small
  ;  The addresses below are segmented, so multiply the first part by
  ;  16 to get the real address.
  ;
  ;   address     contents
  ;   -------     --------
  ;   1000:0000   8K of code and data, including the dictionary
  ;               and any new words defined by colon :
  ;   ??:0000     8K return stack pointed to by es:di
  ;   ??:0000     4K data stack pointed to by ss:sp
  ;

  ; this is silly, can erase memory without writing to disk
  ; in fact, is there any need to erase stack memory?
  ; times 4096 db 0

  ;*** some text/forth code at sector ? which we can load
  ;*** with source.p or with load opcode
  diskcode:

  ; can load this with "block1 2 first read first 1024 source"
  block1:

  ; see os.sed for a proprocessing sed script.
  ; This script allows us to write multiline
  ; forth source code without the "db ’ " guff.

 %if 0 ; code{

  ; switch to video mode 16 (x86 bios colours, no cursor), not portable
  ; or text mode 3
  3 vid

  ; standard ’ tick (?). There is another word TICK which works
  ; a bit differently since it returns a flag as well indicating
  ; if the word is a number, opcode or fcall.

  ; But really this should just be the same as [’] I think,
  ; otherwise it executes before other commands interactively.
  : ’ wparse nfind ; imm

  ; if call, is not immediate then this gets simpler
  ; does it need to be immediate?

  ; ans forth ’postpone’ word. Compiles a call to a word, even if
  ; it is immediate. This is used in words like [char] but is tricky
  ; to think about somethimes.
  : post wparse nfind ’ call, call, ; imm

  ; a ’ to use in colon defs
  : [’] wparse nfind post literal ; imm

  ; makes all words execute immediately until ] ’
  ; This word sets the state variable to true (immediate) ’
  ; so that all words parsed will execute immediately ’
  ; without being compiled. ’
  : [ 1 state ! ; imm

  ; This word sets the state variable to false (normal)
  ; so that all words parsed will be compiled unless they
  ; have their immediate control bit set.
  : ] 0 state ! ; imm

  ; Put the value of the following character on the stack
  : char wparse drop c@ ; imm
  ; standard [char]
  : [char] post char post literal ; imm

  ; Hopefully this will save space
  ; 2r> is an opcode now (19 mar 2019)
  ; : 2R> r> r> swap r> swap >r ;

  ; comments like ( -- )
  : ( [char] ) parse 2drop ; imm

  ; integer division
  : / /mod swap drop ;

  ; type a space
  : space ( -- ) 32 emit ;

  : count c@+ ;

  ; classic forth variables.
  ; The does> is a hack because there is a bug in using create in
  ; defining words. does> forces the dp point to be updated to ’here’
  : var create 0 , does> ; imm
  ; definition of CONSTANT
  : con create , does> @ ; imm

  ; a simple ’block’ word that just reads block n into first buffer
  ; in a proper implementation this would use ’buffer’ to flush
  ; code to disk and then read the block. Also, the block would only
  ; be read if it wasnt already loaded to ram (at ’first’)
  ; ans blk variable
  ; Maybe blk is only updated in the "load" word according to
  ; ansi forth?
  var blk
  : block
      ; ( n -- A )
      dup blk !
      2* block1 + 2 first read drop first ;

  ; The last block loaded. cryptic name because of disk block
  ; space limitations.
  var lb

  ; perhaps source and load should return a flag indicating
  ; success or failure. This may allow us to catch errors
  ; But have to dig down to "inputcompile" .
  : load ( n -- ) dup lb ! dup . block 1024 source ;

  ; get the loop counter, needed for thru, called I in ansi forth
  ;: ii r> r> r> dup >r swap >r swap >r ;
  : ii r> r> dup >r swap >r ;

  ;: compile> wparse tick item, ; imm

  ; lookup a word and return the opcode or zero
  : op’ wparse nfind xt>op ; imm
  ; use this op’ in : defs
  : [op’] post op’ post literal ; imm
```

```
; These words should be called op: and [op:] because
; comma words eg , c, get their argument from the stack
; not from the input stream.
;

; use this in definitions
; This seems archane: And we need a simpler syntax
; for compiling FCALLs  eg [call:] <name>

; This is working!
; Another archane word. Compile an fcall to a word
: [call:] post [’] [’] call, post call, ; imm

; more elaborate versions of [op:]
; : [op:] post op’ post literal ’ c, literal post call, ; imm
;: [op:] post [op’] ’ c, literal post call, ; imm
; : [op:] post [op’] [’] c, post call, ; imm

: [op:] post [op’] [call:] c, ; imm

; (run: limit start -- r: start limit )
; (comp: -- ’here’ )
; At run-time: puts start and limit on return stack
; at compile time: marks a jump back address for loop etc
; the jump address is left on the data stack
; Source code do
;: do [op’] >r dup c, c, here ; imm

; The code below seems to be working. The phrase "[op:] >r"
; will compile the opcode ">r" when "do" runs (not when it
; compiles. So this is a little like a double "post"

; compile >r >r to get loop parameters on the return stack
; and then leave jump-back position on the stack for "loop" to use.

; swapped order of params (counter ontop of return stack)
;: do [op:] swap [op:] >r [op:] >r here ; imm
; 2>r is an opcode now! So can do "[op:] 2>r"
: do [op:] swap [op:] 2>r here ; imm

; increment and check the loop parameters at runtime
; for swapped params to the 1+ a bit earlier. And "ii" will
; become smaller too.

;: (check) r> r> r> 1+ 2dup >r >r = swap >r ;
; could combine into one function (loop), that is, drop the
; rstack params if limit is reached.
; : (check) r> r> 1+ r> 2dup >r >r = swap >r ;
: (check) 2r> 1+ r> 2dup 2>r = swap >r ;

; 2drop is an opcode now.
: (clean) r> 2r> 2drop >r ;

: Loop
  ;( short jump loop )
  [call:] (check)
  [op:] jumpz
  here 1- - c,
  [call:] (clean) ; imm

; a "Forth Inc" word to load blocks x to y
; ( x y -- ) ... load blocks x to y
; here we need to chekc if x=y etc
: thru
   ; ( a b -- )
   ; there is no if yet!!
   ; 2dup = if 2drop exit fi
   swap do ii load Loop ;
```

```
; load required blocks. These phrases (not defining a word) seem
; to get executed last, no matter where they are in the block.
 1 9 thru
;1 load 2 load 3 load

clock

; load the shell block (defines shell and starts it)
33 load

; pad remainder of 1st disk block with zeros
times 1*1024-($-block1) db 0
; block1:


; like the old figforth variables. usage: 31415 var pi
: Var create , does> ; imm

; standard forth comment
: \ 10 parse 2drop ; imm

; This is working!
; Another archane word. Compile an fcall to a word
: [call:] post [’] [’] call, post call, ; imm

;  ( comp: -- A )
;  marks a jump back address for until/again etc’
;  the jump address is left on the data stack ’
: begin here ; imm

; source version (ljump long relative jumps)
;  ( comp: jb -- ) ( run: -- )
;  jumps back at runtime. At compile time
;  gets the absolute jump-back address from the data stack and compiles
;  it a relative ljump.
: again
    ; A (Address of compiled ’begin’)
    ; compile "ljump" at current compile point
    [op:] ljump here 1- - , ; imm
    ; stack-progression is:
    ; A here
    ; A here-1  /align to ljump instruction
    ; A-here-1
    ; compile relative ljump address (2 bytes)

; compile an execution token, either as an opcode or else
; as an FCALL
; : xt, dup [’] nop 1+ < if , else  ;

; Below a standard forth "if" word. Used to be in byte code.
; Need to use a jumpnz/ljump combination here
;
; compiles the "jumpnz" opcode and then a dummy relative
; jump address (+2). This address will be replaced with the
; real relative jump address when "fi" runs.
; I think I will have to use ljumps here to overcome the
; +/- 128 byte limit
; At runtime, if the value n on the stack is zero ’
; skip statements after this until next "fi" ’
; compiles a jumpzero and puts the current ’
; address on the data stack. The "fi" command consumes ’
; that address ’

; A new version of if using long jumps. This wont work until
; fi and else are adjusted to compile 2 byte target addresses.
; eg: jumpnz 4, ljump 2
;  where the "2" will be replaced by the real target when
;  "else" or "fi" is encountered. The argument to ljump is 2bytes long
: if
```

```
      ; ( runtime: n -- /ctime: -- A /compile a conditional long jump *)
      ; compile "JUMPNZ, 5".
      ;[op’] jumpnz c,
      [op:] jumpnz 5 c,
      ; compile "LJUMP, 2"
      ; The 2 (2 bytes will be replaced with a
      ; real target when "else" or "fi" executes.
      [op:] ljump
      ; Put current compilation position on the stack to be used
      ; by next "else" or "fi"
      here
      2 , ; imm

   : else
     ; compile "LJUMP, 2". The 2 will be replaced by "fi"
     [op:] ljump
     ; leave H on stack for fi to use
     here
     ; temporary jump-to-fi target
     2 ,
     ; Aj H
     swap dup here
     ; H Aj Aj H
     swap - 1+
     swap ! ; imm

   ; A new version of fi using long jumps
   ; run-time: ( -- ) compile-time: ( A -- )
   ; At compile time obtains the correct jump
   ; address from the data stack and compiles the correct
   ; target address into a previously compile "if" clause.
   ; This word is called "then" in traditional forths.
   : fi
       dup here
       ; A A H
       swap - 1+
       ; A H-A+1
       swap ! ; imm

   ; just testing new if/else/fi
   ; : ff iff 65 emit eelse 64 emit ffi ;
   ; )

   ; This defer cannot handle opcodes, which is not good. See
   ; another implementation of defer below.
   ;
   ; An alternative is to use a less elegant, but effective, method
   ; of actually compiling code into the word with "is"
   ; For an opcode we compile in OP, EXIT and for a normal word
   ; we compile FCALL, <address>, EXIT. op/exit is 2 bytes, and
   ; fcall/address/exit is 4 bytes. So deferred words will
   ; take up more room this way.
   ; standardish defer word. If the deferred word is not
   ; initialised then it should just do nothing. eg
   ; : defer create 0 , does> @ dup if pcall else drop fi ; imm
   : Defer create 0 , does> @ pcall ; imm
   ; The "4 +" below is a cludge. In my implementation
   ; of does> the data field starts 4 bytes after the execution address.
   ; This is because "create" adds code to push parameter field address
   ; onto the stack, and does> then modifies that code. But there
   ; should be a more elegant way to do this.
   ;
   ; usage:
   ;     defer all ’ ls is all
   ;  or: : print [’] printer is type ;
   ; Also, "is" should check that name and xt are valid
   : Is
      ;( xt -- / set xt for <name> to xt *)
      wparse nfind 4 + ! ; imm

   ; a new defer, to work with opcodes, compile a new name
   ; and then an exit and then allow 3 more bytes
   ; The -4 here+ should reset the code put in by "create"
   ; to push the data/parameter field address onto the stack.
   ; But there should be a more elegant way to do this, I think
   ;

   ; Another way is to define Defer with a buffer: ??
   : defer create reset [op:] exit 0 , 0 c, code>here ; imm

   ; This is much more laborious than the "does>" style defer/is
   ; but it can handle opcodes, which is important I think.
   ; We dont have any good words for compiling ops/calls to
   ; any memory (only to "here/dp") eg:
   ; [op:mem] ( A <name> -- ) etc

   ; either compile OP+EXIT or FCALL+<address>+EXIT
   : is wparse nfind
     ; XT xt
     swap dup
     ; xt XT XT
     xt>op dup if
       ; xt XT op
       swap drop
       ; xt op
       swap c!+
       ; xt+1
     else
       ; xt XT 0
       drop swap
       ; XT xt
       [op’] fcall swap
       c!+
       ; XT xt+1
       !+
     fi
     ; xt’
     [op’] exit swap c! ; imm


   ; allow us to use different versions of accept
  defer accept
   ; ’ accept.byte is accept

   ; so I dont have to type so much
   : A accept ;

   ; type a newline
   : cr 13 emit 10 emit ;

   : xt+
       ; ( xt - xt’ / get exec address of next word in dict *)
       1-
       ; xt’ n
       rcount drop 1- 1-
       ; xt’-2
       @ ;

   ; ans standard tuck. This can be a handy way to save a value
   ; for later processing. Could be an opcode
   : tuck  ( a b -- b a b )
       swap 2dup drop ;

   ; get a copy of topmost return stack item. We need to dig
   ; under the word return pointer. Probably should be opcode.
   ; This is now an opcode, as "r@"
   : R@
     ; ( -- n ) ( R: n -- n )
     r> r> dup >r swap >r ;

   ; just for testing long jump if
   ; : iff 0 if 60 emit fi ;
   ; see iff

   ; determine the used size of a block by searching for the 1st
   ; zero byte. Use like this: "first bsize"
   ; : bsize dup begin c@+ 0 = until swap - ;

   ; Increment a counter variable by 1 and leave the new value
   ; on the stack, because often we want to use the counter value
   ; immediately afterwards. This is like C
   ;    nn++ (in forth: nn ++ )

   : ++ ( A -- [A]+1 / increment counter *)
       dup dup @
       ; A A n
       1+ swap
       ; A n+1 A
       ! @ ;
       ; n+1

   : -- ( A -- [A]-1 / minus 1 value at A *)
      dup dup @ 1- swap ! @ ;

   ; counter variables
   var nn
   var ll

   ; an implementation of the standard forth word 'I' which gets
   ; a copy of the loop counter onto the data stack. We have to
   ; 'dig' under the current word return pointer which is the
   ; current top of the return stack and also under the loop limit
   ; parameter

   ; : ii++ r> r> r> 1+ >r >r >r ;

   ; use this before 'exiting' from a do loop.
   : unloop r> 2r> 2drop >r ;

   ; discard loop parameters and continue execution after next 'loop'
   ; : leave ( -- ) ;

   ; just testing quote preprocessing by os.sed
   ; : quote char " char ' char " ;

   ;: [op:] wparse nfind xt>op literal c, ; imm

   ; pad remainder of 2nd block with zeros
   times 2*1024-($-block1) db 0
    ; block2
    ; 3rd disk block

   ; An alias for [char] that is shorter
   ; : c: post char post literal ; imm
   : c: post [char] ; imm

   ; by putting for/next initial value on the stack twice
   ; (and only decrementing one copy), we can use the same
   ; "unloop", "ii" as do/loop. Also we can use a word ii'
   ; which is a reverse counter for for/next
   : for ( run: u -- / iterate u times *)
       [op:] dup [op:] >r [op:] >r here ; imm

   : next ( short jump back to "for" )
       [op:] rloop
       here 1- - c,
       [call:] (clean) ; imm
```

```
  ; (comp: 'here' -- ) (run: -- )
  ; at run-time: takes loop parameters off stack
  ; increments the iterator, checks if it is = to limit
  ; and jumps back to DO if not. If = then it removes loop
  ; parameters from the stack and discards them.
  ; at compile-time: get begin address from data
  ; stack and compiles a rloop jump
  ; back to begin (address on data stack).

: loop ( long jump loop )
  [call:] (check)
  [op:] jumpnz 5 c,
  [op:] ljump
  here 1- - ,
  [call:] (clean) ; imm

; a modulus operator
: mod /mod drop ;

 ; the last block for searching etc.
 var final 84 final !

 ; parse and find words, more or less working
 ; : findwords term accept term count in0
 ;     begin
 ;      wparse 2dup type space find dup u. space
 ;      0 =
 ;    until ;

 ; frees a buffer (eg first) and writes old contents to disk if
 ; necessary.

 ;: buffer ( n -- adr //to do! ) ;

 : todo! nop ;

 ; logical not! same as 0=
 : not ( n -- f )
    ; leave true [1] if n=0, else leave false [0]
    if 0 else 1 fi ;

 ; 0= is now an opcode
 : zero= ( n -- f )
    ; leave true [1] if n=0, else leave false [0]
    if 0 else 1 fi ;

 ; saves the current buffer (first) to block number n on disk
 : save ( n -- flag=-1/0/1 ) 2* block1 + 2 first write ;

 ; advance the input parse position by 1 character
 : in++ inp ++ drop ;

 ; standard s" word. The in++ skips over the leading space
 : s" [char] " in++ parse post sliteral ; imm

 ; An fcall <type> will get compiled when ." is run,
 ; not when it is compiled. This is an important technique
 ; It is like a "double postpone"
 ;
 ; more elaborate versions of ." ...
 ;: ." post s" [ wparse type nfind ] literal post call, ; imm
 ;: ." post s" [ ' type ] literal post call, ; imm
 ;: ." post s" ' type literal post call, ; imm
 ;: ." post s" ['] type post call, ; imm

 : ." post s" [call:] type ; imm

 ; a convenience to get rid of stuff on the stack
```

```
    ; this is an opcode now (2drop)
    : 2DROP ( a b -- ) drop drop ;

    : 4drop ( a b c d -- ) 2drop 2drop ;

   ; A new and improved "cmove" (see block 2) that is
   ; correct even when data areas overlap
   ; here use if A > B then copy from low to high
   ; but if B > A then copy from high to low. This will be used
   ; when the backspace key is pressed in an editor

   ; Actually! forth.inc forths use cmove and cmove>
   ; cmove> copies to overlapping address in higher memory.
   ; Having 2 words speeds up the copy. The programmer has to
   ; decide if the memory areas overlap and if the target is
   ; in higher mem. If so, use cmove>
   : cmove ( A B n -- / copy n bytes from A to B *)
      ; A B n
      ; If n==0 then crash out.
      dup 0 = if drop 2drop exit fi
      ; A B n
      >r 2dup
      ; A B A B    r: n
      ; if A=B then nothing to do
      = if r> drop 2drop exit fi
      ; A B        r: n
      2dup < if
        ; A < B
        ; A B      r: n
        r@ + 1- swap r@ + 1- swap
        ; A+n-1 B+n-1
        r> 0 do
          ; A+n-1 B+n-1
          ;>r c@- r>
          ; c@-
          >r
          dup c@ swap 1- swap
          r>
          ; A+n-1 c B+n
          ; this is c!-
          dup >r c! r> 1-
          ; A+n-1 B+n-1
        loop 2drop
      else
        ; A > B this works for overlapping data areas
        ; A B      r: n
        r> 0 do
          ; A B
          >r c@+ r>
          ; A+1 c B
          c!+
          ; A+1 B+1
        loop 2drop
      fi ;

   : c> post [char] [op:] emit ; imm

 times 3*1024-($-block1) db 0
   ; start block3:


   ; the 2-2+ trick is to avoid the loop parameters for
   ; for/next which are not interesting, or else use begin/end
   ; with a counter. Cant call this .r because that is right
   ; justified print in standard forth.
   : .rs ( show return stack )
     ;rdepth c: < emit u. c: > emit
     rdepth 2 - for ii 2 + rpick u. space next ;
```

```
   ; handy to compile an opcode by name eg: op: fcall
   ; this can be handy for opcodes that arent normally compiled.
   ; eg jump, jumpz, rloop etc
   ; we may also need code that compiles an opcode at runtime
   : op: wparse nfind xt>op c, ; imm

   ; a non standard type of comment that reads until a new line
   ; this should be EOL not 10 for portability
   : ./ 10 parse 2drop ; imm

   ; a dodgy opcode test based on being less than the xt of "nop"
   ; in the future this will not be true because new opcodes can
   ; be added to the system, probably in their own namespace/wordlist.
   ; just an alias for xt>op
   : opxt? ( xt -- F ) xt>op ;

   ; The simplest dotxt version
   ; : .xt ( A -- / print wordname of xt *)
   ;     1- rcount type ;

     ;   : bcode ( xt -- F )
       ; check if xt is bytecode
       ;dup opxt? IF drop 0 exit FI
       ;last < IF 0 exit FI nop ;

    : .xt ( A -- / print colour coded xt name *)
        dup 0= if 13 fg ." [xt=0]" 7 fg drop exit fi
        ; light blue for normal words
        11 fg
        ; green if words are defined in bytecode
        dup [’] last < if 10 fg fi
        ; Immediate words are red
        dup imm? if 12 fg fi
        ; here make word xt>op which leaves flag
        ; Opcodes are brown
        dup opxt? if 6 fg fi
        1- rcount type
        ; return colour to normal
        7 fg ;

   ; if0 (long jump), does consume top stack item
   : if0 [op:] jumpz 5 c, [op:] ljump here 2 , ; imm

   ; can make a simple alias for this immediate word like
   : ifnot post if0 ; imm

   ; this conserves the current base
   : .hex ( n -- / display top-of-stack as unsigned hexadecimal *)
        base @ swap 16 base c! u.
        ; restore original base
        base ! ;

   ; fill u bytes of memory with 0 starting at addr
   : erase ( A u -- )
       0 do 0 c!+ loop drop ;

   ; fill u bytes of memory with byte b starting at addr
   : fill ( A u b -- )
       swap 0 do dup c!+ loop 2drop ;

   : dumpw ( A u -- / display u words of data *)
       ; A u
       0 do
         ii
         ; A ii
         ; print memory address and newline every 8 chars
         8 mod if0
           cr 11 fg dup u. ." :" 15 fg
         fi
        ; A
        @+ u. space
        ; A+2
      loop drop ;

  ;  allocate u bytes of data-space, beginning at the next available
  ;  location. Normally used immediately after "create".
  : allot ( n -- )
    here + here! ;

  ; allot data space and initialize to 0
  : allot0 ( n -- )
      here swap 2dup erase + here! ;

  ; create a data buffer <name> in the dictionary of n bytes
  : buffer: ( n -- )
       create allot0 does> ; imm

  ; convert from n to cells (in this case 2 bytes)
  : cells ( n -- ) 2* ;

  ; print some machine info
  ;: mach ( -- )
  ;  machine count 2dup ." name: " type cr
  ;  + count ." signature: " type cr ;

  ; print the machine name
  : mname ( -- ) machine count type ;

  times 4*1024-($-block1) db 0
    ; block4:

    ; this is an opcode now
    : ROT ( a b c -- b c a )
      >r swap r> swap ;

    : -rot ( a b c -- c a b ) rot rot ;

    ; some forths just define pad like this, it is not
    ; a permanent buffer
    : pad ( -- A / temporary text buffer *)
       dp @ 128 + ;

    ; gforth "fi"
    ; : endif post fi ; imm
    ; standard forth then
    : then post fi ; imm

    ; put a space on the stack
    : bl 32 ;

    : 3drop 2drop drop ;
    ; an old synonym for negate
    : minus neg ;
    : negate neg ;

    : here+ ( n -- )
        here + here! ;

    : abs ( n -- +n / absolute value of n *)
       dup 0 < if negate fi ;

    ; true if x >= y otherwise false
    : >= ( x y -- true=1/false=0 ) < not ;

    ; simplest ?
    : > swap < ;

    : <= 2dup < if 2drop 1 exit fi = ;
```

```
; tests if string at A (length n) starts with string at P (length m)
: prefix ( A n P m -- true=1/false=0 )
   swap >r 2dup
   ; A n m n m   r: P
   ; if length of string < prefix then false
   < if r> 4drop 0 exit fi
   ; A n m     r: P
   swap drop r> swap
   ; A P m
   ncompare ;
   ; 0=false=different

; a double swap, needed for double numbers and convenience.
; But this is slow in source, better as an opcode, no?
: 2swap ( a b x y -- x y a b )
    >r swap
    ; a x b    r: y
    >r swap
    ; x a      r: y b
    r> r> swap
    ; x a y b
    >r swap
    ; x y a    r: b
    r> ;

; what you would expect
: 4dup
  ; a b c d
  2>r 2dup
  ; a b a b    r: c d
  2r> 2dup 2>r
  ; a b a b c d   r: c d
  2swap
  ; a b c d a b   r: c d
  2r> ;

; Actually the code below should return (A n 0) if the
; string is not found, but it is not doing this.
; standard form
: search ( A n P m -- A’ m flag / search for text in a string )
   ; A n P m
 begin
   2swap dup 0 =
   ; P m A’ n’ flag  .. n==0 so no more string to search
   ; A n false (original string, length flag)
   if 2drop 0 exit fi
   2swap
   ; A’ n’ P m
   swap >r 2dup
   ; A’ n’ m n’ m   r: P
   ; if length of string < substring then leave A n false
   < if
      ; A’ n’ m     r: P
      2drop r> drop
      ; A’
      0 0
      ; A’ 0 0
      exit
     fi
   ; A’ n’ m     r: P
   r> swap 4dup
   ; A’ n’ P m A’ n’ P m
   prefix
   ; A’ n’ P m flag
   ; return: (found address, length, true) if prefix is true
   if
     ; A’ n’ P m
     swap drop
```

```
        ; A’ n’ m
        swap drop 1
        ; A’ m 1
        exit
      fi
      ; A n P m
      2>r
      ; A n    r: m P
      1- swap 1+ swap
      ; A+1 n-1
      2r>
      ; A+1 n-1 P m
    again ;
      ; 0=false=different

 times 5*1024-($-block1) db 0
  ; block5

  ;  (run: n -- / jumps back to begin if n is true, non-zero *)
  ;  until consumes 1 item on the top of the stack.
  ;  (compile: get "begin" address from data
  ;  stack and compile a conditional relative jump
  ;  back to begin.
  : until ( long jump "until" *)
    [op:] jumpnz 5 c,
    [op:] ljump
    here 1- - , ; imm

  : within ( n a b -- F / true if a<=n<b else false [ans94] *)
      2>r dup r> 1- swap
      ; n a-1 n
      < swap
      ; t/f n
      r> <
      ; t/f t/f
      and ;

 : xyemit ( c x y -- / print char c at [x,y], reset pos *)
     getxy 2swap atxy
     ; c x y
     2>r emit 2r>
     atxy ;

 : dump ( A n -- / display n bytes of memory *)
     ; A n
     0 do
       ; A’
       ; print 12 bytes per line
       ii 12 mod 0 = if
         cr dup 6 fg .hex [char] : emit 3 fg
       fi
       c@+ dup
       ; A+1 c c
       .hex
       ; A+1 c
       ; if not printable, just print "."
       dup 32 127 within
       ; A+1 c F
       0 = if drop [char] . fi
       getxy swap drop
       ; A+1 c y
       ; calculate cursor position for asci chars
       ii 12 mod 60 +
       ; A+1 c y [ii mod 12]+50
       swap 14 fg xyemit 3 fg
       space
     loop cr ;

 ; search for all words in the dictionary starting
 ; with prefix. Improved logic with 2swap.

 ; search for all words in the dictionary containing
 ; a string. Improved logic with 2swap. Replace "search"
 ; with "prefix" to only see words starting with string

 ; If a word is immediate, display it "red", if a word is
 ; an opcode, display it "brown"

 ; This needs to be modified for multiple wordlists
 : s/ ( -- )
    ; initialise counter
    0 nn !
    wparse 2dup
    ; P m P m
    last @
    ; P m P m A
    begin
      dup >r
      ; P m P m A     r: A
      1- rcount
      ; P m P m S n   r: A
      2swap
      ; P m S n P m   r: A
      search
      ; P m S’ m flag      r: A
      ; Discard the found address and length
      >r 2drop r>
      ; P m flag      r: A
      if
        ; print words 8 to a line
        nn ++ 8 mod 0 = if cr fi
        ; display the name of the word
        r> dup .xt space >r
      fi
      ; P m           r: A
      2dup r>
      ; P m P m A
      xt+ dup
      ; P m P m A+ A+
      0 =
    until 4drop drop ; imm

 ; source version of type, ignore asci 13 in source but emit
 ; a 13 when 10 found
 : type ( A n -- )
     ; string length is 0 so do nothing
     dup 0 = if 2drop exit fi
     0 do
       c@+
       ; A’+1 n
       ; ignore \r = asci 13. Below we use the trick of putting
       ; a dummy 0 on the stack to get an if/elseif/elseif/fi effect
       dup 13 = if drop 0 fi
       dup 10 = if drop 0 cr fi
       dup if emit else drop fi
     loop drop ;

  ; type a string in quotes
  : qtype c> " type c> " ;
  : btype c> ( type c> ) ;

 times 6*1024-($-block1) db 0

    : .code
    dup [op’] nop 1+ u< not if
      u. ."  ??" exit
    fi
    2* optable + @
```

```
      1- rcount type ;

    : tint create c, does> @ fg ; imm
     [ 1 tint .blue 2 tint .green 10 tint .lgreen 7 tint .white
       3 tint .lblue
       4 tint .red 6 tint .brown 0 tint .black
       12 tint .lred 13 tint .pink 14 tint .yellow ]

     ;  ( A -- )
     ;  receive a line of input from the terminal ’
     ;  and store it as a counted string in the buffer. ’
     : accept.zero
         ;( a -- / get text with no editing *)
         dup 1+ dup
         ; A A+1 A+1
         begin
           key dup
           ; A A+1 A+n k k
           13 = if drop swap - swap c! exit fi
           dup emit
           ; A A+1 A+n k
           swap c!+
           ; A A+1 A+n+1
         again ;

     ’ accept.zero is accept

     ; ans forth word, add n to [A] and store at A
     ; eforth definition. Very similar name to !+ but very
     ; different meaning.
     : +! ( n A -- ) swap over @ + swap ! ;
     ;: +! ( n A -- ) 2dup @ + swap ! drop ;

     ; the opposite of xt+, finds the previous definition in
     ; the dictionary, so goes in the opposite direction from
     ; the linked list. If there are multiple wordlists, then
     ; we cant use xt- to find the compiled size of a word (the
     ; wordlists are interleaved.
     : xt- ( xt -- prev.xt )
         last @
         ; xt last
         begin
           ; xt a
           dup >r
           ; xt a         r: a
           xt+
           ; xt a+        r: a
           2dup =
           ; xt a+ T/F    r: a
           if
             ;  xt a+     r: a
             2drop r> exit
             ; a
           fi
           r>
           ; xt a+ a
           drop
           ; xt a+
           dup
           ; xt a+ a+
           0 =
           ; xt a+ true/false
         until ;

     ; show the (approx?) compiled size (in bytes) of a word
     ; Do an rcount first. But this doesnt work with multiple
     ; wordlists!!
     : wsize
        ; xt
```

```
         dup 1-
         ; xt xt-1
         rcount drop swap
         ; A xt
         xt-
         ; A xt’
         1- rcount drop
         ; A B
         swap - 1- ;
         ; B-A-1

  : head.  ( xt -- / decompile word header *)
     dup 1- rcount
     ; xt A n  / A= start of name
     drop 1- 1-
     ; print address
     ; xt A-2
     cr dup u.
     ; xt A
     ; print link back address
     ." : [" @ dup u. ." ] -> "
     ; xt A
     1- rcount type 1- cr
     ; xt-1
     rcount >r dup u.
     ; a-n    r: n
     ." : " r> dup >r
     ; a-n n   r: n
     ; print word name in quotes
     ; use .xt here?
     ;c. " type c> "
     qtype space
     r>
     ;*** print word length|control in quotes
     ;!! need to handle immediate words which have msb set
     u. cr ;

   ; decompile approximately as many bytes as the word is
   ; minus the header bytes
   ; ( xt -- ) ’
   ; shows how a word is compiled in the dictionary ’
   ; The dictionary header and compiled code is displayed ’
   ; for the word corresponding to the execution address. ’
   ;: word. ( xt -- )
   ;   dup head. dup wsize 4 - de, ;

   : xt? ( A -- / is address A an xt? *)
      last @
      begin
        ; A L
        2dup = if 2drop 1 exit fi
        xt+ dup 0=
      until swap drop ;

 times 7*1024-($-block1) db 0

   ; called definitions in Ans forth
   ; Actually context @ is a pointer to the data-field of a
   ; wordlist, not to its execution token
   : defs
      ; ( -- set "current":="context" *)
      context @ current ! ;

   ; this leaves the address of the next byte after the item
   ; print address as well?
   : de,,
      ;( A -- A’ / decompile one instruction *)
      c@+ dup .yellow .code .white
      ; A+1 op
```

```
        dup args
        ; A+1 op 0/1/2
        ; If opcode takes no arguments, then finished
        dup 0= if 2drop exit fi
        ; handle 1 byte argument opcodes
        ; A+1 op 1/2
        1 = if
          drop
          ; A+1
          c@+ c> : .lgreen c. .white exit
          ; A+2
        fi
        ; A+1 op
        dup [op’] fcall = if
          drop
          ; A+1, also check if A+1 is a named xt or does> etc
          @+ c> : dup xt? if .xt else u. ."  does>?" fi exit
          ; A+3
        fi
        ; A+1 op
        ; need to print signed jump for ljump
        ;dup [op’] ljump = if
        ;  @+ c> : .lgreen . .white exit
          ; A+3
        ;fi ;
        ; A+1 op
        args 2 = if
          @+ c> : .lgreen . .white exit
          ; A+3
        fi ;

     ; a very simple decompiler, just use it like this
     ; ’ name 10 un,
     ; This will decompile 10 instructions for the word <name>
     ; address would be useful too...
     : un, for de,, space next ;
     ; use a limit address instead of for/next

     ; decompiles n bytes starting at address n ’
     ; returns the next address after the decompiled ’
     ; bytes. Need to handle jumps too. This cannot decompile’
     ; machine code, only bytecodes ’
     : Un,
       0 nn ! for
         dup .brown u. .white c> : space de,, cr
         nn ++ 20 mod 0= if
           .lgreen c> > .white
           key c: q = if
             unloop exit
           fi cr
         fi
         ; do paging here
       next ;

   : word. ( xt -- )
      dup head. dup wsize 4 - Un, ;

   : bin
      ; ( -- / set base to binary *)
      2 base ! ;
   : hex
      ; ( -- / set base to hexadecimal 16 *)
      16 base ! ;
   : deci 10 base ! ;

   ; print the stack with signed numbers, source version
   ; display the items on the data stack without
   ; altering it. The top (or most recent) item
   ; is printed rightmost
```

```
   : .s ( -- )
      depth 0 = if exit fi
      depth
      ; ... n n
      begin swap >r depth 1 = until
      ; ... n     r: ....
      begin
         r> dup
         ; n a a
         . space swap
         ; a n
         1- dup 0 =
         ; a n-1 n-1
      until drop ;

   ; the escape keycode
   : esc ( -- n ) 27 ;

   ; returns the lowest number of "a" and "b"
   : min ( a b -- min )
      2dup < if drop else swap drop fi ;

   ; short jump if/fi, maybe a bit faster, for "wfind",
   : IF [op:] jumpz here 2 c, ; imm
   : FI dup here swap - 1+ swap c! ; imm

   ; short-jump until (slightly faster). begin is the same
   : Until
     ; ( +/-128 byte jump "until" *)
     [op:] jumpz here 1- - c, ; imm

   ; loops until top-of-stack is zero
   : Until0
     ; ( +/-128 byte jump "until0" *)
     [op:] jumpnz here 1- - c, ; imm

   times 8*1024-($-block1) db 0

   : xt>name ( xt -- A n / return name from word xt *)
      1- rcount ;

   ; gforth word str=
   : s=
     ;( A u B v -- F / true if strings == *)
     rot over
     ; A B v u v
     ; if lengths not equal, false. 2drop and drop are opcodes
     <> IF 2drop drop 0 exit FI
     ; A B v
     ncompare ;

   ; called I’ in some forths (ron geere)
   ;
   : iimax ( -- n / fetch loop limit *)
      r> r> r> dup >r swap >r swap >r ;
      ;r> r> dup >r swap >r ;

   : 2r@
     r> r> dup >r swap >r ;

   : all ( n -- / load blocks from last block >> n *)
     ; lb is the last block loaded with "load"
     lb @ 1+ swap 1+
     ; a+1 n+1
     thru ;

   ; just display the current disk geometry parameters which
   ; will be used for reads and writes from and to disk (usb etc)
   : geom ( -- )
```

```
    ." boot drive: " drive c@ u. cr
    ." sectors per track: " sectors @ u. cr
    ." sides (platters): " sides @ u. cr ;

  ; on different systems (slower, faster) the mswait needs
  ; to be adjusted. Perhaps there is a way to auto adjust with
  ; clock etc
  ; on qemu asus seems about right
  var mswait 950 mswait !
  ; standard forth
  : ms ( n -- / wait n milliseconds *)
    ; at least >=1
    ; dup 0 <= if drop exit fi
    0 do mswait @ 0 do nop loop loop ;
   ; get the next insertion point in the counted string A after A+i or
   ; just return A A+i if already at the last insertion point.
   ; The last insertion point is one character after the last char
   ; in the string
   : nextc ( A A+i -- A A+i' / next char, insert in string *)
       2dup swap - >r
       ; A A+i       r: i
       ; check the count
       over c@ 1 + r>
       ; A A+i n+1 i
       swap <
       ; A A+i flag (i<n+2)
       if 1+ fi ;

   ; get the previous insertion point in the counted string A before A+i or
   ; just return A A+i if already at the first insertion point (0)
   ; Also, need to skip over cr/nl (13,10) characters for traversing
   ; multiline strings
   : prevc ( A A+i -- A A+i' / previous char, insert in string *)
       2dup swap - 1
       ; A A+i i 1
       swap < if 1- fi ;
       ; A A+i'

   ; returns the maximum number of "a" and "b".
   : max ( a b -- max )
       2dup < if swap drop else drop fi ;

   ; return true if n=b or n=a otherwise return false
   : either ( n a b -- t/f )
       2>r dup r> =
       ; f   r: b
       if r> 2drop -1 exit fi
       r> = ;

    ; not a standard forth word, as far as I know.
    ; Makes copying/initialising strings a bit easier
    : smove ( A n B -- / copy n chars from A. Store counted string at B *)
       >r dup
       ; A n n     r: B
       r> c!+
       ; A n B+1
       swap cmove ;

  times 9*1024-($-block1) db 0

    ; what you would expect
    : 3dup ( a b c -- a b c a b c *)
      ; a b c
      >r 2dup
      ; a b a b    r: c
      r@ swap
      ; a b a c b  r: c
      >r swap
      ; a b c a    r: c b
      r> r> ;

    : @- ( A -- A-2 c / like @+ *)
       dup @ swap 2 - swap ;

    : c@- ( A -- A-1 c / like c@+ *)
       dup c@ swap 1- swap ;

    : c!- ( c A -- A-1 / like c!+ *)
      dup >r c! r> 1- ;

    ; this version just prints until the next newline or max n chars
    ; Another version could use "startline" above to print whole line
    : typeline ( A n -- /print line containing addr A *)
      0 do
        c@+
        ; A'+1 c
        dup 13 10 either if
          ; A'+1 c
          2drop unloop exit
        fi
        emit
        ; A'+1
      loop drop ;

  : append ( A n B -- / append n chars from A to counted string at B *)
     ; append characters at the end of B
     count
     ; A n B+1 m
     4dup
     ; A n B+1 m A n B+1 m
     +
     ; A n B+1 m A n B+1+m
     swap cmove
     ; Now increment the count at B
     ; A n B+1 m
     swap 1- >r
     ; A n m       r: B
     + r>
     ; A n+m B
     c! drop ;

    ; search source code blocks for a word
    : ss/ ( -- / search blocks for a phrase *)
        ; get the whole line, not just a word
        ; actually, when interactive there is no newline 10
        ; char, but parse will advance to the end of the input
        ; anyway. There is a bug in "parse" at the moment which
        ; cuts a character. see bugs
        ; nn is counter for paging
        0 nn !
        10 parse
        ." Searching for "
        ; white, yellow, white
        14 fg 2dup type 7 fg
        ."  on disk... " cr cr
        ; forground bright white
        15 fg
        ." Block     Text " cr
        ." -----     ---- " cr
        ; A n
        ; "final" has the last source code block
        final @ 0 do
          ; A n
          2dup
          ; A n A n
          ii block 1024
          ; A n A n B 1024
          2swap search
          ; A n B’ n flag
          if
            nn ++ 20 mod 0= if
              .lgreen ." ..." .white
              key [char] q = if
                unloop exit
              fi cr
            fi
            ; A n B’ n
            ; just assume lines are < 80 chars
            drop 80
            ; A n B’ 60
            ; green text
            .green ii . [char] > emit
            ; purple text
            13 fg
            space
            typeline cr
          else
            ; A n B’ n
            2drop
          fi
        loop
        ; A n
        2drop .white ; imm


 ; return n if a<n<b or a if n<a or b if n>b
 : limit ( n a b -- n/a/b ) >r min r> max ;

times 10*1024-($-block1) db 0

  24 buffer: defterm

  ; modify this to display the source code for <word>
  ; Puts -1 on the stack if not found
  ; A similar word is called "locate" in many forths.
  : defblock ( A n -- n / find 1st definition block of word at A/n *)
     ; add a colon before the search word
     s" : " defterm smove
     ; append the search word after the colon in "defterm"
     ; A n
     defterm append
     defterm count
     ; A n
     30 0 do
       ; A n
       2dup
       ; A n A n
       ii block 1024
       ; A n A n B 1024
       2swap search
       ; A n B’ n flag
       if
         ; A n B’ n
         4drop ii
         ; ii
         unloop exit
       else
         2drop
         ; A n
       fi
     loop
     2drop -1 ;

 : lo ( <word> -- / loads all blocks until <word> *)
   wparse
   ; A n
   defblock
```

```
   ; x  ( block containing first definition of <word> or -1, not found)
   ; check if -1
   dup -1 = if drop ." not found" cr exit fi
   ; now load all blocks upto and including required block
   all ; imm

 ; display byte value of K value
 : K 1024 u* ;

 ; a word that types one line. What should we do with
 ; unprintable characters? Print a ’?’ Also, what happens
 ; to the remnant 13,10 ?
 : typeline ( A n -- A’ n’ /type one line of input)
   0 do
      c@+ dup emit 13 = if ii iimax swap - unloop exit fi
     loop 0 ;

; we can just count 13,10 chars and pause when they reach
; a certain number. Also, its better to ignore asci 13
: page ( A n -- / page string at A length n *)
    ; initialise newline counter
    0 nn !
    ; if string length = 0 do nothing
    dup if0 2drop exit fi
    0 do
      ; A
      c@+
      ; A’+1 n
      ; ignore \r = asci 13. Below we use the trick of putting
      ; a dummy 0 on the stack to get an if/elseif/elseif/fi effect
      dup 13 = if drop 0 fi
      dup 10 = if
        drop 0 cr
        nn ++ 20 mod 0= if
          10 fg ." ..." 7 fg
          key [char] q = if
            unloop exit
          fi cr
        fi
      fi
      dup if emit else drop fi
    loop drop ;

  ; find out how many bytes are free in source block n
  : bfree ( n -- free )
      block dup
      ; adr adr
      begin c@+ 0 = until
      ; adr adr+x
      swap - 1024 swap - ;
      ; 1024-x

; displays block n on the screen.
: list
   ;( n -- /display block n *)
   .lgreen ." Listing Block " dup . .white
   ."  (" dup bfree .yellow . .white ." /1024 free)" cr
   block 1024 page ;

times 11*1024-($-block1) db 0

 ;." block 49 after vocab" cr
 : only ( -- / reduce s/o to minimum wordlists {Vlist,Forth} *)
    2 s/o ! ;

 ; a reverse counter to use with for/next
 : ’ii iimax ii - ;

  ; breaks out of one level of if/begin etc and jumps back to
```

```
    ; last begin. Preserves jump-back address for 'again' to use later.
    ; ( comp: jb JB -- jb JB ) ( run: -- )
    ; this is not elegant because it fails if there are 2 levels of nesting.
    : continue
        [op:] jump
        2dup drop
        ; jb JB jb
        here 1-
        ; jb JB jb here-1
        - c, ; imm

    ; a simple optable scan with no numbers
    ; : ops optable 2 +
    ; begin @+ dup 0= if exit fi .xt space again

    ; list all opcodes (=< NOP which is the last in the table but
    ; wont be for ever, so scan the optable instead. )
    : ops [op'] nop 1+ 1
        do
          ii 8 mod 0 = if cr fi
          ii dup 6 fg u. [char] : emit 3 fg .code space
        loop ;

    ; A Minimalistic "see"
    ; : see wparse tick drop word. ; imm

    ; a standard (?) word to see a word decompilation
    : see ( <name> )
      wparse 2dup
      ; A n A n
      tick
      ; A n X [0/1/2/3]
      dup 0 = if 2drop type ."  ??" cr exit fi
      dup 1 = if 2drop type ."  (number)"  cr exit fi
      dup 2 = if
         2swap type drop ." opcode=" u. cr exit fi
      ; A n xt 3
      2swap 2drop
      ; xt 3
      drop word. ; imm

    ; db ' flag=0 not number nor word '
    ; db ' flag=1 if number, 2 if opcode, 3 if procedure '

    ; duplicate TOS if non-zero
    ; eforth ?dup
    : ?dup ( n -- n n | 0 / duplicate t-o-s if <> 0 ) dup if dup fi ;

    ; a recursive greatest common divisor word
    ; from r.v.noble
    : gcd  ( a b -- gcd / greatest common divisor *)
        ?dup if tuck mod gcd fi ;

    : startline ( A B -- B' /leave address of start of line/string *)
      begin
        ; error! B cannot be < A
        2dup swap < if swap drop exit fi
        ; exit if B=A (already at start of string)
        2dup = if swap drop exit fi
        ; A B'
        c@-
        ; A B'-1 c
        13 10 either if
          ; A B'-1
          2 + swap drop
          ; B'+1
          exit
        fi
      again ;
```

```
    ; a simple approx * 10K
    : pi 31415 ;

    ; this could/should be a constant
    : 10K 10000 ;

    ; exponential e * 10K
    : E 27182 ;
    ; e == 1 + 1 + 1/1*2 + 1/1*2*3 + ...

    ; golden ratio phi (1 + 5^1/2)/2 * 10K
    : phi 16180 ;

    ; This square root method is very cool and gets a good
    ; approximation within about 6 or so iterations.
    ;  gets the next approximation to the square root, from ron geere.
    ; based on newtons method
    : approx ( n x -- n x' )
       over over / + 2 / ;

    ;  return b, the approximate square root of a (maximum 32767 if the /
    ;  division operator is signed, or else 64K if not),
    ;  this just does the iterations of the "approx" word.
    : sqrt ( a -- b )
      60 5 0 do approx loop swap drop ;

  times 12*1024-($-block1) db 0

   ; not so simple because have to juggle
   ; : 3>r ( push 3 onto ret-stack )
   ;   r> -rot 2>r swap 2>r ;
   ; : 3r> ( pop 3 fromfk ret-stack) 2r> r> ;

   : s+c ( c A -- append char c to counted string A *)
      dup c@
      ; c A n
      1+ over c!
      ; c A
      dup c@ + c! ;

  ; deletes the last word in the dictionary by regressing 'last'
  ; and moving back the dp compile pointer to the end of the
  ; previous word
  : del ( -- )
     last @ 1- rcount drop 2 - dp !
     last @ xt+ last ! ;

    50 buffer: wordname

    ; wordname 2 swap c!+ 58 swap c!+ bl swap c!+

    ; todo! add ": " infront of search term. Print until next ";"
    : showdef ( <word> /shows source definition for word *)
      wparse
      ; A m

      14 fg 2dup type 15 fg
      ; [char] "
      ." source code.. " cr
      ; A n
      30 0 do
        ; A n
        2dup
        ; A n A n
        ii block 1024
        ; A n A n B 1024
        2swap search
        ; A n B' n flag
        if
          ; A n B’ n
          ; just assume lines are < 80 chars
          drop 200
          ; A n B’ 60
          ; green text
          10 fg
          ii . [char] > emit
          ; purple text
          13 fg
          space
          type cr
        else
          ; A n B’ n
          2drop
        fi
      loop
      ; dull white text
      7 fg ; imm


    ; increments value stored at address a
    ; or call this ++
    : !1+ ( a -- ) dup @ 1+ swap ! ;

    ; create a list data type (an array of 16bit cells with length
    ; and capacity). At run time, put reference on stack to the first
    ; byte of the data structure
    : list: ( comp: n -- ) ( run: -- addr )
        create dup , 0 , cells allot0 does> ; imm

    ; addr is a reference to a list: object
    ; check if the list is full (i.e. size==capacity)
    : li/full ( addr -- flag=t/f )
        @+ swap @ = ;

    ; check if the list is empty
    : li/empty
        ; ( addr -- flag=t/f )
        2 + @ 0 = ;

    ; increment list size by 1. But all these words may degrade
    ; performance.
    : li/incr ( addr -- ) 2 + dup !1+ ;

    ; delete last element by decreasing list size
    : li/del ( addr -- )
        2 + dup       ./ a+2 a+2
        @ 1- swap ! ;  ./ a

    ; initialize the list
    ; delete all elements, set size (not capacity) to 0
    : li/0 ( addr -- )
        2 + 0 swap ! ;

    ; return the list size
    : li/size ( addr -- n ) 2 + @ ;

    ; add a new element to the list and increment size
    ; use: 5 obj li/add
    : li/add ( n addr -- )
        dup li/full if 2drop exit fi
        tuck 2 + @+
        ; a n a+4 len
        2* + !
        ; store new element.
        2 + !1+   ;
        ; increment size
```

```
times 13*1024-($-block1) db 0

  ; a b c d e -- a b c d e a b c d e
  : 5dup >r 4dup r> dup >r
  ; a b c d a b c d e   R: e
  -rot 2>r -rot 2r> r> ;

  : memdup>
      ; ( A -- / copy cell [2bytes] at A to A+2 *)
      @+ swap ! ;

  : also
     ;( -- / duplicate top item on search-order stack *)
     s/o @ 0= if
       ; if nothing in s/o then do nothing
       drop exit
     else
       s/o dup @ 2* +
       ; A+l*2
       memdup>
       s/o ++ drop
     fi ;

  : c++
      ; ( A -- A / incr byte value stored at A *)
      dup dup c@
      ; A A n
      1+ swap c! ;

  : c--
      ; ( A -- A / decr byte value stored at A *)
      dup dup c@ 1- swap c! ;

   : spaces ( n -- / type n spaces *)
      begin space 1- dup 0 = until drop ;

   ; an asci block char.
   : ablock 219 ;

   ; print n coloured blocks
   : glow
       ; ( n -- / print n coloured blocks *)
       begin
         dup fg 219 emit 1- dup 1 =
       until drop ;

   ; an asci table
   : asci ( -- / show a table of asci chars *)
      1 begin
        dup u. space dup emit space 1+ dup
      255 =
      until drop ;

  ; list elements
  : li/ls ( addr -- )
      dup li/empty if ." <list empty> " drop exit fi
      dup li/size           ./ a size
      swap 4 + swap 0      ./ a+4 size 0
      do @+ u. space loop drop ;

  ; list elements as opcodes
  : li/ops ( addr -- )
      dup li/empty if ." <list empty> " drop exit fi
      dup li/size
      ; a size
      swap 4 + swap 0
      ; a+4 size 0
      do @+ .code space loop drop ;
```

```
    ; list elements as calls
    : li/calls ( addr -- )
        dup li/empty if ." <list empty> " drop exit fi
        dup li/size
        ; a size
        swap 4 + swap 0
        ; a+4 size 0
        do @+ .xt space loop drop ;

    ; Just testing wordlist creation and management.
    ; The "Forth" vocab is defined in bytecode
    ; the moment.
    ;Vlist defs
    ;  vocab Edit vocab Asm
    ;Forth defs

    ; These lines are crashing (3 oks)
    ; also Edit
    ; also Asm

  times 14*1024-($-block1) db 0


    ; see if list has element n
    : li/has ( n A -- F )
        dup li/empty if 2drop 0 exit fi
        swap >r
        ; a         r: n
        dup r> swap
        ; a n a
        dup li/size
        ; a n a size
        swap 4 + swap 0
        ; a n a+4 size 0
        do
          ; a n a+4
          @+ swap >r
          ; a n m      r: a+x
          >r dup r>
          ; a n n m    r: a+x
          = if r> 2drop drop 1 unloop exit fi
          r>
          ; a n a+x
        loop
        2drop drop 0 ;

   ; add an element only if list doesnt already contain element
   : li/addu ( n addr -- )
       2dup li/has
       ; n A t/f
       if 2drop exit fi li/add ;

   ; Using a background color cursor
   ; Modified ctype to display multiline strings. But
   ; the \r \n characters make the screen scroll, when xy is near
   ; the bottom of the screen
   ; Make ctype responsible for showing itself. So add x y params
   ; on stack and eliminate showbuffer. This will allow us to
   ; have a margin on multiline text. This is used by accept and
   ; by edit (or will be)
   : ctype ( A A+i -- / show string at A with cursor at i *)
      swap count
      ; A+i A n
      dup 0 = if
        drop 2drop
        ; print a block cursor at the end of the string too.
        ; 4 fg 254 emit 7 fg
        ; green background colour cursor
        2 bg space 0 bg
        exit
      fi
      ; A+i A n
      ; set normal colour to white

      0 do
        ; A+i A'
        ; print green background block cursor
        .white
        2dup = if 2 bg .brown fi
        c@+
        ; A+i A'+1 c
        ; ignore \r = asci 13. Below we use the trick of putting
        ; a dummy 0 on the stack to get an if/elseif/elseif/fi effect
        dup 13 = if drop 0 fi
        dup 10 = if
          drop 0 cr
          ; clear the next line
          getxy 60 spaces atxy
        fi
        dup if emit else drop fi
        ; back to normal background colour
        0 bg
      loop
      ; A+i A'
      = if 2 bg space 0 bg fi ;

    ; inserts character c in counted string A at the insertion
    ; point A+i. This could handle 13,10 newlines as well
    : insert ( A A+i c -- A A+i+1 /insert char in string *)
      ; change \r (13) to \n (10) since this system uses unix line endings
      dup 13 = if drop 10 fi
      ; first increment the count
      >r >r c++ r>
      ; A A+i      r: c
      2dup - >r
      ; A A+i      r: c i
      over c@ r>
      ; A A+i n i  r: c
      ; If the insertion point is after the last character in the
      ; string, then all we have to do is copy the character in.
      < if
        ; A A+i    r: c
        r> swap c!+
        ; A A+i+1
        ; count at A already incremented, nothing else to do
        exit
      fi
      ; now the real work: copy string after A+i 1 char right
      ; A A+i      r: c
      2dup - >r
      ; calculate how many chars need to be moved right
      ; A A+i      r: c i
      over c@ 1+ r> -
      ; A A+i n+1-i  r: c
      >r dup dup 1+ r>
      ; A A+i A+i A+i+1 n+1-i  r: c
      cmove
      ; now store the char in its place in the string
      ; A A+i                  r: c
      dup r> swap c!
      ; A A+i
      1+ ;

    ; delete one character from the counted string at A or do
    ; nothing if i=1
    : delchar ( A A+i -- A A+i' /delete 1 char in string, or not *)
      ; do nothing if i==1 (at first character)
      2dup swap - 1 = if exit fi
      ; first decrement the count
      ; A A+i
      swap c-- swap
      ; A A+i
      2dup - >r
      ; Calculate how many chars need to be left shifted
      ; A A+i         r: i
      over c@ 1+ r> -
      ; A A+i n+1-i
      >r dup dup 1- r>
      ; A A+i A+i A+i-1 n+1-i
      cmove
      ; A A+i
      1- ;

  times 15*1024-($-block1) db 0

  ; called previous in ans forth
  : prev ( -- / drop one wordlist from search order *)
      ; Vlist and Forth are the minimum lists in this "forth"
      s/o @ 2 > if s/o -- drop fi ;

  ; leave the value (not pointer) on the stack
  : ++n ( A n -- [A]+n )
      swap dup @
      ; n A m
      swap >r + dup r>
      ; n+m n+m A
      ! ;

  ; check if opcode. Need to scan the opcode table here, not
  ; just compare to "nop"
  : op? ( n -- F / is n an opcode or not? *)
       0 [op’] nop 1+ within ;

   ; get a unique list of dependencies for a word
   ; will have to follow unconditional jumps to their target.

   50 list: dep.oo
   50 list: dep.cc
   : deps ( xt -- / show dependencies for a compiled word )
       ; there is a bug because wsize doesnt work for the
       ; last word, need to check the last variable

       ; Another bug is that wsize includes the name of the
       ; word. So we need "wbytes" which is just the compiled
       ; byte size.
       dup dup wsize + swap ./ xt+size xt
       begin
         c@+ dup ./ end A+1 n n
         op?                     ./ e A+n op flag
         ;if drop drop drop exit fi  ./ bail if not an opcode
         if
           dup
           dep.oo li/addu space      ./ end A+n op
           ;dup .code
         fi
         ; check here if it is an unconditional jump, and jump there
         ; if an fcall get the xt and recurse
         dup [op’] fcall =
         if
           drop dup 2 + swap  ./ end A+n+2 A+n
           @                  ./ E A+x xt  :get fcall xt
           ;dup
           dep.cc li/addu
           ;dup
           ;." <" .xt ." > "
           ;deps  ./ recursive!
         else
           args + ./ end A+x
         fi
         2dup <
       until 2drop ;

  ; finds dependencies, initializes the arrays and calls deps
  : Deps ( -- )
      wparse tick drop
      dep.oo li/0 dep.cc li/0 deps
      cr 11 fg ." opcodes: " 15 fg dep.oo li/ops
      cr 11 fg ." words: " 15 fg dep.cc li/calls  ; imm

  ; another version of 2= using ’and’ logic, 7 ops versus 13 for 2=
  ; This shows how to use and logic to replace if statements.
  : d= ( a b A B -- -1:true, 0:false )
      swap >r =
      ; a flag(-1:true,0:false)        r: A
      swap r>
      ; t/f a A
      =
      ; t/f t/f
      and ;

  : video ( u -- ) dup vid ." video mode " . ;

times 16*1024-($-block1) db 0

  : tmode ( text mode ) 3 video ;
  : vmode ( video mode ) 16 video ;

  : nip ( a b -- b ) swap drop ;

  ; the simplest random number, but only works approx 1/second
  : rnd ( -- n / random number mod 16 *)
      clock drop 16 mod ;

  ; a forth inc very simple random generator
  ( simple random number generation -- high level )
  var seed
  ; or use here... here rnd !
  clock drop seed !
  : rand seed @ 31421 * 6927 + dup seed ! ;
  : random  ( u -- u’ / random # between 0 and u *)
     rand um* nip ;

  ; returns true if ekey is a left arrow
  : arrow< ( n 0/1 -- flag ) 75 1 d= ;
  ; returns true if ekey is a right arrow
  : arrow> ( n 0/1 -- flag ) 77 1 d= ;
  : bs 8 emit ;

  ; this displays a buffer A at a particular point x y on the
  ; screen with the cursor positioned at [x+i, y].
  ; [A] contains the length of the string and "i" is the insertion
  ; point as an offset from A. "i" must be > 0.
  ; This will be called after every keystroke in acceptx to display
  ; the state of the buffer.
  : showbuffer ( x y A A+i -- /shows buffer A at x,y *)
    ; x y A A+i
    2swap 2dup atxy
    ; A A+i x y
    80 spaces atxy
    ; A A+i
    ctype ;

  ; A new version of accept that can edit with the arrow keys.
  ; No character limit
  ; Count, buffer and insertion point A+i are updated with every
  ; keystroke, so we dont need A+n on the stack, nor A+1
```

```
: accept.arrow ( A -- / get typed edited text into buffer A *)
    ; initialise buffer to nothing
    0 over !
    ; hide the text cursor, its annoying
    6 7 cursor
    ; "getxy" to saves the initial screen position
    ; A
    dup getxy 2swap 1+
    ; x y A A+i  (where A points to count byte)
    begin
      ; x y A A+1
      ; for debugging
      ; getxy >r >r 0 22 atxy ." count:" swap dup c@ . swap r> r> atxy
      ; getxy >r >r 0 23 atxy ." stack:" .s r> r> atxy
      ; x y A A+1
      4dup showbuffer
      ; x y A A+1
      ekey
      ; x y A A+i k flag
      ; extended keys (eg: arrows, page-up/down etc) have flag=1
      if
        ; x y A A+i k
        ; up arrow
        ; for single line editing, use up arrow to set
        ; insert point to beginning of line
        dup 72 = if
          ; x y A A+i k
          2drop dup 1+ 0
          ; x y A A+1 0
        fi
        ; x y A A+i k
        ; down arrow
        ; for single line editing, use down arrow to set
        ; insert point to end of line
        dup 80 = if
          ; x y A A+i k
          2drop dup count + 0
          ; x y A A+1+n 0
        fi
        ; left arrow
        dup 75 = if
          ; x y A A+i k
          ; cls
          drop prevc 0
          ; x y A A+i’ 0
        fi
        ; right arrow
        dup 77 = if
          ; x y A A+i k
          drop nextc 0
        fi
        ; cr ." stack: " .s cr
        drop
        ; x y A A+i+1
      else
        ; x y A A+i k
        ; now handle normal keys and characters
        dup 13 = if
          ; reshow the cursor
          7 6 cursor
          drop 4drop exit
        fi

        ; x y A A+i k
        ; backspace key
        dup 8 = if
          ; x y A A+i k
          ; delchar handles all cases
          drop delchar
          ; x y A A+i
        else
          ; No emit! because "showbuffer" does that.
          ; x y A A+i k
          insert
          ; x y A A+i+1
        fi
      fi
      ; if extended or normal char
      ; x y A A+i
    again ;

    ; make backtick ‘ put a newline 13,10 in the buffer
    ; for testing
    ;dup 96 = if
    ;  drop
    ;  13 insert 10 insert 0
    ;fi

times 17*1024-($-block1) db 0

  ; gforth is...
  ; : find ( cA â\200\223 xt +-1 | cA 0 )

  ; "last" is "latest" in many forths
  ; This is slightly different from Ans forth find ( A -- xt)
  ; hence the slightly different name.

  ; This is probably the most important word for speed of
  ; compilation
  ; IF/FI is a short jump if/fi and should be a whisker faster
  : wfind
     ;( A n xt -- xt’|0 / find name at A starting search at xt *)
     begin
       1- rcount
       ; A n B m
       swap 2>r dup r>
       ; A n n m      r: B
       ; if string lengths are equal
       = if
         ; A n        r: B
         r> swap >r 2dup
         ; A A B B     r: n
         r@ swap >r
         ; A A B n     r: n B
         ncompare IF
           ; A         r: n B
           ; now do name>xt
           drop 2r> + 1+
           exit
         FI
         2r> swap
         ; A n B
       else r> fi
       ; A n B
       1- 1-
       ; A n B-2
       @ dup
     Until0 2drop drop 0 ;

  ; find a name in the search order
  : find.so ( A n -- xt/0 )
     s/o @ for
       ; ." in find.so for loop " cr
       2dup
       ; A n A n
       ii 2* s/o + @
       ; for debugging
       ; ." in find.s/o: " .s cr
```

```
        ; dup .vname space
        ; A n A n W
        @ wfind
        dup if
          ( leave xt ) >r 2drop r> unloop exit
        else drop fi
      next
      ; not found, so push 0
      2drop 0 ;
    ' find.so is nfind

    ; Vocabs are unusual because they are manipulated through
    ; the s/o search order buffer. There is no direct way to
    ; access their data field (which contains a pointer to the
    ; last word defined in the wordlist).
    ;
    ; vocabulary in ans forth. Invoking the name of a wordlist
    ; set "current"=<name> (replacing top item in search-order
    ; but we also need to get access to the pointer field of
    ; the vocab (maybe with "nfind" etc ?).
    ; The data field in the vocab is a pointer to the last word
    ; in this wordlist. "vocabulary" does not seem to exist in modern
    ; forth standards. >s/o replaces ( or appends if |s/o| < 3)
    ; the new vocab in the search order buffer
    : vocab
      ; ( <name> -- / create new wordlist <name> *)
      Vlist defs
      create 0 , does> >s/o ; imm
    ; Trying to insert Vlist here causes some strange behaviour
    ; ls is not coloured properly etc.
    ; Vlist


  ; the 2-2+ trick is to avoid the loop parameters for
  ; for/next which are not interesting, or else use begin/end
  ; with a counter.
  : .rcall
    ; ( show return stack as calls )
    rdepth 2 - for
      ; print 4 to a line
      ii 4 mod 0= if cr fi
      ii 2 + rpick
      dup u. space
      ; ip points just after last fcall address FCALL:Addr: <--ip
      ; or points just after PCALL: if pcall sub 3, if FCALL sub 2
      ; but this is dodgy because the byte at r(n)-1 could happen
      ; to be ==pcall==50 but still be part of an address.

      ;dup 1 - c@ [op'] pcall = if
      ;  2 - 3 Un, else
      ;  1 - 3 Un,
      ;fi
      ; check if "pcall" op
      2 - @ .xt
      space
    next ;

    ; recursive factorial. very succint in forth
    ; 16bit forth can only hold 8! no more
    : fact
        ;( n -- n! / leave n factorial *)
        dup 1 = if exit fi dup 1- fact u* ;

    ; recursive arithmetic series
    : arith ( n -- 1+2..+n / sum arithmetic series *)
        dup 1 = if exit fi dup 1- arith + ;

    ; calculate E exponent using infinite series and factorial
    ; we can only loop upto 8 because this is a 16 bit forth.
```

```
    ; accurate to 2 decimal places. All is scaled by 10K
    : Ecc 10K 8 1 do 10K ii fact / + loop ;

    ; But this below is not the simplest. We can write an
    ; accept with no backspace editing.
    ; eg: : accept.0  ...

    ; A simple source version of "accept", with no character limit,
    ; no special keys, only editing with backspace
    : accept.basic ( a -- / get text into buffer *)
        dup 1+ dup
        ; A A+1 A+1
        begin
          key dup
          ; A A+1 A+n k k
          13 = if drop swap - swap c! exit fi
          dup
          ; test if it is a backspace
          8 = if
            ; start of buffer?
            drop 2dup <
            if bs space bs 1- fi
            continue
          fi
          dup emit
          ; A A+1 A+n k
          swap c!+
          ; A A+1 A+n+1
        again ;

    ; if we dont have anything better, just use this
    ' accept.basic is accept

  times 18*1024-($-block1) db 0

    ; colour names
    1 con blue 2 con green 3 con pblue
    4 con red 5 con purple 6 con brown
    7 con white 8 con grey 9 con lblue
    10 con lgreen 11 con llblue 12 con lred
    13 con pink 14 con yellow 15 con lwhite

   ; buffers for holding previous commands and a temp
   ; string. This will allow the up and down arrows to recall
   ; a previous command. Also, we could only create these
   ; variables if they dont exist already.
    80 buffer: h1
    80 buffer: h2
    80 buffer: temp

   ; These different versions of accept can be vectored to
   ; "accept" with "defer accept ' accept.hist is accept"
   ; This is nice because it allows the user to change the behaviour
   ; of the system.
   ;
   ; A version of accept that has command history and allows
   ; editing with arrow keys. The up arrow recalls previous text
   : accept.hist
      ;( A -- / get typed text into buffer A with history *)
      ; initialise buffer to nothing
      ; actually the shell should probably set this to 0, not here.
      0 over !
      ; A
      ; set history to null ??
      ; 0 h1 ! 0 h2 ! 0 temp !
      ; hide text cursor
      6 7 cursor
      ; "getxy" saves the initial screen position
      ; A
  dup getxy 2swap 1+
  ; x y A A+i  (where A points to count byte)
  begin
    ; x y A A+1
    ; for debugging
    ; getxy >r >r 0 22 atxy ." count:" swap dup c@ . swap r> r> atxy
    ; getxy >r >r 0 23 atxy ." stack:" .s r> r> atxy
    ; x y A A+1
    4dup showbuffer
    ; x y A A+1
    ekey
    ; x y A A+i k flag
    ; extended keys (eg: arrows, page-up/down etc) have flag=1
    if
      ; x y A A+i k
      ; up arrow
      ; for single line editing, use up arrow to recall previous
      ; command
      dup 72 = if
        ; copy buff->temp, h1->buff, h2->h1, temp->h2
        ; x y A A+i k
        2drop
        ; x y A
        dup count temp
        ; x y A A+1 n B
        ; copy buff->temp
        smove
        ; x y A
        ; copy h1->buff
        dup >r h1
        ; x y A H1         r: A
        count r>
        ; x y A H1+1 n A
        smove
        ; x y A
        ; copy h2->h1
        h2 count h1 smove
        ; copy temp->h2
        temp count h2 smove
        ; x y A
        ; move to first character of buffer
        dup 1+ 0
        ; x y A A+1 0
      fi
      ; x y A A+i k
      ; down arrow
      ; use down arrow to recall new command
      dup 80 = if
        ; x y A A+i k
        2drop
        ; x y A
        ; copy h2->temp, h1->h2, buff->h1, temp->buff
        ; downarrow
        ; copy h2->buff
        h2 count temp smove
        ; x y A
        h1 count h2 smove
        ; x y A
        dup count h1 smove
        ; x y A
        dup >r temp count r> smove
        ; x y A
        dup count + 0
        ; x y A A+1+n 0
      fi
      ; left arrow
      dup 75 = if
        ; x y A A+i k
        ; cls
        drop prevc 0
        ; x y A A+i’ 0
      fi
      ; right arrow
      dup 77 = if
        ; x y A A+i k
        drop nextc 0
      fi
      ; cr ." stack: " .s cr
      drop
      ; x y A A+i+1
    else
      ; x y A A+i k
      ; now handle normal keys and characters
      dup 13 = if
        ; x y A A+i k
        2drop
        ; here copy h1-h2 buff->h1
        ; x y A
        h1 count h2 smove
        ; x y A
        dup count h1 smove
        ; x y A
        dup
        ; because A=A showbuffer and ctype should not
        ; display any green cursor.
        ; x y A A
        showbuffer
        ; reshow the cursor
        7 6 cursor cr
        exit
      fi

      ; x y A A+i k
      ; backspace key
      dup 8 = if
        ; x y A A+i k
        ; delchar handles all cases
        drop delchar
        ; x y A A+i
      else
        ; No emit! because "showbuffer" does that.
        ; x y A A+i k
        insert
        ; x y A A+i+1
      fi
    fi
    ; if extended or normal char
    ; x y A A+i
  again ;

; this version of accept is probably the most useful, so
; we will use it (because it has command history and editing
’ accept.hist is accept

; slightly easier to use than insert
: c+s
   ;( A c -- / insert char c to at start of A *)
   >r dup 1+ r> insert 2drop ;

times 19*1024-($-block1) db 0

 ; try to create indexes
 ; ./ $-udot.so

  ; testing vocabs
  ;vocab New also New vocab Test also Test

udot.so:
      : u.
        ;( n -- / print t-o-s as unsigned number in current base *)
        ; set pad=""
        0 pad c!
        ; n
        begin
          ; n'
          base @ u/mod
          ; r n''
          swap digits +
          ; n'' D+r
          c@ pad swap c+s
          dup 0=
        until drop pad count type ;

      : xytype ( A n x y -- / print string at [x,y] then reset cursor *)
          getxy 2swap atxy
          ; A n x y
          2swap type atxy ;

      : asc ( -- / show asci codes *)
         255 0 do
           ii 8 mod 0= if
             ii 16 mod 0= if
               cr 15 fg ii u. space
             else
               space 15 fg ii u. space
             fi
           fi
           ii 14 mod 1+ fg ii emit
         loop ;

      : tri ( n -- / a triangle of color blocks *)
        2 do ii glow cr loop ;

      ; list colours
      : color ( -- / show foreground and background colours *)
         16 0 do
           space ii dup fg u.
         loop cr
         16 0 do
           space ii dup bg u.
         loop 0 bg ;

      ; return true if a=A and b=B, or call d= (double equals)
      : 2= ( a b A B -- t/f )
          swap >r =
          ; a flag(-1:true,0:false)        r: A
          not if r> 2drop 0 exit fi r> = ;

      ; another version of 2= using logic, 7 ops versus 13 above
      : d= ( a b A B -- -1:true, 0:false )
          swap >r =
          ; a flag(-1:true,0:false)        r: A
          swap r>
          ; t/f a A
          =
          ; t/f t/f
          and ;

    times 20*1024-($-block1) db 0

    : rclear ( clear the return stack )
       r> begin
         r> drop
         rdepth 0=
       until >r ;
     ; called 0sp in pforth
```

```
 : clear
    ;( -- / clear all items from data stack *)
   depth dup 0= if drop exit fi for drop next ;
 ; do something when an error occurs
 : abort clear rclear shell ;
 : abort" ( -- / clear the stacks etc *)
    post ." [call:] abort ; imm

  ; this should be loaded with into deferred word "missing"
  ; and should abort, so that we can see what went wrong
  : miss.0 ( n/xt/op 0 -- /handle word-not-found in dict *)
     cr 2drop >word
     ; A n
     type .red ."  missing " [char] @ emit
     ; [L] last gets from "current"
     last @ .xt cr
     ;order cr
     ;*** compile exit even when an error occurs
     [op:] exit
     abort ;

  ' miss.0 is missing

  ; compute the nth fibonacci using recursion
  : fib ( n -- fib[n] )
     dup 1 swap < if 1- dup 1- fib swap fib + fi ;

  ; count colons : and semi colons, to show source code slightly
  ; better. We use : and ; to indent definitions.
  var ncolon var nsemi
  ; a space margin
  var margin

  ; list word names and execution token addresses
  ; adjust this for multiple wordlists.
  : listxt ( -- ) last @
      0 nn !  0 ll !
      begin
        ; A'
        nn ++ 5 mod 0 = if
          cr
          ll ++ 20 mod 0= if
            ." ..." key [char] q = if drop exit fi
          fi
        fi
        dup .green u. [char] : emit dup .brown .xt space xt+ dup 0 =
      until drop ;

  ; Display asci codes of keys pressed. See ekeycode for the ekey
  ; version
  : keycode
      begin
        key dup u. space dup emit
        cr [char] q =
      until ;

  : ekeycode ( -- / display keycodes for key-presses [ekey] *)
      begin
        ekey 2dup swap u. space u.
        drop cr [char] q =
      until ;

times 21*1024-($-block1) db 0

 ; direction is 1=east, 2=south, 3=west, 4= north
 ;( drawing with pixel lines, using length, compass direction )

 ; draw a line starting a pixel position xy of length n
 ; ( x y n -- )
```

```
    ; : line 1 do 2dup pix 1+ swap 1+ swap loop ;

    : line ( x y dx dy n -- x+dx*n y+dy*n dx dy / draw n pixels from [x,y] in dir d
x/dy *)
       0 do
         >r >r 2dup plot
         ; x y     r: stepy stepx
         swap r> dup >r +
         ; y x+step   r: stepy stepx
         swap r> r>
         ; x+step y stepx stepy
         swap >r
         ; x+step y stepy    r: stepx
         dup >r +
         ; x+step y+step     r: stepx stepx
         r> r>
         swap
         ; x+step y+step dx dy
       loop ;

     ; algorithm to convert BCD encoded values to binary (normal)
     ; binary = ((bcd / 16) * 10) + (bcd & 0xf)

     ; this should be a matrix multiplication
     : rot45 ( dx dy - dx' dy' / rotate step params by 45 degrees *)
         2dup 1 0 2= if 1 1 2swap 2drop exit fi
         2dup 1 1 2= if 0 1 2swap 2drop exit fi
         2dup 0 1 2= if -1 1 2swap 2drop exit fi
         2dup -1 1 2= if -1 0 2swap 2drop exit fi
         2dup -1 0 2= if -1 -1 2swap 2drop exit fi
         2dup -1 -1 2= if 0 -1 2swap 2drop exit fi
         2dup 0 -1 2= if 1 -1 2swap 2drop exit fi
         2dup 1 -1 2= if 1 0 2swap 2drop exit fi ;

     : rot90 ( dx dy - dx' dy' / rotate step params by 90 degrees *)
         rot45 rot45 ;

     : square ( x y lx ly -- / draw square at x y   )
       todo! ;

     : square ( x y -- / draw square at x y  )
         1 0  ( start direction )
         ; x y 1 0
         4 for
           20 line rot90
         next 2drop 2drop ;

     : octo ( x y -- / draw an octagon at x y *)
         1 0
         ; x y 1 0
         8 for
           20 line rot45
         next 2drop 2drop ;

   times 22*1024-($-block1) db 0

     ; need to debug
     : fsquare ( x y n -- / draw filled square at x y with side length n *)
         >r
         2dup 1 0
         ; x y x y 1 0    r: n
         r> 0 do
           ; x y x y 1 0
           iimax
           ; x y' x y' 1 0 iimax
           line 4drop
           ; x y'
           1+ 2dup
           ; x y'+1 x y'+1
             1 0
             ; x y'+1 x y'+1 1 0
           loop 2drop 4drop ;

   : grid ( -- / make a pattern *)
      cls
      20 20 20
      8 0 do
        ; x y' 20
        3dup ii fg
        fsquare
        ; x y' 20
        >r 30 + r>
      loop ;

   ; this is a very primitive recursive descent parser
   : palindrome ( A n -- flag / checks if a string is a palindrome)
      ; if stringlength <= 1 just call it a palindrome
      dup 0 = if 2drop 1 exit fi
      dup 1 = if 2drop 1 exit fi
      ; check first and last letters then recurse
      ; A n
      >r c@+
      ; A+1 c    r: n
      r> swap >r
      ; A+1 n    r: c
      2 - 2dup +
      ; A+1 n-2 A+n-1    r: c
      c@ r>
      ; A+1 n-2 b c
      = not if 2drop 0 exit fi
      ; for debugging ." recurse.. "
      ; do recursion!
      palindrome ;

   : pal
      ." check: "
      pad accept pad count palindrome
      cr .s if ." palindrome!! " else ." nope " fi ;

   : showpal
      ." check: "
      ;pad 40 accept pad count
      pad accept pad count
      ; A n
      cr
      begin
        2dup
        ; A n A n
        palindrome if 2dup type space fi
        swap 1+ swap 1-    ./ A+1 n-1
        dup 0 =
      until ;


  times 23*1024-($-block1) db 0

   : wc ( xt -- n  count words, not opcodes from word at xt )
      ; assume <5000 words
      5000 0
      do
        ; assume all ops at top of dict, bad assumption!
        dup xt>op if drop ii unloop exit fi
        xt+
      loop drop ;


   : allwords last @ wc ;
   ; count words defined in bytecode
   : bytewords last 4 - wc ;

   ; how many bytes do the opcodes occupy.
   : opspace [’] nop [’] exec - ;
   ; how much space do the bytecodes occupy.
   : bytespace [’] last [’] nop - ;
   : totalspace last @ ;

   : splash ( -- / produces a colourful splash screen *)
       31 spaces 8  glow cr
       30 spaces 10 glow cr
       29 spaces 12 glow cr
       28 spaces 14 glow cr
       28 spaces 14 glow cr
       29 spaces 12 glow cr
       30 spaces 10 glow cr
       31 spaces 8  glow cr cr
       7 fg
       23 spaces ." Teatree, A ’Forthish’ System " cr cr
       13 fg
       20 spaces 6 fg ."    Machine: " 13 fg mname cr
       20 spaces 6 fg ."    Opcodes: " 13 fg
         opspace u. ."  bytes"
        ; ’ nop literal ’ exec literal - u. ." bytes"
         10 fg ." ( " [op’] nop u. ."  opcodes )" 13 fg cr
       20 spaces 6 fg ."  Byte codes: " 13 fg
         bytespace u. ."  bytes"
        ; ’ last literal ’ nop literal - u. ." bytes"
         10 fg ."  ( " bytewords u. ."  bytecode words ) " 13 fg cr
       20 spaces 6 fg ."  Total Size: " 13 fg last @ u. ."  bytes"
         10 fg ."  ( " allwords u. ."  total words ) " 13 fg cr ;

    ./ splash

  times 24*1024-($-block1) db 0

    : size ( show word size )
      wparse nfind dup
      if0 drop ." ?"
      else
        wsize ." (bytes): " .
      fi ; imm

    ; like unix "disk free"
    : df ( -- ) final @ 0
        ." Free bytes/block: " cr
        do
          ii dup 6 fg
          u. [char] : emit 10 fg bfree u. ."  "
          ; print 4 to a line
          ii 4 mod 3 = if cr fi
        loop ;

  ; display u words of data
  : dumpw ( A u -- )
      ; print mem address
      swap dup u. ." : " swap
      0 do @+ u. space loop drop ;

   ; prints out time taken given 2 clock-ticks values (18/sec) on stack
   ; this has a limit of +/- 52 minutes. takes 2 double values
   ; ( D:t1 D:t2 -- )
   : timer
       ; get rid of high clock value on stack
       drop swap drop swap - 55 u* u. ." milliseconds." cr ;

   ; do not change this word because it is being used to
   ; measure performance across different machines and different
   ; exec.x versions.
   : timeloop ( n -- )
       ; juggle number of loops
       >r clock r>
       dup u. ." 000 loops"
       begin
         1000 begin 1- dup 1 = until drop
         1- dup 1 =
       until drop ." took " clock timer ;

  ; the simplest dump, no asci chars
  : dump.basic ( A n -- / display n bytes of memory *)
      ; print mem address
      swap dup u. ." : " swap
      0 do c@+ u. space loop drop ;

  times 25*1024-($-block1) db 0

   : 2over ( a b c d -- a b c d a b /copy second double to top *)
     2>r 2dup 2r> 2swap ;

   ;A naive line drawing algorithm
   ;dx = x2 - x1
   ;dy = y2 - y1
   ;for x from x1 to x2 {
   ;  y = y1 + dy * (x - x1) / dx
   ;    plot(x, y)
   ;}

   ;
   ;
   ; gradient variables.
   var dx var dy

   ; Bresenhams algorithm would be better, or Xaolin Wus but
   ; for the time being, lets just use "naive"

   defer lineto
   : lineto.naive ( a b c d -- / draw line from [a,b] to [c,d] *)
       4dup swap >r swap - dy ! r>
       ; a b c d a c
       swap - dx !
       2drop
       ; a b
       ; if line vertical do it separately
       dx @ 0= if
         ; a b
         dup dy @ +
         ; a b b+dy
         dy @ 0 > if swap fi
         do dup ii plot loop
         ;." 0 dx!! "
         drop exit
       fi
       over dup dx @ +
       ; a b a a+dx
       ; if dx<0 then adjust the order of the loop parameters
       dx @ 0 > if swap fi
       do
         ; a b
         2dup swap ii swap -
         ; a b b ii-a
         dy @ dx @ */ +
         ; a b b+dy(x-a)/dx
         ii swap plot
         ; a b
       loop 2drop ;

    ’ lineto.naive is lineto
    ; stores zero byte at location
    : c!0 ( adr -- ) 0 swap c! ;

    : sspoints ( x y n -- a b c d e f g h
      / vertices of square side 2n, midpoint x y *)
        >r
        ; x y        r: n
        2dup swap r@ - swap r@ -
        ; x y a b    r: n
        2swap
        ; a b x y    r: n
        2dup swap r@ + swap r@ -
        2swap
        ; a b c d x y    r: n
        2dup swap r@ + swap r@ +
        2swap
        ; a b c d e f x y    r: n
        2dup swap r@ - swap r@ +
        2swap
        ; a b c d e f g h x y    r: n
        2drop r> drop
        nop ;

    ; reverse line
    : rline ( a b c d -- a b / trace line from {c,d} to {a,b} *)
        2swap 2dup >r >r lineto r> r> ;

    ; traces lines starting at point (x1,y1), ending at (xn,yn)
    ; This does not "close" the path
    : trace ( xn yn ... x1 y1 n -- xn xn
        / trace lines between n points on stack *)
        1- 0 do rline loop ;

  times 26*1024-($-block1) db 0

    : emitn ( char n -- emit char n times *)
        for dup emit next drop ;

    : utype ( A n -- / type string with underline *)
       tuck type cr [char] - swap emitn ;

    : .u" ( -- / print underline text *)
        post s" [call:] utype ; imm

    ; traces lines starting at point (x1,y1), ending at (xn,yn)
    ; This closes the path
    : ctrace ( xn yn ... x1 y1 n -- / trace path between n points on stack *)
        dup 2swap 2dup >r >r
        ; ... n n x1 y1   r: x1 y1
        2swap drop trace
        r> r> lineto ;

    ; traces lines starting at point (x1,y1), ending at (xn,yn)
    ; This does not "close" the path
    : pplot ( xn yn ... x1 y1 n -- / plot n points from stack *)
        0 do plot loop ;

    : rswap ( r: a b -- r: b a / swap top 2 items on return stack *)
        r> r> r> swap >r >r >r ;

    : rvertex ( x y w h -- a b c d e f g h
      / leave vertices of rectangle, midpoint [x,y], width=2w, heigth=2h  *)
        >r >r
        ; x y        r: h w
        2dup swap r@
        ; x y y x w
        - swap rswap r@ - rswap
        ; x y a b    r: h w
        2swap
        ; a b x y    r: h w
        2dup swap r@
        ; a b x y y x w   r: h w
        + swap rswap r@ - rswap
        2swap
        ; a b c d x y    r: h w
        2dup swap r@ + swap rswap r@ + rswap
        2swap
        ; a b c d e f x y    r: h w
        2dup swap r@ - swap rswap r@ + rswap
        2swap
        ; a b c d e f g h x y    r: h w
        2drop r> r> 2drop ;

times 27*1024-($-block1) db 0

  ; 5 is a magic number, the data field is 6bytes after
  ; the (reverse) count field of the wordlist name.
  : .vname ( A -- / print vocab name from pointer to data field A *)
      5 - rcount type ;

  ; this can be simplified. See the for/next loop in find.s/o
  : order ( -- show wordlists in search-order & "current" *)
     s/o dup @ 2* +
     ; get name of wordlist
     ; L
     ." Search Order: "
     s/o @ 0 do
       ii 4 mod 0= if cr 3 spaces fi
       ; A’
       dup @ dup
       ; A’ B B
       ; print vocab name and last word in that list
       .vname @
       ; show last word if <> 0 or else empty
       ;dup if c> (  .xt c> ) else ." (empty) " fi
       c> (  .xt c> )
       space 2 -
     loop drop cr
     ." current (def): " current @ .vname ;


; list all words in the dictionary
; need to modify this to search all words in all wordlist
; in s/o search order.
; Also, need a word wo ( xt -- ) which list all words starting
; from xt address. Then a word
;  ls/ Asm  which lists all words in the Asm wordlist
;   This can be something like
;   also Asm context @ @ wo prev ;

: ls ( -- / list all words in dictionary search-order *)
   ; last @
   ; ll counts number of lines
   0 ll !
   ; loop through s/o pointers
   s/o @ 1+ 1 do
     ; get next vocab pointer from search order
     ii 2* s/o + @
     dup .yellow cr cr ." Words in vocab " .vname ." :" cr
                      ." ------------------------"
     cr .white
     @
     dup 0= if ." (empty)" fi
     ; number of words
     0 nn !
     begin
       ; print 8 words to a line
       nn ++ 8 mod 0 = if
```

```
          cr
          ; print a prompt every n lines
          ll ++ 16 mod 0 = if
            .lgreen ." any key or 'q' to quit>" key
            ; quit if 'q' pressed
            [char] q = if unloop exit fi
            0 bg cr
          fi
        fi
        dup .xt
        space xt+ dup 0 =
      until drop
    loop ;

  ; when a word is not found we can try to loop through
  ; the blocks and find the missing word.
  ; also, show similar words in the dictionary
  : miss.more ( n/xt/op 0 -- /handle word-not-found in dict *)
     cr 2drop >word
     ; A n
     5 fg type 4 fg ." not found " 2 fg [char] @ emit
     14 fg last @
     ; [L] get last from "current"
     .xt 7 fg cr
     order ( show vocab search order )
     cr
     ;*** compile an exit even when an error occurs
     ; do "abort" here. This will stop more stuff loading
     ; after an error. An clear the return stacks etc.
     [op:] exit ;

times 28*1024-($-block1) db 0

  : ?1- ( n -- n-1/n / decrement n if n<>0 )
     dup if 1- fi ;

  : boxtype ( A n -- / type text surrounded by box *)
     tuck getxy
     ; n A n x y
     ?1- swap ?1- swap atxy
     ; n A n
     218 emit 196 swap emitn 191 emit over getxy
     ; n A n x y
     1+ >r swap 2 + - r>
     ; n A x-n-2 y+1
     atxy 179 emit
     ; n A
     over type 179 emit
     ; n
     dup getxy
     ; n n x y
     1+ >r swap 2 + - r>
     ; n x-n-2 y+1
     ; 2dup swap c> [ . c> , . c> ]
     atxy 192 emit
     196 swap emitn 217 emit ;

  : .b" ( -- / print boxed text line *)
      post s" [call:] boxtype ; imm

  ; traces lines starting at point (x1,y1), ending at (xn,yn)

  ( Double numbers, ie numbers on stack in form [a b] where
    b is high order item, 16bits, and also highest on stack )

  ; some ron geere double number defs

  : 2rot ( a b c d e f -- b c e f a b /do rotate with doubles *)
    2>r 2swap 2r> 2swap ;
```

```
  : 2@  ( A -- d  / fetch double number from address A *)
    dup 2 + @ swap @ ;

  : 2!  ( d A -- / store double number at address A *)
    dup >r ! r> 2 + ! ;

  : 2con ( define 32 bit double constant *)
    ( looks a bit dodgy ) ;
   ; con , does> 2@ ; imm

  ; this is called dminus in older forths.
  : dnegate ( d -- -d / negate double number d *)
    todo! ;

times 29*1024-($-block1) db 0

  ; do defined as source.
  ; : do [op'] >r dup c, c, here ; imm
  ; need to compile code which checks something
  : ?do [op'] >r dup c, c, here ; imm
  ; : (?do) ... ;

  : 0< ( n -- f / leave true if n is negative *)
     todo! ;

  : d- ( d1 d2 -- d3 / do d1-d2 double number subtraction *)
    dnegate d+ ;

  : d0= ( d -- f / leave true flag if double number is zero *)
    or = ;

  : d0< ( d -- f / leave true if double number negative *)
    swap drop 0< ;

  : d= ( d1 d2 -- f / leave true flag if both equal *)
    d- d0= ;

  ; '79 standard double compare.
  : d< ( d1 d2 -- f /leave true flag if d1<d2, signed, else leave false *)
    d- d0< ;

  : d> ( d1 d2 -- f /leave true flag if d1>d2, signed,  else leave false *)
    2swap d< ;

;  : dmin ( d1 d2 -- d3 / leave minimum of 2 double numbers *)
;    2over 2over d< 0= if 2swap fi 2drop ;

;  : dmax ( d1 d2 -- d3 / leave maximum of 2 double numbers *)
;    2over 2over d< if 2swap fi 2drop ;

times 30*1024-($-block1) db 0

  ; list words in "context"
  : words ( -- / list words in 1st wordlist of search-order *)
      context @ @

      dup 0= if ." (empty)" fi
      ; number of words
      0 nn ! 0 ll !
      begin
        ; print 8 words to a line
        nn ++ 8 mod 0 = if
          cr
          ; print a prompt every 22 lines
          ll ++ 20 mod 0 = if
            1 fg c> > key
            ; quit if 'q' pressed
            [char] q = if unloop exit fi
            0 bg cr
          fi
        fi
        dup .xt
        space xt+ dup 0=
      until drop ;

    ; just to test find.s/o
    : sof .s cr wparse .s cr find.so cr .s ; imm
    ; just to test wfind
    : wf .s cr wparse rot .s cr wfind .s ; imm

    : rev ( A n -- / type string in reverse *)
      swap over + 1-
      ; n A+n-1
      swap 0 do
        ; A+n-1
        c@- emit
      loop drop ;

    ; another
    ; rev for dup ii 1- + @ emit next drop ;

  times 31*1024-($-block1) db 0

    ; Handles opcodes and fcalls
    ; Ansi forth "execute", maybe called "perform" in older forths
    ; "obuff" already has an EXIT opcode as its seconds byte,
    ; it is not necessary to write one there.
    : ex ( ... xt -- / execute word at xt *)
       dup xt>op dup if
         swap drop ( op )
         obuff c! ( make opcode callable )
         obuff pcall
       else ( xt 0 ) drop pcall
       fi ;
    : perform ex ;
    : execute ex ;

  times 32*1024-($-block1) db 0

    : time ( <name> -- ms / execution time of <name> in milliseconds *)
      wparse 2dup ." Timing " qtype cr
      ; start time, and save value on rstack
      clock drop >r
      nfind perform
      ; end time, and calculate time taken
      clock drop r> - 55 u* u. ."  ms" ; imm

    : t/wfind 900 for buff count last @ wfind drop next ;
    : t/ffind 900 for buff count ffind drop next ;

    : codepage ( A n -- / format source code from A, length n *)
        ; initialise counters (newlines, : and ; )
        0 nn ! 3 margin !
        ; if string length = 0 do nothing
        dup if0 2drop exit fi
        0 do
          ; A
          c@+
          ; A'+1 n
          ; ignore \r = asci 13. Below we use the trick of putting
          ; a dummy 0 on the stack to get an if/elseif/elseif/fi effect
          dup 13 = if drop 0 fi
          dup 10 = if
            drop 0 cr
            nn ++ 20 mod 0= if
              10 fg ." >" 7 fg
              key c: q = if
```

```
              unloop exit
            fi cr
            ; add margin here
            margin spaces
          fi
        fi
        ; colon, add margin
        ; margin += 2
        ; A'+1 n
        ; dup c: : = if margin 2 n++ drop fi
        ; dup c: ; = if margin -2 n++ drop fi
        dup if emit else drop fi
      loop drop ;

times 33*1024-($-block1) db 0

; toggle a variable
: toggle dup @ not swap ! ;

var seeanon 0 seeanon !
: debug ( toggles debug information )
   seeanon @ not seeanon ! ;

: welcome cr ." Forth Shell has started!" cr ;
; parse, compile and execute words. This is the interpreter.
; called 'quit' in ansi forth. Hopefully "thru" has defined ." etc
; above
: shell
    ; maybe set data and return stacks to 0 here...
    begin
      here>anon [op:] exit here>anon
      term dup accept count in0
      in,
      ; allow debugging of anonymous compilation
      seeanon @ if
         ; also show "dp" here.
         ['] anon word.
      fi
      anon pcall ok
    again ;

 9 33 thru ( dont load 33 again! infinite loop )

 ; testing circle drawing
 69 load
 ; 34 60 thru
 ; ed the editor
 70 82 thru
 ;34 51 thru

 welcome
 shell

times 34*1024-($-block1) db 0

 ; : ascline begin 196 emit loop
 ; 179 side. A 10x10 box is not square...
 ; We dont really need an x,y parameter for location. The
 ; box should just be printed wherever the cursor is.
 : abox ( x y lx ly -- / draw asci box at [x,y] width lx, length ly *)
    4dup
    ; x y lx ly x y lx ly
    2swap atxy
    ; x y lx ly lx ly
    ; top-left corner
    218 emit
    drop 0 do
      ; x y lx ly
      196 emit
      loop
      ; top-right corner
      191 emit
      >r >r 1+ 2dup
      ; x y+1 x y+1     r: lx ly
      atxy r> r>
      ; x y+1 lx ly
      dup 0 do
        ; x y’ lx ly
        179 emit
        4dup drop swap >r + 1+ r>
        ; x y’ lx ly x+lx+1 y’
        atxy
        ; x y’ lx ly
        179 emit
        ; x y’ lx ly
        >r >r 1+ 2dup
        ; x y’+1 x y’+1     r: lx ly
        atxy r> r>
        ; x y’+1 lx ly
      loop
      >r >r atxy r> r>
      ; lx ly
      ; bottom left corner
      192 emit
      swap
      0 do 196 emit loop
     ; bottom right corner
      217 emit ;

    ; need to juggle the return pointer for function "2>r"
    ; 2>r is an opcode now, for speed, not minimalism
    : 2>R ( a b -- r: -- a b / put 2 items on return stack *)
        r> swap >r swap >r >r ;

    : 4>r ( a b c d -- r: -- a b c d *)
        ; can rewrite, in terms of 2>r opcode, but still must
        ; juggle the return pointer.
        r> swap >r swap >r swap >r swap >r >r ;

    ;vocab Matrix
    ;Matrix defs

    : ax+by ( a b x y -- a*x+b*y / row column matrix multiplication *)
        swap >r *   ( a b*y  r: x )
        swap r> *   ( b*y a*x )
        + ;         ( a*x+b*y )

    ; This word does the matrix multiplication (useful for point rotations)
    ;     | a b | * | x |
    ;     | c d |   | y |
    : 2x2Matrix* ( x y a b c d -- x’ y’ / left * 2x2 matrix a b c d *)
        4>r 2dup r> r>
        ; x y x y a b   r: c d
        ax+by
        ; x y ax+by     r: c d
        swap >r swap r>
        ; ax+by x y     r: c d
        r> r> ax+by  ;
        ; ax+by cx+dy

    ; uses rotation matrix to rotate point by left multiplication
    ; Matrix = | 0  1 |
    ;          | -1 0 |

    : xyrot90 ( x y -- x’ y’ / rotate [x,y] 90 degrees about 0 *)
        0 1 -1 0 2x2Matrix* ;

  times 35*1024-($-block1) db 0
```

```
;vocab Set
;Set defs

 ; an xyset is a set of integer pairs [x,y] with a length
 ; points: (0,0) (0,1) (1,1)
 ;


 : xyset: ( -- / create a test xyset *)
      create 4 , 0 , 0 , 1 , 0 , 1 , 1 , 2 , 0 , does> ; imm

 ; remember the count or length of xyset is 16bit not 8bit char.
 : .xyset ( A -- / show values of xyset *)
     @+ dup . c: : emit space
     ; A+1 n
     0 do
       ; A+1
       c: [ emit
       @+ .  c: , emit
       @+ .  c: ] emit
       ii 8 mod if0 cr 4 spaces fi
     loop drop ;

 ; length of xyset is 16bit not 8bit
 : asciglyph ( c A -- / display xyset A as asci glyph with char c *)
     ; get current cursor pos
     getxy >r >r
     @+ 0 do
       ; c A’
       @+ >r @+ r>
       ; c A’+2 y x
       swap atxy over
       ; c A’+2 c
       emit ( asci block char )
     loop 2drop
     ; restore cursor pos
     r> r> atxy ;

 : blockglyph ( A -- )
    219 swap asciglyph ;

 ; show an xyset as a pixel glyph.
 : xyglyph ( A -- / display xyset A as monochrome glyph *)
     ; get the xyset length (2 bytes) and loop
     @+ 0 do
       ; A’
       @+ >r @+ r>
       ; A’+2 y x
       swap plot
       ; A’+2
     loop drop ;

 xyset: tt

 : txty ( dx dy A -- / offset xyset at A by dx dy *)
     @+
     ; dx dy A+2 n
     0 do
       ; dx dy A’
       >r 2dup swap r>
       ; dx dy dy dx A’
       ; or use +! here ...
       dup @
       ; dx dy dy dx A’ x
       swap >r +
       ; store new x value in cell
       ; dx dy dy dx+x   r: A’
       r> !+
          ; dx dy dy A’+2
          dup @
          ; dx dy dy A’+2 y
          swap >r +
          ; store new y value in cell
          ; dx dy dy+y      r: A’+2
          r> !+
          ; dx dy A’+4
        loop drop 2drop ;

    : point ( x y -- / print [x,y] as a point *)
        c> [ swap .  c> , .  c> ] ;

    ; remember the count or length of xyset is 16bit not 8bit char.

 times 36*1024-($-block1) db 0

   ; type text in rainbow colours
   : rainbow ( A n -- )
       ; string length is 0 so do nothing
       dup 0 = if 2drop exit fi
       0 do
         c@+
         ; A’+1 n
         ; ignore \r = asci 13. Below we use the trick of putting
         ; a dummy 0 on the stack to get an if/elseif/elseif/fi effect
         dup 13 = if drop 0 fi
         dup 10 = if drop 0 cr fi
         dup if ii 8 mod 1+ fg emit else drop fi
       loop drop
       ; set colour back to normal
       white fg ;

   : rb" ( -- /rainbow coloured text *)
       post s" [call:] rainbow ; imm

   : dd nop ;

   vocab Tetris
   Tetris defs

    : animblock ( A -- / animate a block falling *)
        12 0 do
          >r
          ;     r: A
          ; show glyph
          ablock r@ asciglyph
          ; wait half a second
          500 ms
          ; erase glyph
          32 r@ asciglyph
          0 1 r@ txty
          r>
          ; A
        loop
        ablock swap asciglyph ;

    ; Tetris blocks are "tetronimoes", 4 orthogonally joined squares.
    ;
    : Iblock: ( -- / tetris I block *)
         create 4 , 0 , 0 , 1 , 0 , 2 , 0 , 3 , 0 , does> ; imm

    : Sblock:
         ;( -- / tetris S block *)
         create 4 , 0 , 0 , 1 , 0 , 1 , 1 , 2 , 1 , does> ; imm

    xyset: b1
    Sblock: b2
    ; translate the set
```

```
    ; this line is causing crash if not immediate
    [ 35 7 b1 txty ]
    [ 35 21 b2 txty ]

   : ftetris ( -- / a fake asci tetris *)
      ; put blocks in initial position
      ; use this below,
      ;35 7 b1 xysetat
      ;35 21 b2 xysetat
      3 vid
      ; hide the text cursor, its distracting
      6 7 cursor
      32 1 atxy 3 fg rb" Forth Tetris! "
      25 2 atxy 7 fg ." Score: 31517 (High Score!) "
      25 3 atxy 15 fg ." ---------------------------"
      7 fg 24 0 30 22 abox
      4 fg ablock b2 asciglyph
      ;6 fg ablock b1 asciglyph
      6 fg b1 animblock
      ; show the text cursor again
      7 0 cursor ;

  also Forth defs

   ; display next char at runtime
   : .c post [char] [op:] emit ; imm

   ;    post s" [’] rainbow post call, ; imm
   ;  post s" ’ rainbow literal post call, ; imm

 times 37*1024-($-block1) db 0

    : ball: ( -- / create a xyset in shape of a ball *)
         create 16 , 1 , 0 , 2 , 0 , 3 , 0 ,
                    0 , 1 , 1 , 1 , 2 , 1 , 3 , 1 , 4 , 1 ,
                    0 , 2 , 1 , 2 , 2 , 2 , 3 , 2 , 4 , 2 ,
                    1 , 3 , 2 , 3 , 3 , 3 ,
           does> ; imm

    : setn ( A n  -- x y / get the nth point [x,y] from set A [0 base] *)
        >r @+ r>
        ; A+2 m n
        ; if n => |S| then error
        2dup <= if
          ; some error number
          drop 2drop 1234 1234 exit
        fi
        ; A+2 m n
        swap drop 4 * +
        ; A+2+4n
        @+ swap @+ swap
        ; x y A+4+n
        drop ;

    : rot90set
       ;( A -- / rotate xyset A by +90 degrees about 0,0 *)
        @+
        ; A+2 n
        0 do
          ; A’
          dup
          ; 2@ here ??
          @+ swap @+ swap drop
          ; A’ x y
          xyrot90
          ; point
          ; A’ x’ y’
          ; a word or opcode 2!  ??
          >r swap !+ r> swap !+
          ; A’+4
         loop drop ;

     ; This is different to Dneg, double number negate.
     : 2neg ( x y -- -x -y / negate 2 stack items *)
         neg swap neg swap ;

     : xyset0 ( A n -- dx dy / translate set A, nth point to 0,0 *)
        ; This word leaves the displacement on the stack so that
        ; it can be used by txty after rot90set.
        over >r
        ; A n         r: A
        setn 2neg 2dup r>
        ; -x -y -x -y A
        txty ;
        ; -x -y

     : xyset00 ( x y A -- dx dy / translate set A, so x,y--> 0,0 *)
        >r 2neg 2dup r>
        ; -x -y -x -y A
        txty ;
        ; -x -y

     : xysetat ( x y A -- dx dy / translate set A, so point 0 --> x,y *)
        dup 0 xyset0 2drop
        ; x y A
        txty ;
        ; -x -y

     ; Terminology: ’’ is a transformation of a set
     ;  ’ is a transformation of an xy point value
     ;  ro’ is a rotation of point [x,y]
     ;  ro’’ is a rotation of a set
     ;  tr’ - translation of point [x,y]
     ;  tr’’ - translation of an xyset/glyph
     ; We could call xysets psets (point sets)

     ; change this to
     ; : ro’’pi/2.about ( A x y -- /rotates pset A about point [x,y] *)
     ; or even
     ; : ro’’about ( A x y -- /rotates pset A about point [x,y] *)
     ; then do
     ;    dup A n setn R’.about
     : rot90setn ( A n -- / rotate 90 degrees about Nth point in set )
       ; A n
       swap dup >r swap
       ; translate A so that Nth point is at zero
       ; A n           r: A
       xyset0 r@
       ; dx dy A       r: A
       rot90set 2neg r>
       ; -dx -dy A
       txty ;

  times 38*1024-($-block1) db 0

      : arot ( A n -- / animate xyglyph A rotation n times *)
      0 do
        ; A
        ablock over
        ; A c A
        asciglyph 500 ms
        ; A
        32 over asciglyph
        ; A
        dup 1 rot90setn
        ; A
     loop ;

      : aarot ( A n -- / animate xyglyph A rotation n times *)
      0 do
        ; A
        ablock over
        ; A c A
        asciglyph 500 ms
        ; A
        32 over asciglyph
        ; A
        dup 1 rot90setn
        ; A
        dup 1 swap 1 swap txty
        ; A
     loop ;

  times 39*1024-($-block1) db 0

  ; : table create does> swap 2* + @ ; imm

  ; the "code>here" is a cludge. Or use sync.


  ; Using : .. create doesnt seem to be working.
  ; : table create
  ; This table contains sine values * 10000 for degrees 0-90
  [ create sinetable
  0 , 175 , 349 , 523 ,
  698 , 872 , 1045 , 1219 ,
  1392 , 1564 , 1736 , 1908 ,
  2079 , 2250 , 2419 , 2588 ,
  2756 , 2924 , 3090 , 3256 ,
  3420 , 3584 , 3746 , 3907 ,
  4067 , 4226 , 4384 , 4540 ,
  4695 , 4848 , 5000 , 5150 ,
  5299 , 5446 , 5592 , 5736 ,
  5878 , 6018 , 6157 , 6293 ,
  6428 , 6561 , 6691 , 6820 ,
  6947 , 7071 , 7193 , 7314 ,
  7431 , 7547 , 7660 , 7771 ,
  7880 , 7986 , 8090 , 8192 ,
  8290 , 8387 , 8480 , 8572 ,
  8660 , 8746 , 8829 , 8910 ,
  8988 , 9063 , 9135 , 9205 ,
  9272 , 9336 , 9397 , 9455 ,
  9511 , 9563 , 9613 , 9659 ,
  9703 , 9744 , 9781 , 9816 ,
  9848 , 9877 , 9903 , 9925 ,
  9945 , 9962 , 9976 , 9986 ,
  9994 , 9998 , 10000 , code>here ]
  ; the "code>here" is a cludge. Or use sync.

    ; This is very old code.
    : tsin ( n -- a’ / get a value from the sinetable *)
       sinetable swap 2* + @ ;

  times 40*1024-($-block1) db 0

    : (sin) ( x -- x’ /adjust sin for 90 < x < 180 *)
      dup 90 > if 180 swap - fi tsin ;

    : sin ( x -- sin[x]*10000 / calculate scaled sin[x] from table *)
      360 mod
      dup 0 < if 360 + fi
      dup 180 > if
        180 - (sin) negate else (sin)
      fi ;

    : cos ( n -- n’) 360 /mod drop 90 + sin ;
    : vals ( show values )
      do
        ii dup . sin ." sin(x):" . cr
        ; ii sin(ii)
      loop ;

    : wave ( n m -- /draw a sine wave from n degrees to m degrees *)
      do
        ii dup sin 80 / 170 + plot
        ; ii sin(ii)
      loop ;

    ; rotate point about 0,0  X degrees counter clockwise
    ; R = | cos(x)  -sin(x) |
    ;     | sin(x)  cos(x) |

  times 41*1024-($-block1) db 0

    ; this is useful for rotations using sin/cos rotation matrix
    ; since the trig values are scaled by 10000. It assumes a & b
    ; are scaled by 10K
    : ax+by/10k ( a b x y -- a*x+b*y/10k / matrix mult scaled by 10000 *)
        swap >r 10000 */  ( a b*y  r: x )
        swap r> 10000 */  ( b*y a*x )
        + ;         ( a*x+b*y )

    ; Since trig functions in forth are usually scaled by 10000
    ; we need to unscale them when calculating the rotation
    ; This word does scaled matrix multiplication (useful for point rotations)
    ;     | a b | * | x |
    ;     | c d |   | y |
    : 2x2Matrix*/10k ( x y a b c d -- x' y' / left * 2x2 matrix a b c d *)
        4>r 2dup r> r>
        ; x y x y a b   r: c d
        ax+by/10k
        ; x y ax+by     r: c d
        swap >r swap r>
        ; ax+by x y     r: c d
        r> r> ax+by/10k  ;
        ; ax+by cx+dy

    : rmatrix ( n -- c -s s c / make an n-degree rotation matrix * 10000 *)
        ; all trig values are * 10000 because basic forth has no
        ; floating point
        dup cos dup >r
        ; n cos      r: cos
        swap sin dup neg swap r> ;
        ; cos -sin sin cos

    ; sin and cos are scaled by 10000

    : xyrotn ( x y n -- x' y' / rotate [x,y] n degrees about 0 anti-clock *)
        rmatrix
        ; x y cos -sin sin cos
        2x2Matrix*/10k ;

    : xyrotnabout ( x y a b n -- x' y' /
          rotate [x,y] n degrees about [a,b] anti-clockwise *)
          ; not not a set here, just x,y
          ; bug!
        >r xyset00
        ; x' y' dx dy    r: n
        r> swap >r swap >r
        ; x' y' n        r: dy dx
        rmatrix
        ; x' y' cos -sin sin cos
        2x2Matrix*/10k
        r> swap >r +
        ; x''+dx  r: dy y''
        r> r> + ;
        ; x''+dx  y''+dy

  times 42*1024-($-block1) db 0

     : rotset ( A n -- / rotate xyset A n degrees about 0,0 anti-clockwise *)
         swap @+
         ; n A+2 m
         0 do
           ; n A'
           swap >r dup
           ; A' A'   r: n
           @+ swap @+ swap drop
           ; A' x y  r: n
           r@
           ; A' x y n   r: n
           xyrotn
           ; A' x' y'          r: n
           ;2dup point cr
           ;.s cr
           >r swap !+ r> swap !+
           ; A'+4              r: n
           r> swap
           ; n A'+4
         loop 2drop ;

     ball: bb
     [ 250 150 bb txty ]

     : swing ( n -- animate with n iterations *)
        ;bb 1 xyset0
        ;250 150 bb txty
        0 do
         3 fg bb xyglyph
         200 ms
         0 fg bb xyglyph
         bb 3 rotset
        loop ;

  times 43*1024-($-block1) db 0



    ; should be able to write turtle graphics now with
    ; trig functions and lineto.

    ; sin(t) = (y1-y0)/r
    ; y1 = r.sin(t) + y0
    ; cos(t) = (x1-x0)/r
    ; y2 = r.cos(t) + x0

    : polar ( x y t r -- x' y' / find co-ords distance r at angle t *)
       >r dup sin
       ; x y t sin(t)*10K   r: r
       r@ 10000 */
       ; x y t r*sin(t)     r: r
       swap >r +
       ; x r.sin(t)+y      r: r t
       swap r>
       ; r.sin(t)-y x t    r: r
       cos r> 10000 */
       ; r.sin(t)+y x r.cos(t)
       + swap ;
       ; r.cos(t)+x r.sin(t)+y

   : ray ( x y t l -- / draw one ray from {x,y} length l, angle a *)
         ; x y t l
         2>r 2dup 2r>
         ; x y x y t l
         polar ( x y x' y') lineto ;

     : ray.ball ( x y l -- / draw rays from {x,y} length l *)
         0 nn !
         begin
           ; x y l
           >r 2dup 2dup
           ; x y x y x y    r: l
           nn ++ 20 * r@
           polar
           ; x y x y x' y'  r: l
           lineto r>
           ; x y l
           nn @ 20 * 360 >
           ; stop when > 360 degrees
        until 2drop drop ;

     ; : mm 100 0 do 100 ii + 100 ii + 30 ray.ball 200 ms loop ;

     [ create somedata 0 , 175 , 349 , 523 , code>here ]

   times 44*1024-($-block1) db 0

     : arc ( a b r -- / draw 1 arc of a circle *) ;

     : ytri ( x h -- y / height of right triangle *)
         ; y = sqrt(h^2-x^2)
         dup * swap dup * - sqrt ;

     : arc ( a b r -- / draw 90 deg arc of radius r midpoint [a,b] *)
       ; x2+y2=r2, so y=sqrt(r2-x2)
       dup 1 do
         ; a b r
         3dup
         ; a b r a b r
         ii over
         ; a b r a b r ii r
         ytri
         ; a b r a b r y
         ; now add midpoint to (x,y)
         swap drop +
         ; a b r a b+y
         swap ii +
         ; a b r b+y a+x
         swap
         ; a b r x' y'
         plot
       loop ;

     : circle ( a b r -- / draw circle of radius r midpoint [a,b] *)
       ; x2+y2=r2, so y=sqrt(r2-x2)
       dup 1 do
         ; a b r
         3dup
         ; a b r a b r
         ii over
         ; a b r a b r ii r
         ytri
         ; a b r a b r y
         swap drop ii swap
         ; a b r a b x y
         rvertex 4 pplot
       loop 2drop drop ;

     : fcircle ( a b r -- / draw filled circle of radius r midpoint [a,b] *)
       ; x2+y2=r2, so y=sqrt(r2-x2)
       dup 1 do
         ; a b r
         3dup
```

```
         ; a b r a b r
         ii over
         ; a b r a b r ii r
         ytri
         ; a b r a b r y
         swap drop ii swap
         ; a b r a b x y
         rvertex 4 ctrace
       loop 2drop drop ;

     ; not working because of immediate problems
     : iskey ( k -- n flag /test for keypress )
        dup wparse
        dup 3 > if
          ; k k A u
          drop c@ = exit
        fi
        dup 1 = if
          drop c@ = exit
        fi
        dup 2 = if
          drop c@ = exit
        fi
        dup 3 = if
          drop c@+ c: ' = if c@ = exit fi
        fi ; imm

   times 45*1024-($-block1) db 0

     ; : quoted? check if 1st & last are quotes '

   times 46*1024-($-block1) db 0

      : rect ( x y w h -- / draw rectangle with midpoint x y *)
        rvertex 4 ctrace ;

      : nrect ( x y w h n -- / draw rectangle, midpoint x y, n thick *)
         0 do
          ; x y w h
          4dup ii + swap ii + swap rect
         loop ;

      ; might be better to rewrite this with a begin/until loop
      ; adding the degree increment each time. This would allow us
      ; to set a start angle for the points (so we could also do
      ; circular animations.
      : ccpoints ( x y r n -- / get n points on circle radius r, mid [x,y] *)
        0 do
         ; x y r
         >r 2dup r>
         ; x y x y r
         360 iimax / ii *
         ; x y r x y r t
         swap >r r@ polar
         ; x y ... x' y'    r: r
         2swap r>
        loop 2drop drop ;

   times 47*1024-($-block1) db 0

   ; standard forth
   : \ 10 parse 2drop ; imm

   : { ( -- / we cant do local variables... yet *)
       [char] } parse 2drop ; imm

   ; Bumgarner sin function  (job-82mar31)
   ; from Ron Geere: Forth: the next step
   ; not working!, not leaving anything on the stack
   var xs 0 xs ! ( square of scaled angle )
   : kn ( a b -- m / m=10000-ax*x/b ..common term in series *)
      xs @ swap / minus 10K */ 10K + ;

   : (sine) ( theta -- 10K*sin /-15708<theta<15708 radians * 10K *)
     dup 10K */ xs ! ( save x^2 ) 10K ( start series )
     72 kn 42 kn 20 kn 6 kn       10K */ ( times x ) ;

   : sine ( theta -- 10K*sin / 0-90 degrees only *)
       17435 100 */ (sine) ; ( deg to radians*10K )

   \ sin(x)=x*(1-x^2/6 (1-x^2/20 (1-x^2/42 (1-x^2/72 )))) approx


     ; at compile time, compile bytes into a buffer. At run-time
     ; fetch an item from the buffer. This is a good test for my
     ; forth machine, probably wont work.
     : cvector ( n -- / compile n byte off stack into buffer *)
         create 0 do c, loop does> + c@ ; imm

     : vector ( n -- / compile n 16bit cells off stack into buffer *)
         create 0 do , loop does> + @ ; imm

     ; create and initialise a list vector (see list: for comparison)
     ; from the stack. lvector has the format
     ; 16bit length, 16bit data cells (no capacity cell)
     : lvector ( n -- / create and init list vector {length,data} *)
         create dup , 0 do , loop does> ; imm

   times 48*1024-($-block1) db 0
     ; create and initialise a capacity list vector (same as list: )
     ; from the stack. llvector has the format
     ; 15bit capacity, 16bit length, 16bit data cells
     ; use the words li/addu etc to manipulate it
     : llvector ( c n -- / create and init list vector {cap,length,data}  *)
         create
           \ 1st 2 bytes are capacity
           swap dup ,
           \ 2nd 2 bytes are length
           swap dup ,
           \ init n words from stack
           ; n
           0 do , loop
           ; set remaining capacity bytes to zero
           ; bug need to do c-l then allot
           ; c
           allot0
           does> ; imm

     ; There is always at least the "forth" (core?) namespace, which
     ; points to the core dictionary
     ;
     ; make NS point to the Forth NS. But when we add new words
     ; this becomes out of date, so create should update this,
     ; or we should just use Forth instead of "last/latest"
     ; Or make "last" do Forth @

     ; s/o is the search order buffer. Consists of a length
     ; (initially 1 ?) and a list of pointers to wordlists
     ; This is defined in bytecode now, feb 2019
     ; 16 buffer: s/o
     ; search order has nothing in it.
     ; s/o 0 swap !
     ; an nset is a set of 16 bit cells with length n (no capacity cell)
     : nset/dump ( A -- / display contents of nset at A *)
        dup @ dup ." [" u. ." items ] "
        dup 0= if drop exit fi
        ; A n
        swap 2 + swap
        ; A+2 n
        0 do
          ; A'
          @+ . space
        loop ;

     ; x and y steps for pong
     var xx var yy var delay
     : pong.header
        10 0 atxy
        .yellow ." Ball Bounce!" cr
        .white
          ." q: quit, r: reverse, f: faster, s: slower" cr
          ." x: xstep++, X: xstep--, y: ystep++, Y: ystep--" cr
        ; erase previous messages
        80 spaces cr 80 spaces cr 80 spaces cr
        0 3 atxy
          ." xstep=" xx @ .
          ."  ystep=" yy @ .
          ."  delay=" delay @ . cr
          ." stack: " .s cr ;

   times 49*1024-($-block1) db 0

     ; look up s/o and get last item
     ; in s/o buffer (s/o is like a stack, and last item is
     ; top of stack). Should context be a p* -> p* like current

     ; Defined in bytecode now
     ;: context ( -- A /
     ;   A is p* to first wordlist in search order *)
     ;   s/o dup @ 2* + ;

     ; ( pointer to data-field of wordlist for new definitions *)
     ; This will probably be defined in bytecode, so that "create"
     ; can use it to compile the dictionary.
     ; defined in bytecode
     ; var current

     ; eg: Forth ' dup setlast
     ; This is useful for pretending that a new wordlist is
     ; an old one
     : setlast ( xt -- / set last word in "context" wordlist *)
         context @
         ; xt A
         ! ;

     ; Make Forth the principal definition wordlist
     ; We need to do "defs" after setting lastword with
     ; setlast...
     ; also Forth
     ;Forth
     ;last @ setlast
     ; Forth defs

     : pong ( -- / bounce a ball around )
         1 xx ! 1 yy ! 20 delay !
         cls
         pong.header
         200 200 115 115 7 nrect
         ; initial pong position
         130 180
         ;2000 0 do
         begin
           ; x y
           key? if
             key
             dup c: q = if cls ." goodbye!" cr exit fi
             ;iskey 'q' if ." goodbye!" cr exit fi
          dup c: f = if
            delay @ 3 - 1 max delay !
          fi
          dup c: s = if
            delay @ 3 + delay !
          fi
          dup c: x = if xx ++ drop fi
          dup c: X = if xx -- drop fi
          dup c: y = if yy ++ drop fi
          dup c: Y = if yy -- drop fi
          c: r = if
            xx @ neg xx !
            yy @ neg yy !
          fi
          pong.header
        fi
        ; x y
        ; set initial colour of shape
        ; 6 fg 2dup square
        .lgreen 2dup 10 fcircle
        ; speed determined by millisecond delay
        delay @ ms
        ; delete the shape by setting colour to black
        .black 2dup 10 fcircle
        ; check y-bounds (vertical walls)
        ; x y
        dup 100 300 within ifnot
          yy @ neg yy !
          pong.header
        fi
        ; or check each wall one by one
        ;dup 200 >= if yy @ neg yy ! fi
        ;dup 20 < if yy @ neg yy ! fi
        swap
        dup 100 300 within ifnot
          xx @ neg xx !
          pong.header
        fi
        swap
        yy @ + swap xx @ + swap
      again ;

times 50*1024-($-block1) db 0

times 51*1024-($-block1) db 0

  : colorboxtype ( A n -- / coloured box with text *)
     tuck getxy
     ; n A n x y
     ?1- swap ?1- swap atxy
     ; n A n
     218 emit 196 swap emitn 191 emit over getxy
     ; n A n x y
     1+ >r swap 2 + - r>
     ; n A x-n-2 y+1
     atxy 179 emit
     ; n A
     over rainbow 179 emit
     ; n
     dup getxy
     ; n n x y
     1+ >r swap 2 + - r>
     ; n x-n-2 y+1
     ; 2dup swap c> [ . c> , . c> ]
     atxy 192 emit
     196 swap emitn 217 emit ;

  : .cbox" ( -- / print boxed coloured text line *)
      post s" [call:] colorboxtype ; imm
```

```
times 52*1024-($-block1) db 0

  ; These always act on double precision numbers ( 2 items on stack )
  : #  ( ud â\200\223 ud’ / divide u by base,
     prefix remainder digit to pad as asci char *)
    nop ;
  : #s  ( ud â\200\223 ud’ / convert all remaining digits using # *)
    nop ;
  : hold ( c -- / prefix char c to numeric string *) ;
  : sign ( n -- / if n<0 prefix ’-’ to numeric string *) ;
  : <# ( -- / clear the numeric output string. *) ;
  : #> ( xd â\200\223 A u / discard xd and return numeric string *) ;

times 53*1024-($-block1) db 0

  ( Ideas from pforth )

  ;: comp? state @ if ( not compiling ) abort" ohoh" fi ;
  (
   If echo is on then forth code will be echoed as it
   is compiling.
  )
  var echo
  ; I like this idea of not having a fixed buffer for
  ; pad. Can use for obuff. This doesnt work in my
  ; current forth because I use an anonymous buffer
  ; to compile to.
  : PAD here 128 + ;

times 54*1024-($-block1) db 0

times 55*1024-($-block1) db 0

  : s->d todo! ;

  ; This is normally a primitive too
  : rshift ( n a -- n’ /right shift integer n, a places *)
    0 do 2/ loop ;

  ; this should be a primitive I think
  : um+ ( n m -- d / leave n+m as double number *)
     2dup + >r ( n m   r: n+m)
     15 rshift swap 15 rshift and r> swap ;

  ; words from Ting/Muench eforth
  : hld ( hold pointer in building numeric string ) pad ;

  ; um+ can be used as a primitive for a number of words
  : negate ( n -- -n)
     not 1 + ;
  : dnegate ( d -- -d)
     not >r not 1 um+ r> + ;
  : d+ ( d e -- d+e )
     >r swap >r um+ r> r> + + ;

  ; defining opcodes as words. This may be useful for creating
  ; systems with very few opcodes (and therefore, in theory)
  ; easier to port to new architectures. But we need a mechanism
  ; to actually register these words as opcodes in the opcode
  ; table, and to execute them.
  : ?dup ( n -- n n | 0 ) dup if dup fi ;
  ; : not -1 xor ;
  ; : + ( a b -- a+b ) um+ drop ;
  ; : - negate + ;
  ; : = ( n n -- F ) xor if 0 exit fi -1 ;
  ; : here dp @ ;
  ; This pad is similar to pforths definition
  ; : pad here 80 + ;
```

```
    : @execute ( A -- ) @ ?dup if execute fi ;

  times 56*1024-($-block1) db 0

    ( classic numeric output words, from eforth, ting/muench )
    ( ans forth seems to use doubles for these words )
    ; The words below assume pad is defined as
    ; : pad here 80 + ;
    ; So hld actually used the data space before pad to
    ; build the numeric string.

    ; this contains some cool trick to convert to hex digits
    ; without using a lookup table.
    : digit ( u -- c / convert u to asci digit [0-15?] *)
       9 over < 7 and + 48 + ;

    ; is um/mod working?
    : extract ( n base -- n’ c / extract 1 {right-most} digit from n *)
       0 swap um/mod swap digit ;

    var hld

    ; initialiase pad to the dictionary pointer. But dp should
    ; be called "dp" or "cp"
    : <# ( -- /initialize numeric string buffer *)
       dp @ 80 + hld ! ;

    : hold ( c -- / left-affix char c to numeric string buffer *)
       hld @ 1 - dup hld ! c! ;

    : # ( u -- u’ / extract and left-affix 1 digit to hld *)
       base @ extract hold ;

    : #s ( u -- 0 / extract all digits *)
       begin # dup 0= until ;
       ; eforth is: begin # dup while repeat

    : sign ( n -- / prefix sign of n to numeric string *)
       0< if 45 hold fi ;

    ; this is crashing because pad is a buffer not : pad here 80 +
    : #> ( w -- A n / leave address and count of numeric string *)
       drop hld @ dp @ 80 + over - ;
      ; The dp @ is a cludge, it should just be "here" but
      ; my "here" bounces around between the dictionary and the
      ; anonymous buffer, which needs to be fixed
      ;
      ; The eforth def is simpler because hld is initialised as pad
      ; drop hld @ pad over - ;

  times 57*1024-($-block1) db 0

    : str ( n -- b u / convert signed integer to numeric string *)
       dup >r abs <# #s r> sign #> ;

    : .R ( n +m -- / display int in m width field, right justified *)
       >r str r> over - spaces type ;

    : U.R ( u +n -- / dislay unsigned int in n width field *)
       >r <# #s #> r> over - spaces type ;

    : U. ( u -- / display unsigned integer in current base *)
       <# #s #> space type ;

    ; if base is 10 then display signed in otherwise show unsigned
    : DOT ( n -- / display un/signed integer in current base *)
       base @ 10 xor if
         U. exit
       fi
       str space type ;

     : ? ( A -- / display contents of memory cell *)
        @ . ;


   times 58*1024-($-block1) db 0

   ; display u bytes of data, better than dump
   : dumpx ( addr u -- )
       ; print mem address
       ; A u
       0 do
         ; swap dup u. ." : " swap
         ii
         ; print memory address and newline every 8 chars
         ; A ii
         8 mod not
         if cr 11 fg dup u. ." :" 15 fg fi
         ; A
         c@+ u. space
         ; A+1 c
       loop drop ;

   times 59*1024-($-block1) db 0

   ; Idea!! what about doing
   ;  s" r> r> 1+ >r" source/compile
   ; as source

   times 60*1024-($-block1) db 0

    ; display all words in buffer
    : inwords ( display all words in buffer *)
        term accept term count in0
        begin
          wparse 2dup type cr
        0= until ;

    ; Testing tick by accepting input and
    ; db ’ displaying the found execution token
    : try.tick ( testing "tick" )
      begin
        term accept term count
        dup 0 = if exit fi
        tick    ./ 0/n/op.xt flag=0/1/2/3
        cr .s cr
      again ;

   times 61*1024-($-block1) db 0

   : ww. last @
      begin dup .xt space xt+ dup 0 = until drop ;

   times 62*1024-($-block1) db 0

   ; shortjump limited to +/- 128 bytes)
   : Again ( +/-128 byte jump "again" *)
       [op:] jump here 1- - c, ; imm

   times 63*1024-($-block1) db 0

   : ffor ( run: u -- / compiled "for", iterate u times *)
       [op:] dup [op:] >r [op:] >r here ; imm
   : nnext ( compiled "next", +/-128 byte jump back to "for" )
       [op:] rloop
       here 1- - c,
       [op:] r> [op:] r> [op:] drop [op:] drop ; imm
```

```
; short jump loop compiling all code (instead of using a helper
; function
: Lloop ( compiled short jump loop )
  [op:] r> [op:] r>
  [op:] 1+ [op:] 2dup
  ; 2>r is an opcode
  [op:] >r [op:] >r
  [op:] =
  [op:] jumpz
  here 1- - c,
  [op:] r> [op:] r>
  [op:] drop [op:] drop ; imm

: lloop ( compiled long jump loop )
  [op:] r> [op:] r>
  [op:] 1+ [op:] 2dup
  [op:] >r [op:] >r
  [op:] =
  [op:] jumpnz 5 c,
  [op:] ljump
  here 1- - ,
  [op:] r> [op:] r>
  [op:] drop [op:] drop ; imm

times 64*1024-($-block1) db 0

  ; parse as source, also handle eol, 10 or end of input
  ; stream
  : Parse ( char -- adr n )
  ; scan input stream for char and return length and address.'
  ;  The current input stream (IN) is scanned for the next char'
  ;  matching char. The parse position of the stream is advanced '
  ;  after this call. If not found, then parse position and 0'
  ;  is returned.'
  >in dup 0= if drop exit fi
  ; c A n
  for
    ; c A'
    c@+ >r over r>
    ; c A'+1 c b
    = if ( advance "in" ) fi
  next
  nop ;

times 65*1024-($-block1) db 0

  var span
  ; basic version working
  : accept.span ( A u -- / get text into buffer A, no more than u chars *)
    0 span !
    swap
    begin
      ekey
      1 = if drop else
        ; u A k
        dup emit
        dup 13 = if exit fi
        ; u A k
        over span ++
        ; u A k A s+1
        + c!
        ; u A
        dup span @
        ; u A A s
        swap c!
        ; u A
        over span @
        ; u A u s
        = if exit fi
      fi
    again ;

times 66*1024-($-block1) db 0

; So the foreach loop has pointer+limit on return stack
; each iteration decrements/increments the point and checks
; against limit Address value.

; this foreach loop should be faster than for/next using ii
; hopefully. But slower than for/next manually advancing pointer

; adapt for "foreach" pointer iterator
: foreach ( run: A u -- / iterates pointer from address A, u times *)
    ; here calculate limit address A' (ie A +/- 2*n)
    [op:] 2* [op:] over [op:] +
    ; A A'=limit-address
    ; put the pointer on top and the limit underneath
    [op:] >r [op:] >r here ; imm

; check for foreach ...
; The top rstack param is the return address for check
: (check.foreach)
    r> r> 2 + r> 2dup >r >r = swap >r ;

: (check.eachfor)
    r> r> 2 - r> 2dup >r >r = swap >r ;

; adapt for foreach
: nextitem ( long jump pointer loop )
  [call:] (check.foreach)
  [op:] jumpnz 5 c,
  [op:] ljump
  ; leave jump-back target on data stack, to be used by
  ; "nextitem"
  here 1- - ,
  [call:] (clean) ; imm

times 67*1024-($-block1) db 0

  : wfind.slow ( A n xt -- xt'|0 / find name at A starting search at xt *)
     begin
       ; A n xt
       >r 2dup r@
       ; A n A n xt    r: xt
       ; this needs to be faster, so dont use xt>name
       xt>name
       ; A n A n B m   r: xt
       ; parsing out s= for speed
       s=
       ; s= inline, but not much difference
       rot over
       ; A B v u v
       ; if lengths not equal, false. 2drop and drop are opcodes
       <> if 2drop drop 0 else ncompare fi

       ; A n F         r: xt
       IF 2drop r> exit FI
       r>
       xt>name
       ; parse out "xt+" for speed
       ; xt+ code:
       drop 1- 1- @
       dup
     Until0 2drop drop 0 ;

times 68*1024-($-block1) db 0

  ; test prefix with a loop.
```

```
    : t:prefix
       ." testing 'prefix': double enter to exit." cr
       begin
        ." Text:" pad accept
        ." Prefix:" buff accept
        pad count buff count
        ;." stack: " .s cr
        dup not if exit fi
        prefix
        ;." stack: " .s cr
        if ." True!" else ." Not a prefix" fi cr
       again ;

    ; a loop to test search.
    : ..search
       ." testing 'search': enter to exit." cr
       begin
        ." Text:" pad accept
        ." Search:" buff accept
        pad count buff count
        dup 0 = if exit fi
        ; A n P m
        search
        ; A n f
        ." stack: " .s cr
        if
          ." Found in 'pad':[" pad 1+ u. ." ]"
          ." at " drop u. cr
        else
          ." Not found" 2drop
        fi cr
       again ;

  times 69*1024-($-block1) db 0

    : summit ( x y r -- a b ) - ;
    : >east ( x y -- x+1 y , one pixel to east) swap 1+ swap ;
    : >south ( x y -- x y+1 ) 1+ ;
    : >west ( x y -- x-1 y ) swap 1- swap ;
    : >southeast >south >east ;
    ; this will fail for big numbers eg 400 squared
    : dist2  ( a b x y -- d^2 , distance between 2 points squared)
       >r swap >r - dup * r> r> - dup * + ;
    ; idea, if the east error is less than southeast error
    ; then it will also be less than south error (at least
    ; for first quadrant of a circle.

    ; [a,b] center of circle radius 'r'
    : easterror ( a b r x y -- error for east point)
       2>r -rot 2r> >east ( r a b x-1 y )
       dist2 swap dup * - abs ;
    : s/easterror ( a b r x y -- e )
       2>r -rot 2r> >southeast ( r a b x+1 y+1 )
       dist2 swap dup * - abs ;
    : southerror ( a b r x y -- e )
       2>r -rot 2r> >south ( r a b x y+1 )
       dist2 swap dup * - abs ;

    : nextpoint ( a b r x y -- x' y' find adjacent point of circle )
       5dup easterror >r
       5dup s/easterror dup r>  ( a b r x y e e e2 )
       > if drop >east exit fi
       >r
       5dup southerror r>
       > if >southeast exit fi
       >south ;
    ; here if easterror<south.easterror then quit.

  times 70*1024-($-block1) db 0
```

```
    ( a new circle drawer using errors )
    : ncircle ( a b r -- )
       3dup summit ( a b r x y)
       begin
         ; a b r x y
         5dup rot drop
         2>r 2dup 2r> ( a b a b x y )
         rot - abs >r - abs r>
         rvertex 4 pplot nextpoint
         5dup >r 2drop swap drop r> =
       until 4drop drop ;

    ; testing animation of a circle
    : anim.circle ( -- )
       .lgreen 50 for 100 ii + 150 50 ncircle next ;

    : xmirror ( x y u -- x y x' y, mirror point about x axis)
       >r 2dup swap r@
       ; x y y x u  R: u
       - r> swap - swap ;

    ; could test xmirror
    : T{xmirror} nop ;

    : ymirror ( x y u -- x y x y', mirror point about y axis 'u')
       >r 2dup r@ - r> swap - ;

    : 2ymirror ( [a,b] [x,y] u -- a b a b' x y x y',
        mirror 2 points about y axis 'u')
       -rot 2>r >r r@
       ; a b u R: u x y
       ymirror 2r> r> rot ymirror ;

    : ball ( a b r -- a filled circle )
       3dup summit ( a b r x y)
       begin
         ; a b r x y
         5dup rot drop
         ; a b r x y a b x y
         2swap >r xmirror r>
         2ymirror lineto lineto
         nextpoint
         5dup >r 2drop swap drop r> =
       until 4drop drop ;


  ; move cursor up one row (check for zero)
  : cur.up getxy 1- atxy ;
  ; move cursor down one row
  : cur.down getxy 1+ atxy ;
  ; move cursor left one (need to check for zero)
  : cur.left getxy swap 1- swap atxy ;
  ; move cursor right one
  : cur.right getxy swap 1+ swap atxy ;

  times 71*1024-($-block1) db 0

   : balls ( testing ball )
    vmode
    cr ." Testing ball drawing..." cr
    16 for ii fg ii 30 * 200 15 ball next ;

   : ball.move
    vmode
    cr ." Testing ball animation..." cr
    100 for
      .green ii 100 + 200 15 ball 1 ms
      .black ii 100 + 200 15 ball
```

```
     next ;

   ; move the cursor around with arrow keys
   : arrow
      ; a video mode with a cursor
      3 vid
      ." arrow keys!! q to exit " cr
      begin
        ekey 1 =
        if
          dup 72 = if cur.up fi
          dup 75 = if cur.left fi
          dup 77 = if cur.right fi
          dup 80 = if cur.down fi
          drop
        else
          [char] q = if 16 vid exit fi
        fi
      again ;
      ; reset the video mode

   : 2var create 0 , 0 ,  does> ; imm

   ; keycodes and values
    72 con k.up
    80 con k.down
    75 con k.left
    77 con k.right
    81 con k.pgdn
    73 con k.pgup
    71 con k.home
    79 con k.end
    82 con k.insert
    83 con k.delete
    8 con k.back

   times 72*1024-($-block1) db 0

       vocab Asci.draw
       Asci.draw defs
       ; make variables for the asci characters to draw the box,
       ; this way we can use the same routines to draw boxes with
       ; different characters
       var ne.corner var topline var nw.corner var side
       var se.corner var botline var sw.corner
       var wjoin var ejoin

       : single/box
         218 nw.corner ! 196 topline ! 191 ne.corner ! 179 side !
         192 sw.corner ! 196 botline ! 217 se.corner !
         ; for title boxes with dividers...
         195 wjoin ! 180 ejoin ! ;

       : margin/box
         214 nw.corner ! 196 topline ! 183 ne.corner ! 186 side !
         211 sw.corner ! 196 botline ! 189 se.corner ! ;

       : double/box
         201 nw.corner ! 205 topline ! 187 ne.corner ! 186 side !
         200 sw.corner ! 205 botline ! 188 se.corner ! ;

       single/box

       : top ( x -- / print top of box, width x )
           nw.corner @ emit 2 - for
             topline @ emit
           next ne.corner @ emit ;

       : sides ( x -- / print 1 row of sides of box width y )
           side @ emit getxy >r + 2 - r> atxy side @ emit ;

       : bottom ( x -- / print bottom of box, width x )
           sw.corner @ emit 2 - for
             botline @ emit
           next se.corner @ emit ;

       ; a horizontal line which divides a box
       : divider ( x -- / horizontal box divider ) ;

   times 73*1024-($-block1) db 0

       : nextrow ( x -- / position cursor back one row under )
           getxy 1+ >r swap - r> atxy ;

       : linebox ( x -- / a one line asci box of width x )
         dup top dup nextrow dup sides dup nextrow dup bottom ;

       ;single/box

       : drawbox ( x y -- / an asci box with width x height y )
           over top for
             dup nextrow dup sides
           next
           dup nextrow bottom ;

       vocab Editor
       Editor defs
       ;( simple block editor, factored  )

       ; either edit=0 or command=1 (like vi modes)
       var ed,mode
       ; current block being edited, also see blk
       var ed,block
       ; start address of buffer
       var ed,buffer
       ; a command buffer
       16 buffer: ed,command
       ; text insertion pointer (an address)
       var ed,insert
       ; buffer address of previous screen
       var ed,prev
       ; buffer address of next screen
       var ed,next
       ; buffer address of current screen
       var ed,screen
       ; default colour for block text
       var ed,tint
       ; colour for editing box
       var ed,boxtint

       0 con edit/mode 1 con com/mode

       : edit/mode? ed,mode @ edit/mode = ;
       : com/mode? ed,mode @ com/mode = ;

       : remaining ed,buffer @ 1024 + ed,screen @ - ;

       ; buffer end address
       : end ed,buffer @ 1023 + ;

       ; number of bytes after insert point
       : after end ed,insert @ - ;

   times 74*1024-($-block1) db 0

       ; address and remaining length of current screen
       ; probably dont need this word
       : ed,at ed,screen @ ed,buffer @ 1024 + ed,screen @ - ;
    : cursor? ed,insert @ = ;

    ; go to start of editing buffer
    : >start
       ed,buffer @ dup ed,screen ! ed,insert ! ;

    ; number of used bytes in buffer
    : bsize ( A -- n ) dup begin c@+ 0 = until swap - ;

    ; how many free bytes in editor buffer
    : buffer.free ed,buffer @ bsize 1024 swap - ;

    : firstchar? ( A -- f ) ed,buffer @ = ;

    ; cursor at the start of the buffer?
    : cur.firstchar? ed,insert @ firstchar? ;

    ; at the end of the buffer?
    : end? ed,insert @ ed,buffer @ 1023 + = ;

    : emit+ ( A n -- A+1 n-1 / emit 1 char and advance *)
       dup 0= if exit fi
       ; set text colour
       ed,tint @ fg
       ; background colour if this char is at the cursor
       over cursor? if
         2 bg
         ; if buffer is full make cursor cyan
         buffer.free 2 < if 5 bg fi
         end? if 5 bg fi
       else 0 bg fi
       1- swap c@+ emit swap 0 bg ;


    ; print until next newline or maximum n chars, then place cursor
    ; just under start of line. Also highlight current insertion point.
    : textline ( A n -- A+x n-x / print until next newline, place cursor *)
       ; save screen cursor pos
       getxy 2>r
       begin
         dup 0= if 2r> 2drop exit fi
         over c@ 10 = if
            1- swap 1+ swap
            ; adjust screen cursor
            2r> 1+ atxy
            exit
         fi
         emit+
       again ;

    : lines# ( -- / screen lines below cursor )
        19 getxy swap drop - ;

times 75*1024-($-block1) db 0

    ( screen positions for messages )
    : at/mode 2 0 atxy ;
    : at/free 50 0 atxy ;
    : at/title 32 0 atxy ;
    : at/line+char 60 23 atxy ;
    : at/info 5 23 atxy ;

    : .info at/info
       ;.brown ."  buffer*:" .green ed,buffer @ u.
       ; insertion point address
       ;.brown ."  cursor*:" .green ed,insert @ u.
       ." command: " ed,command count type
       ; show char under cursor
       ."  [asc=" ed,insert @ c@ u. ." ]" ."  stack depth:" depth u. ;

    : full? ( -- f / is the edit buffer full?)
      ed,buffer @ 1022 + c@ 0 <> ;

    : newline? 10 = ;
    : null? 0 = ;

    ; working
    : cursor.line ( / line number of cursor )
        ; 0 exit
        cur.firstchar? if 1 exit fi
        1 nn ! ed,buffer @
        begin
          dup c@ newline? if nn ++ drop fi
          1+ dup ed,insert @ =
        until drop nn @ ;

    : screen/start? ( -- f /is edit cursor at start of screen? )
       ed,screen @ ed,insert @ = ;

    : line<start ( A -- A' / address of 1st char on line)
       dup firstchar? if exit fi
       begin
         1- dup firstchar? if exit fi
         dup c@ newline? if 1+ exit fi
         dup ed,buffer @ < if ." error!" exit fi
       again ;

times 76*1024-($-block1) db 0
    : line>end ( A -- A' / last char of line)
       begin
         1+ dup c@ null? if 1- exit fi
         dup c@ newline? if 1- exit fi
       again ;

  ;  : cursor.topscreen? ( -- f / is cursor at 1st line of screen )
  ;      ed,cursor @ >startline

    : line<< ( A -- A' / start of previous line)
       line<start dup firstchar? if exit fi 1- line<start ;

    ; name clash with the other line>>
    : line>> ( A -- A' / start of next line)
       line>end 1+ dup c@ null? if 1- exit fi ;

    : screen.line<< ( -- /scroll screen up one line)
       ed,screen @ line<< ed,screen ! ;

    : screen.line>> ( -- /down one line)
       ed,screen @ line>> ed,screen ! ;

    : cursor.line<< ( -- /cursor to start of prev line)
       screen.line<< ed,insert @ line<< ed,insert ! ;

    : cursor.line>> ( -- )
       screen.line>> ed,insert @ line>> ed,insert ! ;

    : screen.page>> ( -- /down one page)
       ;19 for screen.line>> cursor.line>> next ;
       19 for cursor.line>> next ;

    : screen.page<< ( -- )
       ;19 for screen.line<< cursor.line<< next ;
       19 for cursor.line<< next ;

    : linechar ( -- cursor offset for current line )
        ed,insert @ dup line<start - 1+ ;
```

```
times 77*1024-($-block1) db 0

    : .freebytes ( -- free bytes message )
       at/free .blue ." (free " .lblue buffer.free u. .blue ." /1024)"
       full? if .lblue 6 bg ."  block full!" 0 bg fi ;

    : .mode ( -- / show editor mode)
         at/mode .pink ." mode:" .brown
         edit/mode? if ." edit" exit fi
         com/mode? if ." command" exit fi ." ??" ;

    : .title at/title .lgreen ." DISK BLOCK " ed,block @ u. ;

    : .line+char at/line+char
         .pink ." line " cursor.line u.
         ."  char " linechar . ;

    : screen ( A n -- A' n' / display screenful of text *)
        lines# 2 - for textline next ;


    : lines ( A n x -- A' n' / display x lines of text *)
        for textline next ;

times 78*1024-($-block1) db 0

    also Asci.draw

    : edit/box ( draw the edit box)
       ed,boxtint @ fg 0 1 atxy 80 20 drawbox ;


    : layout ( -- / display current block screen in editor layout *)
        cls
        .mode .title .freebytes .info .line+char
        edit/box
        ; put text in the box
        1 2 atxy ed,at 19 lines 2drop ;

    : >edit/mode
        edit/mode ed,mode !
        0 ed,command !
        purple ed,boxtint ! white ed,tint ! layout ;

    : >com/mode
        com/mode ed,mode !
        grey ed,boxtint ! grey ed,tint ! layout ;

    : startup
        0 ed,command !
        margin/box >edit/mode ;

    ; load and initialise variables for block editing
    : getblock ( b -- A )
        dup ed,block !
        block
        ; buffer start pointer
        dup ed,buffer !
        ; current screen address
        dup ed,screen !
        ; previous screen address
        dup ed,prev !
        ; init insertion point to somewhere
        dup 3 + ed,insert ! ;

    ; last address in the buffer
    : buff>> ed,buffer @ 1024 + ;

    : char<< ( back 1 char in buffer )
       cur.firstchar? if exit fi
       ed,insert --
       ; skip over newline char
       c@ 10 = if ed,insert -- drop fi
       ; go back to start (should be one screen)
       ed,insert @ ed,screen @ < if >start fi ;

    : char>> ( forward 1 char in buffer )
       ;ed,insert @ ed,buffer @ 1024 + = if exit fi
       end? if exit fi
       ed,insert ++
       ; skip over newline char
       c@ 10 = if ed,insert ++ drop fi
       ed,insert @ ed,next @ = if screen.page>> fi ;

    ; need to do limit checks here...


times 79*1024-($-block1) db 0

    : ins ( k -- / insert char k )
        ; check for buffer full
        full? if drop exit fi
        ; change carriage return to newline
        dup 13 = if drop 10 fi
        ed,insert @ dup 1+ after 1- cmove
        ed,insert @ c! ed,insert ++ drop ;


    ; bug when buffer full
    : delete< ( -- / delete 1 char left of insert )
        cur.firstchar? if exit fi
        ed,insert @ dup 1- after 1- cmove
        ed,insert -- drop ;

    : delete> ( -- / delete 1 char right of insert )
        end? if exit fi
        ed,insert @ 1+ ed,insert @ after 1- cmove ;

    ; go to end of editing buffer
    : >lastpage
       ed,buffer @ dup bsize + ed,insert !
       ed,insert @ 50 - ed,screen ! ;

    : lastchar ( A n -- last char of string) + 1- c@ ;


times 80*1024-($-block1) db 0

  : next/block ed,block ++ getblock layout ;
  : last/block ed,block -- getblock layout ;

  : do/command ( -- performs an editor command )
    ; just print com buffer
      ed,command count
      dup 0= if 2drop exit fi
      2dup s" i" s= if >edit/mode 0 ed,command ! fi
      2dup s" h" s= if char<< 0 ed,command ! fi
      2dup s" l" s= if char>> 0 ed,command ! fi
      2dup s" j" s= if cursor.line>> 0 ed,command ! fi
      2dup s" k" s= if cursor.line<< 0 ed,command ! fi
      2dup s" J" s= if screen.page<< 0 ed,command ! fi
      2dup s" K" s= if screen.page>> 0 ed,command ! fi
      2dup s" n" s= if next/block 0 ed,command ! fi
      2dup s" N" s= if last/block 0 ed,command ! fi
      lastchar newline? if 0 ed,command ! fi ;

times 81*1024-($-block1) db 0
```

```
      ; simple block editing
      : ed ( n -- / edit block n *)
          startup getblock
          layout
          begin
            ; check if key pressed
            key? if
              ekey
              if
                ; special keys (arrow etc)
                ; k
                dup k.left = if drop char<< 0 fi
                dup k.right = if drop char>> 0 fi
                dup k.up = if drop cursor.line<< 0 fi
                dup k.down = if drop cursor.line>> 0 fi
                dup k.pgdn = if drop screen.page>> 0 fi
                dup k.pgup = if drop screen.page<< 0 fi
                dup k.home = if drop >start 0 fi
                dup k.end = if drop >lastpage 0 fi
                dup k.delete = if drop delete> 0 fi
                drop
              else
                ; "normal" keys
                ; escape goes to command mode or exits in com/mode
                dup esc = if
                   com/mode? if drop exit fi
                   edit/mode? if >com/mode drop 0 fi
                fi
                dup k.back = if drop delete< 0 fi
                ; insert char k at ed,insert
                dup 0 <> if
                  edit/mode? if ins fi
                  com/mode? if
                    ; actually here add the key stroke to
                    ; a command buffer and check for valid command
                    ; execute, then empty command buffer. This
                    ; allows multiple char commands eg 'dd' delete
                    ; line
                    ed,command s+c do/command
                    ;dup [char] i = if >edit/mode fi
                    ;dup [char] j = if screen.line>> fi
                    ; drop
                    ; here give unknown command status message?
                  fi
                else drop
                fi
              fi
              ; only redraw the text if a key is pressed
              layout
            fi
          again  ;
        also Forth defs

   times 82*1024-($-block1) db 0
   times 83*1024-($-block1) db 0

   %endif ; }code

   ; the %if 0 trick below allows including forth source code without
   ; hundreds of "db" lines. But this source needs to be preprocessed
   ; to insert the dbs before every line

   %if 0

FORTH CHESS NOTES

  A sketch of some chessy words. Before implementing this, we
  need working vocabularies, also probably a block editor
  . See also the chess.txt file in the folder "osdev" for more chess code.
```

```
board: 0-11, 12-23, 24-35 ...
.square ( n -- ) print a square number (12*8 board: 0-97) in
   chess algebraic eg a3, b2 h1.
.move ( from to -- ) print a chess move where from and to are squares
   on a 12*8 board.
.game ( C -- ) display the state of the game object, including the
   board and pieces, the moveVector (Pvector), the squareVector (current
   move piece or square).
move>> ( C -- from to )  the next legal move for game "C"
   The move is returned on the stack. The game structure C is not
   altered (move+ does that). This will return 0 0 if there are no
   more legal moves on the board for the current player
move+  ( C from to -- )  update the board position of C with the
   given move. The Pvector in the C game object will be advanced, or
   the Svector (current piece/square) will be advanced with piece+.
piece>> ( C -- square ) the next piece on the board for the player colour
   This will return 0 if no more pieces to check for the current player
   The C game object is not altered (piece+ does that)
piece+  ( C -- square ) update the current piece of position C with the
   given move
piece vectors:
  : vknight [ n c, 1 c, 2 c, ... 0 c, ] ;
  : vpawn   [ ... ] ;
The structure of the piece vector is the displacement value, (eg +12 for a
white pawn), then the multiplier (eg 4 for a rook moving 4 squares), and
then the multiplier limit (one more than the maximum value).

When C move+ executes it updates the value of Pvector in the game object,
either by increasing the multiplier by one, or by advancing the Pvector
pointer to the next piece direction (Pvector + 3), or by calling C piece+
(if the current piece has no more legal moves... ie the termination 0 of the
vector has been reached).

piece vector table:
  : vtable  2 c, ' vwpawn c, 6 c, ' vknight c, etc ... ];

The piece vector table is used to reestablish the Pvector after
C piece+ is called (point to the correct piece vector for the current
move square).

IDEAS FOR WORDS

    A word to tell if a character is a digit.
    : ?digit ( c -- flag ) [char] 0 [char] 9 1+ within

    words ( -- ) list all the words in the dictionary in search order
    quit
      this is the standard forth repl loop. ie read evaluate print
      loop. It is a strange name and I will call it "shell" instead.
      The quit loop allows the user to type and execute commands.
      and compile new words.

    d. ( d -- ) display the top 2 stack items as a signed
       double precision integer.
    d+ d2* d2/ double precision integer operations
       The high cell is the high order byte of the double number.

    m* ( n1 n2 -- d ) multiple n1 by n2 and leave double precision
       result d on stack

    var creates a new variable
       : var ( -- ) create
    we could define colon : with an immediate create.
    eg [create] : ... source ;
    : bl 32 ; put a space character on the stack (20H)


   ; ideas for words
```

```
of3 ( a b c n -- n/0 )
   if n == a or b or c return that value, otherwise return 0.
: of3 dup >r = if drop drop r> exit fi
      r> dup >r = if drop r> exit fi
      r> dup >r = if r> else r> drop 0 fi

flags ( -- flag-reg )
  put the overflow/carry etc flags register on the stack.
  I always wonder why standard forths dont have this. How
  do you know when an overflow or carry occurs? How can you
  call forth a virtual machine if it does not have this?

; input stream
>in ( -- a-addr )
  return address containing offset in characters from the
  start of the current input source to the start of the
  current parse point. In the current forth, ">in" works
  differently. Need to think about this design.

evaluate ( ... addr u -- ... )
  set input source to addr with length u. set >in to zero etc
  The current forth calls this in0 or setin and works differently
  A similar word might be "source" or "load".

parse <text> ( char -- addr n )
  parse text at starting at current parse point ( >in ) using
  char as the delimiter. Also copy to temp location. But
  current forth is different. Word is not copied. Also we need
  a wparse which parses to any whitespace (eg newline is a word
  delimiter too!). Maybe parse should just update the >in variable
  as well which should make things simple.

( a comment)  brackets for multiline comments
a definition of "(" but I may parse bracket comments until a dot .
so as to be able to write more "literate" forth.
  : ( char ) parse/word ...

word ( char - c-adr n ) the same as parse but skip all
  leading occurences of char. My implementation is call WPARSE
  and works slightly differently (only parses on whitespace)

;*** stack
over ( a b -- a b a )
nip ( a b -- b )
rot ( a b c -- b c a )
tuck ( a b -- b a b )

; characters
bl ( -- 20H ) return asci char 32 which is a space.

char n ( -- char )
  put the asci value of the first character
  in the next word in the input stream on the stack.
  In the current forth, this will be an immediate word.
  Not working in : COLON defs, needs to push a literal
  In many forths [char] must be used in : definitions.
  This is because the current forth compiles even words
  entered interactively.

;*** Strings
; we can implement string functions by compiling right in the
; midst of the current word and compile a JUMP over the string
; if necessary
," compile a string at current position in data-space (at "here")
s" compile a string and put its address and length on stack at
   runtime
." compile a string and print it out at runtime.

search
  find one string in another

; text display
space
  display one space on terminal
spaces ( n -- )
  display n spaces on terminal
cr
  emit one newline/ carriage return to terminal

; time and date
ms ( u -- )
  wait for u milliseconds
time&date ( -- secs mins hours days months years )
  return a structure representing time and date

; arithmetic
: */ ( a b c -- a*b/c).
  multiply 3TOS by NOS, keeping precision (double?) and then
  divide by TOS.

; mathematics
: min ( a b -- min )
  returns the lowest number of "a" and "b" (but signs ?).
: max ( a b -- max )
  returns the maximum number of "a" and "b".
: mod ( m n -- p )
  the modulus of m with n.

Euclids greatest common divisor (after R.V.Noble)
This is amazingly succinct. And recursion works!
: gcd ( a b -- gcd) ?dup if tuck mod gcd fi  ;

; sum of 1..n arithmetic series ( n -- ); eg 100 asum = 1+2+3...+100
: sum 0 do ii 0 = if 0 fi ii + loop ." sum = " u. ;
; or eg sum = n*(n-1)/2
: sum dup 1- u* 2 / ;

The following gives pi to 2 decimal places, scaled by 100
: pi 35500 113 / ;

Better, if double arithmetic is available
: pi 103993. 33102. / ;

pi.approx ( )
  pi=4/1-4/3+4/5-4/7+4/9-4/11 an infinite series, but this
  may take 500000 iterations to get to 4 decimal places
  In forth we can scale by 10000 to get 5 decimal places
  See also the inverse tangent function.

: 40K 40000 ;
: pi.approx ( -- pi-ish)
  0 1
  begin
    dup 40K swap / swap >r + r> 2 +
    dup 40K swap / swap >r + r> 2 +
    dup 64001 =
  until drop ;
The code above does 64K iterations quickly but only gets to
2 decimal places accurately ie 31407 (3.1407). Also we scale the result
by 10K because forth doesnt have floating arithmetic. We really
need double arithmetic (32 bit, 2 cells) to get a better
approximation

; ans 94 forth word
: within ( n a b )
  return true != 0 if a <= n < b, otherwise return false==0
  notice <b not <=b !
```

```
twixt ( a b n )
  return true != 0 if a <= n <= b, otherwise return false==0
  can be implemented as ": twixt dup >r min max r> = ;"
  I have used the reverse names compared to those used by
  Ron Geere, since "within" is now a fairly standard word
  with its parameter order.

squareroot ( n -- n^0.5 )
  use (n/x + x)/2 for successive approximations

; This is very cool and gets a good approximation within about
; 6 or so iterations.
: approx ( n x -- n x' )
  gets the next approximation to the square root, from ron geere.
  over over / + 2 / ;
: sqrt ( a -- b )
  return b, the approximate square root of a (maximum 32767 if the /
  division operator is signed, or else 64K if not),
  this just does the iterations of the "approx" word.
  60 5 0 do approx loop swap drop ;

;*** execution
execute ( ... xt -- ... )
  execute word specified at address xt. Called 'pcall' in this
  forth

recurse ( -- )
  append execution behaviour to current definition to allow
  for recursive functions. In this implementation words can just
  call themselves eg:
   : gcd ( a b -- gcd) ?dup if tuck mod gcd fi  ;

; dictionary and defining words
state ( -- addr )
  return address of state variable (either compile or run...)
  This is a way to implement the [ ] words which in this version
  of forth will make all words immediate when between "[" and "]".

[  ( -- )
  Enter "immediate" state. All words are executed, not compiled
  In standard forths, these are called "compile" and "interpret"
  states.
]
  leave "immediate" state. All words are compiled, unless they
  are marked immediate in their control bits (1st bit of the
  name count).

literal ( n -- )
  compile value n so that value n is put on the data stack
  at run-time. In this forth there is also and opcode LITERAL
  which could be confusing although they do almost the same
  thing.

compile, ( xt -- )
  append execution behaviour of xt to current definition.
  But because I am using opcodes as well, I have modified this
  word. Was going to call "item," because my version also
  compiles literal numbers. "opcode," is just "c,"

: unused ( -- u )
  return number of bytes of memory remaining for new dictionary
  entries.

: free first here - . ." bytes cr ;
  an older name for "unused".
  where first is the address of the first disk buffer.

>body ( xt -- addr )
  given execution token for a word, return the start of
  the parameter field (which is just after code). But
  current forth doesnt maintain a link to parameter field
  normally.

: ' <name> ( -- xt )
  search dictionary for name and return execution token
  example: A table of function pointers
  [create] buttons ' start , ' slow , ' fast , ' finish

: allot ( n -- )
  allocate u bytes of data-space, beginning at the next available
  location. Normally used immediately after "create".
  here + here! ;

create <name>
  compile "name" in the dictionary and put address on stack
  at runtime (no data space is allocated). Because all words
  are first compiled in this forth we need an immediate version
  [create] to use this outside of a : definition.
  Or use [ create ] name ...
  This was called "<builds" I believe in figforth or early
  forths.

does>
  really important word. Allows the creation of new defining words
  (eg "constant") that have a special behaviour.
  Needs to be used with create. Implentation is slightly tricky
  because no room in new word code field for a fcall to the
  does> code. So need to code a jump after the parameter field.
  Compiles a call to following words in defined words.
  * create a constant defining word
  : constant create 1 allot does> @ ;

buffer: <name> ( n -- )
  create a dictionary entry for "name" associated with n bytes of
  data-space.
  create allot ;

dump ( adr +n -- )
  display the contents of a memory region
  of length n. print address on left and 8 values per line
  in hex values or current base.

;*** blocks and buffers
list ( n -- )
  display the contents of disk block "n" (1K of source code).
  blocks are good because no file system is required.

buffer ( n -- addr )
  return address where block n may be loaded. No read is done.
  A disk write is done if necessary. Buffer manages a list
  of buffers and blocks and finds a suitable place to put the
  requested disk block, but it doesnt actually read it into
  memory.

block ( n -- addr )
  return buffer address containing disk block n. Data is
  read if necessary. No write is done

update ( -- )
  mark the last disk block accessed as having changed data
  (needs to be written to disk). This could be stored in
  the last byte of the block itself

load ( n -- )
  load disk block (1024bytes) "n" and interpret the contents
  of the block as forth source code.
```

```
    flush
      ensure all updated buffers are written to disk and free
      all buffers.

    // create a dictionary entry for name associated with 1 cell
    // eg: variable data 6 data !
    : variable ( -- ) create 1 allot ; immediate

    // create a new variable and assign a value to it.
    : value ( n -- ) create , ; immediate

    : to  ( newval to name )
    // a constant value
    : constant create , does> @ ; immediate

  %endif ; 0
```