```
%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.

  This code is 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 about 5K 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.
    "Add to itself"
      via disk, serial connection, or tcp, using globally unique and
      locatable object names (words)
    "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 a reading of several days.

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

  The question above is important because it has implications for our
  consumption of resources, impact on the environment, beholdeness
  to obsolescence etc 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, arm etc),
  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 moores 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. Some avr assembly.
  Markdown and code that produces code.

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.
   IF/THEN is IF/FI
     because I find THEN just too confusing.
   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 ...
   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 def do "char * literal" or just use [char]

 Need to do things like "[ create this ] " to test
 create interactively (but not in a : def) because this is a
 "compiling" forth.

STATUS

The system is edging towards a usable forth-style system but
still lacks a name or any way to write back to disk.

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" maybe the opcode to read a 3 axis accelerometer
 The number 1234567 must be unique, universal and defined in
   some acceptable public standard.
   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. But I found this also introduced
complications.

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 ;
```

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 no mainstream relevance apart from, but its ideas remain powerful.

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

  . 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.

 (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.

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

```
"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
```

FORTH HISTORY AND VERSIONS

```
Figforth, forth 79 standard, forth 94 standard
LMI-Forth for the IBM pc
MVP-Forth not sure where it ran
Open Firmware, by Mitch Bradley. Still used at OLPC for
  booting laptops. A major project.
cforth: based on mitch bradley olpc forth
gforth: a gnu forth probably still being developed.
amforth: for avr chips
flashforth: for microcontrollers using flash memory
```

TO DO (june 2018):

immediate tasks:

```
* write a 'dependency' compiler. Find a word on disk, leave
  its block number + offset and then find all sub-words and
  leave their block numbers + offset on the stack or in
  some kind of buffer. Check that no duplicates are added.
* incorporate boot.asm mike gonta code for translating from
  logical sector to chs. also incorporate michael petch code
  for reading sector.
* reform opcodes using jump instea of 'call' in exec.x
* think about using ax as top of stack
* understand disk geometries so we can 'read' sectors past
  sector 16(?). Since we have already hit that limit
all defining words need to be immediate, so 'does>' could
just do that automatically.
change 'decomp' so that it displays the address as well
as the name of the fcall functions.
add ", 13, 10 to the end of lines in the sed preprocessor
because normal source text files will have them.
fix sed prepocess to allow " and ' in words. Put in
own file
write search to find all words containing text
change ( to stop at . not )
xt- find previous xt in dict. Opposite of xt+
use xt- to write SIZE to show size of a word.
then use size in see see, etc to decompile whole word.
write DEPS ( xt -- ) create a list of xts which are
  dependencies of word represented by xt.
fix do/loop to act like a standard forth do/loop

   of words like var/variable, constant, value, :buffer etc

 eg
   : message create cr parse does> type count ; immediate

  is there any reason to loop on the return stack as opposed
  to the data stack. It would be more convenient for rloop.x
  to decrement the top data stack item!!!

* 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.
* are SOURCE and IN,/INPUT, the same or just similar? . Actually they are
* 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 "domain" field in the dictionary with a lookup list
    so that all names in the dictionary will be universally unique.
       eg a = core.math. b = net.tcp. etc
    then in each word a domain field a/b/c which gets attached to
    the name to make a full name. eg core.math.squareroot etc
  * "dictionary full" check, ie. is there any more memory in which
    to place new words? 'unused' word or 'free'
  * stack underflow check?
  * write sector/block to memory
  * implement block and buffer. Buffer writes to disk and frees
    a buffer and block reads from disk. block call buffer

 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.

    The data and return stack (ES:DI)
    Further source code can

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!

    We must write the dictionary or source code back into the
    usb or disk file system, which is not easy. Need to get
    disk geometry, and understand what emulated disk geometry is.

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

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)

* 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.

     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
     }

    HISTORY

    15 june 2018
       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 ." in a roundabout way
       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 s" using postpone. 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.

   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 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
   wrote r.v.nobles recursive euclid "gcd" function. Recursive functions
   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 a char limit would be good
   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+

   >CODE is a pointer but HERE is a value and >IN is 2 values
   This is a bit confusing and not consistent. Could make >CODE
   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 be called by CREATE and thus all defining
   words. It also writes an EXIT to 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
   drive: db 0      ; a variable to hold boot drive number
   db 'bootload...'
 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
  TWODUP equ OVER+1
  FLAGS equ TWODUP+1
  DEPTH equ FLAGS+1
  RDEPTH equ DEPTH+1
  RON equ RDEPTH+1
  ROFF equ RON+1
  STORE equ ROFF+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@+
  EQUALS equ CFETCHPLUS+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
  KEY equ EMIT+1
  EKEY equ KEY+1
  PLUS equ EKEY+1
  MINUS equ PLUS+1
  INCR equ MINUS+1
  DECR equ INCR+1
```

```
NEGATE equ DECR+1
LOGOR equ NEGATE+1       ; logical or
LOGXOR equ LOGOR+1
LOGAND equ LOGXOR+1
DIVMOD equ LOGAND+1
TIMESTWO equ DIVMOD+1
MULT equ TIMESTWO+1
UMULT equ MULT+1
FCALL equ UMULT+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
LOAD equ DIVTWO+1  ; load a sector from disk
SAVE equ LOAD+1
FG equ SAVE+1    ; foreground colour
BG equ FG+1      ; background colour
VID equ BG+1     ; video mode
PIX equ VID+1    ; one pixel on screen at xy
GLYPH equ PIX+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

; a table of code pointers. The pointers have the
; same offset in
; table as value of the opcode

; Machine signature goes here
op.table:
  dw 0, dup.x, drop.x, swap.x, over.x, twodup.x
  dw flags.x, depth.x, rdepth.x
  dw ron.x, roff.x
  dw store.x, storeplus.x, fetch.x, fetchplus.x
  dw cstore.x, cstoreplus.x
  dw cfetch.x, cfetchplus.x
  dw equals.x, notequals.x,
  dw lessthan.x, ulessthan.x, lit.x, litw.x
  dw emit.x, key.x, ekey.x
  dw plus.x, minus.x, incr.x, decr.x, neg.x
  dw logor.x, logxor.x, logand.x
  dw divmod.x, timestwo.x, mult.x, umult.x
  dw fcall.x, pcall.x, exit.x
  dw ljump.x, jump.x, jumpz.x, jumpnz.x, rloop.x
  dw divtwo.x
  dw readn.x, save.x, fg.x, bg.x, vid.x, pix.x
  dw glyph.x
  dw rtc.x, clock.x
  dw noop.x, -1

; 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 0
  db 'exec', 4
exec.x:
  ; save the return ip for 'exec' since the code
  ; below call [op.table+bx] changes the stack and
  ; registers. Before this technique the same byte
  ; code would get executed over and over again because
  ; exec was not returning properly
  pop word [returnexec]     ; save return ip
  pop si      ; get pointer to code
.nextopcode:
  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 (1-6 etc) into bx
  shl bx, 1       ; double bx because its a word pointer
  call [op.table+bx] ; use opcode as offset into
                     ; code pointer table

  ;*** check for stack underflow here ???
  jmp .nextopcode
.exit:
  push word [returnexec]     ; restore fn return ip
  ret

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. What should happen in '
  ; db ' the case of an overflow ? '
  ; db '  eg: LIT, 4, LIT, '0', PLUS, EMIT '
  ; db '   displays the digit "4" '
  ; dw $-plus.doc
plus:
  dw exec.x
  db '+', 1
plus.x:
  pop dx      ; juggle return pointer
  pop bx
  pop ax
  add ax, bx
  push ax
  push dx     ; restore return pointer
  ret

minus.doc:
  ; db 'subtract the top element of stack from next top'
  ; db ' ( n1 n2 -- n1-n2 ) '
  ; dw $-minus.doc
minus:
  dw plus.x
  db '-', 1
minus.x:
  pop dx      ; juggle return pointer
  pop bx
  pop ax
  sub ax, bx
  push ax
  push dx     ; restore return pointer
  ret

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:
  pop dx      ; juggle return pointer
  pop ax
  inc ax
  push ax
  push dx     ; restore return pointer
  ret

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:
  pop dx      ; juggle return pointer
  pop ax
  dec ax
  push ax
  push dx     ; restore return pointer
  ret

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:
  pop dx      ; juggle return pointer
  pop ax
  neg ax
  push ax
  push dx     ; restore return pointer
  ret

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 dx      ; juggle return pointer
  pop ax
  pop bx
  or ax, bx
  push ax
  push dx     ; restore return pointer
  ret

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 dx      ; juggle return pointer
  pop ax
  pop bx
  xor ax, bx
  push ax
  push dx     ; restore return pointer
  ret

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 dx      ; juggle return pointer
  pop ax
  pop bx
  and ax, bx
  push ax
  push dx     ; restore return pointer
  ret

divtwo.doc:
  ; db '(n1  - n1/2) '
  ; db ' divide n1 by 2 '
  ; dw $-divtwo.doc
divtwo:
  dw logand.x
  db '/2', 2
divtwo.x:
  pop dx      ; juggle return pointer
  pop ax      ; dividend is next element
  shr ax, 1   ; do ax := (ax+1)/2
  push ax
  push dx     ; restore return pointer
  ret

divmod.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
divmod:
  dw divtwo.x
  db '/mod', 4
divmod.x:
  pop cx      ; juggle return pointer
  xor dx, dx  ; set dx := 0
  pop bx      ; divisor is top element on stack
  pop ax      ; dividend is next element
  div bx      ; does dx:ax / bx remainder->dx; quotient->ax
  push dx     ; put remainder on stack
  push ax     ; put quotient on top of stack
  push cx     ; restore return pointer
  ret

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 divmod.x
  db '*2', 2
timestwo.x:
  pop dx      ; juggle return pointer
  pop ax      ; dividend is next element
  shl ax, 1   ;
  push ax     ;
  push dx     ; restore return pointer
  ret
```

```
mult.doc:
  ; db '(n1 n2 -- n1*n2 ) '
  ; db ' signed multiplication '
  ; dw $-mult.doc
mult:
  dw timestwo.x
  db '*', 1
mult.x:
  pop dx      ; juggle return pointer
  pop ax
  pop bx
  ; ax * bx ...
  push ax
  push dx     ; restore return pointer
  ret

umult.doc:
  ; db '(n1 n2 -- n1*n2 ) '
  ; db ' unsigned multiplication '
  ; dw $-umult.doc
umult:
  dw mult.x
  db 'u*', 2
umult.x:
  pop cx      ; juggle return pointer
  xor dx, dx  ; set dx := 0
  pop ax
  pop bx
  ; ax * bx ...
  mul bx        ; do dx:ax := ax*bx
  push ax
  push cx     ; restore return pointer
  ret

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 umult.x
  db 'fcall', 5
fcall.x:
  lodsw
  mov [es:di], si
  add di, 2
  mov si, ax      ; adjust the si code pointer
  ret

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". '
  ; dw $-pcall.doc
pcall:
  dw fcall.x
  db 'pcall', 5    ; or call it exec/ex/execute
pcall.x:
  pop dx          ; juggle return
  mov [es:di], si ; save ip to return stack
  add di, 2
  pop si          ; get proc exec address from stack
  push dx
  ret

; Need to think about this stack op call, is it really necessary?
; Not working! ocall doesnt return
ocall.doc:
  ; db ' ( opcode -- ) '
  ; db ' Remove opcode from stack and execute it on machine. '
  ; db ' This may allow immediate execution of opcodes....'
  ; dw $-pcall.doc
ocall:
  dw pcall.x
  db 'ocall', 5    ; or call it exec/ex/execute
ocall.x:
  ; experimental!
  ; add to optable, EQUs.
  pop word [ocall.return]     ; save return ip
  pop bx          ; get opcode (1...NOP etc) into bx
  shl bx, 1       ; double bx because its a word pointer
  call [op.table+bx] ; use opcode as offset into
  push word [ocall.return]     ; restore fn return ip
  ret
ocall.return dw 0

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

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:
  pop dx      ; juggle fn return address
  pop ax      ; get param to duplicate
  push ax
  push ax
  push dx     ; restore fn return address
  ret

drop.doc:
  ; db 'removes the top item on the stack.'
  ; dw $-drop.doc
drop:
  dw dup.x       ; link to previous word
  db 'drop', 4     ; strings are 'counted'
drop.x:
  pop dx      ; juggle fn return address
  pop ax      ; remove top element of stack
  push dx     ; restore fn return address
  ret

swap.doc:
  ; db 'swaps the top 2 items on the stack.'
  ; dw $-swap.doc
swap:
  dw drop.x         ; link to previous word
  db 'swap', 4
swap.x:
  pop dx      ; juggle fn return address
  pop ax      ; get top stack item
  pop bx      ; get next stack item
```

```
    push ax     ; put them back on in reverse order
    push bx
    push dx     ; restore fn return address
    ret

  over.doc:
    ; db ' ( n1 n2 -- n1 n2 n1 ) '
    ; db ' Puts a copy of 2nd stack item on top of stack. '
    ; db ' dont use this, will probably remove. '
    ; dw $-over.doc
  over:
    dw swap.x         ; link to previous word
    db 'over', 4
  over.x:
    pop dx      ; juggle fn return address
    pop ax      ; get top stack item
    pop bx      ; get next stack item
    push bx     ;
    push ax     ;
    push bx     ; add copy of 2nd item on top of stack
    push dx     ; restore fn return address
    ret

  twodup.doc:
    ; db ' ( n1 n2 -- n1 n2 n1 n2 ) '
    ; db ' copies 2 stack items onto stack '
    ; dw $-twodup.doc
  twodup:
    dw over.x
    db '2dup', 4
  twodup.x:
    pop dx      ; juggle fn return address
    pop ax      ; get top stack item
    pop bx      ; get next stack item
    push bx     ;
    push ax     ;
    push bx     ;
    push ax     ; n1 n2 n1 n2
    push dx     ; restore fn return address
    ret

  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:
    pop dx      ; juggle fn return address
    ; to do
    push dx     ; restore fn return address
    ret

  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:
    pop dx        ; juggle fn return address
    mov bx, sp
    mov ax, 4096  ; 4K stack (but could change!)
    sub ax, bx    ;
    shr ax, 1     ; div by 2 (2 byte stack cell)
    ;causing problems ???
    ;dec ax        ; the exec.x call doesnt count
    push ax
    push dx     ; restore fn return address
    ret

  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:
    pop dx        ; juggle fn return address
    mov ax, di
    shl ax, 1
    push ax
    push dx       ; restore fn return address
    ret

  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:
    pop dx         ; juggle fn return address
    pop ax         ; value to store at address
    stosw          ; [es:di] := ax, di+2
    push dx        ; restore fn return address
    ret

  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:
    pop dx          ; juggle fn return address
    sub di, 2       ;
    mov ax, [es:di] ; get top item off return stack
    push ax
    push dx         ; restore fn return address
    ret

  store.doc:
    ; db '( w adr -- ) '
    ; db ' place 2 byte value w at address "adr" '
    ; dw $-store.doc
  store:
    dw roff.x
    db '!', 1
  store.x:
    pop dx         ; juggle fn return address
    pop bx         ; pointer to address
    pop ax         ; value to store at address
    mov [bx], ax   ; 2 byte is stored
    push dx        ; restore fn return address
    ret

  storeplus.doc:
```

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

  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:
    pop dx      ; juggle fn return address
    pop bx
    mov ax, word [bx]
    push ax     ; save value on top of stack
    push dx     ; restore fn return address
    ret

  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:
    pop dx      ; juggle fn return address
    pop bx
    mov ax, word [bx]
    add bx, 2   ; increment address by 1 word (2 bytes)
    push bx     ; save address on stack
    push ax     ; save value on top of stack
    push dx     ; restore fn return address
    ret

  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:
    pop dx         ; juggle fn return address
    pop bx         ; pointer to address
    pop ax         ; value to store at address
    mov [bx], al   ; only the low value byte is stored
    push dx        ; restore fn return address
    ret

  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:
    pop dx         ; juggle fn return address
    pop bx         ; pointer to address
    pop ax         ; value to store at address
    mov [bx], al   ; only the low value byte is stored
    inc bx         ; advance address and put on stack
    push bx
    push dx        ; restore fn return address
    ret

  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:
    pop dx      ; juggle fn return address
    pop bx
    xor ax, ax  ; set ax := 0
    mov al, byte [bx]
    push ax
    push dx     ; restore fn return address
    ret

  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:
    pop dx      ; juggle fn return address
    pop bx
    xor ax, ax  ; set ax := 0
    mov al, byte [bx]
    inc bx      ; increment address by 1
    push bx     ; save address on stack
    push ax     ; save value on top of stack
    push dx     ; restore fn return address
    ret

  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 cfetchplus.x     ; link to previous word
    db '=', 1
  equals.x:
    pop dx         ; juggle fn return address
    pop ax         ; top stack item
    pop bx         ; 2nd stack item
    cmp ax, bx
    je .true
  .false:
```

```
    push 0
    jmp .exit
.true:
    push -1
.exit:
    push dx        ; restore fn return address
    ret

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 dx         ; juggle fn return address
    pop ax         ; top stack item
    pop bx         ; 2nd stack item
    cmp ax, bx
    jne .true
.false:
    push 0
    jmp .exit
.true:
    push -1
.exit:
    push dx        ; restore fn return address
    ret

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 dx         ; juggle fn return address
    pop ax         ; top stack item
    pop bx         ; 2nd stack item
    cmp ax, bx
    jg .true       ; jg is a signed comparison
.false:
    push 0
    jmp .exit
.true:
    push -1
.exit:
    push dx        ; restore fn return address
    ret

ulessthan.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 an unsigned comparison'
    ; dw $-ulessthan.doc
ulessthan:
    dw lessthan.x
    db 'u<', 2
ulessthan.x:
    pop dx         ; juggle fn return address
    pop ax         ; top stack item
    pop bx         ; 2nd stack item
    cmp ax, bx
    ja .true       ; jg is an unsigned comparison
.false:
    push 0
    jmp .exit
.true:
    push -1
.exit:
    push dx        ; restore fn return address
    ret

; maybe call this "char"
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:
    pop dx         ; juggle fn return address
    xor ax, ax     ; set ax := 0
    lodsb          ; al := [si]++ get literal char into AL
    cbw            ; convert signed byte to signed word ax
    push ax        ; put literal value on stack
    push dx        ; restore fn return address
    ret

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:
    pop dx         ; juggle fn return address
    lodsw          ; ax := [si]++ get literal char into AX
    push ax        ; put literal value on stack
    push dx        ; restore fn return address
    ret

emit.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
emit:
    dw litw.x
    db 'emit', 4
emit.x:
    pop bx         ; juggle return pointer
    pop ax         ; char in al
    push bx
    ; 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        ; x86 bios
    ret

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 emit.x    ; link to prev
    db 'key', 3  ; reverse counted string
key.x:
```

```
    mov ah, 0    ; wait for keypress bios function
    int 16h      ; ah := asci code and al := scan code
    pop bx       ; juggle function return pointer
    mov ah, 0    ; set ah = 0
    push ax      ; save asci code onto stack, high byte zero
    push bx      ; restore return pointer to stack
    ret

  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:
    mov ah, 0    ; wait for keypress bios function
    int 16h      ; ah := asci code and al := scan code
    pop bx       ; juggle function return pointer
    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
    push 0       ; false flag
    push bx      ; restore return pointer to stack
    ret

  .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
    push 1       ; true flag
    push bx      ; restore return pointer to stack
    ret

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

  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
    ret

  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:
    pop dx          ; juggle return pointer
    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
    pop bx          ; get top stack item into bx
    cmp bx, 0       ; if dx != 0 continue
    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:
    push dx         ; restore call return
    ret

  ; 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)
    pop dx          ; juggle return pointer
    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)
    pop bx          ; get top stack item into bx
    cmp bx, 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:
    push dx         ; restore call return
    ret

  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)
    pop dx          ; juggle return pointer
    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:
    push dx         ; restore call return
    ret


  ; this data is used by 'read' below in order to
  ; translate logical track number into chs. But this is maybe
  ; only for floppies.
  SectorsPerTrack: dw 18  ; standard floppy config
  Sides: dw 2             ; floppies only have one 'platter'

  ; we might need to find out emulated floppy or hard disk
  ; geometry- see Mike Gonta usb booting info on the net
  ; should return a flag indicating success or no.
  ; Also, on error this loops infinitely! no so good.
  ; Also, sectors start at one not zero !!
  ; So sector 1 is the boot sector on disk.
  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 'readit', 6
  read.x:
                    ; 1st-sect n address
   .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

     ; current segment is 64K=0x1000*16
     mov ax, 1000h   ; 64K segment, or mov ax, ds
     mov es, ax      ; es:bx determines where data loaded to

     ; load to memory address on top of stack
     ;*** or ax=1100h bx=0 is 4K after 64K segment
     ;mov bx, 4096    ; ES:BX = 1000:4096

     pop dx          ; juggle return pointer
     pop bx          ; destination memory address in this segment
     pop cx          ; number of sectors (-> AL)
     pop ax          ; start sector
     push dx         ; restore fn return

     ; -- mike gonta code
  logical:
     ; Calculate head, track and sector settings for int 13h
     ; IN: logical sector in AX, OUT: correct registers for int 13h

     push bx
     push ax

     ;mov ax, cx     ; popped start sector into cx
     mov bx, ax                ; Save logical sector
     ;ax has the number of sectors to read
     mov dx, 0
     div word [SectorsPerTrack] ; 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]
     mov dx, 0
     div word [Sides]
     mov dh, dl                ; Head/side
     mov ch, al                ; Track
     pop ax
     pop bx
     mov dl, byte [drive.d]    ; Set correct device
     ; ----end mike gonta code

     mov ah, 2       ; Load disk data to ES:BX
     ;mov al, 2       ; Load 2 sectors 512 bytes * 2 == 1K
     int 13h         ; Read!
     jc .readerror   ; ERROR => Try again

     ;*** restore es since rstack points with it!!
     mov es, [saven.es]
     pop dx          ; juggle fn return
     push 1          ; return true flag for success
     push dx         ; restore fn return
     ret

   .readerror:
     ;*** restore es since rstack points with it!!
     mov es, [saven.es]
     pop dx          ; juggle fn return
     push 0          ; return false flag for read error
     push dx         ; restore fn return
     ret


  saven.es: dw 0

  ; this is the old version that can only read one track (18 sectors)
  readn.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
  readn:
    dw read.x
    db 'read', 4
  readn.x:
                    ; sect n
   .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       ; ERROR => reset again
  .read:

    ;*** need to save es since rstack points with it!!
    mov [save.es], es

    ; current segment is 64K=0x1000*16
    mov ax, 1000h   ; 64K segment, or mov ax, ds
    mov es, ax      ; es:bx determines where data loaded to

    ; load to memory address on top of stack
    ;*** or ax=1100h bx=0 is 4K after 64K segment
    ;mov bx, 4096    ; ES:BX = 1000:4096
    pop dx          ; juggle return pointer
    pop bx          ; destination memory address in this segment
    pop ax          ; number of sectors (-> AL)
    pop cx          ; start sector
    push dx         ; restore fn return

    mov ah, 2       ; Load disk data to ES:BX
    ;mov al, 2       ; Load 2 sectors 512 bytes * 2 == 1K

    ; but what about disk geometry ?? How many sectors in a
    ; cylinder etc ?
    ; try mov cx, 0x0002 ; cylinder 0, sector 2
    mov ch, 0       ; Cylinder=0
    ;mov cl, 10      ; Sector=10 (sector 1 = boot sector)

    mov dh, 0       ;  Head=0
    mov dl, byte [drive.d] ;
    int 13h         ; Read!
    jc .readerror   ; ERROR => Try again

    ;*** restore es since rstack points with it!!
    mov es, [save.es]
    pop dx          ; juggle fn return
    push 1          ; return true flag for success
    push dx         ; restore fn return
    ret

  .readerror:
    ;*** restore es since rstack points with it!!
    mov es, [save.es]
    pop dx          ; juggle fn return
    push 0          ; return false flag for read error
    push dx         ; restore fn return
    ret

;*** load alters es (used for rstack)
;*** need to save
save.es: dw 0

save.doc:
  ; db ' ( -- ) ... '
  ; db ' save sectors to disk '
  ; dw $-save.doc
save:
  dw readn.x
  db 'save', 4
save.x:
  ; to do
  ret

; 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 save.x
  db 'fg', 2
fg.x:
  pop dx      ; juggle fn return address
  pop bx      ; get foreground colour
  push dx     ; restore fn return address
  mov [fg.d], bl  ;
  ret
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:
  pop dx      ; juggle fn return address
  pop bx      ; get foreground colour
  push dx     ; restore fn return address
  mov [bg.d], bl  ;
  ret
bg.d: db 5    ;

vid.doc:
  ; db ' ( n -- ) '
  ; db ' set video mode to n'
  ; db ' on x86 try 13h mode'
  ; dw $-vid.doc
vid:
  dw bg.x
  db 'vid', 3
vid.x:
  pop dx      ; juggle fn return address
  pop ax      ; get video mode
  push dx     ; restore fn return address
  mov ah, 0   ; ah=0 set video mode function, al=mode
  int 10h
  ret

pix.doc:
  ; db ' ( x y -- ) '
  ; db ' display one pixel at position xy'
  ; dw $-pix.doc
pix:
  dw vid.x
  db 'pix', 3
pix.x:
  ; using interrupts to draw pixels is very slow but ok
  ; for playing around
  pop bx      ; juggle fn return address
  pop cx      ; get x-coordinate
  pop dx      ; get y-coordinate
  push bx     ; restore fn return address
  ;mov     cx, 10      ; x-coordinate
  ;mov     dx, 10      ; y-coordinate
  mov al, 15      ; white
  mov ah, 0ch     ; put pixel
  int 10h         ; draw pixel
  ret

glyph.doc:
  ; db ' ( x y -- ) '
```

```
    ; db ' display a colour glyph at pixel position xy'
    ; dw $-glyph.doc
  glyph:
    dw pix.x
    db 'glyph', 5
  glyph.x:
    ; using interrupts to draw pixels is very slow but ok
    ; for playing around
    pop bx      ; juggle fn return address
    pop cx      ; get x-coordinate
    pop dx      ; get y-coordinate
    push bx     ; restore fn return address
    mov al, 15      ; white
    mov ah, 0ch     ; put pixel
    int 10h         ; draw pixel
    ret

  clock.doc:
    ; db ' ( -- D ) '
    ; db ' number of clock ticks since midnight '
    ; dw $-clock.doc
  clock:
    dw glyph.x
    db 'clock', 5
  clock.x:
    ; using interrupts to draw pixels is very slow but ok
    ; for playing around
    pop bx      ; juggle fn return address
    pop cx      ; get x-coordinate
    pop dx      ; get y-coordinate
    push bx     ; restore fn return address
    mov al, 15      ; white
    mov ah, 0ch     ; put pixel
    int 10h         ; draw pixel
    ret

  ; There are "issues" here. It is not alway 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. '
    ; dw $-rtc.doc
  rtc:
    dw glyph.x
    db 'rtc', 3
  rtc.x:
    pop dx        ; juggle fn return pointer
    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
    ; push ax
    ; jne .updating    ; makes an infinite loop in qemu

    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
    push ax

    push dx          ; restore fn return pointer
    ret

  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:
    ret


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

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

  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.
  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 implementation, the start of the parameter field '
    ; db ' is just the next available byte in the dictionary. '
    ; db ' ** A standard definition of VARIABLE (VAR) is: '
    ; db ' : CREATE 0 , ; '
  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
    db FCALL
    dw last.p       ; /get pointer to last word in dictionary
    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
    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 >code 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    ;
    ; compile an exit for the new word being defined
    db LIT, EXIT      ; op=exit
    db FCALL
    dw ccomma.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:
  ; 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

; 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 >code 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 does.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
   ; HERE and >CODE @ 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 tocode.p       ; >code
   db DUP            ; >code >code
   db FETCH, LIT, 4  ; >code code* 4
   db MINUS          ; >code code*-4
   db SWAP           ; code*-4 >code
   db STORE          ;
   db FCALL
   dw heretocode.p   ; set here to point to >code
   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 >code 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

 begin.doc:
```

```
    ; db ' ( -- )
    ; db ' marks a jump back address for until/again etc'
    ; db ' the jump address is left on the data stack '
    ; db ' forth-  : begin here ; immediate '
  begin:
    dw semicolon.p
    db 'begin', IMMEDIATE | 5
  begin.p:
    db FCALL
    dw here.p         ; ad /current compilation adr
    db EXIT

  again.doc:
    ; db ' ( -- R: back-adr )
    ; db ' jumps back to begin'
  again:
    dw begin.p
    db 'again', IMMEDIATE | 5
  again.p:
    db LIT, JUMP
    db LIT, 2         ; op 2
    db FCALL
    dw itemcompile.p      ; compile to current position (here)
    db FCALL
    dw here.p         ; jb here
    db DECR           ; jb here-1 /align to jump
    db MINUS          ; jb-here-1
    db FCALL
    dw ccomma.p        ; compile to current position (here)
    db EXIT

  until.doc:
    ; db ' ( n -- )
    ; db ' at run-time: jumps back to begin if n is true'
    ; db ' at compile-time: get begin address from data '
    ; db ' stack and compiles a conditional relative jump '
    ; db ' back to begin. '
  until:
    dw again.p
    db 'until', IMMEDIATE | 5
  until.p:
    db LIT, JUMPF
    db LIT, 2         ; jb op 2
    db FCALL
    dw itemcompile.p      ; compile to current position (here)
    db FCALL
    dw here.p         ; jb here
    db DECR           ; jb here-1 /align to jump
    db MINUS          ; jb-here-1
    db FCALL
    dw ccomma.p        ; compile to current position (here)
    db EXIT

  ; need to change this to make it more like standard forth
  ; eg "do" takes 2 args on the data stack and pushes to
  ;   return stack. loop increments counter and compares to
  ;   limit value and loops if less, or else exits and
  ;   removes the 2 args from the return stack and drops.
  loop.doc:
    ; db ' ( R: n -- R: n-1 )
    ; db ' at run-time: jumps back to begin if return stack > 0'
    ; db ' decrements top item of return stack '
    ; db ' at compile-time: get begin address from data '
    ; db ' stack and compiles a rloop jump '
    ; db ' back to begin (address on data stack). '
  loop:
    dw until.p
    db 'loop', IMMEDIATE | 4
  loop.p:
    db LIT, RLOOP      ; jb opcode
    db FCALL
    dw ccomma.p         ; compile opcode to current position (here)
    ;db LIT, 2         ; jb op 2
    ;db FCALL
    ;dw itemcompile.p      ; compile to current position (here)
    db FCALL
    dw here.p         ; jb here
    db DECR           ; jb here-1 /align to jump
    db MINUS          ; jb-here-1
    db FCALL
    dw ccomma.p        ; compile to current position (here)
    db EXIT

  if.doc:
    ; db ' run-time: ( n -- ) '
    ; db ' compile-time: ( -- adr ) '
    ; db ' if the value n on the stack is zero '
    ; db ' skip statements after this until next "fi" '
    ; db ' compiles a jumpzero and puts the current '
    ; db ' address on the data stack. The "fi" command consumes '
    ; db ' that address '
  if:
    dw loop.p
    db 'if', IMMEDIATE | 2
  if.p:

    db LIT, JUMPF     ; op
    ;db LIT, 2         ; op 2
    db FCALL
    dw ccomma.p        ; compile to current position (here)

    db FCALL
    dw here.p         ; ad /current compilation adr

    ; we compile a 2 byte jump which will do nothing.
    ; eg JUMPF, 2  which just goes to the next instruction.
    ; but this will be replaced by the real target when "fi"
    ; compiles
    db LIT, 2         ; compile temporary jump target (2)
    db FCALL
    dw ccomma.p        ;
    db EXIT

  else.doc:
    ; db ' compile.time: ( ad -- jad )
    ; db ' at compile time, get the "if" jump address  '
    ; db ' from the data stack and completes the if jump '
    ; db ' using offset from here. Then it leaves the '
    ; db ' else jump address on stack for "fi" to deal with '
  else:
    dw if.p
    db 'else', IMMEDIATE | 4
  else.p:
                  ; ja
    db LIT, JUMP    ; ja op
    db FCALL
    dw ccomma.p      ; compile opcode to current position (here)

    db FCALL
    dw here.p         ; ja ad /current compilation adr

    db LIT, 2         ; compile temporary jump target 2
    db FCALL
    dw ccomma.p        ;
    db SWAP           ; ad ja
    db DUP            ; ad ja ja
    db FCALL
    dw here.p         ; ad ja ja target
```

```
    db SWAP           ; ad ja target ja
    db MINUS          ; ad ja t-ja
    db INCR           ; .... adjust jump target
    db SWAP           ; ad n ja
    db CSTORE         ; ad
    db EXIT


  ; better to use the data stack for compile time
  ; behaviors
  fi.doc:
    ; db ' run-time: ( -- )
    ; db ' compile-time: ( ad -- ) '
    ; db ' at compile time obtains the correct jump '
    ; db ' address from the data stack and compiles the correct '
    ; db ' target address into a previously compile "if" clause. '
    ; db ' This word is called "then" in traditional forths. '
  fi:
    dw else.p
    db 'fi', IMMEDIATE | 2
  fi.p:
    ;*** we use the data stack for compile time calculations
    ; "ja" means jump address

    db DUP            ; ja ja
    db FCALL
    dw here.p         ; ja ja target
    db SWAP           ; ja target ja
    db MINUS          ; ja t-ja
    db INCR           ; .... adjust jump target
    db SWAP           ; a1 a2 n ja
    db CSTORE         ; a1 a2
    db EXIT

  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 fi.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
    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

  dotrstack.doc:
    ; db ' ( -- )
    ; db ' display the top 2 items on the return stack '
  dotrstack:
    dw dotstack.p
    db '.r2', 3
  dotrstack.p:
    db LIT, ' ', EMIT
    db ROFF, ROFF ; n m
    db DUP        ; n m m
    db FCALL
    dw udot.p     ; n m
    db LIT, ' ', EMIT
    db RON        ; n
    db DUP        ; n n
    db FCALL
    dw udot.p     ; n
    db RON
    db EXIT

  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 dotrstack.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

  dotxt.doc:
    ; db ' ( adr - ) '
    ; db ' given a valid execution token on the stack '
    ; db ' the name of the procedure'
    ; db ' The definition might be
    ; db ' : .xt -1 rcount type ; '
    ; dw $-dotxt.doc
  dotxt:
    dw rcount.p
    db '.xt', 3
  dotxt.p:
                      ; xt
    db DECR           ; adr-1
    db FCALL
    dw rcount.p
    db FCALL
    dw type.p
    db EXIT

  nextxt.doc:
    ; db ' ( xt -- XT ) '
```

```
    ; db ' get next execution address in the dictionary '
  nextxt:
    dw dotxt.p
    db 'xt+', 3
  nextxt.p:

                      ; xt
    db DECR           ; xt-1  /points to count|control byte
    db FCALL
    dw rcount.p       ; adr n  /start of name
    db DROP           ; adr
    db DECR, DECR     ; adr-2
    db FETCH          ; XT   / get word pointer
    db EXIT

  opcode.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 (-1?) 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 '
  opcode:
    dw nextxt.p
    db 'opcode', 6
  opcode.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
    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

  dotcode.doc:
    ; db ' ( opcode - ) '
    ; db ' given a valid opcode on the stack, print '
    ; db ' the textual version of the opcode. '
    ; dw $-dotcode.doc
  dotcode:
    dw opcode.p
    db '.code', 5
  dotcode.p:
                      ; op
    ;*** check for invalid code, nop always last
    db DUP            ; op op
    db LIT, NOOP, INCR  ; op op nop+1
    db ULESSTHAN        ; op flag
    db JUMPT, 12        ; op

    ;*** invalid code
    db FCALL
    dw udot.p
    db LIT, '?', EMIT
    db LIT, '?', EMIT
    db EXIT

    ;*** valid opcode
    db DUP, PLUS      ; op*2
    db LITW
    dw op.table       ; op*2 op.table
    db PLUS           ; op*2+op.table
    db FETCH          ; [op*2+op.table] / get execution adr
    db DECR           ; a-1

    db FCALL
    dw rcount.p       ; adr n

    ;*** does the same as rcount
    ;db DUP, CFETCH   ; adr n  / get the count
    ;db DUP           ; adr n n
    ;db RON, MINUS    ; adr-n
    ;db ROFF          ; adr-n n

    db FCALL
    dw type.p
    db EXIT

  udot.doc:
    ; db ' ( n -- ) '
    ; db ' display top stack element as unsigned '
    ; db ' number in the current base. '
    ; dw $-udot.doc
  udot:
    dw dotcode.p
    db 'u.', 2
  udot.p:
    db LIT, 0, RON      ; n            r: 0
    db LITW
    dw base.d           ; n adr        r: 0
    db CFETCH           ; n base
    db DIVMOD           ; rem quotient
    db ROFF, INCR, RON  ; rem quotient r: 0+1
    db DUP, JUMPNZ, -9
                        ; rem rem rem ... 0
    db DROP             ; rem rem ...   r: x

    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, -6        ; rem ...        r: x-1
    db ROFF, DROP       ; clear rstack
    db EXIT

  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

dothex.doc:
  ; db ' ( n -- ) '
  ; db ' display top stack element as an unsigned '
  ; db ' number in hexadecimal '
  ; dw $-dothex.doc
dothex:
  dw ddot.p
  db '.hex', 4
dothex.p:
                    ; n
  db LITW
  dw base.d         ; n adr
  db CFETCH         ; n base
  db SWAP           ; base n     /save original base
  db LIT, 16        ; base n 16
  db LITW
  dw base.d         ; base n 16 adr
  db CSTORE         ; base n     /set new base to 16
  db FCALL
  dw udot.p         ; base       /print number
  db LITW
  dw base.d         ; base adr
  db CSTORE         ;            /restore original base
  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 dothex.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

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.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 todigit.p
  db 'base', 4
base.p:
  db LITW
  dw base.d
  db EXIT
base.d: db 10

bin.doc:
  ; db ' ( -- ) '
  ; db ' sets the numeric base to binary '
  ; dw $-bin.doc
bin:
  dw base.p
  db 'bin', 3
bin.p:
  db LIT, 2
  db LITW
  dw base.d
  db CSTORE
  db EXIT

hex.doc:
  ; db ' ( -- ) '
  ; db ' sets the numeric base to hexadecimal '
  ; dw $-hex.doc
hex:
  dw bin.p
  db 'hex', 3
hex.p:
  db LIT, 16
  db LITW
  dw base.d
  db CSTORE
  db EXIT

deci.doc:
  ; db ' ( -- ) '
  ; db ' sets the numeric base to 10 '
  ; dw $-deci.doc
deci:
  dw hex.p
  db 'deci', 4
deci.p:
  db LIT, 10
  db LITW
  dw base.d
  db CSTORE
  db EXIT

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 deci.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

test.tonumber.doc:
  ; db ' Testing >number by accepting input and '
  ; db ' displaying the number '
test.tonumber:
```

```
    dw tonumber.p
    db 'test.>number', 12
  test.tonumber.p:
    db LIT, 10, EMIT
    db LIT, 13, EMIT
    db LITW
    dw term.d
    db FCALL
    dw accept.p
    db LITW
    dw term.d
    db COUNT          ; a n
    db DUP            ; a n n
    db JUMPNZ, 3      ; a n    /exit if no input
    db EXIT
    db FCALL
    dw tonumber.p
    db LIT, ' ', EMIT
    db FCALL
    dw dot.p
    db LIT, ' ', EMIT
    db DUP
    db FCALL
    dw udot.p
    db LIT, ' ', EMIT
    db FCALL
    dw dot.p
    db LIT, 10, EMIT
    db LIT, 13, EMIT
    db JUMP, -42
    db EXIT

  toword.doc:
    ; db ' ( -- adr n ) '
    ; db ' put on the stack a pointer to the current '
    ; db ' word and its length. '
  toword:
    dw test.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 toin.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
  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 $-parse.doc
  parse:
    dw wparse.p
    db 'parse', 5
  parse.p:
    db RON          ;  r: char
    db FCALL
    dw toin.p       ; adr n      r: char
    db DUP          ; a n n      r: char
    ;* no more text so nothing to parse
    db JUMPZ, .notext-$  ; 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
    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 RON, DUP, ROFF  ; a a a+p-1
    db SWAP, 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

  ; this will not work when parse is fixed, rewrite as source
  seecomp.doc:
    ; db ' ( -- ) '
    ; db ' compiles one line of text to anon and '
    ; db ' displays the decompilation. This is for testing'
    ; db ' the compilation process. Enter exits the loop'
    ; dw $-seecomp.doc
  seecomp:
    dw parse.p
    db 'see,', 4
  seecomp.p:

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

    db LITW
    dw pad.d            ; a : the address of pad buffer
    db DUP              ; a a
    db FCALL
    dw accept.p         ; a
    ;*** exit if no input, i.e 'pad' count == 0
    db COUNT            ; a+1 n /count of pad
    db DUP              ; a+1 n n
    db JUMPNZ, .notempty-$  ; a+1 n
    db DROP, DROP       ;
    db EXIT

  .notempty:
    db FCALL            ; a+1 n
    dw resetin.p        ; reset in >in >word etc

  .nextword:
    db FCALL
    dw toin.p           ; a n
    db DUP              ; a n n
    ;*** if no more characters, exit loop
    db JUMPZ, .end-$    ; a n
    db FCALL
    dw parse.p          ; a+x n
    ;*** if parse returns 0 then no more words (all space)
    db DUP              ; a+x n n
    db JUMPZ, .end-$    ; a+x n

    db TWODUP           ; a+x n a+x n
```

```
    ;*** update >word etc
    db FCALL
    dw inplus.p           ; a+x n
    db FCALL
    dw tick.p             ; n/op/xt flag
    ;*** if flag==0 abort, unknown word/number
    db DUP                ; n/op/xt flag flag
    db JUMPZ, .error-$    ; n/op/xt flag

    db FCALL              ; n/op/xt
    dw itemcompile.p        ; ... (if/fi may leave data here)

    db JUMP, .nextword-$
  .end:
    ;*** no more words (>in returned zero)
    db DROP, DROP         ; clear stack
    db LIT, EXIT
    db LIT, 2             ; compile opcode exit
    db FCALL
    dw itemcompile.p

    ;*** show the decompilation

    db LIT, 'S', EMIT
    db LIT, ':', EMIT
    db FCALL
    dw dotstack.p

    db LIT, ' ', EMIT
    db LIT, 'R', EMIT
    db LIT, ':', EMIT
    db FCALL
    dw dotrstack.p

    db LITW
    dw anon.d
    db LIT, 25          ; ad n
    db FCALL
    dw decomp.p         ; ad2
    db DROP

    db LIT, 13, EMIT
    db LIT, 10, EMIT

    ;*** loop forever
    db LJUMP
    dw .again-$
  .error:
    ;db FCALL
    ;dw dotstack.p
    db LIT, 13, EMIT
    db LIT, 10, EMIT
    db DROP, DROP         ; a+x n
    db FCALL
    dw toword.p           ; a+x n
    db FCALL
    dw type.p
    db LIT, ' ', EMIT
    db LIT, '?', EMIT
    db LIT, '?', EMIT
    db LIT, 13, EMIT
    db LIT, 10, EMIT
    db LJUMP
    dw .again-$
    db EXIT

  ; 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 $-inputcompile.doc
  inputcompile:
    dw seecomp.p
    db 'in,', 3
  inputcompile.p:
  .nextword:
    db FCALL
    dw wparse.p           ; a+x n  /address+length of next word
    ;*** if parse returns 0 then no more words (all space)
    db DUP                ; a+x n n
    db JUMPZ, .end-$      ; 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-$
  .end:
    ;*** 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
                          ; n/op/xt flag
    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, 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

  ; 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 toin.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 LIT, 2           ; compile opcode exit
    db FCALL
    dw itemcompile.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
    db LITW
    dw anon.d
    db PCALL

    db LIT, 13, EMIT
    db LIT, 10, EMIT
    db LIT, 2, FG     ; set to green
    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

  ; I would like to call this "shell". Also this word can be
  ; compiled from source.
  shell.doc:
    ; db ' ( -- ) '
    ; db ' parse compile and execute words'
    ; dw $-interp.doc
  shell:
    dw source.p
    db 'shell', 5
  shell.p:
    ;*** compile to anon buffer or dictionary
  .again:
    ; source
    ; op, reads ahead gets the name and opcode and compiles it
    ; at current code point
    ; : shel
    ;     begin
    ;       here>anon op, exit here>anon
    ;       pad dup accept count in0
    ;       in,
    ;       anon pcall cr ." ok" cr
    ;     again
    ;
    ;
    ; call this 'op,' etc
    ; [ wparse exit ' drop ]
    ; default compile point is anon buffer
    db LITW
    dw anon.d
    db FCALL
    dw ishere.p

    ;*** rub out anon, so it doesnt execute twice
    ; eg when colon calls create which calls here>code which
    ; calls fcall anon.

    db LIT, EXIT
    db FCALL
    dw ccomma.p        ; compile opcode exit

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

    db LITW
    dw pad.d             ; a    (address of pad buffer)
    db DUP               ; a a
    db FCALL             ; get some text into input buffer
    dw accept.p          ; a
    db COUNT             ; a+1 n
    ;*** set "in" var to beginning of "pad" buffer
    db FCALL
    dw resetin.p         ; reset >in >word atin etc

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

    ;db LIT, 0           ; zero terminate for convenience
    ;db FCALL
    ;dw ccomp.p
  .run:
    ;*** now run the compiled stuff
    ;db LITW
    ;dw anon.d
    ;db PCALL
    db FCALL
    dw anon.d

    db LIT, 13, EMIT
    db LIT, 10, EMIT
    db LIT, 2, FG     ; set to green
    db LIT, 'o', EMIT
    db LIT, 'k', EMIT
    db LIT, 7, FG     ; set to white
    db LIT, 13, EMIT
    db LIT, 10, EMIT

    ;*** loop forever
    db LJUMP
    dw .again-$
    db EXIT       ; never reaches here
```

```
find.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 $-find.doc
find:
  dw shell.p
  db 'find', 4
find.p:
                ; a n
  db FCALL      ; pointer to last word in dictionary
  dw last.p     ; a+1 n A
  db FETCH      ; a+1 n last
.again:
  db DECR       ; point to count|control byte of name
  db FCALL
  dw rcount.p   ; a n A N

  ;***
  ;db DUP    ; a+1 n A-1 A-1
  ;db CFETCH      ; .. A-1 N  / get the count
  ;db DUP         ; .. A-1 N N
  ;db RON, MINUS  ; .. adr-N       r: N
  ;db ROFF        ; .. adr-N N
                ; a n A N

  ;db FCALL
  ;dw print.p

  ;*** 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, RON, RON;           r: n A a
  db ROFF, DUP  ; a a         r: n A
  db ROFF, DUP  ; a a A A     r: n
  db ROFF, DUP  ; a a A A n n
  db RON, SWAP, RON  ; a a A n  r: n A

  ;*** compare two strings
  db FCALL      ; are strings equal?
  dw compare.p  ; a flag      r: n A
  ;
  db JUMPF, .notequal-$     ; a           r: n A
  ; if strings same clear stacks
  db DROP       ;             r: n A
  db ROFF, ROFF ; 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 ROFF, ROFF ; 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 DROP, DROP, DROP  ; clear stack
  db LIT, 0     ; zero means not found
  db EXIT

test.find.doc:
  ; db ' Testing find by accepting input and '
  ; db ' displaying the found execution token '
test.find:
  dw find.p
  db 'test.find', 9
test.find.p:
  db FCALL
  dw listxt.p
  db LIT, 10, EMIT
  db LIT, 13, EMIT
  db LITW
  dw term.d
  db FCALL
  dw accept.p
  db LITW
  dw term.d     ; adr
  db COUNT      ; adr+1 n
  db FCALL
  dw find.p
  db LIT, ' ', EMIT
  db FCALL
  dw udot.p
  db LIT, 10, EMIT
  db LIT, 13, EMIT
  db JUMP, -25
  db EXIT

immediate.doc:
  ; db ' ( -- ) '
  ; db ' make the last word immediate. '
  ; db ' Set the immediate control bit'
immediate:
  dw test.find.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 ' ( p n -- n flag) '
    ; db ' given a pointer "p" 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 $-tick.doc
  tick:
    dw isimmediate.p
    db 'tick', 4
  tick.p:
                    ; a n
    db TWODUP       ; a n a n
    db FCALL
    dw find.p         ; a n xt/0
    db DUP            ; a n xt xt
    db JUMPNZ, .opcode-$ ; 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 opcode.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

  test.tick.doc:
    ; db ' Testing tick by accepting input and '
    ; db ' displaying the found execution token '
  test.tick:
    dw tick.p
    db 'test.tick', 9
  test.tick.p:
    db LIT, 10, EMIT
    db LIT, 13, EMIT
    db LITW
    dw term.d
    db FCALL
    dw accept.p
    db LITW
    dw term.d     ; adr
    db COUNT      ; adr+1 n
    ;*** exit if no input
    db DUP
    db JUMPNZ, 3
    db EXIT

    db FCALL
    dw tick.p     ; 0/n/op/xt flag
    db LIT, 10, EMIT
    db LIT, 13, EMIT
    db FCALL
    dw dotstack.p
    db LIT, 10, EMIT
    db LIT, 13, EMIT
    db JUMP, -32
    db EXIT

  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 test.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

  ccomma.doc:
    ; db ' ( n -- ) '
    ; db ' compile byte value n at next available byte '
    ; b  ' as given by here.p '
    ; dw $-ccomp.doc
  ccomma:
    dw args.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
    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

  ; call this "create"
  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 ' ( n flag -- ) '
    ; db ' compile item "n" at next availabe position '
    ; b  ' as given by "here" variable where flag indicates '
    ; db ' the type of item. '
    ; db ' flag=1 literal, 2 opcode, 3 procedure'
    ; dw $-compile.doc
  itemcompile:
    dw sliteral.p
    db 'item,', 5
  itemcompile.p:
    ;*** 1=literal number
                      ; n flag
    db DUP            ; n flag flag
    db LIT, 1, EQUALS  ; n flag 0/-1
    db JUMPZ, .notnumber-$    ; 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

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

    ;*** 2 is opcode
    ;*** compile the bytecode to given address
    db DROP         ; op
    db FCALL        ; /get compile point
    dw here.p       ; op adr
    db CSTOREPLUS   ; adr+1
    db FCALL        ; /set compile point
    dw ishere.p     ;
    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

  headdot.doc:
    ; db ' ( xt -- ) '
    ; db ' show the dictionary header for a word function. '
    ; db ' The word header is shown in a form suitable for  '
    ; db ' debugging. '
    ; dw $-headdot.doc
  headdot:
    dw itemcompile.p
    db 'head.', 5
  headdot.p:
                    ; xt
    db DUP          ; xt xt
    db DECR         ; xt xt-1  /points to count|control byte
    db FCALL
    dw rcount.p     ; xt adr n  /start of name
    db DROP         ; xt adr
    db DECR, DECR   ; xt adr-2
    ;db FETCH        ; XT   / get word pointer

    ;*** print head memory address
    db LIT, 13, EMIT
    db LIT, 10, EMIT
    db DUP          ; xt a a
    db FCALL
    dw udot.p       ; xt a
    db LIT, ':', EMIT
    db LIT, ' ', EMIT
    ;*** print the link address to next word in dictionary
    db LIT, '[', EMIT
    db FETCH        ; xt [a]
    db DUP          ; xt [a] [a]
    db FCALL
    dw udot.p       ; xt [a]
    db LIT, ']', EMIT
    db LIT, ' ', EMIT
    db LIT, '-', EMIT
    db LIT, '>', EMIT
    db LIT, ' ', EMIT
    ;*** print the name of the linked word
    db DECR         ; xt A-1   /point to count|control byte
```

```
    db FCALL
    dw rcount.p      ; xt A-n n
    db FCALL
    dw type.p        ; xt
    db LIT, 13, EMIT   ;
    db LIT, 10, EMIT   ;
    db DECR          ; xt-1    /point to count|control byte
    db FCALL
    dw rcount.p      ; a-n n
    db RON, DUP      ; a-n a-n     R: n
    db FCALL
    dw udot.p        ; a-n         R: n
    db LIT, ':', EMIT
    db LIT, ' ', EMIT
    db ROFF          ; a-n n
    db DUP, RON      ; a-n n       R: n
    ;*** print word name in quotes
    db LIT, '"', EMIT
    db FCALL
    dw type.p        ;             R: n
    db LIT, '"', EMIT
    db ROFF          ; n
    db LIT, ' ', EMIT
    ;*** print word length|control in quotes
    ;!! need to handle immediate words which have msb set
    db FCALL
    dw udot.p          ;
    db LIT, 13, EMIT   ;
    db LIT, 10, EMIT   ;
    db EXIT

worddump.doc:
    ; db ' ( xt -- ) '
    ; db ' shows how a word is compiled in the dictionary '
    ; db ' The dictionary header and compiled code is displayed '
    ; db ' for the word corresponding to the execution address. '
    ; dw $-worddump.doc
worddump:
    dw headdot.p
    db '..word', 6
worddump.p:
    db DUP          ; xt xt
    db FCALL
    dw headdot.p    ; xt
    db LIT, 50      ; xt n  /decompile up to n bytes
    db FCALL
    dw decomp.p
    db EXIT
; : word.. dup head. 20 decomp ;

decomp.doc:
    ; db ' ( adr n -- a2 ) '
    ; db ''
    ; db ' decompiles n bytes starting at address n '
    ; db ' returns the next address after the decompiled '
    ; db ' bytes. Need to handle jumps too. This cannot decompile'
    ; db ' machine code, only bytecodes '
    ; dw $-decomp.doc
decomp:
    dw worddump.p
    db 'un,', 3
decomp.p:
                  ; adr n
    db RON        ; adr     r: n

.again:
    db LIT, 13, EMIT
    db LIT, 10, EMIT
    db DUP        ; a a
    db FCALL
    dw udot.p     ; a
    db LIT, ':', EMIT
    db LIT, ' ', EMIT
    db CFETCHPLUS  ; a+1 c   r: n
    db DUP         ; a+1 c c
    db JUMPZ, .invalid-$ ; a+1 c

    db DUP, LIT, NOOP, INCR   ; a+1 c c op+1    r: n
    db ULESSTHAN   ; a+1 c flag        r: n
    db JUMPT, .valid-$   ; a+1 c             r: n

;*** invalid opcode
.invalid:
                      ; a+x c
    db FCALL
    dw udot.p
    db LIT, '?', EMIT

    ;*** clear rstack
    db ROFF, DROP
    db EXIT

.valid:

    ;*** valid opcode
    db DUP         ; a+1 c c         r: n
    db FCALL
    dw dotcode.p   ; a+1 c           r: n
    db LIT, ' ', EMIT

    ;*** is fcall
    db DUP         ; a+1 c c         r: n
    db LIT, FCALL  ; a+1 c c op      r: n
    db EQUALS      ; a+1 c flag    ...
    db JUMPF, 21   ; a+1 c         ...

    ;*** fcall, get xt
    db DROP        ; a+1           ...
    db FETCHPLUS   ; a+3 xt       ...
    db LIT, '<', EMIT
    db FCALL
    dw dotxt.p     ; a+3            r: n
    db LIT, '>', EMIT
    db LIT, ' ', EMIT
    db RLOOP, .again-$   ; a+3      r: n-1
    db ROFF, DROP  ; clear rstack
    db EXIT

    ;*** check if 1 or 2 byte argument or non
    db DUP         ; a+1 c c         r: n

    db FCALL
    dw args.p      ; a+1 c args      r: n
    db DUP         ; a+1 c args args r: n
    db JUMPZ, .zerobytes-$ ; a+1 c args  ...
    db LIT, 1, EQUALS      ; a+1 c flag
    db JUMPT, .onebyte-$   ; a+1 c

    ;*** opcode takes 2 byte argument
.twobytes:
    db DROP        ; a+1           ...
    db FETCHPLUS   ; a+3 xt       ...
    db LIT, '<', EMIT
    db FCALL
    dw dot.p       ; a+3            r: n
    db LIT, '>', EMIT
    db LIT, ' ', EMIT
    db RLOOP, .again-$   ; a+3      r: n-1
```

```
    db ROFF, DROP  ; clear rstack
    db EXIT

    ;*** opcode takes 1 byte argument
  .onebyte:
    db DROP        ; a+1           ...
    db CFETCHPLUS  ; a+2 n         ...
    db LIT, '<', EMIT

    ;*** cdot prints 8 bit number as signed
    db FCALL
    dw cdot.p       ; a+2            r: n
    db LIT, '>', EMIT
    db LIT, ' ', EMIT
    db RLOOP, .again-$  ; a+2      r: n-1
    db ROFF, DROP  ; clear rstack
    db EXIT

  ;*** opcode takes no argument
  .zerobytes:
                       ; a+1 c args  ...
    db DROP, DROP
    db RLOOP, .again-$  ; a+1      r: n-1
    db ROFF, DROP  ; clear rstack
    db EXIT

  tocode.doc:
    ; db ' ( -- adr ) '
    ; db ' puts on the stack the next available byte'
    ; db ' in the dictionary or just the variable ?? '
    ; dw $-tocode.doc
  tocode:
    dw decomp.p
    db '>code', 5
  tocode.p:
    db LITW
    dw tocode.d
    ; db FETCH  / maybe not
    db EXIT
  tocode.d: dw dictionary

  codetohere.doc:
    ; db ' ( -- ) '
    ; db ' sets the dict compile point to the here var'
    ; dw $-codetohere.doc
  codetohere:
    dw tocode.p
    db 'code>here', 9
  codetohere.p:
    db FCALL
    dw here.p    ; a
    db LITW
    dw tocode.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=froth
  ; 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 tocode.d  ; >code*
    db FETCH     ; [>code]
    db LITW
    dw here.d    ; [>code] a
    db STORE     ;
    db EXIT

  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 LITW
    dw here.d     ; adr b
    db STORE      ;
    db EXIT

  ; according to standard forth docs, this should not be modified by
  ; forth words except [ ] etc.
  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

; [ and ] can be defined in source as
; : [ 1 state ! ; immediate
; : ] 0 state ! ; immediate
; but I want to use them to test now
imode.doc:
  ; db ' ( -- ) '
  ; db ' makes all words execute immediately until ] '
  ; db ' This word sets the state variable to true (immediate) '
  ; db ' so that all words parsed will execute immediately '
  ; db ' without being compiled. '
  ; dw $-state.doc
imode:
  dw state.p
  db '[', IMMEDIATE | 1
imode.p:
  db LIT, 1     ; 1
  db LITW
  dw state.d    ; 1 state
  db STORE      ; set state=true
  db EXIT

nmode.doc:
  ; db ' ( -- ) '
  ; db ' normal mode: only immmediate words are executed immediately. '
  ; db ' This word sets the state variable to false (normal) '
  ; db ' so that all words parsed will be compiled unless they '
  ; db ' have their immediate control bit set. '
  ; dw $-state.doc
nmode:
  dw imode.p
  db ']', IMMEDIATE | 1
nmode.p:
  db LIT, 0     ; 0
  db LITW
  dw state.d    ; 0 state
  db STORE      ; set state=false
  db EXIT

first.doc:
  ; db ' ( -- adr ) '
  ; db ' address of first buffer (for loading source from disk)'
  ; dw $-first.doc
first:
  dw nmode.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 2+ is a hack because this is 2 short for some
  ; reason
  dw 2+(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

compare.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 on stack as flag if the strings are '
  ; db ' the same or flag=0 on stack if the strings '
  ; db ' are different. '
  ; dw $-compare.doc
compare:
  dw copy.p
  db 'compare', 7
compare.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

  listxt.doc:
    ; db ' ( adr --  ) '
    ; db ' list all words by name and execution address '
    ; dw $-listxt.doc
  listxt:
    dw compare.p
    db 'listxt', 6
  listxt.p:
    db FCALL
    dw last.p       ; adr
    db FETCH
  .again:
    db DUP          ; adr adr
    db FCALL
    dw udot.p       ; adr
    db LIT, ' ', EMIT    ; print space

    db DECR
    db FCALL
    dw rcount.p       ; a-n n

    ;*** display the word name
    db RON, DUP, ROFF ; A A n
    db FCALL
    dw type.p         ; A
    db LIT, ' ', EMIT
    db DECR, DECR     ; A-2 /point to next pointer
    db FETCH          ; [A-2]
    db DUP            ; *p *p
    db JUMPNZ, .again-$   ; *p
    db DROP               ; clear 0 pointer
    db LIT, '~', EMIT
    db EXIT

  test.type.doc:
    ; db ' testing accept and type '
  test.type:
    dw listxt.p
    db 'test.type', 9
  test.type.p:
    db LIT, '>', EMIT   ;
    db LITW
    dw term.d           ; a
    db DUP              ; a a
    db FCALL
    dw accept.p         ; a
    db COUNT            ; a+1 n
    ;*** show count of buffer
    db DUP              ; a+1 n n
    db FCALL
    dw udot.p           ; a+1 n
    db LIT, ':', EMIT   ; a+1 n
    ;*** show contents of buffer
    db FCALL
    dw type.p
    db LIT, 13, EMIT   ;
    db LIT, 10, EMIT   ;
    db KEY, LIT, 13, EQUALS   ; <escape> terminates
    db JUMPT, -28
    db EXIT

  type.doc:
    ; db ' ( adr n -- ) '
    ; db ' Prints out n number of characters starting '
    ; db ' at address adr. '
    ; dw $-type.doc
  type:
    dw test.type.p
    db 'type', 4
  type.p:
                  ; adr n
    ;*** if count zero, do nothing
    db DUP          ; adr n n
    db JUMPNZ, .sometext-$  ; 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

  accept.doc:
    ; db ' ( buffer -- )
    ; db ' receive a line of input from the terminal '
    ; db ' and store it as a counted string in the buffer. '
    ; db ' should have a character limit. backspaces are mainly
    ; db ' handled. '
    ; dw $-accept.doc
  accept:
    dw type.p
    db 'accept', 6
  accept.p:
    ;*** to elimate backspace problems, need to
    ;    emit character after finding out what it is
    ;    not before
                    ; a
    db DUP, RON     ; a       r: a
    db INCR         ; a+1     r: a
  .nextkey:
    db KEY, DUP     ; a+1 c c r: a
    db DUP, EMIT    ; a+1 c c r: a

    ;*** enter terminates input
    db LIT, 13, EQUALS; a+1 c flag   r: a  /enter press
    db JUMPT, 32    ; a+1 c        r: a

    ;*** handle backspace
    db DUP          ; a+1 c c      r: a
    db LIT, 8, EQUALS ; a+1 c flag   r: a  /backspace
    db JUMPF, 22    ; a+1 c        r: a
    db DROP         ; a+1          r: a
    ;*** test for at 1st char
    db DUP          ; a+1 a+1      r: a
    db ROFF, DUP, RON ; a+1 a+1 a    r: a
    db MINUS        ; a+1 n        r: a
    db LIT, 1       ; a+1 n 1      r: a
    db EQUALS       ; a+1 flag     r: a
    db JUMPT, -24

    ;*** not 1st char so go back 1 space
    db LIT, ' ', EMIT ; a+n      r: a
    db LIT, 8, EMIT   ; go back
    db DECR           ; a+n-1    r: a
    db JUMP, -33      ; a+n-1    r: a /get next char

    ; put the character limit test here.
    db SWAP         ; c a+1        r: a
```

```
    db CSTOREPLUS     ; a+2          r: a
    db JUMP, -37      ; a+2          r: a
                      ; a+n 13       r: a
  .exit:
    db LIT, 10, EMIT  ; print newline if enter pressed
    db LIT, 13, EMIT  ;
    db DROP           ; a+n          r: a
    db ROFF           ; a+n a
    db DUP, RON       ; a+n a        r: a
    db MINUS          ; n            r: a
    db DECR           ; n-1          r: a
    db ROFF           ; n-1 a
    db CSTORE
    db EXIT

  in.doc:
    ; db ' ( -- adr )
    ; db 'Puts on the stack the address of the '
    ; db 'current input source/ buffer '
    ; dw $-in.doc
  in:
    dw accept.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 64 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, toin.d and in.d are
    ; pointers and need to be fetched before use... *p
    ;
    db LITW
    dw toin.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
  toin.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 toin.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 toin.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 toin.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 toin.d       ; adr adr toin.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

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

  pad.doc:
    ; db ' ( -- adr )
    ; db 'Puts on the stack the address of the '
    ; db 'general purpose text buffer '
    ; dw $-pad.doc
  pad:
    dw buff.p
    db 'pad', 3
  pad.p:
    db LITW
    dw pad.d
    db EXIT
  pad.d: times 256 db 0

  one.doc:
    ; db ' A loop to compile and execute one word typed'
    ; db ' at the terminal and interpreted from the term.d'
    ; db ' buffer. The word can be either opcode '
    ; db ' procedure or number '
  one:
    dw pad.p
    db 'one', 3
one.p:
    db FCALL
    dw listxt.p
    db LIT, 10, EMIT
    db LIT, 13, EMIT

.again:
    db LITW
    dw term.d
    db FCALL
    dw accept.p

    db LITW
    dw term.d    ; adr

    db CFETCH       ; n
    db JUMPZ, -10   ;
    db LITW
    dw term.d       ; adr

    db COUNT        ; adr+1 n
    db FCALL
    dw find.p       ; xt
    db LIT, 'x', EMIT
    db LIT, 't', EMIT
    db LIT, '=', EMIT

    db DUP          ; xt xt
    db FCALL
    dw udot.p       ; xt
    db DUP          ; xt xt
    db FCALL
    dw opcode.p     ; xt op/0
    db DUP          ; xt op op
    db LIT, ' ', EMIT
    db LIT, 'o', EMIT
    db LIT, 'p', EMIT
    db LIT, '=', EMIT
    db FCALL
    dw udot.p       ; xt op
    db LIT, ' ', EMIT

    ;*** check if valid opcode
    db DUP          ; xt op op
    db JUMPZ, 17    ; xt op    /opcode or procedure

    ;*** compile the bytecode to anon buffer
    db SWAP, DROP   ; op
    db LITW
    dw anon.d       ; op adr
    db CSTOREPLUS   ; adr+1
    db LIT, EXIT    ; adr+1 n
    db SWAP         ; n a+1
    db CSTORE       ;

    db FCALL
    dw anon.d
    db JUMP, 21
    ;

    ;*** check if valid word
                    ; xt op
    db DROP         ; xt
    db DUP          ; xt xt
    db JUMPZ, 19    ; xt

    ;**** here compile fcall and xt
```

```
    db LITW         ;
    dw anon.d       ; xt a
    db LIT, FCALL   ; xt a n
    db SWAP         ; xt n a
    db CSTOREPLUS   ; xt a+1
    db STOREPLUS    ; a+4
    db LIT, EXIT    ; a+4 n
    db SWAP         ; n a+4
    db CSTORE       ;
    db FCALL
    dw anon.d
    db JUMP, 43

    ;*** check if valid number
                    ; xt
    db DROP
    db LITW         ;
    dw term.d       ; adr
    db COUNT        ; adr+1 n
    db FCALL
    dw tonumber.p   ; a/n flag
    db JUMPNZ, 5    ; a/n

    db DROP         ;
    db JUMP, 30
                    ; n
    ;*** valid number, so print
    db LIT, 'n', EMIT
    db LIT, '=', EMIT
    db DUP          ; n n
    db FCALL
    dw udot.p       ; n
    db LIT, ' ', EMIT

    db LITW         ;
    dw anon.d       ; n a
    db LIT, LITW    ; n a op
    db SWAP         ; n op a
    db CSTOREPLUS   ; n a+1
    db STOREPLUS    ; a+3
    db LIT, EXIT    ; a+3 op
    db SWAP         ; op a+3
    db CSTORE       ;

    db FCALL
    dw anon.d       ;
    ;
    db LIT, 10, EMIT
    db LIT, 13, EMIT
    db LJUMP
    dw .again-$
    db EXIT

  drive.doc:
    ; db ' ( -- adr )
    ; db ' a variable to hold virtual drive number '
    ; db ' but this should be an opcode '
  ; drive: /another drive
    dw one.p
    db 'drive', 5
  drive.p:
    db LITW
    dw drive.d     ; ad
    db EXIT
  drive.d: db -1

  lib.doc:
    ; db ' ( -- adr )
    ; db ' some source code to load'
    ; dw $-lib.doc
  lib:
    dw drive.p
    db 'lib', 3
  lib.p:
    db LITW
    dw lib.d
    db EXIT
  lib.d:
     db lib.end-$-1
     ; immediate state vs normal/compile state
     ; block1 is the sector (512 byte block) of the first
     ; source code block (forth block=1K bytes) on disk
     db ' : 2* dup + ; '
     ; load any 1K source code block with eg: 0 load 1 load 2 load
     ; do drop after read because it returns a flag
     db ' : load 2* block1 + 2 first read first 1024 source ; '
     ; actually parse and compile this source code
     db ' 0 load '
     db ' 1 load '
     db ' 2 load '
     ; block3 is probably outside of 16 sector limit (need to deal
     ; with head, cylinder, track disk geometry to get more)
     ; db ' 3 load '

  ; : word.. dup head. 20 un, ;
  ; : in.. in count dup u. sp type ;

  lib.end:

  last.doc:
    ; db ' ( -- adr )
    ; db 'Puts on the stack a pointer to 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 lib.p
    db 'last', 4
  last.p:
    db LITW
    dw last.d      ; ad
    ;db FETCH       ; [ad]
    db EXIT
  last.d: dw last.p

  ; testing multisector stack machine byte code

  code:

    ; lib.p just creates and runs LOAD which loads the first
    ; disk source code block
    db FCALL
    dw lib.p
    db COUNT  ; source takes address + length
    db FCALL
    dw source.p
    ; source needs to save current input stream?
    ; maybe not because all code is compiled...
    db FCALL
    dw shell.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
    mov [drive.d], byte dl ;

    ; 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.
; The code block below can be preprocessed with a sed line
; such as the one below. This allows us to write multiline
; forth source code with the "db ' " guff.
;


%if 0 ; code{

; 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.
: ' wparse find ; imm

; if call, is not immediate then this gets simpler
; does it need to be immediate?
: post wparse find ' call, call, ; imm
;: post wparse find [ wparse call, find ] call, ; imm

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

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

; a modulus operator
: mod /mod drop ;
; a divide word
: / /mod swap drop ;
; a standardish word to see a decompilation: check!
: see wparse tick drop ..word ; imm
; type a space
: space ( -- ) 32 emit ;
; n spaces
: spaces ( n -- ) begin space 1- dup 0 = until drop ;
; put a space on the stack
: bl 32 ;
; type a newline
: cr 13 emit 10 emit ;
; ?dup, duplicate TOS
: ?dup dup 0 = if exit else dup fi ;
; standard tuck. This can be a handy way to save a value
; for later processing
: tuck  ( a b -- b a b )
   swap 2dup drop ;
; a recursive greatest common divisor word
; from r.v.noble
: gcd  ( a b -- gcd )
   ?dup if tuck mod gcd fi ;

: count c@+ ;
```

```
  : K 1024 u* ;

  ; Display keyboard event codes of keys pressed
  : ekeycode
      begin
        ekey 2dup swap u. space u.
        drop cr [char] q =
      until ;

  ; classic forth variables.
  : var create 0 , ; imm

  ; definition of CONSTANT
  : con create , does> @ ; imm

  ; 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 - ;

  ; list all words in the dictionary
  : list last @
     begin
       ; dup .xt space xt+ dup 0 =
       dup .xt dup imm?
       ; make red if immediate word
       if 12 fg else 11 fg fi
       space xt+ dup 0 =
     until drop ;

  ; 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.
  : ii r> r> dup >r swap >r ;
  : ii++ r> r> 1+ >r >r ;

  16 vid

  ; Display asci codes of keys pressed
  ; cant use this char in colon defs
  ; could recode this to use ekeycode
  : keycode
      begin
        key dup u. space dup emit cr 113 =
      until ;

%endif ; }code



  ;db ' : [ 1 state ! ; imm '
  ;db ' : ] 0 state ! ; imm '

  ;db ' : testnl ." stuff with 13,10" ; ',13,10
  ; put a space on the stack
  ; pad remainder of 1st disk block with zeros
  times 1024-($-block1) db 0

block2:
  ; 2nd disk block


%if 0 ; code{

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

  ; a non standard type of comment that reads until a "."
  : ./ [char] . parse drop drop ; imm

; compute the nth fibonacci using recursion
: fib ( n -- fib[n] )
  dup 1 swap < if 1- dup 1- fib swap fib + fi ;

; return stack
: r@ ( -- n ) ( R: n -- n )
  ;./ get a copy of the top most item from the return stack to
  ;   the data stack.
  r> dup >r ;

; lookup a word and return the opcode or zero
: >op wparse find opcode ; imm

; use this >op in : defs
: [>op] post >op post literal ; imm

; list all opcodes (55=NOP which is the last in the table)
: opcodes [>op] nop >r
    begin
      ii dup u. [char] : emit .code space
    loop r> drop ;

; 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 find opcode c, ; imm

; list colours
: color 0 begin dup fg space dup u. 1+ dup 33 = until drop ;

; print n coloured blocks
: glow ( n -- )
    begin
      dup fg 219 emit 1- dup 1 =
    until
    drop ;

 ; make an asci box n wide, eg: 7 box
 ; could factor with 'line'
 ; : ascline begin 196 emit loop
 : box 218 emit dup >r
    begin
      196 emit
    loop
    r> drop 191 emit cr
    192 emit dup >r
    begin
      196 emit
    loop
    r> drop 217 emit ;

 ; 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 ;

; 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')
: block ( n -- ) 2* block1 + 2 first read first ;
; displays block n on the screen. In a real implementation this
; would use block
: screen ( n -- ) 2* block1 + 2 first read first 1024 type ;
```

```
%endif ; }code


; display all words in buffer
; : inwords term accept term count in0
;     begin
;       wparse 2dup type cr
;       0 =
;     until ;

; pad remainder of 2nd block with zeros
times 1024-($-block2) db 0

block3:
; 3rd disk block

%if 0 ; code{
 : B3! [ here ] literal ;

 ; algorithm to convert BCD encoded values to binary (normal)
 ; binary = ((bcd / 16) * 10) + (bcd & 0xf)

 ; find out how many bytes are free in source block n
 : bfree ( n -- free )
     block dup begin c@+ 0 = until swap - 1024 swap - swap drop ;

 ; draw a line starting a pixel position xy of length n
 ; ( x y n -- )
 : line
   >r begin 2dup pix 1+ swap 1+ swap loop r> drop ;

 ; an asci table
 : asci 1
    begin
      dup u. space dup emit space 1+ dup 255 =
    until drop ;

 ; a colourful asci list. call with eg: 4 asc
 : asc begin dup fg dup emit 1+ dup 255 = until drop ;

 ; standard s" word
 : s" [char] " 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...
 ;: ." post s" [ wparse type find ] literal post call, ; imm
 ;: ." post s" [ ' type ] literal post call, ; imm
 : ." post s" ' type literal post call, ; imm

 ; a convenience to get rid of stuff on the stack
 : dd drop drop drop ;

 : 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 ." Froth, A 'Forthish' System " cr cr
     13 fg
     24 spaces ."    Opcodes = "
       ' nop literal ' exec literal - u. ." bytes" cr
     24 spaces ." Byte codes = "
       ' last literal ' nop literal - u. ." bytes" cr
    24 spaces ."      Total = " last @ u. ." bytes" cr ;

   splash

%endif ; }code

times 1024-($-block3) db 0

; start block4:
%if 0 ; code{

%endif ; }code

; can leave for testing preprocessor
;db ' : testnl 66 emit ; ',13,10
;db ' : newnl 66 emit ; ',13,10

times 4*1024-($-block1) db 0

; 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

; below contains lots of standardish forth words.
; This is just a guide to naming conventions and to avoid name clashes
; with standard words. This is also an attempt to create a "literate"
; style of forth coding, where the explanation and documentation of
; the forth word is clearly visible next to the words definition.

%if 0

   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

  ; better to use r@ here, more readable than r> dup >r
  of2 ( a b n -- n/0 )
     if n == a or b return that value, otherwise return 0.
  : of2 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

; double numbers, which are 2 stack item numbers
; high (top) stack position is high order value
; eg D == a b where "b" is high value and a low
d+ ( a b c d -- b:a + d:c )
d-
d2*
d2/
d>s convert double number


; 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  ;

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

within ( n a b )
  return true != 0 if a <= n <= b, otherwise return false==0

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
```