Collapse OS Documentation

This document was created at 2025-12-17 23:00:00 from documentation in CollapseOS v1.

Table of contents

Documentation Files

1 General Documentation

1.1 Introduction to Collapse OS (`intro.txt`)

Collapse OS is a minimal operating system created to preserve the ability to program microcontrollers through civilizational collapse. Its author expects the collapse of the global supply chain means the loss of our computer production capability. Many microcontrollers require a computer to program them.

Collapse OS innovates by self-hosting on extremely tight resources and is thus able to operate and be improved in a world without modern computers.

Forth

This OS is a Forth. It doesn't adhere to any pre-collapse standard, but is pretty close to the Forth described in Starting Forth by Leo Brodie. It is therefore the recommended introductory material to learn Forth in the context of Collapse OS.

If you don't have access to this book and don't know anything about Forth, learning Collapse OS could be a rough ride, but I'm sure you can do it. Begin with doc/usage.

Documentation and self-hosting

Collapse OS is self-hosting, its documentation is not, that is, Collapse OS cannot read this document you're reading. Text blocks could, of course, be part of Collapse OS' blocks, but doing so needlessly uses blocks and make the system heavier than it should.

This documentation is expected to be printed before the last modern computer of your community dies.

Virgil's workspaces

When you explore the contents of Collapse OS' code, you'll notice "Virgil's workspace" littered around. Those blocks are of no importance to you directly, but they contain code that I use myself while I work from within Collapse OS.

They can provide great insights as to how tools are supposed to be used.

Where to begin?

If you're reading this and don't know where to begin, you're likely to have access to a modern computer. The best place to begin is to build the C VM of Collapse OS in /cvm. You can then begin playing with it with the help of doc/usage and doc/impl.

When you're ready to move to real hardware, read doc/hw/intro.

Other topics in this documentation

* Dictionary of core Forth words (doc/dict)
* Understanding Collapse OS (doc/grok)
* Design considerations (doc/design)
* Editing text (doc/ed)
* Editing binary memory (doc/me)
* Assembling binaries (doc/asm/intro)
* Disassembling binaries (doc/dis)
* Emulators (doc/emul)
* Code conventions (doc/code/intro)
* Cross-compilation mechanisms (doc/cross)
* Architecture management (doc/arch)
* Bootstrap Collapse OS to a new system (doc/bootstrap)
* Hardware Drivers (doc/drivers)
* Self-hosting notes (doc/selfhost)
* Frequently asked questions (doc/faq)

There are other subjects, you are invited to browse the doc/ folder to discover them.

1.2 Collapse OS usage guide (`usage.txt`)

This usage guide begins with a Forth primer, but a complete introduction to Forth is out of the scope of this document.

Therefore, it is strongly recommended to read an introduction to Forth before reading this below. "Starting Forth" be Leo Brodie is a very good one.

If you don't have access to such documentation, you can try yourself with the primer below, but it's likely to be a steep learning curve.

First steps

Before you read this primer, let's try a few commands, just for fun.

```
42 .
```

This will push the number 42 to the stack, then print the number at the top of the stack.

```
4 2 + .
```

This pushes 4, then 2 to the stack, then adds the 2 numbers on the top of the stack, then prints the result.

If you just type ".", you'll see "stack underflow" because the "." word tries to fetch a number from the stack, which is empty. You can inspect your stack with ".S".

```
42 $8000 C! $8000 C@ .
```

This writes the byte "42" at address $8000 ($ prefix is for hex notation), and then reads back that byte from the same address and print it.

Interpreter loop

Forth's main interpeter loop is very simple:

1. Read a word from input.
2. Is it a number literal? Put it on the stack.
3. No? Look it up in the dictionary.
4. Found? Execute.
5. Not found? Error.
6. Repeat

Word

A word is a string of non-whitepace characters. We consider that we're finished reading a word when we encounter a whitespace after having read at least one non-whitespace character.

Character encoding

Collapse OS doesn't support any other encoding than 7bit ASCII. A character smaller than $21 is considered a whitespace, others are considered non-whitespace.

Characters above $7f have no special meaning and can be used in words (if your system has glyphs for them).

Comments

```
Both () and \ comments are supported. The word "(" begins a
comments and ends it when it reads a ")" word. It needs to be a
word, that is, surrounded by whitespaces. "\" comments the rest
of the line.
```

Dictionary

```
Forth's dictionary link words to code or data. Unless you're
cross compiling (doc/cross), there is only one dictionary. On
boot, this dictionary contains the system's words (look in
doc/dict for a list of them), but you can define new words with
the ":" word. For example:

: FOO 42 . ;

defines a new word "FOO" with the code "42 ." linked to it. The
word ";" closes the definition. Once defined, a word can be
executed like any other word.

You can define a word that already exists. In that case, the new
definition will overshadow the old one. However, any word def-
ined *before* the overshadowing took place will still use the
old word.

: foo 42 . ;
: bar foo ;
: foo 43 . ;
foo \ prints 43
bar \ prints 42

You can get the address of a word with "'":

' foo
DUP .X \ prints an address
EXECUTE \ prints 43

You can "rewind" the dictionary with FORGET. This word "forgets"
the specified word along with all words following it:

FORGET bar
bar \ error: word does not exist
foo \ prints 42

FORGETing a system word breaks the system.
```

Cell size and endian-ness

The cell size in Collapse OS is 16 bit, that is, each item in stacks is 16 bit, @ and ! read and write 16 bit numbers. Whenever we refer to a number, a pointer, we speak of 16 bit.

Endian-ness is arch-dependent and core words dealing with words will read-write according to native endian-ness. On a z80, "$8000 @" puts $8000 in LSB and $8001 in MSB, but on a 6809, it's the opposite.

To read and write bytes, use C@ and C!.

Number literals

Traditional Forths often uses HEX/DEC switches to go from decimal to hexadecimal parsing. Collapse OS has no such mode.

Straight numbers are decimals, numbers starting with "$" are hexadecimals (example "$12ef"), char literals are single characters surrounded by ' (example 'X'). Char literals can't be used for whitespaces (conflicts with the concept of "word" as defined above).

Signed-ness

For simplicity purposes, numbers are generally considered unsigned. For convenience, decimal parsing and formatting support the "-" prefix, but under the hood, it's all unsigned.

This leads to some oddities. For example, "-1 0 <" is false. To compare whether something is negative, use the "0<" word which is the equivalent to "$7fff >".

Parameter Stack

Unlike most programming languages, Forth executes words directly, without arguments. The Parameter Stack (PS) replaces them. There is only one, and we're constantly pushing to and popping from it. All the time.

For example, the word "+" pops the 2 numbers on the Top Of Stack (TOS), adds them, then pushes back the result on the same stack. It thus has the "stack signature" of "a b -- n". Every word in a dictionary specifies that signature because stack balance, as you can guess, is paramount. It's easy to get confused so you need to know the stack signature of words you use very well.

Return Stack

```
There's a second stack, the Return Stack (RS), which is used to
keep track of execution, that is, to know where to go back after
we've executed a word. It is also used in other contexts, but
this is outside of the scope of this primer.
```

Conditional execution

```
Code can be executed conditionally with IF/ELSE/THEN. IF pops
PS and checks whether it's nonzero. If it is, it does nothing.
If it's zero, it jumps to the following ELSE or the following
THEN. Similarly, when ELSE is encountered in the context of a
nonzero IF, we jump to the following THEN.

Because IFs involve jumping, they only work inside word defin-
itions. You can't use IF directly in the interpreter loop.

Example usage:

: FOO IF 42 ELSE 43 THEN . ;
0 FOO \ prints 43
1 FOO \ prints 42
```

Loops

```
Loops work a bit like conditionals, and there's 3 forms:

BEGIN..AGAIN --> Loop forever
BEGIN..UNTIL --> Loop conditionally
X >R BEGIN..NEXT --> Loop X times

UNTIL works exactly like IF, but instead of jumping forward to
THEN, it jumps backward to BEGIN.

NEXT decreases RS' TOS by one and if zero isn't reached, jumps
backward to BEGIN.

Why not have a FOR which would be the equivalent of ">R BEGIN"?
Because in many cases, maybe even most, the order of arguments
in PS is such that it's more convenient to perform the ">R" a
little earlier. Doing so right before BEGIN results in needless
stack juggling. The lack of FOR makes all loops begin with
BEGIN, which helps overall readability.

You can use the word "LEAVE" to exit a NEXT loop early. When
```

used, it will finish the current loop and then stop looping when NEXT is reached.

```
: foo 5 >R BEGIN R@ 3 = IF LEAVE THEN R@ . NEXT ;
foo \ prints 543
```

Exiting early

You can leave a word early with EXIT:

```
: foo 42 . EXIT 43 . ;
foo \ only 42 is printed
```

When you're inside a BEGIN..AGAIN or BEGIN..UNTIL, you can use EXIT just fine, but if you're inside a NEXT loop, you have to drop RS' TOS with R~ before calling EXIT or else you have a messed up Return Stack and all hell breaks loose.

Memory access and HERE

We can read and write to arbitrary memory address with @ and ! (C@ and C! for bytes). For example, "1234 $8000 !" writes the word 1234 to address $8000. We call the @ and ! actions "fetch" and "store".

There's a 3rd kind of memory-related action: "," (write). This action stores value on PS at a special "HERE" pointer and then increases HERE by 2 (there's also "C," for bytes).

HERE is initialized at the first writable address in RAM, often directly following the latest entry in the dictionary. Explaining the "culture of HERE" is beyond the scope of this primer, but know that it's a very important concept in Forth. For example, new word definitions are written to HERE.

Linking names to addresses

Accessing addresses only with numbers can become confusing, us humans often need names associated to them. You can do so with CREATE. This word creates a dictionary entry of the "cell" type. This word, when called, will put its own address on the stack. You are responsible for allocating a proper amount of memory to it.

For example, if you want to store a single 16-bit number, you would do "CREATE foo 2 ALLOT". You can then do stuff like "42 foo ! foo @ . (prints 42)"

```
Cells can store more than just a number, they can hold
structures and array. Simply ALLOT appropriately and then use
this memory as you wish.

Another way to link a name to an address is VALUE. The "VALUE"
word takes a value parameter and creates a special "value" type
word. This word type always allocates 2 bytes of memory and when
called, instead of spitting its address, spits the 16-bit value
at that address.

You can change the number associated with a VALUE with TO.
Example:

42 VALUE foo foo . ( prints 42 )
43 TO foo foo . ( prints 43 )

VALUEs make more readable code in cases where the value is more
often read than written. Also, reading it is faster (writing is
slower). Compactness is the same.

Multiple values can be declared at once with VALUES and CONSTS:

3 VALUES foo bar baz \ all values are 0
3 CONSTS 1 foo 2 bar 3 baz \ foo=1 bar=2 baz=3
```

The semantics of TO

```
The word "TO" as described above might seem a bit like magic
and requires further explanation. The mechanism through which
the call to the VALUE "foo", which normally reads the value
becomes a write is special.

TO does one very simple thing: it sets the "TO?" flag in SYSVARS
(see doc/impl). Then, the code that handles VALUE calls (which
is a core routine, see doc/impl) checks whether the flag is set.
If it's not, it's a regular read. If it is, it resets the flag
and does a write.

Because the TO? flag is global, the call to TO has to be very
close to its target, ideally adjacent. If you call other words
in between, the value of the TO? flag will mess things up and
transform reads into writes, writes into reads, hell freezes
over, cats and dogs living together. Be responsible with TO
placement.
```

DOER and DOES>

```
DOER and DOES> allow to bind data and behavior together in a
```

space-efficient way. Those words are called "does words" and, when created, behave a bit like a cell (a CREATE word): it pushes its own address to PS. But then, instead of just continuing along, it executes its DOES> instructions. Example:

```
: printer DOER , DOES> @ . ;
42 printer foo
43 printer bar
foo \ prints 42
bar \ prints 43
```

DOER creates a special "does" entry and DOES> tells the latest DOER entry where to jump for its behavior. The instructions following DOES> are not executed when the DOER is defined, only when it's executed. This execution always happen in a context where the DOER's address in on PS. This is why, in the example above, we call "@" before ".".

IMMEDIATE

So far, we've covered the "cute and cuddly" parts of the language. However, that's not what makes Forth powerful. Forth becomes mind-bending when we throw IMMEDIATE into the mix.

A word can be declared immediate thus:

```
: FOO ; IMMEDIATE
```

That is, when the IMMEDIATE word is executed, it makes the latest defined word immediate.

An immediate word, when used in a definition, is executed immediately instead of being compiled. This seemingly simple mechanism (and it *is* simple) has very wide implications.

For example, The words "(" and ")" are comment indicators. In the definition:

```
: FOO 42 ( this is a comment ) . ;
```

The word "(" is read like any other word. What prevents us from trying to compile "this" and generate an error because the word doesn't exist? Because "(" is immediate. Then, that word reads from input stream until a ")" is met, and then returns to word compilation.

Words like "IF" and "BEGIN" are all regular Forth words, but their "power" come from the fact that they're immediate.

Starting Forth by Leo Brodie explains all of this in detail.

Read this if you can. If you can't, well, let this sink in for a while, browse the dictionary (doc/dict) and try to understand why this or that word is immediate. Good luck!

Memory map

Memory is filled by 4 main zones:

1. Boot binary: the binary that has to be present in memory at boot time. When it is, jump to the first address of this binary to boot Collapse OS. This code is designed to be able to run from ROM: nothing is ever written there.
2. Work RAM: As much space as possible is given to this zone. This is where HERE begins.
3. SYSVARS: Hardcoded memory offsets where the core system stores its things. It's $60 bytes in size. If drivers need more memory, it's bigger. See doc/impl for details.
4. PS+RS: Typically around $100 bytes in size. Their implementation is entirely arch-specific. Overflows aren't checked, PS underflows are checked through SCNT.

Unless there are arch-related constraints, these zones are placed in that order (boot binary at addr 0, PSP at $ffff).

Strings and lines

Strings in Collapse OS are an array of characters in memory associated with a byte length. There are no termination.

This length, when refering to that string in the different string handling words, is usually passed around as a separate argument in PS. It is common to see "sa sl", "sa" being the string's address, "sl" being its length.

How that "sl" is encoded depends on the situation. For example, the S" word, which writes the enclosed string and, at runtime, yields "sa sl", is wrapped around a branch word (so that the string isn't evaluated by forth) followed by 2 number literals.

When we refer to a "line", it's a string that is of size LNSZ, a constant that is always 64. It corresponds to the size of the input buffer and to the size of a line in a Block (16 lines per block).

Because those lines have a fixed length, we sometimes want to know the length of the actual content in it (for example, to EMIT it). When we do so, for example in LNLEN, we go through the whole line and check when is that last visible character, that is, the last one that is higher than $20 (space). That's where

our line ends.

We don't use any termination character for lines, it's too messy. Blocks might not have them, and when we want to display lines in a visual mode (that is, always the full 64 characters on the screen), we need complicated CR handling. It's simpler to fill lines in blocks with spaces all the way.

Branching

Branching in Collapse OS is limited to 8-bit. This represents 64 word references (or a bit less if there are literals and branches) forward or backward. While this might seem a bit tight at first, having this limit saves us a non-negligible amount of resource usage.

The reasoning behind this intentional limit is that huge branches are generally an indicator that a logic ought to be simplified. So here's one more constraint for you to help you towards simplicity.

When you compile branches, if you go over that limit, you'll get a "br ovfl" (branch overflow) error.

Interpreter and I/Os

Collapse OS' main I/O loop is line-based. INTERPRET calls WORD which then iterates over the current "input buffer" (INBUF) for characters to eat up. That input buffer is a 64 characters space in SYSVARS where typed characters are buffered from KEY, but that's not always the case.

During a LOAD, the input buffer pointer changes and points to one of the 16 lines of the BLK buffer. WORD eats it up just the same, but it ain't coming from KEY anymore. When the 16th line is read, we come back to the regular program.

Back to KEY. It always yields a characters, which means it blocks until it yields. It loops over KEY? which returns a flag telling us whether a key is pressed, and if there is one, the character itself.

KEY? is an alias which points to a driver implementing this routine. It can also be overridden at runtime for nice tricks. For example, if you want to control your computer from RS-232, you can do "' RX<? 'KEY? !".

Interpreter output is unbuffered and only has EMIT. This word can also be overriden, mostly as a companion to the *raison

```
d'etre* of your KEY? override.
```

Interpreting and compiling words

```
When the INTERPRET loop reads from INBUF, it separates its input
in words which yields chunks of characters.

Whenever we have a word, we begin by checking if it's a number
literal with PARSE. If yes, push it on the stack and get next
word. Otherwise, check if the word exists in the dictionary.
If yes, EXECUTE. Otherwise, it's a "word not found" error.

Compiling words with ":" follows the same logic, except that
instead of putting literals on the stack, it compiles them with
LITN and instead of executing words, it writes their address
down (except immediates, which are executed).

This "PARSE then FIND" order is the opposite of many traditional
Forths, which generally go the other way around. This is because
traditional forths often don't have hexadecimal prefixes for
their literals and the "PARSE then FIND" order would prevent the
creation of words like "face", "beef", cafe", etc. This is not
a problem we have in Collapse OS.

"PARSE then FIND" is faster because it saves us a dictionary
lookup when parsing a literal.
```

Word Not Found override

```
It's possible to override the "word not found" behavior and
instead execute some kind of "catch all" word. You do so through
the '(wnf) sysvars.

By default, this variable points to (wnf), which simply spits
out the "word not found" error. You can make this variable point
to any word with a ( -- ) signature.

To access the word currently being parsed, use CURWORD.
```

Native words

```
Native words are regular forth words wrapping binary executable
code.

With the proper assembler loaded in memory, you can compile
words that directly execute native code. Here's a z80 example:
```

```
CODE foo BC PUSH, BC 42 LDdi, ;CODE
```

See doc/asm/intro for more details.

Aliases

Sometimes, often for fulfilling protocols, we want to "plug" a word into another, for example, we want FOO and BAR to mean the same thing. Of course, you can do ": BAR FOO ;", but this represents an annoying overhead, both in terms of speed and RS space. In this case, you'll want to create an alias like this:

```
ALIAS FOO BAR
```

Which means "make BAR point to FOO". This generates a native jump which is pretty much as low overhead as it can be.

Those aliases are read-only. Once created, they can't be changed. If you want to use a word as an indirection, you need to use execute like this:

```
: FOO ;
' FOO VALUE 'BAR
: BAR 'BAR EXECUTE ; \ BAR executes FOO
: BAZ ;
' BAZ TO 'BAR \ BAR EXECUTES BAZ
```

System aliases

Core words have 2 special aliases, which jump to an address determined in their corresponding SYSVAR. These are EMIT and KEY?.

Each of these system aliases have their corresponding "'" SYSVAR address CONSTANT. You go through them to modify where the alias jumps to. Example:

```
' RX<? 'KEY? !
' TX> 'EMIT !
```

System values

Most SYSVARS described in doc/impl have a constant corresponding to their absolute address. For example, you get the value of "NL" with "NL @" and set it with "NL !".

Some SYSVARS are very often used and necessitate faster access.

These SYSVARS are split in 2 words: the accessor and the address. For example, we have HERE and 'HERE. HERE returns HERE's value directly and 'HERE returns HERE's address. Therefore, you get HERE with "HERE" and set it with "'HERE !".

The list of such SYSVARS is:

HERE CURRENT IN(IN>

The A register

The A register is an out of stack temporary value that often helps minimize stack juggling. Its location is arch-dependent, but it's often in SYSVARS. On register-rich CPUs, it's a register.

Access to it is fast, but its downside is that words using it must be careful not to use words that also use the A register. doc/dict indicate such words with *A*.

Dealing with performance bottlenecks

Because Collapse OS runs on multiple CPUs, dealing with bottlenecks is a bit tricky. We want to avoid, in arch-independant application code (VE, ME, assemblers, emulators), to maintain bottleneck words in all supported architectures.

The way we deal with this situation is by declaring bottleneck words as "back-overridable" with the word ?: (instead of :).

This word creates a new word only if the specified name doesn't already exist in the dictionary. With this, what you can do is optionally load "speedup words" for your arch, and then load your app. Your sped-up version will superseed the default, slow version and your bottlenecks will be faster. Example:

```
\ My super app
?: slowstuff ( ... ) ;
: myapp ( ... ) slowstuff ( ... ) ;

\ My arch-specific speedup
CODE slowstuff ( ... ) ;CODE
```

If you load the app without loading speedups, "slowstuff" will be slow, but will work under all arches. If you load your speedups first, then the forth version of "slowstuff" will never be created and "myapp" will refer to the fast "slowstuff" instead.

Mass storage through disk blocks

Collapse OS can access mass storage through its BLK subsystem. See doc/blk for more information.

It is through this subsystem that applications are loaded, so you'll want to look at doc/blk on this subject too.

Useful little words

In Collapse OS, we try to include as few words as possible into the cross-compiled core, making it minimally functional for reaching its design goals.

However, in its source code, it has a section of what is called "Useful little words" at B1-B9 and you'll probably want to load some of them quite regularly because they make the system more usable.

Contexts

B3 provides the word "context" allowing multiple dictionaries to exist concurrently. This allows you to develop applications without having to worry too much about name clashes because those names exist in separate namespaces.

A context is created with a name like this:

```
context foo \ creates context "foo"
```

When a context is created, it is "branched off" CURRENT as it was at the moment the context was created.

To activate a context, call its name (in the case, "foo"). This will do two things:

1. Save CURRENT in the previously active context.
2. Restore CURRENT to where it was the last time "foo" was active (or created).

Note that creating a context doesn't automatically activate it.

Code generation

The kernel has 2 words that generate native code and although they're there as support for define words (:, VALUE, etc.), they can be used for interesting thing.

```
These words are JMPi! CALLi! and have the same signature of
"n a -- len".

For example, let's say that you're debugging the kernel and want
to ruthlessly patch a word with another behavior you're trying
out. You could do:

' newword ' wordtopatch JMPi! DROP

And poof! wordtopatch is now an alias to newword.
```

1.3 Implementation notes (`impl.txt`)

Execution model

```
At the end of BOOT, we call ABORT which triggers our main loop,
which is in lblmain. It's very simple: Initialize input buffer,
then call INTERPRET.

INTERPRET itself is very simple: repeatedly call RUN1 (run one
word).

RUN1 implements this logic:

1. Read a word from input.
2. Is it a number literal? Put it on the stack.
3. No? Look it up in the dictionary.
4. Found? Execute.
5. Not found? Error.
```

Dictionary entry

```
A forth binary is, in its vast majority, a big dictionary of
words. The dictionary is a list of entries, the address of its
last entry being kept in CURRENT. A dictionary entry has this
structure:

- Xb name. Arbitrary long number of character (but can't be
  bigger than input buffer, of course).
- 2b previous entry
- 1b name size + IMMEDIATE flag (7th bit)
- The word content (see DTC explanation below)

The previous entry field is the address of the previous dict
entry, which is used when iterating over the dict.

The size + flag indicate the size of the name field, with the
7th bit being the IMMEDIATE flag.
```

The Direct Threaded Code model

Forths come in different flavors with regards to their execution model and Collapse OS is a Direct Threaded Code (DTC) forth.

This means that each word (except CODE words, which directly begin with native code) begins with a jump or call to a "word type" routine, which then does its thing, optionally using the words Parameter Field (PF, that is, the memory area following the word routine jump).

At the heart of all those word types is the "next" routine, defined at lblnext in all ports. This is the "beating heart" of our system. Whenever it's called, it increases IP by 2 and then jumps to the word referenced at IP-2. In other words, it "continues" along the path of the currently active stream of eXecution Tokens (XT). See "Executing a XT word" below.

These are the word types of Collapse OS:

native: nothing is done, native code is executed directly.

xt: eXecution Tokens. CALL lblxt which pushes IP to RS, pop PS (the PC pushed during the CALL) into IP and does "next".

cell: CALL lblcell, which is the same as lblnext on "regular" forths. On forths having a register assigned to TOS, we have to pop that value and "properly" push it back to PS.

does: CALL lbldoes, which pops the pushed addr, inc by 2 (this is the DOES data address) and push it back. Then, take the original addr value, dereference it (it's the address of DOES>), then jump to it as we would with any other word. The DOES> addr contains a regular word handler (generally a xt handler).

value: CALL lblvalue, which pops addr from PS, dereferences it, then push the value back to PS, then continue to lblnext.

Executing a XT (eXecution Tokens) word

At its core, executing a word is simply jumping to its wordref address. Then, we let the word do its things. Some words are special, but most of them are of the XT type, and that's their execution that we describe here.

First of all, at all time during execution, the Interpreter Pointer (IP) points to the word we're executing next.

When we execute a XT word, the first thing we do is push IP to

the Return Stack (RS). Therefore, RS' top of stack will contain a wordref to execute next, after we EXIT.

At the end of every XT word is an EXIT. This pops RS, sets IP to it, and continues.

A XT word is simply a list of word addresses, but not all those "tokens" are 2 bytes in length. Some tokens are special. For example, a reference to (n) will be followed by an extra 2 bytes number. It's the responsibility of the (n) word to advance IP by 2 extra bytes.

To be clear: It's not (n)'s word type that is special, it's a regular "native" word. It's the compilation of the (n) type, done in LITN, that is special. We manually compile a number constant at compilation time, which is what is expected in (n)'s implementation. Similar special things happen in (br), (?br) and (next).

For example, the word defined by ": FOO 345 EMIT ;" would have an 8 bytes PF: a 2b ref to (n), 2b with $0159, a 2b ref to EMIT and then a 2b ref to EXIT.

When executing this word, we first set IP to PF+2, then exec PF+0, that is, the (n) reference. (n), when executing, reads IP, pushes that value to PS, then advances IP by 2. This means that when we return to the "next" routine, IP points to PF+4, which next will execute. Before executing, IP is increased by 2, but it's the "not-increased" value (PF+4) that is executed, that is, EMIT. EMIT does its thing, doesn't touch IP, then returns to "next". We're still at PF+6, which then points to EXIT. EXIT pops RS into IP, which is the value that IP had before FOO was called. The "next" dance continues...

Stack management

In all supported arches, The Parameter Stack and Return Stack tops are tracked by a register assigned to this purpose. For example, in z80, it's SP and IX that do that. The value in those registers are referred to as PS Pointer (PSP) and RS Pointer (RSP).

The way those stacks are managed are arch-specific and opaque. Our only "meta-access" to stacks are through SCNT and RCNT which give us counts for each stacks.

Register roles

In the code, many registers have special meaning, and it's crucial to keep this in mind when reading or writing native

code. As written above, we reserve a register for PSP and RSP, but also for IP (Interpreter Pointer). You can see what register is reserved for what in the cpu-specific document of doc/code/.

With CPUs that have very few registers, we might end up using memory for IP, but it greatly impacts speed.

With CPUs that have a lot of registers, we can reserve some for stack elements. For example, on the z80, BC is reserved for PS' Top Of Stack. It makes all of the native code a bit weird because pushes and pops are no longer this clean, symmetrical set of operations, but gains (both in speed and binary size) are significant, especially with words that have a symmetrical stack signature (same number of stack elements before and after execution).

Stack underflow and overflow

When words pop and push from the stack, nothing stops them. If the stack goes out of bounds, bad things happen.

When a pop results in the stack pointer going out of bounds, it's a "stack underflow". We could check, in each native word, whether the stack is big enough to execute the word, but these checks are expensive.

Instead, what we do is that we check for stack underflow in the INTERPRET loop after each EXECUTE, through the word "STACK?". If SCNT < 0, it's a stack underflow.

Would a word like ": foo DROP 42 ;" trigger an underflow if executed on an empty PS? Well, no. That's the tradeoff. In exchange for simplicity and speed, we don't catch all underflow errors.

We don't check RS for underflow because the cost of the check is significant and its usefulness is dubious: if RS isn't tightly in control, we're screwed anyways, and that, well before we reach underflow.

Overflow condition happen when RS or PS outstep their bounds during a push. That condition is not checked because it's too expensive for what it's worth.

Overflow happens much less often than underflow. However, when it happens, it can means that your RS gets overwritten and will catastrophically crash your machine.

When you know you have a deep stack, or before you do fancy recursion, make sure you know the state of your stack well.

You can use .S for this.

System variables

There are some core variables in the core system that are referred to directly by their address in memory throughout the code. The place where they live is configurable by the SYSVARS constant in xcomp unit, but their relative offset is not.

SYSVARS occupy $60 bytes in memory in addition to driver memory, which typically follows SYSVARS.

This system is a bit fragile because every time we change those offsets, we have to be careful to adjust all system variables offsets, but thankfully, there aren't many system variables. Here's a list of them:

```
SYSVARS
+00       IOERR
+02       CURRENT
+04       HERE
+06       A register
+08       N register
+0a       NL characters
+0c       'LN<
+0e       'EMIT
+10       'KEY?
+12       CURWORD
+16       '(wnf)
+18       TO?
+19       RESERVED
+1c       IN(*
+1e       IN>
+20       INBUF
+60       DRIVERS
```

CURRENT points to the last dict entry.

HERE points to current write offset.

IN> and INBUF: See "Input Buffer" below.

LN< is called whenever the stream needs to be fed with a new line in IN(. Generally points to RDLN, but is overridden during LOAD.

CURWORD is a 4 bytes buffer containing a reference to the word last read with WORD. First byte is length, the next 2 are the address to the character string, and the last is a "don't read" flag.

```
IOERR: When an error happens during I/Os, drivers can set this
to indicate an error. For example, the AT28 driver sets this
when it detects write corruption. Has to be reset to zero man-
ually after an error.

NL is 2 bytes. NL> spits them, MSB first. If MSB is zero, it's
ignored. For example, $0d0a spits CR and then LF.

'KEY? and 'EMIT default to (key?) and (emit) but can be
overwritten to other routines.

The N register is like the A register but only accessible from
native code. You can then be sure that by using it you will not
break some high-level words.

'(wnf): see "Word Not Found override" in doc/usage.

DRIVERS section is reserved for recipe-specific drivers.

RESERVED sections are unused for now.
```

Initialization sequence

```
A Collapse OS binary is initialized by jumping to the first
address of its binary. This runs the "early initialization"
routine, written in native code. This does 2 things: initialize
PSP and RSP and then jump to BOOT.

Then, BOOT does this:

1. Initialize CURRENT and HERE.
2. Initialize system aliases and values in this way:
     EMIT -> (emit)
     KEY? -> (key?)
     NL -> CRLF
3. Call INIT, which is system-specific.  This usually
   initializes drivers.
4. Print "Collapse OS"
5. Call ABORT. See Execution Model above for the rest.
```

The "main" routine

```
The "main" routine is Collapse OS "soft boot" entry point. It
simply resets the input buffer (so that an ABORT/QUIT ran during
a LOAD brings us back to a usable system) and runs the
INTERPRET loop.

This routine is anonymous because it's not meant to be called
```

directly. You call it through QUIT. It's only there because QUIT is a low level word and "main" references high level words, so QUIT needs to do a forward jump to it (see doc/bootstrap for gory details).

Input buffer (INBUF)

As indicated above, the Input Buffer lives in SYSVARS and is $40 bytes in length (the value of LNSZ).

This buffer contains a stream of characters that, unlike regular strings, is not sized. It is also not terminated by any kind of character.

Words IN(and IN) indicate its bounds and IN> is a pointer (in absolute address) pointing to the current character being read.

This buffer will generally be filled by RDLN and then consumed by IN<. These words take care of not stepping out of bounds.

When you type characters in the prompt, it's RDLN that handles it. When you type CR (or LF), it stops reading and begins feeding IN<. If you type LNSZ characters without typing CR, an additional CR will be fed to IN< after INBUF has gone through.

In rare occasions, you need to know when you've reached the end of INBUF, for example in ED where some words read "rest of the buffer". In these cases, you can use IN<? instead of IN< which, when the end of INBUF is reached, instead of calling RDLN, will simply return 0.

1.4 Dictionary (`dict.txt`)

List of words defined in arch-specific boot code and core words.

Glossary

Stack notation: "<stack before> -- <stack after>". Rightmost is top of stack (TOS). For example, in "a b -- c d", b is TOS before, d is TOS after. "R:" means that the Return Stack is modified.

Some words have a variable stack signature, most often in pair with a flag. These are indicated with "?" to tell that the argument might not be there. For example, "-- n? f" means that "n" might or might not be there.

Some words consume contents from input and this is indicated by "x", "y" and "..." elements next to the word itself, not in the

stack signature. For example, ": x ... ;" means that the word ":" will consume an element "x" (which is usually explained in the description), followed by an arbitrary length of contents, which is ended by ";".

Word references (wordref): When we say we have a "word reference", it's a pointer to a word's entry point. That is, making native jump to the address contained in the wordref will execute the word.

For example, the address that "' DUP" puts on the stack is a word reference to DUP.

"*I*" in description indicates an IMMEDIATE word.
"*A*" in description indicates A register usage.

Symbols

Across words, different symbols are used in different contexts, but we try to be consistent in their use. Here's their definitions:

! - Store
@ - Fetch
$ - Initialize
^ - Arguments in their opposite order
< - Input
> - 1. Pointer in a buffer 2. Opposite of "<".
(- Lower boundary
) - Upper boundary
' - Address of
* - Word indirection (pointer to word)
? - "Is it ...?" or "do ... if flag"
[...] - Indicates immediateness

Placement of those symbols is often important. In I/O-related words for example, symbold to the left of the words refer to input and to the right, output. For "?", placement at the right refer to the first form, placement at the left refer to the second form.

System variables

See doc/usage for details. These ones have a "'" pair:

BLK> Currently selected Block.
CURRENT Address of the last word of the dictionary.
HERE Addr of next available space in dict
IN(Beginning of the input buffer.

```
IN>      Current pos in input buffer.

These ones are addresses and must be accessed with @ and !:

IOERR    Nonzero when an IO error occurred in some drivers.
NL       1 or 2 chars to spit during NL>, MSB first. If MSB is
         0, it's ignored.
LN<      Routine that feeds lines to the interpreter. Generally
         RDLN.
BLKDTY   Whether current block is dirty
```

Entry management

```
'? x      -- f Find x it in dict. If found, f=1. Otherwise, f=0.
' x       -- w Push addr of word x to w. If not found, aborts.
['] x     --   *I* Like "'", but spits the addr as a number
               literal. If not found, aborts.
FIND      sa sl -- w? f
          Find "sa sl" in dict. If found, w=wordref, f=1.
          Otherwise, f=0.
FORGET x  --   Rewind the dictionary (both CURRENT and HERE) up
               to x's previous entry.
```

Defining words

```
: x ... ;   --     Define a new word.
ALIAS x y   --     Define an alias y with a starting value of x
CREATE x    --     Create cell named x. Doesn't allocate.
CODE x      --     Define a new native word.
[COMPILE] x --     *I* Compile word x and write it to HERE.
                   IMMEDIATE words are *not* executed.
COMPILE x   --     *I* Meta compiles: write wordrefs that will
                   compile x when executed.
CONSTS ...  n --   Creates n new constants. See usage.txt Page 7.
VALUE x     n --   Creates cell x that when called pushes its
                   value.
VALUES ...  n --   Create a serie of n values. See usage.txt Page 7
DOER        --     See doc/usage
DOES>       --     See doc/usage
IMMEDIATE   --     Flag the latest defined word as immediate.
LITN        n --   Write number n as a literal.
```

Code generation

```
JMPi!  n a -- len  Write a native jump to n at address a
CALLi! n a -- len  Write a native call to n at address a
```

"len" is the length in bytes of the written binary contents.

Flow

Note that flow words can only be used in definitions. In the INTERPRET loop, they don't have the desired effect because each word from the input stream is executed immediately. In this context, branching doesn't work.

f IF A ELSE B THEN: if f is true, execute A, if false, execute B. ELSE is optional.
[IF] .. [THEN]: Meta-IF. Works outside definitions. No [ELSE].
BEGIN .. f UNTIL: if f is false, branch to BEGIN.
BEGIN .. AGAIN: Always branch to BEGIN.
n >R BEGIN .. NEXT: Loop n times.

```
(           --   *I* Comment. Ignore input until ")" is read.
\           --   *I* Line comment. Ignore input until EOL.
[           --   *I* Begin interpretative mode. In a definition,
                 execute words instead of compiling them.
]           --   End interpretative mode.
ABORT       --   Resets PS and RS and returns to interpreter.
ABORT" ..." --   *I* Compiles a ." followed by a ABORT.
EXECUTE     a -- Execute wordref at addr a
INTERPRET   --   Main interpret loop.
LEAVE       --   In a BEGIN..NEXT, exit at the next NEXT call.
QUIT        --   Reset RS, return to interpreter prompt.
```

Parameter Stack

```
DROP        a --
DUP         a -- a a
?DUP        DUP if a is nonzero
NIP         a b -- b
OVER        a b -- a b a
ROT         a b c -- b c a
ROT>        a b c -- c a b
SWAP        a b -- b a
TUCK        a b -- b a b
2DROP       a a --
2DUP        a b -- a b a b
```

Return Stack

```
>R          n -- R:n        Pops PS and push to RS
R>          R:n -- n        Pops RS and push to PS
R@          -- n            Copy RS TOS to PS
```

```
R~          R:n --          Drop RS TOS
```

Stacks meta

```
.S   --   *A* Prints stack information as well as the contents
          of PS.
SCNT -- n Size of PS in bytes
RCNT -- n Size of RS in bytes
```

Memory

```
@         a -- n        Set n to value at address a
!         n a --        Store n in address a
,         n --          Write n in HERE and advance it.
+!        n a --        Increase number at addr a by n.
[]=       a1 a2 u -- f Compare u bytes between a1 and a2. Returns
                        true if equal.
[C]?      c a u -- i    Look for c in the u bytes at addr a. If
                        found, return index i. Otherwise, i=-1.
C@        a -- c        Set c to byte at address a
C@+       a -- a+1 c    Fetch c from a and inc a.
C!        c a --        Store byte c in address a
C!+       c a -- a+1    Store byte c in a and inc a.
C,        b --          Write byte b in HERE and advance it.
nC,       n --          Parse next n words and write them as
                        bytes.
ALLOT     n --          Move HERE by n bytes.
ALLOT0    n --          *A* ALLOT and fill with zero.
FILL      a n b --      *A* Fill n bytes at addr a with val b.
L,        n --          Write n in little-endian regardless of
                        native endianess (L=LSB first)
M,        n --          Write n in big-endian regardless of
                        native endianess (M=MSB first)
MOVE      a1 a2 u --    *A* Copy u bytes from a1 to a2, starting
                        with a1, going up.
MOVE,     a u --        *A* Copy u bytes from a to HERE.
TO        --            Next VALUE call will be in write mode.
                        See doc/usage.
```

A register

```
>A        n -- A:n
A>        A:n -- n A:n
R>A       R:n -- A:n
A>R       A:n -- R:n A:n
A+        A:n -- A:n+1
A-        A:n -- A:n-1
```

```
AC@      A:a -- c A:a
AC!      c A:a -- A:a
AC@+     A:a -- c A:a+1
AC!+     c A:a -- A:a+1
```

Arithmetic / Bits

```
+           a b -- a+b
-           a b -- a-b
-^          a b -- b-a
*           a b -- a*b
/           a b -- a/b
<>          n1 n2 -- l h Sort n1 and n2, highest on TOS.
<<          n -- n       Shift n left by one bit
<<8         n -- n       Shift n left by 8 bits
>>          n -- n       Shift n right by one bit
>>8         n -- n       Shift n right by 8 bit
L|M         n -- lsb msb Split n word in 2 bytes, MSB on TOS
1+          n -- n+1
1-          n -- n-1
MAX         n1 n2 -- hi
MIN         n1 n2 -- lo
MOD         a b -- a%b
/MOD        a b -- r q   r:remainder q:quotient
AND         a b -- a&b
OR          a b -- a|b
XOR         a b -- a^b
LSHIFT      n u -- n     Shift n left by u bits
RSHIFT      n u -- n     Shift n right by u bits
```

Logic

```
=    n1 n2 -- f Push true if n1 == n2
<    n1 n2 -- f Push true if n1 < n2
>    n1 n2 -- f Push true if n1 > n2
>=   n1 n2 -- f Push true if n1 >= n2
<=   n1 n2 -- f Push true if n1 <= n2
0<   n -- f     Push true if n-as-signed is negative
0>=  n -- f     Push true if n-as-signed is positive
NOT  f -- f     Push the logical opposite of f. Always 0 or 1.
```

Strings and lines

```
See doc/usage for the concepts of strings and lines.

S" ..."   --     Read following characters and write to HERE as
                 a string literal.
```

```
LNLEN     a -- n Return length of line at a, the line ending at
                 the last visible char of it.
S=        sa1 sl1 sa2 sl2 -- f
          Returns whether string s1 == s2.
```

Number formatting

```
.         n --          Print n in its decimal form
.x        n --          Print n's LSB in hex form. Always 2
                        characters.
.X        n --          Print n in hex form. Always 4 characters.
                        Numbers are never considered negative.
                        "-1 .X" --> ffff
FMTD      n a -- sa sl  Formats n as decimal in memory and return
                        its string as "sa sl".
FMTx      n a -- sa sl  Formats n's LSB in hex form.
FMTX      n a -- sa sl  Formats n in hex form.
```

I/O

```
," ..." --         Write ... to HERE
." ..." --         *I* Compiles string literal ... followed by a
                   call to STYPE.
CURWORD -- sa sl   Yield the last read word (see WORD).
EMIT    c --       Spit char c to output stream
EMITLN  a --       *A* EMIT line at addr a
IN<     -- c       Read one char from buffered input, if end of
                   input is reached, read new line.
IN<?    -- c-or-0  Read from buffered input if its end hasn't
                   been reached, 0 otherwise.
IN(     -- a       Beginning of input buffer.
IN)     -- a       End of the input buffer, exclusive.
IN$     --         Flush input buffer
KEY?    -- c? f    Polls the keyboard for a key. If a key is
                   pressed, f is true and c is the char. Other-
                   wise, f is false and c is *not* on the stack.
KEY     -- c       Get char c from direct input.
NL>     --         Emit newline
PARSE   sa sl -- n? f *A*
        Parses string s as a number and push the result in n if
        it can be parsed, with f=1. Otherwise, push f=0.
PC!     c a --     Spit c to port a
PC@     a -- c     Fetch c from port a
SPC>    --         Emit space character
STYPE   sa sl --   *A* EMIT all chars of string.
WAITW   sa sl --   Call WORD until we get the same string as
                   sa sl.
WORD    -- sa sl   Read one word from buffered input and push it.
                   That word is a string (begins with a length
```

```
                byte).
WORD!   sa sl --  The next WORD call will not read from input
                  and yield this string instead.

These ASCII consts are defined:
EOT BS CR LF SPC

KEY? and EMIT are aliases to (key?) and (emit) (see doc/drivers)
KEY is a loop over KEY?.

NL> spits CRLF by default, but can be configured to spit an
alternate newline char. See impl.txt Page 21.
```

BLK subsystem (see doc/blk)

```
\S      --        Interrupts LOAD of current block.
BLK(    -- a      Beginning addr of blk buf.
BLK)    -- a      Ending addr of blk buf.
BLK@    n --      *A* Read block n into buffer and make n active.
BLK!    --        *A* Write currently active block, if dirty.
COPY    s d --    *A* Copy contents of s block to d block.
FLUSH   --        Write current block to disk if dirty and inval-
                  idates current block cache.
LIST    n --      *A* Prints the contents of the block n on
                  screen in the form of 16 lines of 64 columns.
LOAD    n --      *A* Interprets Forth code from block n
LOADR   n1 n2 --  *A* Load block range between n1 and n2,
                  inclusive.
WIPE    --        *A* Empties current block

Note: Most BLK words don't actually use the A register them-
selves, but we want to allow BLK drivers to make usage of it,
so you *should* guard yourself again A changes when using those.
```

RX/TX subsystem (see doc/rxtx)

```
RX<?    -- c? f   If a char is available on RX, return it in c
                  with f=1. Otherwise, f=0.
RX<     -- c      Block until a char is available in RX.
RX<<    --        Consume RX<? and drop result until there's
                  nothing to be received.
RX[     --        Replace KEY? with RX<?.
]RX     --        Put back the old KEY? handler.
TX>     c --      Spit c to TX, blocking until it can do it.
TX[     --        Replace EMIT with TX>.
]TX     --        Put back the old EMIT handler.
```

Other

```
BOOT    --       Boot back to a fresh system.
CRC16   c b -- c Computes byte b into c, a 16-bit CRC with a
                 $1021 polynomial (XMODEM CRC).
DUMP    n a --   *A* Prints n bytes at addr a in a hexdump
                 format. Prints in chunks of 8 bytes. Doesn't do
                 partial lines. Output is designed to fit in 32
                 columns.
NOOP    --       Do nothing.
TICKS   n --     Wait for approximately n*100 microseconds.
                 Don't use with n=0.
```

Loaders

```
These words load the related application from blocks:

ARCHM  Arch-specific loader words and macros
ED     Block Editor
VE     Visual Editor
ME     Memory Editor
RSH    Remote shell and XMODEM implementation
XCOMP  Cross-compilation tools
```

Kernel internals

```
Some words from the kernel are designed to be internal but
ended up being used in "userland". Let's document them:

_bchk   n -- n     Checks whether n is a valid 8-bit signed
                   branching offset, that is, in the range -128
                   to 127. If not, abort with "br ovfl".
```

1.5 The BLK subsystem (`blk.txt`)

```
Disk blocks are Collapse OS' main access to permanent storage.
The system is exceedingly simple: blocks are contiguous
chunks of 1024 bytes each living on some permanent media such
as floppy disks or SD cards. They are mostly used for text,
either informational or source code, which is organized into
16 lines of 64 characters each.

Blocks are referred to by number, 0-indexed. They are read
through BLK@ and written through BLK!. When a block is read,
its 1024 bytes content is copied to an in-memory buffer
starting at BLK( and ending at BLK). Those read/write
operations are often implicit. For example, LIST calls BLK@.

When a word modifies the buffer, it sets the buffer as dirty
```

by calling BLK!!. BLK@ checks, before it reads its buffer, whether the current buffer is dirty and implicitly calls BLK! when it is.

The index of the block currently in memory is kept in BLK>.

Most blocks contain code. That code can be interpreted through LOAD. LOAD operations cannot be nested, that is, you can't call LOAD from a block or you can't call a word that calls LOAD from a block.

Using blocks

You will typically interact in 2 main ways with blocks: LISTing them or LOADing them. You LIST a block with a block number argument. For example, "0 LIST" spits the master index block.

LOAD interprets the specified block as if you typed it. You also invoke it with a block number. "42 LOAD" reads block 42 in the buffer and, for each of the 16 lines, interprets it as if it was typed in the input buffer.

You can see the whole list of words supplied by the BLK sub-system at doc/dict.

Exploring blocks

Block 0 in Collapse OS is a text block describing the whole contents in all blocks, organized in sections. Sections are typically 5, 10 or 20 blocks in size.

The first line of each block is often a comment describing the contents of the block. To take advantage of this, we have the INDEX word which prints the first line of each block in a range.

So, for example, if you see in the master index that Collapse OS core words spans from B210 to B229 and you want to quickly find a word in it, you'd run "210 229 INDEX".

LOADing applications

The first block of each section (a section often contains an application) will typically contain loading instructions in comments. These generally involve "application loaders", words that LOAD the appropriate blocks for the application to be loaded in memory.

The BLK subsystem contains these loader words:

```
ED     Text editor (doc/ed)
VE     Visual text editor (doc/ed)
ME     Memory editor (doc/me)
XCOMP  Cross-compilation tools (doc/cross)
ARCHM  Arch-specific macros, constants and loaders (doc/arch)
RXTX   Serial communication tools (doc/rxtx)
```

For example, it has the "VE" word which loads VE. Therefore, on a freshly booted system, if you want to run VE, simply type "VE". IF VE isn't loaded yet, it will LOAD. If it is loaded, it will run.

Some of these words, such as ARCHM are "doors" opening the way to further loader words, such as assemblers (doc/asm/intro).

How blocks are organized

Organization of contiguous blocks is an ongoing challenge and Collapse OS' blocks are never as tidy as they should, but we try to strive towards a few goals:

1. B0 is a textual master index of blocks. LIST it to see the whole contents of the blkfs.
2. B1-B199 are for applications.
4. B200-B299 are for arch-independent cross-compiled code, including xcomp tools and subsystems.
5. B300+ is for arch-specific code.

In the POSIX package of Collapse OS, arch-specific code is kept in separate "blk.fs" files so that depending on the arch being built, the content of B300+ varies.

B300 is always an "arch-specific" master index and B301 is always the "macros" block for this architecture (the block you want to load before XCOMPC during bootstrapping). This block defines all subsequent loader words for this archtecture.

When collapse comes and you want to build your final Collapse OS media, you'll probably want to keep all arch-specific contents at once. You will then need to organize those blocks yourself in the way you see fit.

The BLK subsystem enables disk access and provides all disk-related words (LOAD, LIST, FLUSH, etc.).

Including the BLK subsystem in a kernel

```
Before assembling, this requires 3 words:

BLK_MEM: where the 1024 bytes block buffer will live as well as
         BLK variables. The total size used is $409 bytes.

(blk@) blkno dest --  Reads blkno into dest (almost always BLK(
                      is passed there).
(blk!) blkno dest --  Write contents of buffer at dest into
                      blkno.

Then, you can call BLKSUB in your xcomp unit.

These are the variables defined in BLKSUB:

BLK>    Currently active block number
BLKDTY  Whether current block is dirty (needs to be saved on
        FLUSH). Nonzero means dirty.
BLKIN>  Upon LOAD, old IN> value is saved there so that when
        LOAD is finished, we can restore it and continue
        interpreting INBUF where we were.
BLK(    Address of the BLK buffer.
BLK)    Address where the BLK buffer ends.

Some subsystems provide an implementation to the BLK protocol:

* SD card subsystem (doc/sdcard)
```

1.6 RX/TX subsystem (`rxtx.txt`)

```
If your machine has a serial device (often a RS-232 device),
adding the RX/TX subsystem to your kernel can open interesting
possibilities. That subsystem is added through RXTXSUB during
xcomp and requires the following words to be defined by drivers:

RX<?  -- c? f   Check if a character has been received by the
                device. If it has, f=1 and c is the received
                char. Otherwise, f=0.
TX>   c --      Transmit character c to the device.

From these words, The RX/TX subsystem them defines a handful of
extra words, which are listed in the "RX/TX" section of
doc/dict.txt Page 27.

Some of those words deserve special mention, such as TX[ ]TX and
RX[ ]RX. What those words do is that they temporarily replace
EMIT and KEY? (respectively) to facilitate some processing.

Example: let's say that you have a number on PS that you'd like
to spit, hex-formatted, to TX. What do you do? Re-implement .X
and replace EMIT with TX>? Well, you could so that. Or... you
could do "TX[ .X ]TX".
```

Other example: You want to give total control of your computer to the RX/TX link. What do you do? "TX[RX[". Your keyboard and screen are now unresponsive, RX/TX has control now. Want to go back? "]TX]RX" (from the serial link, of course).

RX/TX tools

Collapse OS also has tools at B10 that you can load at runtime. The loader words for these tools is "RXTX". These tools include:

* A remote shell
* A way to upload binary contents to a Collapse OS remote.
* A XMODEM implementation.
* A blksrv client (see doc/blksrv).

Remote shell

You can control a remote system from Collapse OS using the "rsh" (--) word.

When you run "rsh", it will repeatedly poll RX<? and emit whatever is coming from there and at the same time, poll KEY? and push whatever key you type to TX>.

You can stop the remote shell by typing CTRL+D (ASCII 4).

Uploading data

You can also upload data to your remote if it runs Collapse OS. Use the "rupload" word. It takes a local address, a remote address and a byte count. For example, "$8000 $a000 42" copies' 42 bytes from the local machine's $8000 address and uploads it to the remote's $a000 address.

When you execute the word, it's doing to remotely (and tempo-rarily) define helper words and that's going to result in some gibberish on the screen. Then, it's going to start spitting "." characters, one per byte uploaded. After that, it's going to spit two checksum: one for the data received by the remote and one for the data sent locally. If they match, you're all good.

XMODEM

XMODEM is a simple protocol for reliable data transfer over a serial line. The reference document for it is titled:

```
XMODEM/YMODEM PROTOCOL REFERENCE, A compendium of documents
describing the XMODEM and YMODEM File Transfer Protocols

By Chuck Forsberg

On POSIX systems, the tool generally used for it is "lrzsz".
To use with Collapse OS, you'll want to use the rx/sx versions
of it.

Words defined in RXTX that implement the XMODEM protocol are:

MEM>TX  a u --   Send u bytes to TX from addr a.
BLK>TX  b1 b2 -- Send contents of block range b1-b2
RX>MEM  a --     Receive packets into a until EOT
RX>BLK  --       Receive packets into blocks, starting with
                 currently active BLK>, increasing by one when-
                 ever a block is filled.

As it is now, the XMODEM implementation is a bit fragile, but
all the important parts are there. They just need to be
solidified.
```

blksrv client

```
RXTX tools also have words to fetch blocks from and push blocks
to a blk server (see doc/blksrv). These words are:

blksrv<  blk --  Receive remote "blk" from and put it in
                 currently active block.
blksrv>  blk --  Send currently active block to remote "blk".
```

1.7 Block Server (blksrv) (`blksrv.txt`)

```
In POSIX tools/, there's a program called "blksrv" which is a
"block server", that is, a program that sends and receives
blocks to/from a serial link.

Admittedly, the goal of this program is very much pre-collapse,
but we can imagine some post-collapse uses for such a setup too:
I use it to facilitate the synchronisation between my retro
machines and my modern environment.

At some point during Collapse OS' development, I began
developing Collapse OS from within Collapse OS from a retro
machine. It worked well, but because I published Collapse OS on
something we called the "internet", I needed to fetch my work
from my retro machine and into my modern machine.
```

I tended to do this only once in a while, and even worse, sometimes my modern machine also had some changes on it. Merging was complicated.

Also, because I tended to not remember exactly what I had modified recently, I tended to transfer the whole blkfs each time (to be sure), which is a bit time consuming on a 9600 bauds link.

The idea with the Block Server is that the retro machine is "in charge". It controls what to push/pull and when, so the modern machine is a slave to it. Because it's easy to do so while working on the retro machine, then I can do it more often and avoid tricky merging problems down the road.

How it works

Very simple. There are 2 commands: Get and Put. The client (the retro machine) initiates a command by sending a 'G' or a 'P' followed by a formatted 16-bit hex number (this means 4 chars).

If it's a 'G' command, the server reads the asked block from its local file and spits its 1024 bytes as-is. Then, it spits the checksum of those 1024 bytes as a hex-formatted 16-bit number.

The checksum is a simple sum of all bytes.

If it's a 'P' command, then it's the client that sends its block in the same fashion. If the checksum matches, it writes it to its local file.

The client and the server don't tell each other of checksum failures or anything. You're supposed to see those because you have access to both ends locally. At least that's the idea.

You remember when I talked about spitting contents as-is? You're thinking that it could possibly cause transmission problems in the ASCII control chars range, don't you? Well yes, it can, but this system is designed to send text blocks. Those blocks don't contain ASCII control chars.

So, don't transmit binary data through this system or you're going to have a bad time.

1.8 Understanding Collapse OS (`grok.txt`)

Collapse OS is designed for maximum "grokkability", that is, the possibility for a single person to understand the system in its entirety.

To this end, it helps that there is so little code. It is astounding to realize that there's a whole self-hosting system in those few thousand lines! Surely you can simply read them and grok it all, no?

No.

This code is extremely dense and can't be easily understood by only looking at the code casually like you would with another project. This reading has to be guided. This is, I hope, the guide.

After following this guide, you should normally be able to read any part of the code and understand its place in the whole.

1. The core dictionary

The first step you have to take is to know all the words from the core dictionary (doc/dict). After having gone through doc/usage, take the time to understand every word from that dictionary. Those words are used everywhere.

Toy with them, try them out. Master them.

2. Applications

Code from applications (VE, ME etc.) are the easiest to grok. The pattern is always the same: small words from which bigger things are made.

For applications, it's often better to begin from the end. The last defined word (for example, "VE" in VE) is often the main loop. Because you have an idea of what the app is supposed to do, you can estimate the behavior from reading the main loop alone. You don't have understanding, but you can begin guessing.

From there, I suggest drilling down each word of the main loop until you hit "rock bottom" (words from the core dictionary).

That will make you go through all words of the app. Once you're finished, you understand the app. doc/code/intro might help.

Don't be hasty. Forth code reads slower than code from other languages. When you drill down, accept partial understanding until you hit "rock bottom". Only then can you begin perfectly understanding words. From there, you go back up, with solid understanding.

3. Assemblers

Gaining ability to understand application code will put your mind in the right mood. You're already well set for greater understanding.

The next step is mastering assemblers. You must begin, of course, by knowing your CPU. It is assumed here that you familiarized yourself with its documentation and know its opcodes well. Then, continue with doc/asm.

Play with assemblers, create your own native words. Crash your machine. Try creating macros. Be well aware of register roles for your target CPU (doc/code/).

4. Cross compilation

Cross compilation is seriously mind bending. Take the time to understand it, understand what the tooling does. See doc/cross.

Begin with trying to cross compile native code. This is easier to understand because all we have to think about is XORG. Maybe try to compile a dummy bootloader. That will make you practice xcomp.

5. Boot code

You can now begin looking at your arch's boot code, which is covered by doc/bootstrap as well as the corresponding page in doc/code/.

Because it's native code, it's relatively straightforward to read. The part that gains by being read slowly and carefully is the first part, before words begin. It's initialization and core routines.

You don't get more fundamental than this. This is the heart of the whole thing. Take your time, it's one, maybe two blocks at most, so there's not a lot to read.

Its important that this code is as fast as possible so all tricks in the books (well, those that I know of!) are used, hindering readability. You have to take your time.

Then come native words, which are more straightforward. It's still worth it to read them to train yourself to understand native code that is sometimes complex (/MOD * []= FIND).

6. Core words

Core words are mostly written in regular Forth, so most of it is grokkable easily. But not all of it.

To understand core words, and thus the kernel, as a whole, you need to understand XCOMPC (covered in doc/bootstrap) which is dark magic. It's so twisted that even though I master it, my mind still treats it as a black box and only "opens" the box when there's a problem with it.

Fortunately, it's only five blocks, so its not so bad. Give it a look, read doc/bootstrap.

If you're looking at a way to properly understand its purpose, I would suggest writing a new set of core words yourself, from scratch, using XCOMPC and your boot words. From boot words, you can go quite quckly to some kind of prompt, so it could be a fun exercise and will let you experience XCOMPC in a minimal environment.

Quick tip: begin with BOOT. That's all you really need to have (with lblboot set). Your first step could be to simply (emit) and BYE. Then, you grow from there.

7. Driver code

At this point, you can consider that you know Collapse OS already. However, because it needs to run on some kind of machine at some point, you'll also want to understand your drivers.

Driver code has to be the hardest to read. It is often deeply tied to the way hardware is organized. For compactness reasons, we keep comments terse, and on top of that, we can't have complete hardware specifications in Collapse OS itself. Therefore, it is highly recommended to have technical specifications handy when trying to read this code.

In the hardware documentation ("hw" folder), we try to document hardware specs directly related to driver code, but this kind of documentation is always going to be incomplete.

So, hum, what can I say. Begin with the tech spec of your hardware. Everthing flows from that. Once you know it, the code should become clear.

Conclusion

That's it! You should now be able to look at any part of Collapse OS and understand where it fits, how to improve it, how to make it your own. Congratulations!

1.9 Design considerations (`design.txt`)

The primary goals of Collapse OS are to:

* Run on minimal and improvised machines.
* Interface through improvised means.
* Edit text and binary contents.
* Compile assembler source for a wide range of MCUs and CPUs.
* Read and write from a wide range of storage devices.
* Assemble itself and deploy to another machine.
* Achieve this with as little external means as possible, for example, internet or a functional global supply chain.

It follows, from these goals, the following priorities in code qualities:

* Simplicity
* Compactness
* Speed

It is paramount that this project stays understandable in its whole by a single person so that it can be adapted to as many contexts as possible.

However, it also needs to stay compact so that it can run on as many machines as possible, however small. So far, there's a few hard limits that have been identified:

* The ROM for the RC2014 port with SD card has to fit in 8K.
* Self-hosting must stay possible on the Sega Master System with its 8K of RAM.

If these limits are respected, we consider the compactness criteria met.

Lastly, Collapse OS should be usable on its target machines. If it's slow, it's less usable. Efforts should be made towards this as long as it doesn't break the two other constraints.

That last part is hard to remember. When going through the code, there are some obvious inefficiencies and when we look at them, the mind immediately thinks of ways to smooth them out. But those ways add complexity. If the solution you're thinking about adds significant complexity compared to the speedup, then it has likely been considered already and dismissed.

The idea behind this is that those easy pickings can easily be picked by the post-collapse user. Simplicity is paramount and, besides, why not leave this pleasure to them, those poor souls?

1.10 Editing text (ed.txt)

Collapse OS has 2 levels of text editing capabilities: command-based editing and visual editing. This 2-fold application is located at B20.

The command-based editor is a "traditional" Forth text editor as described in Starting Forth by Leo Brodie. This editor can be loaded with "ED".

The visual editor is a full-blown application that takes over the interpreter loop with its own key interpreter and takes over the whole screen using the Grid subsystem. We call this editor the "Visual Text Editor" and can be loaded with "VE" once loaded it can be ran with "VE".

When available, the Visual editor is almost always preferable to the command-line editor. It's much more usable. We have the command line editor around because not all machines can use the Grid subsystem. For example, a machine with only a serial console can't.

Command-line editor

The command-line editor augments the built-in word LIST with words to modify the block currently being loaded. Block saving happens automatically: Whenever you load a new block, the old block, if changed, is saved to disk first. You can force that with FLUSH.

Editing works around 3 core concepts: cursor, insert buffer (IBUF), find buffer (FBUF).

The cursor is simply the character index in the 64x16 grid. The word T allows you to select a line. For example, "3 T" selects the 3rd line. It then prints the selected line with a "^" character to show your position on it. After a T, that "^" will always be at the beginning of the line.

You can insert text at the current position with "I". For example, "I foo" inserts "foo" at cursor. Text to the right of it is shifted right. Any content above 64 chars is lost.

You can "put" a new line with "P". "P foo" will insert a new line under the cursor and place "foo" on it. The last line of

```
the block is lost. "U" does the same thing, but on the line
above the cursor.

Inserting anything also copies the inserted content into IBUF.
Whenever an inserting command is used with no content (you still
have to type the whitespace after the word though), what is in-
serted is the content of IBUF.

This is all well and good, but a bit more granularity would be
nice, right? What if you want to insert at a specific position
in the line? Enter FBUF.

"F foo" finds the next occurrence of "foo" in the block and
places the cursor in front of it. It then spits the current line
in the same way "T" does.

It's with this command that you achieve granularity. This allows
you to insert at arbitrary places in the block. You can also
delete contents with this, using "E". "E" deletes the last found
contents. So, after you've done "F foo" and found "foo", running
"E" will delete "foo", shifting the rest of the line left.

List of commands:

T ( n -- ): select line n for editing.
P xxx: put typed IBUF on selected line.
U xxx: insert typed IBUF on selected line.
F xxx: find typed FBUF in block, starting from current
       position+1. If not found, don't move.
I xxx: insert typed IBUF at cursor.
Y: Copy n characters after cursor into IBUF, n being length of
   FBUF.
X ( n -- ): Delete X chars after cursor and place in IBUF.
E: Run X with n = length of FBUF.
L: LIST current block.
N: Show next block.
B: Show previous block.
```

Visual Text Editor

```
This editor, unlike the command-line editor, is grid-based
instead of being command-based. It requires the Grid subsystem
(see doc/grid)

It is loaded with "VE" and invoked with "VE". Note that this
also fully loads the command-line editor.

This editor uses 19 lines. The top line is the status line and
it's followed by 2 lines showing the contents of IBUF and
FBUF. There are then 16 contents lines. The contents shown is
```

that of the currently selected block.

The status line displays the active block number, then the "modifier" and then the cursor position. When the block is dirty, an "*" is displayed next. At the right corner, a mode letter can appear. 'R' for replace, 'I' for insert, 'F' for find.

All keystrokes are directly interpreted by VE and have the effect described below.

Pressing a 0-9 digit accumulates that digit into what is named the "modifier". That modifier affects the behavior of many keystrokes described below. The modifier starts at zero, but most commands interpret a zero as a 1 so that they can have an effect.

'G' selects the block specified by the modifier as the current block. Any change made to the previously selected block is saved beforehand.

'[' and ']' advances the selected block by "modifier".

'h' and 'l' move the cursor by "modifier" characters. 'j' and 'k', by lines. 'g' moves to "modifier" line.

'H' goes to the beginning of the line.

'L' goes to the char following the last non-whitespace char. If everything following the cursor is whitespace, goes to the end of the line.

'w' moves forward by "modifier" words. 'b' moves backward. 'W' moves to end-of-word.

'I', 'F', 'Y', 'X' and 'E' invoke the corresponding command from command-based editor.

'n' finds the next occurrence of FBUF.

'N' is the same as 'n', but if it doesn't find anything, it continues the search in the, at most, "modifier" next blocks.

'C' is the "change" command. In essence, it behaves like "E" followed by "I", with this important difference: unlike "E", it doesn't place the selection in IBUF. This allows you to reuse what you previously had there. Very useful for repetitive search and replace.

'o' inserts a blank line after the cursor. 'O', before.

'D' deletes "modifier" lines at the cursor. The first of those

```
lines is copied to IBUF.

'R' goes into replace mode at current cursor position.
Following keystrokes replace current character and advance
cursor. Press return to return to normal mode.

'f' puts the contents of your previous cursor movement into
FBUF. If that movement was a forward movement, it brings the
cursor back where it was. This allows for an efficient combi-
nation of movements and 'E'. For example, if you want to delete
the next word, you type 'w', then 'f', then check your FBUF to
be sure, then press 'E'.

*** 'f' is the key you're looking for. It enables all copy/
pasting capabilities in VE. Try it.

't' and 'm' are for bookmarks. There are 10 bookmarks, select-
able through the modifier. 'm' saves current position and block,
't' recalls it.

*** If a recall happens on the same line, it is 'f' compatible,
that is, you can use m/t as a text selection tool.

'@' re-reads current block even if it's dirty, thus undoing
recent changes.

'!' writes the current block to disk.

'&' WIPE's the current block but doesn't save it. You can still
undo a mistyping with '@'.

'q' quits VE
```

Cheat sheet

```
h left
l right
j down
k up
H beginning of line
L end of line
w next word
W next end-of-word
b previous word
F find
n next match
N next match across blocks
f select
I insert
X delete char
```

```
E delete selection
Y copy selection to IBUF
C change selection
o insert line after
O insert line before
D delete line
R replace mode
G load block
[ previous block
] next block
@ re-read block
! write block
& wipe block
m mark
t goto mark
q quit
```

Tight screens

Blocks being 64 characters wide, using the Visual editor on a screen that is not 64 characters wide is a bit less convenient, but very possible.

When VE is in a "tight screen" situation, it behaves differently: no gutter, no line number. It displays as much of the "left" part of the block as it can, but truncate every line.

The right part is still accessible, however. If the cursor moves to a part of the block that is invisible, VE will "slide" right so that the cursor is shown. It will indicate its "slid" mode by adding a ">" next to the cursor address in the status bar.

To slide back left, simply move the cursor to the invisible part of the left half of the block.

Other than that, VE works the same.

1.11 Memory Editor (`me.txt`)

The Memory Editor at B35, which can be loaded with "ME" and then invoked with "ME" is a Grid application (doc/grid) allowing you to explore and modify binary contents in the memory.

Such applications are often called "hex editors" in the modern world.

The application uses the whole screen and has 3 main sections: the status bar, the hex display and the ASCII display.

The status bar has 3 fields:

```
A: Base address begin displayed. The top left cell of the
   display is the value at that address.
C: Cursor position. This is the address relative to the Base
   address and represents where the cursor presently is.
S: Stack. This displays PS exactly like the ".S" word does.
   Because some actions affect the stack, it's useful to see it
   in real time.

The 2 other sections display the contents of the memory
following the Base Address, with the left part being mirrored by
the right part.

While running, ME repeatedly waits for single keystrokes and
performs the associated action, if any. Unlike VE (doc/ed), it
has no concept of accumulator that affects all commands.

The Base address is always divisible by 16.

Press q to quit.
```

Tight mode

```
The regular mode of ME requires 60 columns and shows 16 bytes
per line. When the screen doesn't have enough columns, it falls
back to a 8 bytes per line mode, requiring 32 columns.
```

Navigating

```
You can increase/decrease the Base Address in a page-by page
fashion with [ and ].

You can do it in a line-by-line fashion with J and K.

The Cursor determines where most actions will take place and the
cursor can be moved with h/l (left/right) and j/k (down/up),
like in VE. There is no accumulator though, single mode only.

You can jump to a specific address with G. When pressing G, you
will be prompted for a hexadecimal address. You can type 4
characters or less-than-4-plus-return.

When you do that, the base address is changed to what you've
specified. If it's not divisible by 16, the Cursor is moved to
make up the difference.

When jumping to a new address, ME checks whether that address is
in the currently visible page. If it is, only the cursor moves,
not the base address.
```

Playing with the stack

You can read the 16-bit cell number at Cursor and place it in the stack with @. You can write from Stack to Cursor with !.

You can put the current cursor position on the Stack with m.

You can jump to the address currently on the top of the Stack with g.

You can "follow" the Cursor, that is, jump to the address where the cursor currently points with f.

You can "enter" the Cursor, that is, save the current Cursor position to stack and then "follow". When you want to come back, press g.

Modifying memory

When you press R, you are in "replace" mode. As long as you enter valid hexadecimal pairs, they will be written to the Cursor and the Cursor will advance. As soon as you enter an invalid value or Enter, the replace mode stops.

You can also press A to toggle the ASCII mode. In this mode, the Cursor will keep its position, but will go on the right side of the screen.

When you go in "replace" mode while also in ASCII mode, you can enter ASCII values directly. Enter to stop.

1.12 Disassemblers (`dis.txt`)

Some architectures (6502 and 6809 for now) include a disassembler in addition to an assembler. The loader word follows the same pattern as the assemblers: it lives in ARCHM and ends with "D". Examples: 6502D 6809D

All disassemblers require the corresponding assembler to be loaded first.

Once loaded, they supply the word "dis (addr --)" which prints DISCNT lines (by default 20) of disassembled memory starting at "addr". DISCNT is a VALUE, so you can change it with TO.

Disassembly formatting tries to stay close to the "manufacturer language" rather than the assembler language. For example, the 6809 disassembly of "$42 X+N ADDA," is "ADDA 42,X".

We lose symmetry with assembler, but we gain general readability. During assembly, we are constrained by Forth semantics, but with disassembly, we aren't. We can afford to make ourselves closer to manufacturer language.

Numbers are always hexadecimal and width matter. "2a" means that an 8b literal was extracted from the opcode and "002a" means that a 16b literal was.

Some opcodes are invalid, so you'll get "???" outputs. From the first of these that you get, you can consider the rest of the output to be garbage because opcodes are "out of sync".

1.13 Emulators (`emul.txt`)

Some architectures (6502 and 6809 for now) include an emulator, allowing you to run foreign code on any host. The loader word follows the same pattern as the assemblers: it lives in ARCHM and ends with "E". Examples: 6502E 6809E

All emulators require the corresponding disassembler to be loaded first.

Once loaded, the following words are supplied, regardless of the arch:

```
cpu.  --   Print CPU register values
run1  --   Run a single instruction
runN  n -- Run n instructions
run   --   Run until CPU is halted
```

You can define breakpoints through the "'BRK?" VALUE, which points to a word with a (-- f) signature. If it returns nonzero, the run loop halts. If 'BRK? is zero (default value), there is no breakpoint.

Breakpoints are checked after having run an op. This means that, after your run is interrupted by a breakpoint condition, you don't have to disable breakpoints to resume, resuming will always run at least one op.

There is also the "VERBOSE" value, defaulting to 0. If set to nonzero, every run step will also execute "cpu.".

"MEM" points to the area of memory allocated for the emulated machine. Usually, 2048 bytes are allocated there. Each emulated memory operation are done relative to "MEM".

Each emulator have an "initializer" word which initializes registers that need it (PC, DP, etc.). It has the name of the

```
loader word + $. Examples: 6502E$ 6809E$

6502 init: PC: $200
6809 init: PC: $100 DP: 0

"run" words don't initialize CPU registers.

Each emulator supply pointer words for each register. For
example, 'D in 6809 would point to a 16b value, 'A to the exact
same space (but we have to use it as a byte), etc. 16b registers
are in target byte order, which means that 'D points to a big
endian value regardless of the host architecture.

6502 registers: A X Y S P PC
6809 registers: D X Y U S PC A B CC DP
```

Usage

```
These emulators are designed to debug small pieces of code. You
could use them to emulate complete machines, but you'd have to
develop quite a bit of tooling around it.

For example, if you just want to run a few ops and see how it
goes, you could do something like this:

6502E$ 1 TO VERBOSE
HERE MEM $200 + 'HERE !
$02 # LDA, TAY, $12 # ADC, 1 <> SBC, BRK,
'HERE !
run

If you want to debug Collapse OS ports within context, things
get a bit more complex. One option is to emulate a full COS
binary. You can do it by identifying the address of the part
you want to debug and use BRK? with a word that checks the
value of PC. Then, you single step to your heart's content.

However, this is slow: it can take a while before your emulator
gets where you want. Also, you have to develop tooling around
the emulator because you'll need a (emit) and (key?) word that
feeds content to the interpreter loop.

But it doesn't need to be super complicated. It could be as
simple as mapping (emit) to $700 and (key?) to $780. If, for
example, you want to test the "." word, you write "42 . BYE"
+ CR to MEM+$780, call "run", then verify that you end up with
"42 ok" + CR in MEM+$700.

One thing I favor, however, is working with partial binaries.
That is, I copy Collapse OS code around and keep only the bare
```

minimum for the word I want to test, and then I run the code I want to run in BOOT. Quicker to build, quicker to emulate.

1.14 Programming AVR chips (avr.txt)

(In this documentation, you are expected to have an AVR binary ready to send. To assemble an AVR binary from source, see asm/avr.txt Page 90)

To program AVR chips, you need a device that provides the SPI protocol. The device built in the rc2014/sdcard recipe fits the bill. Make sure you can override the SPI clock because the system clock will be too fast for most AVR chips, which are usually running at 1MHz. Because the SPI clock needs to be a 4th of that, a safe frequency for SPI communication would be 250kHz.

The programmer device

The AVR programmer device is really simple: Wire SPI connections to proper AVR pins as described in the MCU's datasheet. Note that this device will be the same as the one you'll use for any modern SPI-based AVR programmer, with RESET replacing SS.

This device should have an on/off switch that controls the chip's power for a very simple reason: Because we can't control what's on the chip, it could mess up your whole SPI bus when RESET is not held low. This means that as long as it's connected and powered, it is likely to mess up your other devices, such as the SD card.

You could put the AVR chip behind a buffer to avoid this, but an on/off switch also does the trick and satisfies the low-tech lover in you.

Programming software

The AVR programming code is at B160.

Before you begin programming the chip, the device must be deselected. Ensure with "0 (spie)".

Then, you initiate programming mode with "asp$", and then issue your commands.

Each command will verify that it's in sync, that is, that its 3rd exchange echoes the byte that was sent in the 2nd exchange. If it doesn't, the command aborts with "AVR err".

Ensuring reliability

The reliability of your communication depends a lot on the soundness of your SPI relay design. If it's good, you will seldom see those "AVR err".

However, there are worse things than "AVR err": wrong data. Sync checks ensure communication reliability at every command, but in the case of commands getting data, you might be out-of-sync when you receive your result without knowing it! To ensure that you're still in sync, you need to issue a command, which might spit "AVR err". If it does, your previous result is unreliable.

Here's an example word that reliably prints the high fuse value from SPI devid 1:

```
: get 1 asp$ asprdy aspfh@ asprdy .x 0 (spie) ;
```

Another very important matter is clock speed. As mentioned above, the safe clock speed is 250kHz. If you use the SPI design in rc2014/sdcard recipe, this means that your input clock speed can theoretically be 500kHz because the '161 divides it by 2.

In practice, however, you can't really do that because depending on the timing of your SPI write, the first "bump" of the SPI clock might end up being nearly 500kHz, which will result in occasional communication errors.

The simplest and safest way to avoid this is to reduce your raw input clock by 2, which will reduce your effective communication speed by 2. There certainly are options allowing you to keep optimal speed, but they're significantly more complex than accepting slower speed.

Access fuses

You get/set they values with "aspfx@/aspfx!", x being one of "l" (low fuse), "h" (high fuse), "e" (extended fuse).

Access flash

Writing to AVR's flash is done in batch mode, page by page. To this end, the chip has a buffer which is writable byte-by-byte.

Writing to the flash begins with a call to asperase, which erases the whole chip. It seems possible to erase flash page-by-page through parallel programming, but the SPI protocol doesn't expose it, we have to erase the whole chip. Then, you write to

```
the buffer using aspfb! and then write to a page using aspfp!.
Example to write $1234 to the first byte of the first page:

asperase $1234 0 aspfb! 0 aspfp!

Please note that aspfb! deals with *words*, not bytes. If, for
example, you want to hook it to C!*, make sure you use MOVEW
instead of MOVE. You will need to create a wrapper word around
aspfb! that divides dst addr by 2 because MOVEW use byte-based
addresses but aspfb! uses word-based ones. You also have to make
sure that C@* points to @ (or another word-based fetcher)
instead of its default value of C@.
```

Access EEPROM

```
Accessing EEPROM is simple and is done byte-by-byte with words
aspe@ and aspe!. Example:

$42 0 aspe! 0 aspe@ .x ( prints 42 )
```

1.15 Word tables (`wordtbl.txt`)

```
Word tables are arrays of pointer to words. B5 provide words
allowing to conveniently create and use these tables. These
words are:

WORDTBL x   n -- a  Initialize a word table named x witn n
                    elements.
:W ... ;    a -- a? Add a new anonymous word to the active tbl
'W x        a -- a? Find x in dict and add it to active tbl

The idea is that when you call WORDTBL, it becomes the active
table by pushing its first address to PS. Then, for each new
element you add, current address is increased and when all
elements are added, that address is dropped from PS. Example
usage:

5 LOAD
: foo 42 . ;
: bar 43 . ;
3 WORDTBL w 'W foo 'W bar :W 44 . ;
w 0 WEXEC \ prints 42
w 1 WEXEC \ prints 43
w 2 WEXEC \ prints 44

When the count (3 in the example) is reached, table's address
is dropped from PS. For this reason, when you create a WORDTBL,
you have to create this exact number of words afterward. If you
create less, you have a PS leak, if you create more, a PS
underflow.
```

1.16 Cross-compilation (`cross.txt`)

When you naively compile binary (see doc/asm) or forth code, the resulting binary will only run on the machine it was compiled on.

If you want to compile for another machine, you need to cross-compile. Collapse OS includes tools to do so.

There are two distinct tasks that require distinct tools: binary xcomp and forth xcomp.

1. Binary xcomp

Assemblers, when encoding absolute addresses, do so naively. If you write "$1234 JMPi,", $1234 will always be the encoded address. So far, so good.

Where there's a problem is when labels are involved. For example, the result of "LSET L1 L1 JMPi," depends on where in memory this opcode was created. As long as the code runs on the machine that compiled it at the address it was compiled, this will always do the right thing. Otherwise, we need to stop being naive. We do so with the XCOMP loader word (it loads B200).

This loader introduces 2 new VALUEs that determine how labels work: XORG and BIN(.

XORG is the address at which our binary starts on the host machine. When you're ready to spit your cross-compiled binary, you'll want to do "HERE TO XORG".

BIN(is the address at which our binary is expected to live in the target machine. By default, it's 0.

There's a convenience "XSTART (bin(--)" word which sets BIN(and XORG in one fell swoop.

Together, these words give you control over what the assembler considers it's current "PC" (program counter) at any given moment. For example, right after a "XSTART", "LSET L1" will give L1 the value of BIN(. $100 bytes later, PC will be BIN(+$100.

To do binary xcomp, you will want to load XCOMP before you load your assembler and only call XSTART when you're ready to spit binary.

Example of xcomp from fresh Z80 Collapse OS boot:

```
ARCHM XCOMP Z80A 0 XSTART LSET L1 NOP, L1 JP,
```

This produces a binary that is designed to run an infinite loop from address 0. Without XCOMP, the jump would be incorrect and jump somewhere in the middle of the memory, where HERE was during compilation.

Cross-compiling for another CPU architecture is the same thing, all you need to do is to load the proper assembler. You just have to be extra careful if compiling for a different endianness. See below.

2. Forth xcomp

Binary xcomp is relatively straightforward. Forth xcomp is a bit hairy. Because forth words are nothing but references to other words all the way until we "hit rock" (hit native code), that code is tricky to relocate.

Collapse OS has tools (which builds upon the tools explained in section 1) to produce a Collapse OS forth dictionary designed to run on another machine at another address.

These tools are loaded with the "XCOMPC" (XCOMP for Collapse OS) which requires "XCOMP" to be loaded first.

As with binary xcomp, XCOMPC requires XORG and BIN(to be properly set before you begin spitting cross forth words.

XCOMPC overrides defining words (:, CREATE, CONSTANT, etc.) so that it adds an offset (XORG) to every wordref it compiles. With this override, you end up with a dictionary that is separate from the host dictionary and is internally consistent.

A useful word supplied by XCOMPC is "X'", which behaves like "'" except that it looks in the xcomp dictionary and it yields addresses with offset applied, that is the addresses as it will be in the target system.

Dual-CURRENT

Although the principle behind cross-compilation is simple, the devil's in the details. While building our new binary, we still need access to a full-fledged Forth interpreter. To allow this, we'll maintain two CURRENT: the regular one and XCURRENT, the CURRENT value of the cross-compiled binary.

XCURRENT's value is a *host* address, not a cross one. For example, if XORG is $1000 and the last word added to it was at offset $234, then XCURRENT is $1234.

During cross compilation, we *define* in XCURRENT and we *execute* in CURRENT.

When we encounter an IMMEDIATE during compilation, we execute the *host* version of that word. The reason for this is simple: any word freshly cross-compiled is utterly un-runable because its wordrefs are misaligned under the current host.

XCOMPCL and XCOMPCH

In some cases, you might want to split your XCOMPC invocation in two: XCOMPCL (low) and XCOMPCH (high). This give you the opportunity to define some macros that use the "X'" word, which is defined in XCOMPCL.

You can't do that if you invoke XCOMPC because as soon as this invocation is done, we're in xcomp mode, we can't define compile time macros anymore.

xcomp unit

Cross-compilation of a Collapse OS binary is achieved through the writing of a cross-compilation unit of code, xcomp unit for short.

The xcomp toolset at XCOMPC alters core words in a deep way, so ordering is important. First, we load our tools. XCOMP, assembler.

We also define some support words that will not be part of our resulting binary, but will be used during xcomp, for example, declarations units and macros.

Then, it's time to apply XCOMPC overrides. From this point on. every defining word is messed up and will produce offsetted binaries.

The XCOMPC loader implicitly calls "0 XSTART", so if your BIN(is 0, you can start spitting right away. Otherwise, call XSTART with a proper BIN(value before you spit.

What to spit? See doc/bootstrap for details, but in short it's:

1. Arch-specific port
2. COREL
3. Drivers
4. XWRAP (which loads COREH and wraps it up)

Once XWRAP is called, and if you did things the right way, what is between XORG and HERE is your fancy new Collapse OS binary!

After you're done, you can run "FORGET PS_ADDR" (or whatever is the first word declared by your xcomp unit) to go back to a usable system.

Immediate compiling words trickyness

When using an immediate compiling word such as "IF" during xcomp, things are a bit tricky for two reasons:

1. Immediates used during xcomp are from the host system.
2. The reference of the word(s) they compile is for the host system.

Therefore, unless the compiled word (for example (?br) compiled by IF) has exactly the same address in both the host and target, the resulting binary will be broken.

For this reason, we re-implement many of those compiling words in xcomp overrides, hacking our way through, so that those compiling words compile proper target references. We don't do this for all compiling words though. This means that some words can't be used in core and drivers, for example, ABORT" and .".

DOES words

DOES> can't work in an xcomp environment. It's too tricky because the word that DOES> compiles has to work in both the host and the target system at the same time. The hoops to jump through to make this kind of word work are horrific.

However, DOES words do allow for some juicy space saving, so the xcomp program has an alternate way to compile does words: ~DOER.

Instead of creating a "generator" word as you do with DOER, you first create an anonymous word with ":~", which is the equivalent of the code following the "DOES>" word. And then, you create your DOES words with ~DOER at runtime. Example:

```
:~ ( n 'n ) @ + . ;
~DOER foo 42 T,
~DOER bar 54 T,
```

In the compiled system, "1 foo" will print 43 and "1 bar" will print 55. "T," is for endian-ness. See below.

Endian-ness

16 bit numbers you write when cross-compiling will often need to follow your target's endian-ness, which might not be the same as your host's. To this end, XCOMP defines these words:

```
|T: Split word into 2 bytes, using Target's endian-ness.
T!: Like "!", but uses Target's endian-ness.
T,: Like ",", but uses Target's endian-ness.
T@: Read a word using Target's endian-ness. Used, for example,
    in XFIND to read prev to traverse a cross-compiled dict.
```

Constants and IMMEDIATE-ness, oh my!

One thing that is particularly tricky with xcomp code is the management of constants. VALUEs declared before XCOMPC is loaded are *only* accessible outside of compilation mode. For example, PS_ADDR will not be a word in the target system. When writing assembly, you can reference it just fine because you're in runtime mode. However, if you're inside a ":", you can't reference PS_ADDR. You have to add a literal of its value with "[PS_ADDR LITN]" (or by creating a VALUE inside the target, but this will take precious binary space!).

Extra words

xcomp tools define a couple of extra words that are specific to it:

```
OALLOT   oa --   ALLOT0 n bytes where n = oa-PC. In other words,
                 make current binary oa bytes, filling with 0.
*VALUE   --      A read-only, indirect VALUE
*ALIAS   --      An indirect ALIAS
~DOER x  --      Create a xcomp-compatible DOES word. See above.
PC2A     pc -- a Transforms a target's PC into a host's addr.
```

1.17 Architecture management (`arch.txt`)

To facilitate the development of the Collapse OS project, code related to specific architectures all live in their separate blk.fs file in /arch. This arch-specific code is organized to live at B300. This means that, out-of-the-box, Collapse OS can only be built with one architecture at once.

For example, /cvm/Makefile builds a blkfs with the /cvm/cvm.fs architecture. /arch/z80/rc2014/Makefile builds a blkfs with a /arch/z80/blk.fs architecture.

How then can you cross-compile from within Collapse OS? Out of

the box, you can't. You have to craft your own blkfs. The good news is, it's not complicated.

For example, if you want a z80/8086 blkfs, you can start with a z80 blkfs and graft /arch/8086/blk.fs on top of it. This could mean, for example, that 8086 blocks start at B440. If you want "round" blocks, you can add a "phantom" 199 marker at the end of /arch/z80/blk.fs which would make your 8086 arch start at B500.

Then, to have a clean system, adjust bock numbers in 8086 "ARCHM" block (B1 of 8086) to have their base offset B440 instead of B300. Finally, adjust your "ARCHM" loader word to also load B441. You now have a clean z80/8086 Collapse OS!

1.18 Bootstrap guide (`bootstrap.txt`)

This guide tells you about the gory details you need to know to create or maintain a port of Collapse OS. This is some pretty hairy stuff and you should have read doc/usage, doc/impl and doc/cross first. Is is also recommended that you use the z80 port (arch/z80/blk.fs) as a reference as you read this guide.

What is Collapse OS? It is a binary placed either in ROM on in RAM by a bootloader. That binary, when executed, initializes itself to a Forth interpreter. In most cases, that Forth interpreter will have some access to a mass storage device, which allows it to access Collapse OS' disk blocks and bootstrap itself some more.

This binary can be separated in 5 distinct layers:

1. Arch-specific boot code (B302 for Z80)
2. Arch-specific words (B303 for Z80)
3. Arch-independant core words (low) (B210)
4. Drivers, might contain arch-specific code
5. Arch-independant core words (high) (B225)

Boot code

The boot code, which is arch-specific, contains these elements:

1. Early initialization code, which initializes RSP and PSP and then jumps to BOOT.
2. Core routines. Set lblnext, lblxt, lblcell, lblval and lbldoes.

Boot words

Then come the implementation of core Forth words in native assembly. This is a limited set of words that implement core operations:

QUIT ABORT EXIT BYE RCNT SCNT * /MOD TICKS (b) (n) (br) (?br) (next) C@ @ C! ! AND OR XOR NOT + - R> >R R~ DUP ?DUP DROP SWAP OVER ROT EXECUTE

If your arch's native absolute jumps and calls aren't in the form "1b opcode + 2b addr", you also need to implement a native version of JMPi! and CALLi!. Otherwise, the forth layer will provide implementations.

On CPUs having I/O ports, PC! and PC@ are also needed.

This is the absolute minimum set of words that a port needs to be functional. If it only implements those words, however, it's going to be very slow.

Some forth core words are defined with "?:" instead of ":". If those words are part of the native words, they're going to be used instead of their forth version and will result in a much faster binary.

Core words (low)

Then comes the part where we begin defining words in Forth. Core words are designed to be cross-compiled, from a full Forth interpreter. This means that it has access to more than boot words. This comes with tricky limitations. See doc/cross.

Drivers

Core words don't include (key?) and (emit) implementations because that's hardware-dependant. This is where we need to load code that implement it, as well as any other driver code we want to include in the binary. This includes subsystems.

We do it now because if we wait until the high layer of core words is loaded, we'll have messed up immediates and ":" will be broken. If we load our code before, we won't have access to a wide vocabulary.

See doc/drivers for more details.

Core words (high)

The final layer of core words contains the BOOT word as well as tricky immediates which, if they're defined sooner, mess cross compilation up. Once this layer is loaded, we become severly limited in the words we can use without messing up.

Forward labels (BOOT, main and HERE)

Because Collapse OS assemblers have limited support for forward labels, we organize the code in order to avoid it. When compiling Collapse OS, however, there are some cases where we reference an address without knowing it yet. These are:

1. The initial jump to BOOT
2. The jump to the "main" (see doc/impl) routine in QUIT.
3. Initial values for HERE and CURRENT.

The XCOMPC program declares 3 labels, lblboot, lblmain and lblhere where address of these forward references must be written (in PC value, like any label, *not* host address).

Core words manage the lblhere references by itself, arch-specific ports don't need to do anything about it. However, it needs to set lblhere and lblmain at the exact PC where the address of the corresponding routines will be written when they're created. You can look at the z80 port for examples.

Building it

So that's the anatomy of a Collapse OS binary. How do you build one? If your machine is already covered by a recipe, you're in luck: follow instructions.

If you're deploying to a new machine, you'll have to write a new xcomp (cross compilation) unit. Let's look at its anatomy. First, we have constants. Some of them are device-specific, but some of them are always there. SYSVARS is the address at which the RAM starts on the system. System variables will go there and use $80 bytes. See doc/impl.

HERESTART determines where... HERE is at startup. 0 means "same as CURRENT".

You will likely need more constants than that, but this depends on your architecture and drivers.

Then comes time time to load the blocks that will compile the thing. Order is important.

First come XCOMP followed by the assembler (example, Z80A). Then

comes CPU-specific macros, constants further loader words such as the "C" units. They always live in B301 (see doc/blk). The loader word for this is ARCHM. It's important that it's loaded before XCOMPC because it's being executed during xcomp, not included in the target binary.

Now comes the real deal: XCOMPC. It's the "forth" part of xcomp and from this point on, we're in "target" mode. Everything we define ends up in the target binary (see doc/cross).

The first unit that comes after this is the "C" unit (example: Z80C). "C" is for "code". It's the layers 1 and 2 from the layer list at the top of this document.

We're done with the CPU-specific part! Now comes COREL, for "core words (low)".

Then comes the custom part: drivers and subsystems. This part is heavily dependant on the target system and varies a lot.

After that, you need to define a INIT word. This will be called by BOOT right before spitting the prompt. This is usually used to call init words of all subsystems.

All xcomp unit end with XWRAP, a helper word that loads "high" core words and then wrap things up (set CURRENT and LATEST in the stable ABI). You're done!

To produce a Collapse OS binary, you run that xcomp unit and then observe the values of XORG and HERE. That will give you the start and stop offset of your binary, which you can then copy to your target media.

Good luck!

1.19 Hardware Drivers (`drivers.txt`)

To be able to run on a wide variety of hardware, Collapse OS needs to abstract away interactions with it. It does so with drivers, which are words that conform to a protocol and whose job is to talk with the specific hardware they support.

This way, core words can be independant of implementation details for particular hardware.

Running a minimal Collapse OS requires very little drivers:

```
(key?)     -- c? f Returns whether a key has been pressed and,
                   if it has, returns which key. When f is
                   false, c is *not* placed in the stack.
(emit)     c --    Spit a character on the console.
```

To have a functional (albeit minimal) Collapse OS running on your fancy machine, all you need to do is to insert these words in the "Drivers" layer of your xcomp (see doc/bootstrap) and you're golden.

Be aware that these words are cross-compiled, so xcomp rules apply. See doc/cross.

Most of the time, you'll want to implement those words in native code. But Forth code is also an option. At the driver layer, the whole "low" part of core words are available, which is a majority of core words. And, because we're in xcomp, all immediate words are provided by the host. Therefore, the only words that are off-limit for Forth driver code are non-imm words defined in the "high" part of core words.

Subsystems

Having a minimal Collapse OS is already awesome, but maybe you'd like to go a bit further and support fancy stuff like mass storage or RS-232.

To that end, Collapse OS has subsystems, which are chunks of logic sitting on top of wider hardware abstractions. For example, the SD card subsystem depends on having hardware that can somehow do SPI communication with a particular device.

So, if you make the effort of implementing the protocol required by the SD card subsystem, then you win the prize of being able to access SD cards!

Each subsystem in Collapse OS has its own documentation page which details its required protocol and sub-subsystems:

* BLK subsystem (doc/blk)
* Grid subsystem (doc/grid)
* RX/TX subsystem (doc/rxtx)
* SPI protocol (doc/spi)
* PS/2 subsystem (doc/ps2)

1.20 The Grid subsystem (`grid.txt`)

The grid subsystem at B240 supplies a set of words on top of the Grid protocol (see "Grid Protocol" below) that facilitates the development of programs presenting a complex text interface, for example, the Visual Editor.

It creates the concept of a cursor, always being a some position on screen. That position is in the variable XYPOS, which is a

simple integer following the same "pos" logic as in the Grid protocol.

It implements (emit), which sets the cell under the cursor to the specified character, then moves the cursor right. If the cursor is at the last column of the screen, it overflows to the next line. If it's on the last line, it overflows to the first line.

Grid's (emit) handles $d by moving the cursor to the next line and $8 by moving the cursor left.

AT-XY (x y --) moves the cursor to the specified position. It is equivalent to setting XYPOS directly, but uses separate X and y numbers.

When the grid's cursor enters a new line, it clears its contents through a repeated call to CELL!. That implementation is in its world named NEWLN (ln --). This word can be overridden. If it exists when the grid subsystem is loaded, the existing NEWLN will be used.

At build time, the Grid subsystem needs 3 bytes of system memory through the GRID_MEM constant. At run time, GRID$ needs to be called to initialize the system.

Grid protocol

A grid is a device that shows as a grid of ASCII characters and allows random access to it.

```
COLS      -- n          Number of columns in the device
LINES     -- n          Number of lines in the device
CELL!     c pos --      Set character at pos

Optional (default implementation provided if arch doesn't):
NEWLN     old -- new    Go to a new line from old, into new.
CURSOR!   new old --    Move cursor from old pos to new pos
CELLS!    a pos u --    *A* Update u contiguous cells, starting
                        at pos, using characters starting at
                        address a.
FILLC!    pos n c --    Fill n cells with character b as pos.

Words provided:
XYPOS!    pos --        Set cursor position to pos.
AT-XY     x y --        Like XYPOS!, but with separate x y.
STYPEC    sa sl pos --  Like STYPE, but at specific position.
                        Much faster than AT-XY + STYPE.
CLRSCR    --            Clears the screen and move cursor to 0.
```

"pos" is a simple number (y * cols) + x. For example, if we have 40 columns per line, the position (x, y) (12, 10) is 412.

CELL! allows all possible values of "c", including ASCII control characters. The driver implementation isn't expected to filter them out. Many systems have glyphs for this ASCII range, so the driver should just show that glyph.

NEWLN is called when we "enter" a new line, that is, when we overflow from previous line or when $0d (ASCII CR) is emitted.

When this is called, the line being entered should be cleared of its contents. If the video driver needs to scroll, now is the time. The NEWLN implementation has to return the new current line. Most of the time, it's old+1, but if you scroll, you might want to return old+0.

If it's not defined, the default implementation simply wraps to the first line when reaching the end of the screen.

CURSOR! is called whenever we change the cursor's position. If not implemented, it will be a noop. It is never called with an out of range "pos" (greater than COLS*LINES).

It is the driver's responsibility to preserve the contents under the cursor. When CURSOR! is call, we expect "old" to be restored to the character it was before the cursor came on it. Before doing that, however, it should make sure that the contents hasn't been overwritten by CELL!.

CELLS! is a speed optimization. With some hardware, it's much faster to update in batch. If the driver implements this an the application uses it properly, it results in big speed gains.

1.21 The PS/2 subsystem (ps2.txt)

This subsystem translates keycodes received by a PS/2 keyboard and provides a (key?) word. To work, drivers need to provide this:

```
(ps2kc)   -- kc    Returns the next typed PS/2 keycode from the
                   console. 0 if nothing was typed.
```

Then, it's as simple as loading PS2SUB to your xcomp.

1.22 Sega Master System ROM signatures (sega.txt)

When loading ROM, the SMS' BIOS checks for a special signature at the end of that ROM. If that signature is incorrect, the ROM doesn't load.

Collapse OS has a program to generate that signature at B165.

This document describes what it does.

At boot, the BIOS checks $10 bytes before the $8000, then $4000, then $2000 mark for a signature. This signature has the following structure.

$00-$07: String constant: "TMR SEGA"
$08-$09: null bytes
$0a-$0b: checksum
$0c-$0e: null bytes
$0f : "size" flag

The checksum is a simple 16-bit sum of all bytes up to the beginning of the signature.

The size flag can have 3 values: $4a for an 8K ROM, $4b for 16K and $4c for 32K. It can have other values for other kinds of sizes, but we don't care about them in the context of Collapse OS.

Generating the signature

Before generating the signature, you need to have the contents of your ROM somewhere in memory. Then, you load B165 and you call "segasig" which has the signature "addr size". "addr" is the adress of the beginning of the ROM and "size" is 0, 1 or 2 depending on whether your ROM is 8K, 16K or 32K.

Calling the word will write the $10 bytes signature at the end of the ROM.

Note that all I/O use the "Addressed device" words (see usage.txt Page 7), so I/O indirections will work.

1.23 Assembling Collapse OS from within it (selfhost.txt)

This is where we tie lose ends, complete the circle, loop the loop: we assemble a new Collapse OS *entirely* from within Collapse OS.

Build Collapse OS' from within Collapse OS is very similar to how we do it from the makefiles in /arch. If you take the time to look one, you'll see something that look like "cat xcomp.fs | $(STAGE)". That's the thing. Open "xcomp.fs" in a text editor and take a look at it. Some xcomp units are simple proxy to a block, which you'll find in the blk/ subfolder for this recipe.

To assemble Collapse OS from within it, all you need to do is execute the content of this unit. When you run makefiles, it's already Collapse OS building itself from within it, so it's not

different when it's the real deal.

When you do so, it will yield a binary in memory. To know the start/end offset of the binary, You'll use ORG and HERE. ORG is where your first byte starts in your host's memory, "HERE ORG -" is the size of your binary.

With that, you can write that binary between those offsets on your target media. That binary should be the exact same as what you get in "os.bin" when you run "make". You now have a new Collapse OS deployment.

See more details on bootstrapping at doc/bootstrap.

What to do on SDerr?

If you self host from a machine with a SD card and you get "SDerr" in the middle of a LOAD operation, something went wrong with the SD card. The bad news is that it left your xcomp operation in an inconsistent state. The easiest thing to do it to restart the operation from scratch. Those error are not frequent unless hardware is faulty.

Cross-compiling directly to EEPROM

If your target media is a RAM mappable media, you can save precious RAM by cross-compiling Collapse OS directly to it. It requires special handling.

You can begin the process in a regular manner, but right before you're about to assemble the boot code, take a pause.

Up until now, you've been loading your cross compiling tools in RAM, now, you're about to write Collapse OS. So what you need to do is change HERE to the address of your EEPROM. Example:

```
$2000 'HERE !
```

Then, you can continue the process normally.

1.24 Media Spanning subsystem (`mspan.txt`)

The Media Spanning (MSPAN) subsystem allows systems with a BLK (see doc/blk) media having a capacity that is lower in size than the size of the blkfs to conveniently manage the spanning of blocks over multiple media.

For example, in the case of 5 1/4 floppies, you can't have the whole of Collapse OS on one floppy. You'll probably want to have

100 blocks per disk (with double density, you can have about 180, but if you put 180 blocks per disk, spanning gets at odds with Collapse OS blkfs organization, which is 100-based).

You can do this without any subsystem, but you'll have a problem with loader words. For example, if you want to load Z80A, you'll have 2 distinct problems:

1. B320 is outside the range of a floppy, your disk driver won't find the proper track/sector.
2. When B007 (Flow words) is loaded at the end, you need to swap disks.

The MSPAN media inserts itself between your blk drivers and the BLK subsystem and whenever (blk@) or (blk!) is called, it checks whether the block currently being requested is on the same disk as the one inserted before. If it is, it lets the request continue, while adjusting the block number (for example, if disk 3 is inserted with a span of 100 blocks per disk and that B342 is requested, the block number is adjusted to 42). If it's not, it prompts the user with a message asking her to insert the proper disk and then press a key.

Media block adjustments are made through the (msdsks) array defined at compile time. It's the list of each media block count. For example, if your list is 100,200,100, then block 42 is assigned to disk 0, 142 to disk 1, 399 to disk 2. If the requested block is out of bounds, the block is assigned to the last disk.

Dual drives system

If the system has multiple drives, it can implement DRVSEL. If it is, you can press keys 0-9 to indicate to the MSPAN system in which drive the needed disk is.

MSPAN protocol

At compile time, the following constants must be defined:

```
MSPAN_MEM  Address for MSPAN sysvars. 1 byte used.
```

```
Required words:
(ms@)    blkno dest -- Replaces (blk@)
(ms!)    blkno dest -- Replaces (blk!)
(msdsks) -- 'dsks      0-terminated array of 1b disk sizes

Optional words:
DRVSEL   drv -- Make drv the active drive. This word is always
```

```
                called on a FLUSHed system, assume cleanliness.

Provided words:
(blk@)  blkno dest -- Plugs in BLKSUB
(blk!)  blkno dest -- Plugs in BLKSUB
MSPAN$  --            Initializes the subsystem

Defined variables:
MSPAN_DISK  1b. Currently selected disk. Initializes at 0.

Example implementation:

[...]
SYSVARS $80 + VALUE MSPAN_MEM
[...]
COREL
ALIAS (myfd@) (ms@)
ALIAS (myfd!) (ms!)
CREATE (msdsks) 100 C, 100 C, 180 C, 0 C,
[...]
MSPANSUB
[...]
: INIT [...] MSPAN$ [...] ;
XWRAP
```

1.25 Algorithmic notes (`algo.txt`)

Multiply a number by another

```
Let's say we want to multiply 6 by 5 for a result of 30.

First, let's look at binary forms:

110 ( 6 )
101 ( 5 )

The idea is that we'll loop 16 times (for 16 bit) through one of
the numbers (let's use 6), left-shifting through it. At each
step, we check if we've shifted a 1. If yes, then we add the
second number to our running result, which we also left shift
at each step.

Let's try our first (well, 13th for a 16 bit number) step. We
left-shift a 1 from 6, so we add 5 to our running result. That
gives us:

 10 ( remainder 2 )
101 ( result 5 )

Step 2, we do the same thing, left-shift a 1 again. We when left
shift our result, than add 5 to it again:
```

```
   0 ( remainder 0 )
1111 ( result 15 )
```

Then, for your last (16th) step, we left-shift a 0, so we don't add anything to our result, but we still left-shift it, which gives us our final result:

```
    0 ( remainder 0 )
11110 ( result 30 )
```

Divide a number by another, with remainder

Let's say we want to divide 249 by 7 so that we end up with 35 rem 4.

First, let's look at binary forms:

```
11111001 ( 249 )
     111 ( 7 )
```

The general idea is that we try to take the 7 and "fit" it leftmost as much as possible so that we can subtract it. That gives us 2 things: an order of magnitude and a remainder. Then, we repeat until we can't do it any more.

For the first step, we can shift 7 5 times, which gives us:

```
11111001 ( 249 )
11100000 ( 224 )
```

We subtract, which gives us:

```
 11001 ( remainder 25 )
100000 ( quotient 32 )
```

Then, we fit the divisor in the remainder again:

```
11001 ( 25 )
 1110 ( 14 )
```

Which gives us:

```
 1101 ( remainder 11 )
10010 ( quotient 34 )
```

We have wiggle room for one last step:

```
1101 ( 11 )
 111 ( 7 )
```

```
Which gives us:

  100 ( remainder 4 )
10011 ( quotient 35 )

In terms of computing, the hard part is the "fitting". All
/MOD words in Collapse OS use the same fitting logic:

We begin with a remainder and quotient at 0 and we have a loop
that executes 16 times (for 16 bit numbers). At each step, we
left-shift the dividend into the remainder and try to subtract
the divisor from it. If it fits, we left shift a 1 into the
quotient, otherwise we left shift a 0 into the quotient.

To save space, we can even use the same memory space for the
input dividend and the output quotient because the result never
overlap while we left-shift.
```

1.26 Frequently asked questions (`faq.txt`)

What is the easiest way to run Collapse OS on a modern computer?

```
Run the C VM in folder "/cvm". Run "make", then "./cos-grid".
See doc/usage for the rest.
```

How do I use the different emulators?

```
Ah, you've noticed that /emul contains quite a few emulators.
Code in this folder only build emulators, not the binary to run
under it. It's the /arch folder that contains the makefiles to
build Collapse OS binaries to run under those.

When a binary built in /arch has a corresponding emulator, the
makefile has a "emul" target that you can use.

For example, "cd arch/z80/rc2014 && make emul" builds RC2014's
Collapse OS, the RC2014 emulator and then invokes the emulator.
```

How do I fill my SD card with Collapse OS' FS?

```
Very easy. You see that "/cvm/blkfs" file? You dump it to your
raw device.  For example, if the device you get when you insert
your SD card is "/dev/sdb", then you type "cat emul/blkfs | sudo
tee /dev/sdb > /dev/null".
```

2 Assemblers

2.1 Assembling binaries (`asm/intro.txt`)

Collapse OS features many assemblers. Each of them have their specificities, but they are very similar in the way they work.

This page describes common behavior. Some assemblers stray from it. Refer to arch-specific documentation for details.

Initial setup

Assemblers live in their arch-specific blkfs. To load it, you first need to run "ARCHM" to have arch-specific loaders, and then call your assembler loader (for example, "Z80A").

Loaded alone, an assembler will spit opcodes for a "live" target, that is, the computer it's running on.

As long as you don't relocate the code, you will be able to run it just fine, but if you need to relocate it, you will need to load XCOMP *before* you load your assembler so that you have the necessary tooling to craft relocatable binaries.

See doc/cross for details.

Wrapping native code

You will often want to wrap your native code in such a way that it can be used from within forth. You do that with CODE.

CODE allows you to create a new word, but instead of compiling references to other words, you write native code directly. Example:

```
CODE 1+ BC INC, ;CODE
```

This word can then be used like any other (and is of course very fast).

Unlike the regular compiling process, you don't go in "compile mode" when you use CODE. You stay in regular INTERPRET mode. All CODE does is spit the proper word header.

Be sure to read about your target platform in doc/code. These documents specify which registers are assigned to what role.

Usage

To spit binary code, use opcode words, such as "LD,", preceded by proper arguments. For example, "A B LD," in the z80 assembler spits the "LD" opcode in it's "rr" (8b register transfer) form.

Each assembler has its own little specificities as to how to spit ops in their different forms. See arch-specific asm docs.

In all assemblers, arguments precede the opcode word ("A B LD," instead of "LD A B").

Some assemblers do basic argument checking and will let you know if you're trying to spit something that doesn't make sense, but none of them has complete error checking, so they will let you spit nonsensical opcodes.

Why insist on prefix notation?

Compared to regular assemblers, which place their arguments after the opcode mnemonic, Collapse OS assemblers are a bit mind bending. Why do we adopt this notation?

First and foremost, simplicity. By using a notation that is the same as forth's, we get the parsing part for free.

However, one can think of many simple ways of achieving regular notation and these ways, compared the the overall system complexity, wouldn't be such a complexity burden. Why insist in prefix?

Macros. By sticking to regular forth, we have macro-ability for free. If you add a postfix parsing mechanism in there, you need to add special provisions for macros, and then things get icky.

So, that's why.

Labels and flow

Assemblers, of course, implement their "flow" ops (jumps) but these are often awkward to use directly. To help with that, Collapse OS has a unified "flow" interface:

```
IFZ, .. ELSE, .. THEN, \ part 1 if Z is set, part 2 otherwise
IFNZ, .. THEN, \ execute if Z is unset
IFC, .. THEN, \ execute if C is set
IFNC, .. THEN, \ execute if C is unset
BEGIN, .. BR JRi, \ loop forever
```

```
BEGIN, .. BR JRZi, \ loop if Z is set
FJR JRi, .. THEN, \ unconditional forward jump
```

This unified flow layer lives at B007 and is loaded with assemblers. This layer requires the assembler to supply these words (which are ofter simple aliases):

```
JRi,    off --    relative unconditional jump
JRZ,    off --    relative conditional jump if Z is set
JRNZ,   off --    relative conditional jump if Z is unset
JRC,    off --    relative conditional jump if C is set
JRNC,   off --    relative conditional jump if C is unset
```

This is not related to flow, but for xcomp, these words are also defined by every assembler:

```
JMPi,   addr --   unconditional absolute jump
JMP(i), addr --   unconditional indirect jump
CALLi,  addr --   unconditional absolute call
i>,     n --      push n to Parameter stack
(i)>,   addr --   push value at addr to Parameter Stack
```

These words generate the appropriate native code to perform the described actions.

These structured flow are elegant, but limited because they need to be symmetric. There is no way, for example, to jump out of an infinite loop using only those words.

Labels can also be used with those flow words for more flexibility:

```
LSET L1 .. L1 BR JRi, .. L1 JMPi, \ backward jumps
FJR JRi, TO L1 .. L1 FMARK \ forward jump
BEGIN, FJR JRi, TO L1 .. BR JRi, .. L1 FMARK \ exiting loop
```

Labels are simple VALUEs. For example, you can create a label with "0 VALUE lblmylabel". If in an XCOMP context, make sure you declare your labels before XSTART. XCOMP pre-declares L1 L2 and L3 which can be used in local contexts.

2.2 Z80 assembler specificities (`asm/z80.txt`)

Load with "Z80A".

All instructions have a corresponding word, which is its name followed by "," (to indicate that the word writes the opcode). For example, you write "AND" with "and,".

Instructions can take zero, one or two operands, depending on their type. These operands come in multiple types:

```
r: 8b register. A B C D E H L (HL)
d: 16b register. HL BC DE AF SP
c: condition. CZ CNZ CC CNC CPO CPE CP CM
i: immediate. Create with "i)", like in "42 i)"
m: memory. Create with "m)", like in "$1234 m)"
special: see below. AF' (BC) (DE) (SP) (C) R I

All operand word share one common behavior: they result in
exactly one element being pushed to PS. These element follow a
particular bit structure described below. This means that you
can predictably juggle your operands on PS, making this assem-
bler easily macroable.

This property of operands presents a challenge with "i)" and
"m)" words, which cannot fit entirely in 16-bit with their
represented number. To work around this, a "number bank" system
is implemented, and only an index in that bank, a rolling buffer
containing 8 16-bit numbers, is kept in the bit structure.

This means that although the operand system is macroable, those
operands can't be kept around permanently. They need to be used
in the short term.

To assemble an op, you use a mix of those operand words and then
use the corresponding instruction word to spit the result. The
instruction word automatically determines the form of the op
from the preceding arguments.

When there are two arguments, the one on PS top is the "source"
and the one under it is the "destination". For example,
"A B ld," copies the value of register B into register A.

(HL) is considered an r argument for assembling purposes because
z80 consistently allows (HL) to be used in all "r" forms of ops.

SP and AF have the same value. Some 16b ops affect AF, some
affect SP, but never both at the same time.

Assembling a number with an immediate or a memory operand looks
like this: A 42 i) add, $1234 m) BC ld,

Special arguments are single purpose. AF' and (SP) are for ex,:

AF AF' ex,
(SP) HL ex,

(BC) and (DE) are for ld,:

A (BC) ld,
(DE) A ld,
```

```
(C) is for in, and out,:

(C) B out,
E (C) in,

On the subject of IN and OUT: z80 data sheet includes () in its
immediate form, but in this assembler, we don't use "m)", so
the immediate form of in, and out, is used thus:

42 i) A out,
A 42 i) in,

Some forms only work with A or HL. You must explicitly specify
A and HL in your arguments when you use them:

A 42 and, (not "42 and,")
HL BC add, (not "BC add,")

Purely inherent ops don't have this requirement (it's "neg," not
"A neg,").
```

Jump words

```
None of the jump words use the operand system described above.
They all work with straight numbers. For example, "$1234 jp,"
writes a jump to address $1234. The "conditional" versions of
the jump words have a "c" suffix to their name and must receive
a conditional constant (which doesn't follow operand bit
structure) in PS top: $1234 CZ callc,

The three "jump to register value" instructions each have their
own special word: jp(HL), jp(IX), jp(IY),
```

Flow examples

```
IFZ, nop, ELSE, nop, THEN,
BEGIN, nop, BR jr, ( unconditional )
BEGIN, nop, BR CZ jrc, ( conditional )
LSET L1 nop, L1 BR jr, ( backward jump )
FJR jr, TO L1 nop, L1 FMARK ( forward jump )
```

ix+) iy+)

```
As a general rule, IX and IY are equivalent to spitting an extra
$dd / $fd and then spit the equivalent of HL or (HL).
```

In "HL" op types, IX and IY words can be used simply. Examples:

```
IX push,
IY pop,
IX $1234 i) ld,
IY HL add,
```

In "(HL)" op types, all IX/IY words contain displacements and need to be used with ix+) and iy+) prefix words.

Examples:

```
0 ix+) E ld,
-2 iy+) inc,
```

Instructions list

Letters in [] brackets indicate operand types that can be used with it. Order is important, the last letter is the one on PS top. Sometimes, only "A" or "HL" is possible, in which case the signature indicates it.

```
r => A B C D E H L (HL)
d => BC DE HL AF/SP
c => CNZ CZ CNC CC CPO CPE CP CM
i => immediate
m => memory reference

ld  [rr, ri, di, dm, md, (DE/BC)A, A(DE/BC), mA, Am]
ex  [DE/HL, AF/AF', (SP)/HL, (SP)/IX, (SP)/IY]
add [Ar, Ai, HLd]
adc [Ar, Ai, HLd]
sbc [Ar, Ai, HLd]
and [Ar, Ai]
cp  [Ar, Ai]
or  [Ar, Ai]
sub [Ar, Ai]
xor [Ar, Ai]
inc [r, d]
dec [r, d]
out [iA, (C)r]
in  [Ai, r(C)]
jp  [, c, (HL), (IX), (IY)]
jr  [, CZ, CNZ, CC, CNC]
call[, c]
ret [, c]

push       pop
set        res         bit
rl         rlc         sla         rla         rlca
```

```
rr          rrc         srl         rra         rrca
rst         djnz
di          ei          exx         halt
nop         reti        retn        scf         ccf
cpi         cpir        cpd         cpdr        im0
im1         im2         ini         ldi         ldir
ldd         lddr        neg         outi

Macros:

subHL       Clear carry + sbc
pushA       Push value of A. Destroys BC
HLZ         Set Z according to HL. Destroys A
DEZ         Set Z according to DE. Destroys A
BCZ         Set Z according to BC. Destroys A
ldDE(HL)    16-bit LD from (HL) to DE. HL+1
ldBC(HL)    16-bit LD from (HL) to BC. HL+1
ldHL(HL)    16-bit LD from (HL) to HL. Destroys A
outHL       ( port -- ) OUT H, then OUT L. Destroys A
outDE       ( port -- ) OUT D, then OUT E. Destroys A
clrA        Sets A to 0
```

Operand encoding

```
Operand yielded by constants above follow this bit structure:

b15:8  IX+/IY+ displacement
b7     unused
b6     IX+/IY+
b5     Special register
b4:3   type: 0=8b register
             1=16 register
             2=immediate
             3=memory
b2:0   register, condition or number bank ID

In 8-bit mode, register IDs are

000 B
001 C
010 D
011 E
100 H
101 L
110 (HL)
111 A

In 16-bit mode, they are:

000 BC
```

```
001 DE
010 HL
011 AF or SP depending on context

In special mode, they are:

000 (BC)
001 (DE)
010 (SP)
011 AF'
100 I
101 R
110 (C)

Conditions are:

000 CNZ
001 CZ
010 CNC
011 CC
100 CPO
101 CPE
110 CP
111 CM
```

2.3 8086 assembler specificities (`asm/8086.txt`)

```
Load with "8086A".
```

Argtypes

```
Mnemonics are followed by argument types. For example, MOVri,
moves 8-bit immediate to 8-bit register.

'r' = 8-bit register           'x' = 16-bit register
'i' = 8-bit immediate          'I' = 16-bit immediate
's' = SREG register

Mnemonics that only have one signature (for example INT,) don't
have operands letters.
```

Mod/rm mnemonics

```
Mnemonics with "[]" argtypes are "mod/rm" mnemonics are are
designed to be fed with a "modrm argument". For example, if we
want to INC the byte in memory where DI points to, we would
write "[DI] [b] INC[]," If we want to increase the word at DI+1,
it would be "[DI] 1 [w]+ INC[],".
```

```
There are 2 kinds of modrm mnemonics: single and dual. Single
are for ops like "INC[]" or ops pairing a modrm with an
immediate such as "CMP[]i". Dual are for ops like "ADD[]" which
pairs a register with a memory address.

Single:

[m]    Direct memory address (byte)
[M]    Direct memory address (word)
[r]    8b register
[x]    16b register
[b]    Indirect byte
[w]    Indirect word
[b]+   Indirect byte + displacement (8b)
[w]+   Indirect word + displacement (8b)

Dual:

r[]    Indirect byte to 8b register
x[]    Indirect word to 16b register
[]r    8-bit register to indirect byte
[]x    16-bit register to indirect word
r[]+   Indirect byte + displacement (8b) to 8b register
x[]+   Indirect word + displacement (8b) to 16b register
[]+r   8b register to indirect byte + displacement (8b)
[]+w   16b register to indirect word + displacement (8b)

Remember that BP is only valid with displacement mod/rm.

NOTE: the []i form also works with [x]. It auto-detects whether
"i" is 16b or 8b and writes the proper form.
```

Flow examples

```
IFZ, NOP, THEN, ( no ELSE, yet )
BEGIN, NOP, BR JRi, ( unconditional )
BEGIN, NOP, Z? BR ?JRi, ( conditional )
LSET L1 NOP, L1 JMPi, ( backward near jump )
FJR JRi, TO L1 NOP, L1 FMARK ( forward short jump )

BR, LSET, FMARK come from the HAL convenience layer, see
```

doc/hal.txt Page 57

Instructions list

```
r -> AL BL CL DL AH BH CH DX
x -> AX BX CX DX SP BP SI DI
s -> ES CS SS DS
[] -> mod/rm
```

```
i -> immediate

RET   CLI   STI   HLT   CLD   STD   NOP   CBW   REPZ  REPNZ
LODSB LODSW CMPSB SMPSW MOVSB MOVSW SCASB SCASW STOSB STOSW

CALLi
JMPr is for "register jump" and takes a register as an agument
JMPf is for "far jump" and has signature "segment offset --"

INC[r,x,[]]
DEC[r,x,[]]
POP[x,[]]
PUSH[x,[],s]
MUL[r,x]
DIV[r,x]
XOR[rr,xx]
OR[rr,xx]
AND[rr,xx,ALi,AXI]
ADD[rr,xx,[]i,ALi,AXI]
ADC[rr,xx,[]i,ALi,AXI]
SUB[rr,xx,[]i,ALi,AXI]
INT

CMP[rr,xx,[],[]i]
MOV[rr,xx,[],ri,xI,sx,rm,xm mr,mx,ALm,AXm,mAL,mAX]

("1" means "shift by 1", "CL" means "shift by CL")
ROL[r1,x1,rCL,xCL]
ROR[r1,x1,rCL,xCL]
SHL[r1,x1,rCL,xCL]
SHR[r1,x1,rCL,xCL]
```

2.4 6809 assembler specificities (`asm/6809.txt`)

```
Load with "6809A".

First, the 6809 stands out by being big-endian. It doesn't
change much in terms of assembler usage, but it's a good idea
to keep it in mind.

Then, it stands out by having few "targetable" registers. It
onlt has A, B and D accumulators and X, Y, U and S registers are
targeted directly by only a handful of operations. Therefore,
6809 assembly language designer decided to decline every ops
with all their possible targets. For example, the "ADD" op
has 3 forms: ADDA, ADDB and ADDD. This assembler follow this
design and has an op word for every form.

Then, it stands out by having a vast array of addressing modes.
This significantly impact usage: Except for inherent operations
```

(ops that don't require any argument), all arguments passed to operations have to first pass through an "adressing word". For example, "<>" means the "Direct addressing". Example usage:

```
$42 <> CMPA,
```

This line is equivalent to "cmpa $42" in "regular assembly". Addressing words are:

* "#" --> Immediate
* "()" --> Extended addressing
* "[]" --> Indirect Extended
* Indexed:
  * "R+N" --> Constant Offset indexed
  * "R+0" --> Shortcut for "0 R+N"
  * "R+R" --> Accumulator Offset indexed
  * "R+", "R++", "-R", "--R" --> Auto-increment indexed
  * All index words have their indirect forms: "[R+N]", "[R++]", etc..

Index words above are declined and R is a placeholder. Actual words have actual registers, for example, "X+N", "Y+D", "[S+]", etc. Example full usages:

```
42 # CMPB,
L1 @ () LDA,
X+A ADDB,
[Y++] ADCA,
```

The case of PSH, PUL, TFR, EXG

TFR and EXG are exceptions to the above rule that all arguments go through an adressing word. The 6809 define register constants for usage with TFR and EXG and can be used directly. Example:

```
A B TFR, ( copy A into B )
U S EXG, ( exchange U and S )
```

PSH and PUL are even bigger exceptions. Their argument *follow* the op mnemonics and this argument is a list of single letter registers: $ (for PC), S, U, Y, X, % (for DPR), A, B, D
C (for CCR), @ for all. Order doesn't matter. S/U mean the same thing. D means A and B. Examples:

```
PSHS, ABUXY
PULU, $
PSHU, @
```

Branching

The 6809 assembler supports regular branching words but has special provisions for 16-bit relative branching, something that not all arches support.

The "L" versions of relative branches are present, but because flow words only support 8-bit branching, it's not of much use.

Instructions

Next to each operation, in [] brackets, are supported addressing modes:
M = Immediate D = Direct I = Indexed E = Extended H = Inherent

When forms have the same signature, they are grouped in () brackets.

```
ABX           [H]
ADC(A,B)      [MDIE]
ADD(ABD)      [MDIE]
AND(AB)       [MDIE]    ANDCC [M]
ASL(AB)       [H]       ASL   [DIE]
ASR(AB)       [H]       ASR   [DIE]
BIT(AB)       [MDIE]
CLR(AB)       [H]       CLR   [DIE]
CMP(ABDXYUS)  [MDIE]
COM(AB)       [H]       COM   [DIE]
CWAI          [M]
DAA           [H]
DEC(AB)       [H]       DEC   [DIE]
EOR(AB)       [MDIE]
EXG           SPECIAL
INC(AB)       [H]       INC   [DIE]
JMP           [DIE]
JSR           [DIE]
LD(ABDXYUS)   [MDIE]
LEA(XYUS)     [I]
LSL(AB)       [H]       LSL   [DIE]
LSR(AB)       [H]       LSR   [DIE]
MUL           [H]
NEG(AB)       [H]       NEG   [DIE]
NOP           [H]
OR(AB)        [MDIE]    ORCC  [M]
PSH(US)       SPECIAL
PUL(US)       SPECIAL
ROL(AB)       [H]       ROL   [DIE]
ROR(AB)       [H]       ROR   [DIE]
RTI           [H]
```

```
RTS          [H]
SBC(AB)      [MDIE]
SEX          [H]
ST(ABDXYUS)  [DIE]
SUB(ABD)     [MDIE]
SWI          [H]
SWI2         [H]
SWI3         [H]
SYNC         [H]
TFR          SPECIAL
TST(AB)      [H]      TST   [DIE]

Branches: All words below have a "L" form for a 2b displacement.
Example: BRA --> LBRA

BCC BCS BEQ BGE BGT BHI BHS BLE BLO BLS BLT BMI BNE BPL BRA BRN
BSR BVC BVS
```

2.5 6502 assembler (`asm/6502.txt`)

```
6502 is one of the simplest CPUs out there and its assembler is
also simple. We have 3 types of opcodes: inherent, addressed
and branches.

As with other assemblers, all ops described below have a ","
suffix. For example, you write "NOP," rather than "NOP"
```

Inherent

```
Inherent opcodes are called without argument.

BRK NOP RTI RTS
CLC CLD CLI CLV
SEC SED SEI
DEX DEY INX INY
PHA PLA PHP PLP
TAX TXA TAY TYA TSX TXS
```

Addressed

```
Addressed opcodes take an address argument which needs to be
filtered through address mode words.

#    Immediate
<>   ZeroPage
<X+> ZeroPage+X
<Y+> ZeroPage+Y
()   Absolute
(X+) Absolute+X
```

```
(Y+) Absolute+Y
[X+] Indirect+X
[]Y+ Indirect+Y

The indirect notations are not a typo, they're to illustrate
the difference in indirection scheme between X and Y. See 6502
datasheet.

Example usage:

42 # LDA,
$fe <> LDX,
$1234 () STY,

Not all address modes are legal with all ops below. This
assembler is not going to tell you when your combo is illegal,
it's just going to spit invalid code. The op list below indicate
valid address modes for each op.

We have a special situation with ASL/LSR/ROL/ROR: they can
target the accumulator. We have no addressing mode for this.
Instead, we have a special "inherent" op (no argument) for these
4 cases: ASLA/LSRA/ROLA/RORA. The "A" in the list below indicate
that.

ADC # <> <X+> () (X+) (Y+) [X+] []Y+
SBC # <> <X+> () (X+) (Y+) [X+] []Y+
CMP # <> <X+> () (X+) (Y+) [X+] []Y+
CPX # <> ()
CPY # <> ()
AND # <> <X+> () (X+) (Y+) [X+] []Y+
ORA # <> <X+> () (X+) (Y+) [X+] []Y+
EOR # <> <X+> () (X+) (Y+) [X+] []Y+
BIT <> ()
ASL A <> <X+> () (X+)
LSR A <> <X+> () (X+)
ROL A <> <X+> () (X+)
ROR A <> <X+> () (X+)
DEC <> <X+> () (X+)
INC <> <X+> () (X+)
LDA # <> <X+> () (X+) (Y+) [X+] []Y+
LDX # <> <X+> () (Y+)
LDY # <> <X+> () (X+)
STA # <> <X+> () (X+) (Y+) [X+] []Y+
STX <> <X+> ()
STY <> <X+> ()
```

Branches

```
Conditional branches are all relative, unconditional branches
```

```
are absolute.

There are 2 absolute branching ops: JMP and JSR. They are called
with a single numerical argument. The indirect mode of JMP is
called through the special JMP[] op. Examples:

$1234 JMP,
$1234 JMP[],
$1234 JSR,

Relative branch words are called with a single byte argument
and are compatible with regular flow words:

$fe BEQ,
CLC, BEGIN, NOP, BR BCC,

An important limitation with 6502 is that there is no relative
unconditional branch word! This has important implications with
our regular flow words because it means that "JRi," for 6502
has to be hackish and spit out an absolute JMP. This works with
BR, BUT NOT FOR FMARK.

Therefore, in 6502 code, FMARK is broken with unconditional
jumps and can't be used. Conditional is fine though, so
IF,..THEN, works.

Relative jump words:

BCC BCS (C=0/1)
BNE BEQ (Z=0/1)
BPL BMI (N=0/1)
BVC BVS (V=0/1)
```

2.6 AVR assembler specificities (`asm/avr.txt`)

```
Load with "AVRA".

All mnemonics in AVR have a single signature. Therefore, we
don't need any "argtype" suffixes.

Registers are referred to with consts R0-R31. There is
X, Y, Z, X+, Y+, Z+, X-, Y-, Z- for appropriate ops (LD, ST).
XL, XH, YL, YH, ZL, ZH are simple aliases to R26-R31.

Branching works differently. Instead of expecting a byte to be
written after the naked op, branching words expect a displace-
ment argument.

This is because there's bitwise ORing involved in the creation
of the final opcode, which makes z80a's approach impractical.
```

```
This makes labelling a bit different too. Instead of expecting
label words after the naked branching op, we rather have label
words expecting branching wordref as an argument. Examples:

' BRTS L2 TO, ( branch forward to L2 )
' RJMP L1 LBL, ( branch backward to L1 )
```

Model-specific constants

```
Model-specific constants must be loaded separately. AVRA
supplies loader words. Here's a list:

ATMEGA328P

Those units contain register constants such as PORTB, DDRB, etc.
Unlike many modern assemblers, they do not include bit
constants. Here's an example use:

DDRB 5 SBI,
PORTB 5 CBI,
R16 TIFR0 IN,
R16 0 ( TOV0 ) SBRS,
```

Instructions list

```
OPRd
ASR  COM   DEC   INC   LAC   LAS   LAT   LSR   NEG   POP   PUSH
ROR  SWAP  XCH

OPRdRr
ADC  ADD   AND   CP    CPC   CPSE  EOR   MOV   MUL   OR   SBC
SUB

OPRdA
IN   OUT

OPRdK
ANDI CPI   LDI   ORI   SBCI   SBR   SUBI

OPAb
CBI   SBI   SBIC  SBIS

OPNA
BREAK  CL[C,H,I,N,S,T,V,Z] SE[C,H,I,N,S,T,V,Z] EIJMP ICALL
EICALL IJMP  NOP   RET   RETI  SLEEP WDR

OPb
BCLR  BSET
```

```
OPRdb
BLD   BST   SBRC  SBRS

Special
CLR   TST   LSL   LD    ST

Flow
RJMP  RCALL
BR[BC,BS,CC,CS,EQ,NE,GE,HC,HS,ID,IE,LO,LT,MI,PL,SH,TC,TS,VC,VS]

Flow macros
LBL! LBL, SKIP, TO, FLBL, FLBL! BEGIN, AGAIN? AGAIN, IF, THEN,
```

3 How to read the code

3.1 Code conventions (`code/intro.txt`)

Because compactness is a primary design goal of Collapse OS, comments in the code itself are terse. This represents an extra challenge when comes the time of understanding it.

The code is designed to be accompanied by the documentation. If a piece of code seems underdocumented, you should look for more context in the documentation.

Stack comments

Most comments in Collapse OS describe the expected stack at a point in time. Those comments almost always describe PS with Top-Of-Stack being the rightmost element. For example, a "(a b c)" indicate that at this point, we expect a PS of at least 3 items with "c" being on top of it.

When we play with the Return Stack, we'll also include its signature with "R:". Example: (a b R:c d) means that b is PS' TOS and d is RS' TOS.

Those elements can be seen (and are often called such) as variables.

Names used for those variables are contextual. They're supposed to be context-obvious, but to allow more compactness, some conventions are used:

* A repeat of a previous variable are often 1 or 2 letters. For example, "firstchar" would become "fc" in following comments.
* "a" is an address.
* "sa sl" is an unpacked string. 2 elements in the stack, sl being the length, sa being the address of sl characters.

```
* "w" is a "word reference".
* "b" is a byte, "c" is a char (also a byte). You can generally
  assume the MSB to be 0.
* "n" is a cell-sized (2 bytes) number.
* "u" is a byte count. Often used in ranges.
* "f" is a boolean flag. 0 is false, nonzero if true.
* "r" is a "result", often an accumulator in an algorithm.
* For clarity purposes, the result of complex processing is
  often described in comments (ex: "a*b+c"), but only once.
* In loops, for clarity purposes, the same stack comment is
  often put at the beginning and end of the loop to show that
  we're looping in a balanced manner.
* We indent by 2 (used to be 4) spaces in word defs, loops,
  conditions. We do it loosely though: we often don't have
  enough screen space to do it strictly.

## Idioms

Here are some common patterns you'll see:

<<8 >>8: removes MSB. Faster than "$ff AND".
>>8 IF: Checks if MSB > 0. Faster than "$ff >".
```

3.2 Z80 Boot code (code/z80.txt)

```
Let's walk through Z80 Boot code in arch/z80/blk.fs.

This assembles the boot binary. It requires the Z80 assembler
(B5) and cross compilation setup (B200). It requires some
constants to be set. See doc/bootstrap.txt Page 63 for details.

RESERVED REGISTERS:

* SP points to PSP
* IX points to RSP
* DE hold IP (Interpreter Pointer)
* BC holds PSP's Top Of Stack value
* IY is the A register

The boot binary is loaded in 2 parts. The first part, "macros"
before xcomp overrides, with "301 LOAD". The rest, after xcomp
overrides, with "Z80C".

As with any boot binary, it begins with the Stable ABI (see
doc/impl.txt Page 21), all of it at this point being a placeholder.

We do things a bit differently in Z80 because we also add RST
placeholders in case we want to graft some RST handlers in
there.
```

Right after that comes the early boot code. This is the very first code being run. Initialization sequence is documented in doc/impl.txt[Page 21].

Then comes the "next" routine which is called at the end of every word execution. We can see that it:

1. Read wordref where IP currently points.
2. Continue to Execute

The execute routine begins by checking the byte where our wordref in DE points to: it's the word type. Choosing the proper behavior for the proper word type is most of the noise of this code.

PFA fiddling is central to all word types and HL holds it. We try to group word types to minimize operations, which is why alias, ialias and DOES> are lumped together (they de-reference their PFA).

Regular "compiled" words being special, it's implemented last. Note that the DOES> word "continues" to this code after having de-referenced its PFA: HL points to the right place. Then, executing the "compiled" word is as simple as:

1. Push IP to RS
2. Checks for stack overflow (if SP and IX cross) if needed. See doc/impl.txt[Page 21].
3. Set IP to PFA+2
4. De-reference PFA+0 into DE
5. Recurse into execute

chkPS: This routine is called by every word needing to pop from PS. What we do is that after we've popped everything we needed to pop, we call chkPS with the "chkPS," macro and this then verifies that SP hasn't gone over PS_ADDR. If it did, we call lbluvfl which prints "stack underflow" and ABORTs.

The undeflow method requires high level words and because we call it from very early code, it needs to be in the Stable ABI so that we can call it from its binary offset recorded in it.

The comes the native words. It's important that the first word of the dict has a 0 prev field so we can detect the end of it, which is why we muck with XCURRENT.

We only document words that aren't self-evident.

PROTECTING REGISTERS: Avoiding using IX is rather easy, but DE is sometimes hard to live without. Because we're already using the stack for PS in our words, and because so far we've never

had to use shadow registers, we use EXX, whenever we need to use DE. This way, DE is protected when we EXX, back.

FIND is the most complex of native words. It's implemented natively because otherwise, loading code from storage is really slow. Its logic goes as follow:

```
while not end-of-dict:
  if cur-entry.len ( with IMMEDIATE ANDed out ) == word.len:
    if cur-entry.name == word:
      found, push cur-entry, 1
    else:
      prev-entry
  else:
    prev-entry
else:
  not found, push word addr, 0
```

In this code, DE generally holds cur-entry, HL holds the searched word.

One oddity in this implementation is that we hold searched word "by the tail", that is, we hold the address of its last char. Because of the dict structure, it's easier to compare chars in a reverse order.

(br): When it's called IP points to the byte we need to offset our IP by. That byte is signed, so it needs to be sign-extended before it's added to IP.

(n): Literal value to push to stack is next to (n) reference in the atom list. That is where IP is currently pointing. Read, push, then advance IP.

*: The idea for DE*BC is to loop 16 times left-shifting DE. HL, which begins at 0, doubles in every loop and every time that DE carries, we add BC into the mix. For example, if BC is 3 and DE is 2, HL will stay to zero until the 15th loop, at which points it becomes 3, which is then doubled to 6 on the 16th loop. If DE was 3, then the 16 looped would have carried BC once more for a total of 9.

Carry flag management is a bit complicated here. We can't simply use the flag of the last ADDHLd. The logic is as is: if any ADDHLd carried during the loop, we have carry.

/MOD: The idea for AC /MOD DE is a bit like *. We loop 16 times with AC left-shifting and HL accumulating and at each step, we try to see if DE "fits in" HL. If it does, a 1 is added at the right of the rotating AC. If it doesn't, DE is re-added back to HL for the next loop.

For example, with AC=5 and DE=2, HL becomes 1 at 14th loop. DE fails to fit, so a 1 is not integrated to AC, but HL stays at 1. On the 15th loop, HL is doubled to 2. DE fits, so AC gets its 1, HL becomes 0. 16th loop, AC is doubled to 2, HL gets a carry, DE fails to fit. Final result: AC=2, HL=1.

3.3 8086 Boot code (`code/8086.txt`)

Let's walk through 8086 Boot code at B400. This walkthrough is a bit less detailed that the "canonical" z80 one, which is contains comments that are common to all CPUs.

This assembles the boot binary. It requires the 8086 assembler (B20) and cross compilation setup (B200). It requires some constants to be set. See doc/bootstrap.txt Page 63 for details.

In general, this code works like the Z80 boot code. We only document when it differs.

RESERVED REGISTERS:

* AX is the Work register
* SP points to PSP TOS
* BP points to RSP TOS
* DX hold IP (Interpreter Pointer)
* BX holds PSP's Top Of Stack value

Master Boot Record

So far, the only platform where the 8086 boot code is used is the PC/AT and this has the peculiarity of booting through the Master Boot Record (MBR), which you can see in arch/8086/pcat/mbr.fs. This is loaded at $7c00 on boot and does:

1. skip the next few bytes because it's the BIOS Parameter Block (BPB) and having values other than 0 there messes boot.
2. Set all segments to $800.
3. DX holds the boot drive no. Push it to SP so it can be popped at Collapse OS init.
4. Read Collapse OS binary from boot drive to memory through INT13h.
5. Jump to Collapse OS's address 0.
5. Have the proper $aa55 signature at the end of the 512 bytes block.

driveno in stable ABI

We use byte $03 in stable ABI to store the boot drive no. On startup, this boot drive has been placed on SP's TOS be the MBR and we write it to $03 so that PC/AT floppy drivers pick it up.

3.4 6809 Boot code (code/6809.txt)

Let's walk through 6809 Boot code at B280. This walkthrough is a bit less detailed that the "canonical" z80 one, which is contains comments that are common to all CPUs.

This assembles the boot binary. It requires the 6809 assembler (B50) and cross compilation setup (B200). It requires some constants to be set. See doc/bootstrap.txt Page 63 for details.

RESERVED REGISTERS:

* D is the Work register
* S points to PSP
* U points to RSP
* Y holds IP (Interpreter Pointer)

The boot binary is loaded in 2 parts. The first part, "declarations" before xcomp overrides, with the loader word 6809M. The rest, after xcomp overrides, with 6809C.

As with any boot binary, it begins with the Stable ABI (see doc/impl.txt Page 21), all of it at this point being a placeholder.

Right after that comes next and execute routines, the heart of Collapse OS' runtime. 6809 adressing mode come handy here and it allows us to have quite compact code.

In next, we can read wordref from (Y) and increase IP by 2 in a single op, then continue to exec, which expects a wordref in X.

Then, it's a matter of reading the first byte and to bit-fiddling along with conditional jumps to get to the proper logic for the word contents, which begins 1 byte after the initial X position. TFR ops used in XT and DOES are a bit expensive, but they're hardly avoidable.

Then comes the initialization code, that is, set PSP, RSP, and call BOOT from the stable ABI.

Then come the base native words. They're all straightforward and we can see that we benefit greatly from 6809's superior indexing ops. We rarely use PSH/PUL. We work directly with S because it's generally faster for what we want to do.

Sometimes, we lack register space so we use the zero page as

```
a temporary holding area (<> indexing).

FIND: something not so straightforward happens here. Unlike in
z80, we don't hold our string by the tail, so comparison happens
in "forward" mode. We even re-use code from []= for this.
String length, which is held in B, is re-used in the "length
matched!" part of the code (because, you know, it matched...).
However, to go to the beginning of the string in the dict entry,
we need LEAX to go backward, so we NEGB. However, because B hold
our reference length, we need to NEGB again afterwards.
```

3.5 6502 Boot code (code/6502.txt)

```
6502 boot code lives in arch/6502/blk.fs.

RESERVED REGISTERS:

* X is reserved for PS
* S is reserved for RS

The PS lives in the zero page (ZP) and begins at $ff, growing
downwards. X always points to it.

RS lives in the hardware stack (Page 1) and begins at $1ff.

The IP is held in the ZP at a hardcoded offset, defined by the
IPL (low) and IPH (high) xcomp constants.

Because 6502 has a peculiar way of indirectly addressing memory
(it needs a space in the ZP pointing to the target), we have
INDL (low) and INDH (high) hardcoded offsets in the ZP. These
are preceded by INDJ, which at initialization is filled with $6c
which is the opcode for an indirect jump.

Therefore, you have 2 levels of indirect jumping available to
you once you fill INDL/INDH: Jump to it with "INDL JMP[]," or
jump to where the address described in INDL/INDH points to with
"INDJ JMP,".

In native words, INDL/INDH is very often used as a regular
holding space, a second "N" register. That is often needed
because the 6502 is severly limited compared to other CPUs.

In 6502, the N register has to live in the ZP. It is thus
mandatory to override "'N" constant in xcomp, which by default
lives in SYSVARS (which is not necessarily in the ZP on 6502).
```

4 Hardware documentation

4.1 Running Collapse OS on real hardware (hw/intro.txt)

```
Collapse OS is designed to run on ad-hoc post-collapse hardware
```

build from scavenged parts. These machines don't exist yet.

To make Collapse OS as likely as possible to be useful in a post-collapse world, we try to give as many examples as possible of deployment on hacked-up hardware.

For example, we include a recipe for running a Sega Master System with a PS/2 keyboard plugged to a hacked up controller cord with an AVR MCU interfacing between the PS/2 connector and the controller port.

This setup, for which drivers are included in Collapse OS, exist in only one copy, the copy the author of the recipe made.

However, the idea is that this recipe, which contains schematics and precise instructions, could help a post-collapse engineer to hack her way around and achieve something similar. She would then have a good example of schematics and drivers that are known to work.

Organisation of this folder

While /doc's top folder contain documentation about software, this folder contains instructions and schematics about ways to get Collapse OS running on actual hardware.

Each CPU architecture has its own subfolder with recipes about specific machines of that arch, while /doc/hw's top folder contain instructions on broader topics, such as SD cards, floppies, EEPROM, etc.

Most instructions have companion code in /arch that is conveniently wrapped in Makefiles for easy building.

How to use

If you want to run Collape OS on real hardware, browse this folder's contents until you find something that closely matches your own hardware (or hardware-to-be).

If you live in a pre-collapse world and are looking for an easy platform to try Collapse OS on, easy pickings are PC/AT (which run on modern PCs supporting legacy BIOS), Sega Genesis w/ Everdrive and TI-84+. Those options don't require any soldering.

Drivers

Most instructions in this subfolder tell you to add drivers to your Collapse OS. What is meant by this is that you need to rebuild your binary with an augmented xcomp unit. See doc/bootstrap for details, but the short version is that you'll want to load your driver code between the COREL call and subsystems. Then, if your driver needs initialization, then you'll add it to INIT.

4.2 Asynchronous Communications Interface Adapters (hw/acia.txt)

Machines talking to each other is generally useful and they often use ACIA devices to do so. Collapse OS has drivers for a few chips of this type and they all implement those words:

```
TX>  c --    Send char c through the device
RX<? c? f -- Poll device for character
```

The rest of the implementation is device-specific, but those two words are enough for applications like the Remote Shell and the XMODEM implementation to work.

Flow control

All drivers in Collapse OS have a similar approach: unbuffered communication using RTS/CTS handshaking as flow control.

The reason for being unbuffered is simplicity and RAM. The logic to implement input buffering is non-trivial and, alone, doesn't buy us much in terms of reliability: you still have to signal the other side when your buffer is nearly full.

Because we don't really need speed, we adopt a one-byte-at-once approach: The RTS flag is always high (signalling that it's not ready for communication) *except* when calling the ACIA driver's "read" word, which is blocking.

That "read" word will pull RTS low, wait for a byte, then pull it high again.

This slows down communication, but it's simple and reliable.

Note that this doesn't help making communications with modern systems (which are much faster than a typical Collapse OS machine and have their buffer output faster than the RTS flag can be raised) very much. We have to take extra care, when communicating from modern system, not to send too much data too fast. But for COS-to-COS communication, this simple system works.

Broken hardware

Some designs are broken with this scheme. For example, the RS2014 SIO module hard-wires CTS to GND because the FTDI connector doesn't have such a pin (modern computers can always handle the load).

In these cases, a solution would be to use Break signals as a workaround, but I prefer avoiding complexity for now. So when you deal with broken design, you'll have to sidestep it either by implementing your own Break handling or by lowering communication speed.

4.3 Writing to a AT28 from Collapse OS (`hw/at28.txt`)

Gathering parts

* A RC2014 Classic
* An extra AT28C64B
* 1x 40106 inverter gates
* Proto board, RC2014 header pins, wires, IC sockets, etc.

Building the EEPROM holder

The AT28 is SRAM compatible so you could use a RAM module for it. However, there is only one RAM module with the Classic version of the RC2014 and we need it to run Collapse OS.

You could probably use the 64K RAM module for this purpose, but I don't have one and I haven't tried it. For this recipe, I built my own module which is the same as the regular ROM module but with WR wired and geared for address range $2000-$3fff.

If you're tempted by the idea of hacking your existing RC2014 ROM module by wiring WR and write directly to the range $0000-$1fff while running it, be aware that it's not that easy. I was also tempted by this idea, tried it, but on bootup, it seems that some random WR triggers happen and it corrupts the EEPROM contents. Theoretically, we could go around that by putting the AT28 in write protection mode, but I preferred building my own module.

I don't think you need a schematic. It's really simple.

Writing contents to the AT28

If you wait 10ms between each byte you write, you can write directly to the AT28 with regular memory access words. If you don't

wait, the AT28 writing program will fail. Because it's not very pratical to insert waiting time between each byte writes, you need another solution.

B321 contains an override routine called AT28$. When you call this, It defines new "C!" and "!" words and those words ensure that data is properly written to EEPROM before returning.

Note that because it's new definitions for "C!" and "!", these are only going to work for direct execution or for words defined after you've called "AT28$".

When you're done writing to the AT28, you can unset the override with "FORGET C!".

When polling, AT28 routines also verifies that the final byte in memory is the same as the byte written. If it's not, it will place a non-zero value in the IOERR 1b variable. Therefore, if you want to see, after a big write operation to your AT28, whether any write failed, do "IOERR C@ .". Re-initialize to zero before your next write operation.

4.4 Making an ATmega328P blink (hw/avr.txt)

Collapse OS has an AVR assembler and an AVR programmer. If you have a SPI relay (see doc/hw/spi.txt Page 105), then you almost have all
it takes to make an ATmega328P blink.

First, read doc/avr.txt Page 55. You'll see that it tells you how to build an AVR programmer that works with your SPI relay. You might already have such device. For example, I use the same device as the one I connect to my Sparkfun AVR Pocket Programmer, but I've added an on/off switch to it. I then use a 6-pin ribbon cable to connect it to my SPI relay.

If you have a SD card connected to the same SPI relay, you'll face a timing challenge: SD specs specifies that the minimum SPI clock is 100kHz, but depending on your setup, you might end up with an effective SCK below that. My own clock setup looks like this:

I have a RC2014 Dual clock which allows me to have easy access to many clock speeds, but the slowest option is 300kHz, not slow enough. My SPI relay has a pin for input clock override, and I built a pluggable 4040 with a switch that selects a divisor. I plug that module in my SPI relay, then I plug that into my RC2014 Dual clock. When doing SD card stuff, I select the "no division" position, and when I communicate with the AVR chip, I move the switch to increase the divisor.

Once you've done this, you can test that you can communicate with your AVR chip by doing "160 163 LOADR" (turn off your programmer or else it might mess up the SPI bus and prevent you from using your SD card) and then running:

```
1 asp$ aspfl@ .x 0 (spie)
```

(Replace "1" by your SPI device ID) If everything works fine, you'll get the value of the low fuse of the chip.

Building the blink binary

A blink program for the ATmega328P in Collapse OS would look like this:

```
50 LOAD ( avra ) 65 66 LOADR ( atmega328p ) H@ ORG !
DDRB 5 SBI, PORTB 5 CBI,
R16 TCCR0B IN, R16 $05 ORI, TCCR0B R16 OUT,
R1 CLR,
L1 LBL! ( loop )
    R16 TIFR0 IN,
    R16 0 ( TOV0 ) SBRS,
        L1 ( loop ) ' RJMP LBL, ( no overflow )
    R16 $01 LDI, TIFR0 R16 OUT,
    R1 INC,
    PORTB 5 CBI,
    R1 7 SBRS,
        PORTB 5 SBI,
    L1 ( loop ) ' RJMP LBL,
```

See doc/asm.txt Page 90 for details. For now, you'll paste this into an arbitrary unused block. Let's use 999.

```
$ cd arch/z80/rc2014
$ xsel > blk/999
$ rm blkfs
$ make
$ dd if=blkfs of=/dev/<your-sdcard> bs=1024
```

Now, with your updated SD card in your RC2014, let's assemble this binary:

```
999 LOAD
H@ CREATE end ,
CREATE wordcnt end ORG - 2 / ,
: write 1 asp$ asperase wordcnt 0 DO
ORG I 2 * + @ I aspfb! LOOP
0 aspfp! 0 (spie) ;
write
```

The first line assembles a 16 words binary beginning at ORG, then the rest of the lines are about writing these 16 words to the AVR chip (see doc/avr.txt Page 55 for details). After you've run this, if everything went well, that chip if it has a LED attached to PB5, will make that LED blink slowly.

4.5 Remote access to Collapse OS (hw/tty.txt)

If you interface to your machine through a serial communication device and that you have a POSIX environment on the other side, Collapse OS provides tools in /tools which can be very useful to you.

Uploading data to Collapse OS' memory is a frequent need and /tools/upload can help you there.

See details in the /tools folder directly.

4.6 Accessing SD cards (sdcard.txt)

SD cards support the SPI protocol. If you have a SPI relay and a driver for it that implement the SPI protocol (doc/spi), you're a few steps away from accessing SD cards!

What you need to do is to add the SDC subsystem to your Collapse OS binary. First, define SDC_DEVID to a mask selecting the proper device on your SPI relay (this is what is sent to "(spie)"). For example, a SDC_DEVID or 1, 2, 4, or 8 would select SPI device 1, 2, 3 or 4.

The subsystem is loaded with "250 258 LOADR".

Once that subsystem is loaded, you need to create aliases that will plug into the BLK subsystem (doc/blk). Add this to your xcomp:

```
ALIAS SDC@ (blk@)
ALIAS SDC! (blk!)
```

You can now load BLKSUB and end the rest of your xcomp normally.

At runtime, the SD card that was inserted needs to be initialized. You can do it with SDC$. If you have no error, it means that the system can spek to your card, that sync is fine, etc. You can read/write right now. SDC$ needs to run every time a new card is inserted.

Collapse OS' SDC drivers are designed to read from the very first 512 sector of the card, mapping them to blocks sequentially, 2 sectors per block.

4.7 Communicating through SPI (`spi.txt`)

Many very useful device are able to communicate through the SPI protocol, for example, SD cards and AVR MCUs. In many cases, however, CPUs can't "speak SPI" because of their inability to bit-bang.

In most cases, we need an extra peripheral, which we can build ourselves, to interface with devices that "speak SPI". We call this peripheral a SPI relay.

The design of those relays depend on the CPU architecture. See spi.txt Page 105 in arch-specific folders for more information.

SPI Relay protocol

This protocol enables communication with a SPI relay. This protocol is designed to support devices with multiple endpoints. To that end, (spie) takes a device ID argument, with a meaning that is up to the device itself. To disable all devices, supply 0 to (spie).

We expect relay devices to support only one enabled device at once. Enabling a specific device is expected to disable the previously enabled one.

```
(spie)    n --     Enable SPI device
(spix)    n -- n   Perform SPI exchange (push a number, get a
                   number back)
```

There is no SPI subsystem, but other subsystems depend on the SPI protocol being fulfilled:

* SD Card subsystem (doc/sdcard)

5 Hardware: z80 hardware interfaces

5.1 Interfacing a PS/2 keyboard (`hw/z80/ps2.txt`)

Collapse OS needs a way to input commands and keyboards are one of the most straightforward ways to proceed. The PS/2 protocol is very widespread and relatively simple.

We explain here how to interface a PS/2 keyboard with a RC2014.

Gathering parts

* A RC2014 Classic that could install the base recipe
* A PS/2 keyboard. A USB keyboard + adapter also works, if it's not too recent (if it still speaks PS/2).

* A PS/2 female connector.
* ATtiny85/45/25 (main MCU for the device)
* 74xx595 (shift register)
* 40106 inverter gates
* Diodes for A*, IORQ, RO.
* Proto board, RC2014 header pins, wires, IC sockets, etc.
* AVRA (https://github.com/hsoft/avra). The code for this recipe hasn't been translated to Collapse OS' AVR assembler yet.

Building the PS/2 interface

Let's start with the PS/2 connector (see img/ps2-conn.png[Page 107]), which has two pins.

Both are connected to the ATtiny45, CLK being on PB2 to have INT0 on it.

The DATA line is multi-use. That is, PB1 is connected both to the PS/2 data line and to the 595's SER. This saves us a precious pin.

The ATtiny 45 (img/ps2-t45.png[Page 108]) hooks everything together. CE comes from the z80 bus (img/ps2-z80.png[Page 109]).

The 595 (img/ps2-595.png[Page 108]) allows us to supply the z80 bus with data within its 375ns limits. SRCLR is hooked to the CE line so that whenever a byte is read, the 595 is zeroed out as fast as possible so that the z80 doesn't read "false doubles".

The 595, to have its SRCLR becoming effective, needs a RCLK trigger, which doesn't happen immediately. It's the ATtiny45, in its PCINT interrupt, that takes care of doing that trigger (as fast as possible).

Our device is read only, on one port. That makes the "Chip Enable" (CE) selection rather simple. In my design, I chose the IO port 8, so I inverted A3. I chose a 40106 inverter to do that, do as you please for your own design.

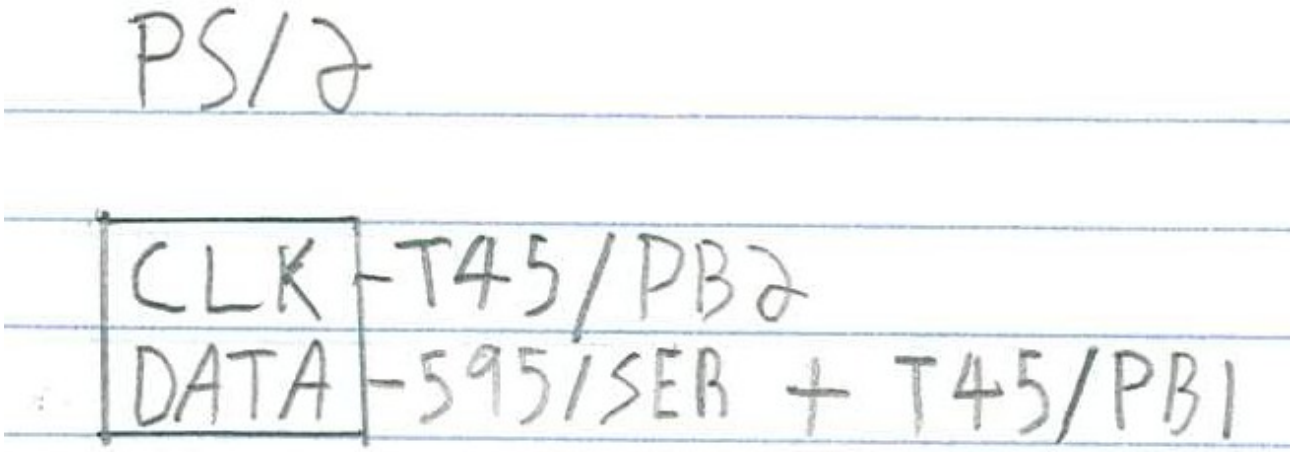
I wanted to hook CE to a flip flop so that the MCU could relax a bit more w.r.t. reacting to its PB4 pin changes, but I didn't have NAND gates that are fast enough in stock, so I went with this design. But otherwise, I would probably have gone the flip-flop way. Seems more solid.

Then, all you need to do is to assemble code/ps2ctl.asm and load it onto your ATtiny.

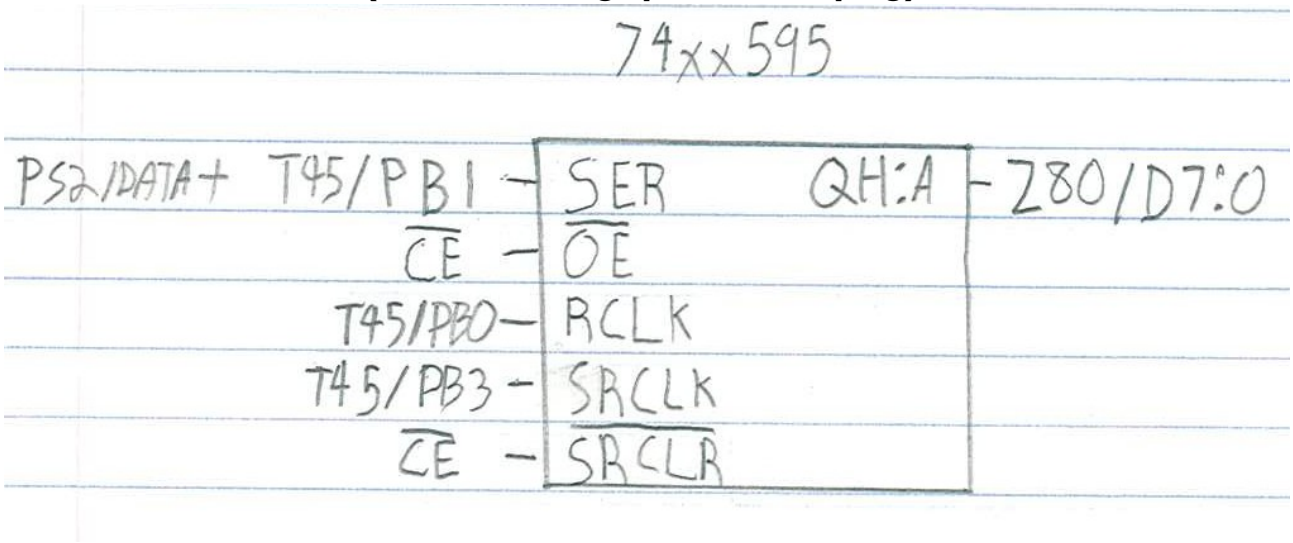
Using the PS/2 interface

```
To use this interface, you have to build a new Collapse OS
binary. This binary needs two things.

First, we need a "(ps2kc)" routine (see doc/ps2). In
this case, it's easy, it's ": (ps2kc) 8 PC@ ;". Then, we can
load PS/2 subsystem. You add "411 414 LOADR". Then, at
initialization, you add "PS2$".  You also need to define PS2_MEM
at the top. You can probably use "SYSVARS + $aa".

The PS/2 subsystem provides "(key)" from "(ps2kc)".

For debugging purposes, you might not want to go straight to
plugging PS/2 "(key)" into the system. What I did myself was to
load the PS/2 subsystem *before* ACIA (which overrides with its
own "(key)") and added a dummy word in between to access PS/2's
key.
```

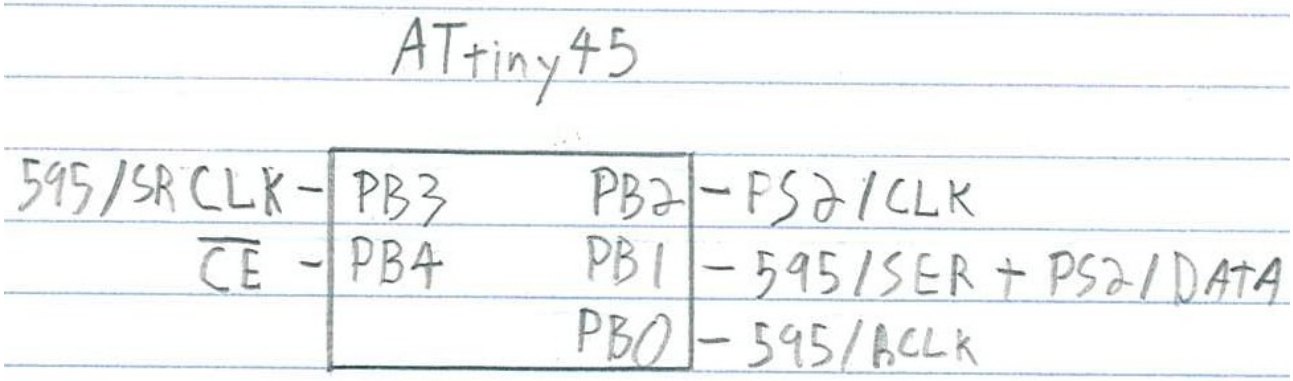
5.2 PS/2 Connector (`hw/z80/img/ps2-conn.png`)



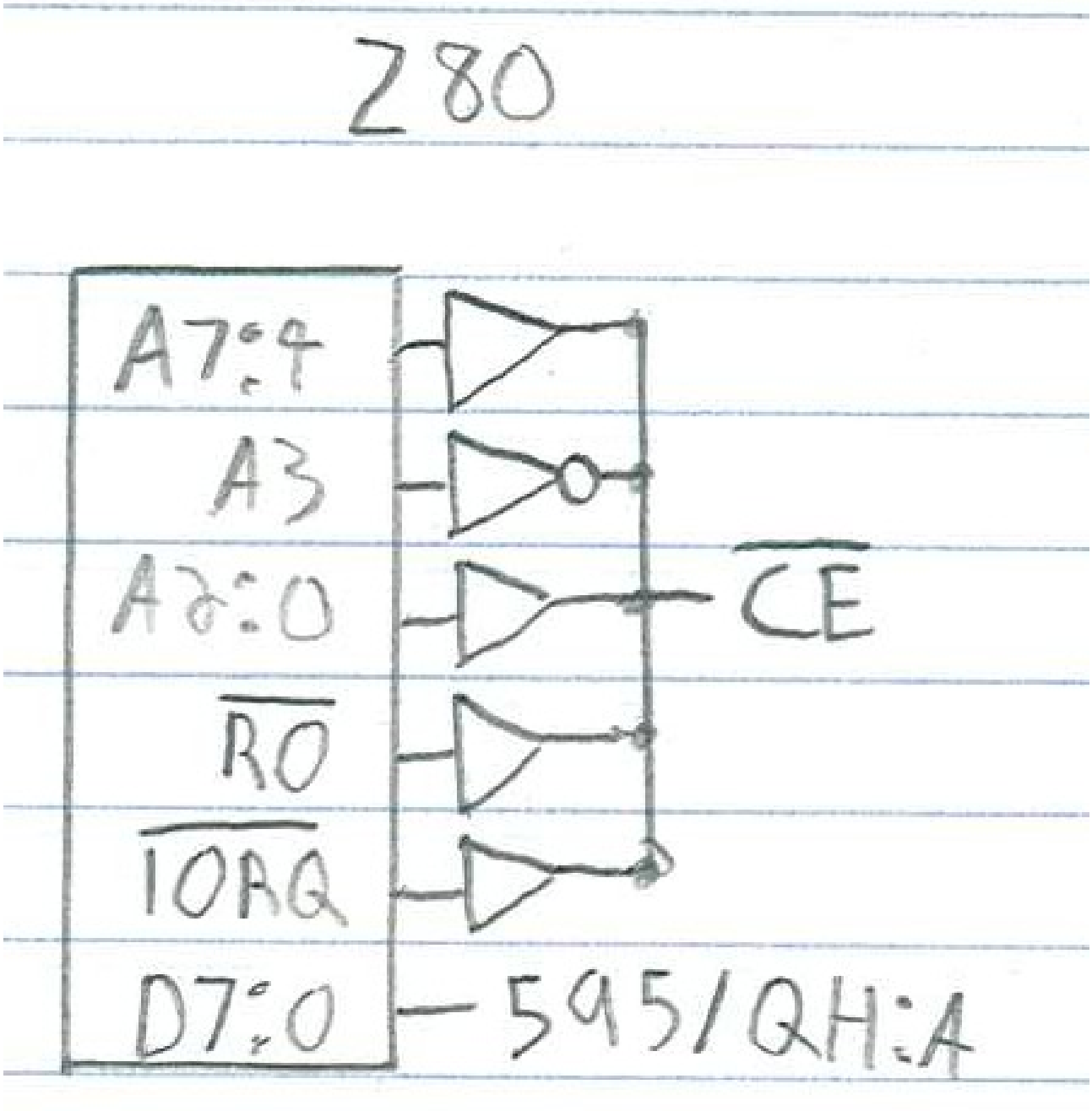
5.3 PS/2 74xx595 (hw/z80/img/ps2-595.png)



5.4 PS/2 ATtiny45 (hw/z80/img/ps2-t45.png)



5.5 PS/2 Z80 (hw/z80/img/ps2-z80.png)



5.6 Building a SPI relay for the z80 (hw/z80/spi.txt)

In this recipe, we build a SPI relay (see /doc/hw/spi.txt Page 105) for a RC2014.

Gathering parts

```
* A RC2014 Classic
* A proto board + header pins with 39 positions so we can make
  a RC2014 card.
* Diodes, resistors and stuff
```

* 40106 (Inverter gates)
* 74xx138 (Decoder)
* 74xx375 (Latches)
* 74xx125 (Buffer)
* 74xx161 (Binary counter)
* 74xx165 (Parallel input shift register)
* 74xx595 (Shift register)

Building the SPI relay

The schematic (img/spirelay.jpg[Page 111]) works well with the SD Card subsystem (B420). Of course, it's not the only possible design that works, but I think it's one of the most straighforwards.

This relay communicates through the z80 bus with 2 ports, DATA and CTL and allows up to 4 devices to be connected to it at once, although only one device can ever be active at once. This schema only has 2 (and the real prototype I've built from it), but the '375 has room for 4. In this schema, DATA is port 4, CTL is port 5.

We activate a device by sending a bitmask to CTL, this will end up in the '375 latches and activate the SS pin of one of the device, or deactivate them all if 0 is sent.

You then initiate a SPI exchange by sending a byte to send to the DATA port. This byte will end up in the '165 and the '161 counter will be activated, triggering a clock for the SPI exchange. At each clock, a bit is sent to MOSI from the '161 and received from MISO into the '595, which is the byte sent to the z80 bus when we read from DATA.

When the '161 is wired to the system clock, as it is in the schema, two NOPs are a sufficient delay between your DATA write and subsequent DATA read.

However, if you build yourself some kind of clock override and run the '161 at something slower than the system clock, those 2 NOPs will be too quick. That's where that '125 comes into play. When reading CTL, it spits RUNNING into D0. This allows you to know when the result of the SPI exchange is ready to be fetched. Make sure you AND away other bits, because they'll be garbage.

The '138 is to determine our current IORQ mode (DATA/CTL and WR/RO), the '106 is to provide for those NOTs sprinkled around.

Please note that this design is inspired by https://www.ecstaticlyrics.com/electronics/SPI/fast_z80_interface.html

Advice 1: Make SCK polarity configurable at all 3 endpoints (the 595, the 165 and SPI connector). Those jumpers will be useful when you need to mess with polarity in your many tinkering sessions to come.

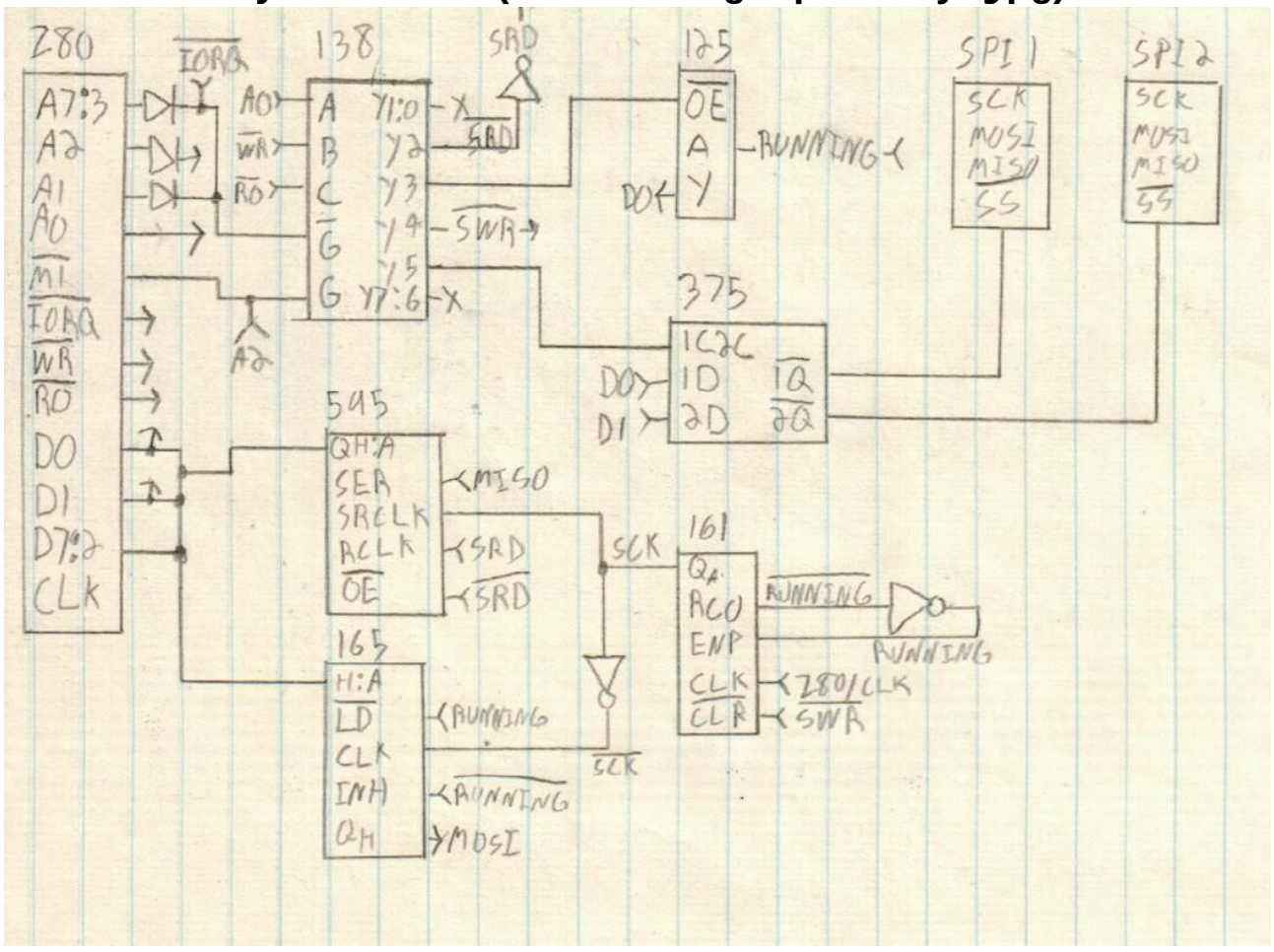
Advice 2: Make input CLK override-able. SD cards are plenty fast enough for us to use the system clock, but you might want to interact with devices that require a slower clock.

Driving the relay

There is a provider for the SPI protocol (doc/spi) that works with this device in B312. It needs SPI_DATA and SPI_CTL constants which in this case are 4 and 5 respectively.

When writing to SPI_CTL, we expect a bitmask of the device to select, with 0 meaning that everything is de-selected. Reading SPI_CTL returns 0 if the device is ready or 1 if it's still running an exchange. Writing to SPI_DATA initiates an exchange.

5.7 SPI Relay Schematic (hw/z80/img/spirelay.jpg)



5.8 Using Zilog's SIO as a console (hw/z80/sio.txt)

The RC2014 has an optional module called the Dual Serial Module SIO/2 which is built around Zilog's SIO chip. This module is nice because when paired with the Dual Clock Module and when using port B, it's possible to run a UART with a baud rate lower than 115200.

Collapse OS has a driver for it (although for now, only port A is supported by it). Let's use it.

* Let's assume a xcomp unit similar to the one in /arch/z80/rc2014.
* Locate SIO driver in /arch/z80/rc2014/blk
* The driver main page gives you references for declarations and for code.
* In the base xcomp unit, replace ACIA declarations with SIO's
* Replace ACIA code with SIO's
* At the bottom, replace "ACIA$" with "SIO$".

Rebuild the binary and you're done. "(key)" and "(emit)" will go through the SIO.

6 Hardware: Sega Master System (z80 based)

6.1 Sega Master System (hw/z80/sms/intro.txt)

The Sega Master System was a popular gaming console running on z80. It has a simple, solid design and, most interestingly of all, its even more popular successor, the Megadrive (Genesis) had a z80 system for compatibility!

This makes this platform *very* scavenge-friendly and worth working on.

SMS Power[1] is an awesome technical resource to develop for this platform and this is where most of my information comes from.

This platform is tight on RAM. It has 8k of it. However, if you have extra RAM, you can put it on your cartridge.

Gathering parts

* A Sega Master System or a MegaDrive (Genesis).
* A Megadrive D-pad controller.
* A way to get an arbitrary ROM to run on the SMS. Either through a writable ROM cartridge or an Everdrive[2].

Hacking up a ROM cart

SMS Power has instructions to transform a ROM cartrige into a battery-backed SRAM one, which allows you to write to it through another device you'll have to build. This is all well and good, but if you happen to have an AT28 EEPROM, things are much simpler!

Because AT28 EEPROM are SRAM compatible, they are an almost-drop-in replacement to the ROM you'll pop off your cartridge. AT28 are a bit expensive, but they're so handy! For SMS-related stuff, I recommend the 32K version instead of the 8K one because fitting Collapse OS with fonts in 8K is really tight.

The ROM cartridge follow regular ROM pinout, which means that A14 are just under VCC, where WE is on the AT28. We need WE to be perma-disabled and A14 to be properly connected.

1. De-solder the ROM
2. Take a 28 pins IC socket
3. Cut off its WE pin (the one just under VCC), leaving a tiny bit of metal.
4. Hard-wire it to VCC so that WE is never enabled.
5. Solder your socket where the ROM was.
6. With a cutter, cut the trace leading to A14.
7. Wire A14 to the trace just under WE (which doesn't actually touch WE because we've cut the IC socket's pin).
8. Insert Collapse OS-filled EEPROM in socket.

As simple as this! (Note that this has only been tested on a SMS so far. I haven't explored whether this can run on a megadrive).

Build the ROM

Running "make" in /arch/z80/sms will produce a "os.sms" ROM that can be put as is on a SD card to the everdrive or flashed as is on a writable ROM cart. Then, just run the thing!

To run Collapse OS in a SMS emulator, run "make emul".

Usage

Our input is a D-Pad and our output is a TV. The screen is 32x28 characters. A bit tight, but usable.

D-Pad is used as follow:

* There's always an active cursor. On boot, it shows "a".
* Up/Down increase/decrease the value of the cursor.
* Left/Right does the same, by increments of 5.
* A button is backspace.
* B button skips cursor to next "class" (number, lowcase, upcase, symbols).
* C button "enters" cursor character and advance the cursor by one.
* Start button is like pressing Return.

Of course, that's not a fun way to enter text, but using the D-Pad is the easiest way to get started which doesn't require soldering. Your next step after that would be to build a PS/2 keyboard adapter! See smsps2.txt Page 115

[1]: http://www.smspower.org
[2]: https://krikzz.com

6.2 Writing to a AT28 from a SMS (hw/z80/sms/at28.txt)

Writing on the EEPROM that is currently running Collapse OS is as easy as enabling the WE pin on your hacked up cartridge. However, this is not practical: If you want to deploy Collapse OS (or something else) to another machine, or even if you want to upgrade your current Collapse OS, you will likely want to write to another EEPROM.

The easiest way to do so is to build yourself a dual EEPROM cartridge. It's very similar to a simple cartridge, except it has two AT28 sockets and a '139 decoder to select between the two.

The design proposed here sacrifices access to the upper 16K of your AT28C256 for the sake of simplicity because it uses A14 as the chip selector. Therefore, addrs $0000-$3fff belong to the first chip and $4000-$7fff belong to the second.

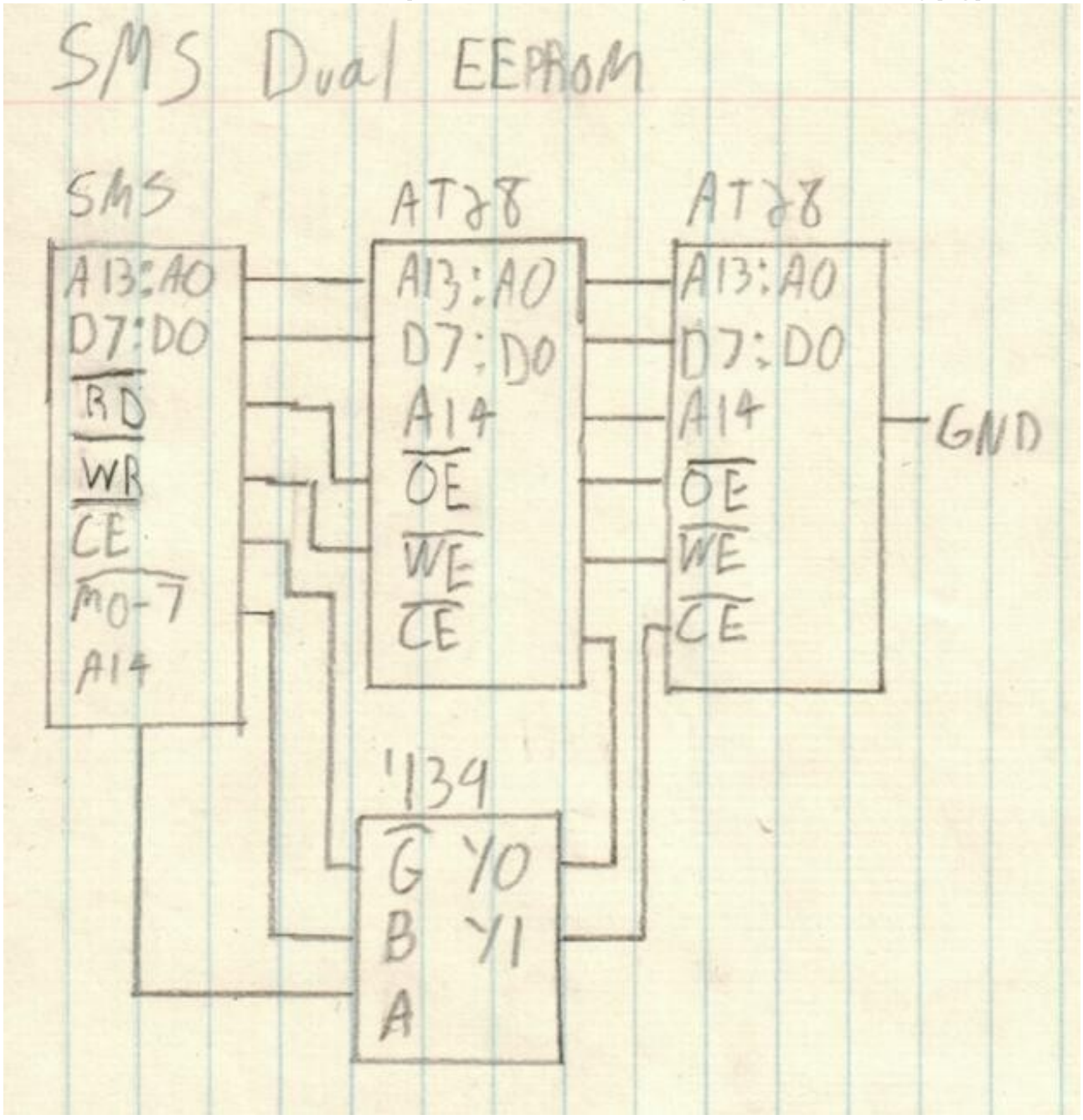
You can see the schematic in dual-at28.jpg Page 115.

The schematic enables WE on both EEPROMs, but in my actual prototype, I hard-wired the first chip's WE to high because I never want to write to it, despite bugs I might introduce in hardware or software (I try a lot of dangerous stuff on my machines...).

On top of that, you will likely want to add a physical CE-inhibit jumper (a jumper hard-wired to VCC) on the AT28 socket. The reason for this is that if the EEPROM you have on your socket ends with a SEGA TMR signature, it will be a wrong one, but it will still be picked up by the BIOS and Collapse OS will refuse to boot. A CE-inhibit switch that you can remove after

boot will solve the problem.

6.3 SMS Dual EEPROM (hw/z80/sms/img/dual-at28.jpg)



6.4 PS/2 keyboard on the SMS (hw/z80/sms/ps2.txt)

Using the shell with a D-pad on the SMS is doable, but not fun at all! We're going to build an adapter for a PS/2 keyboard to plug as a SMS controller.

The PS/2 logic will be the same as the regular PS/2 adapter (see doc/hw/ps2.txt Page 105) but instead of interfacing directly with the bus, we interface with the SMS' controller subsystem (that is, what we poke on ports $3f and $dc).

How will we achieve that? A naive approach would be "let's limit ourselves to 7bit ASCII and put TH, TR and TL as inputs". That could work, except that the SMS will have no way reliable way (except timers) of knowing whether polling two identical values is the result of a repeat character or because there is no new value yet.

On the AVR side, there's not way to know whether the value has been read, so we can't do like on the RC2014 and reset the value to zero when a RO request is made.

We need communication between the SMS and the PS/2 adapter to be bi-directional. That bring the number of usable pins down to 6, a bit low for a proper character range. So we'll fetch each character in two 4bit nibbles. TH is used to select which nibble we want.

TH going up also tells the AVR MCU that we're done reading the character and that the next one can come up.

As always, the main problem is that the AVR MCU is too slow to keep up with the rapid z80 polling pace. In the regular adapter, I hooked CE directly on the AVR, but that was a bit tight because the MCU is barely fast enough to handle this signal properly. I did that because I had no proper IC on hand to build a SR latch.

In this recipe, I do have a SR latch on hand, so I'll use it. TH triggering will also trigger that latch, indicating to the MCU that it can load the next character in the '164. When it's done, we signal the SMS that the next char is ready by resetting the latch. That means that we have to hook the latch's output to TR.

Nibble selection on TH doesn't involve the AVR at all. All 8 bits are pre-loaded on the '164. We use a 4-channel multiplexer to make TH select either the low or high bits.

Gathering parts

* A SMS that can run Collapse OS
* A PS/2 keyboard. A USB keyboard + PS/2 adapter should work, but I haven't tried it yet.
* A PS/2 female connector.
* A SMS controller you can cannibalize for the DB-9 connection. A stock DB-9 connector isn't deep enough.
* ATtiny85/45/25 (main MCU for the device)
* 74xx164 (shift register)
* 74xx157 (multiplexer)
* A NOR SR-latch. I used a 4043.
* Proto board, wires, IC sockets, etc.

Historical note

As I was building this prototype, I was wondering how I would debug it. I could obviously not hope for it to work as a keyboard adapter on the first time, right on port A, driving the shell. I braced myself mentally for a logic analyzer session and some kind of arduino-based probe to test bit banging results.

And then I thought "why not use the genesis?". Sure, driving the shell with the D-pad isn't fun at all, but it's possible. So I hacked myself a temporary debug kernel with a "a" command doing a probe on port B. It worked really well!

It was a bit less precise than logic analyzers and a bit of poking-around and crossing-fingers was involved, but overall, I think it was much less effort than creating a full test setup.

There's a certain satisfaction to debug a device entirely on your target machine...

Building the PS/2 interface

See schematic at img/ps2-to-sms.png Page 118. The PS/2-to-AVR part is identical to doc/hw/ps2.txt Page 105.

We control the '164 from the AVR in a similar way to what we did in rc2014/ps2, that is, sharing the DATA line with PS/2 (PB1). We clock the '164 with PB3. Because the '164, unlike the '595, is unbuffered, no need for special RCLK provisions.

Most of the wiring is between the '164 and the '157. Place them close. The 4 outputs on the '157 are hooked to the first 4 lines on the DB-9 (Up, Down, Left, Right).

In my prototype, I placed a 1uf decoupling cap next to the AVR. I used a 10K resistor as a pull-down for the TH line (it's not always driven).

If you use a 4043, don't forget to wire EN. On the '157, don't forget to wire ~G.

The code expects a SR-latch that works like a 4043, that is, S and R are triggered high, S makes Q high, R makes Q low. R is hooked to PB4. S is hooked to TH (and also the A/B on the '157). Q is hooked to PB0 and TL.

Building the firmware

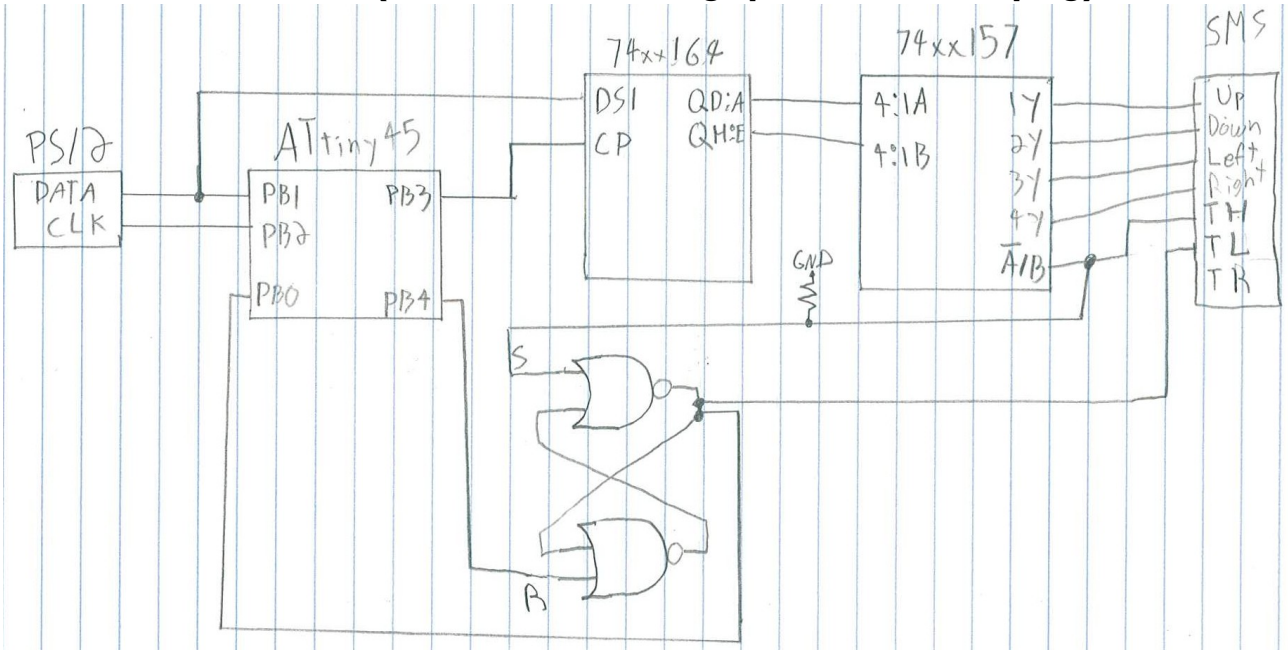
The code for the ATtiny is in B501. It is built with the AVR assembler (doc/asm/avr.txt Page 90). Once built, the binary begins at ORG and can be sent to the ATtiny using the AVR programmer (doc/avr.txt Page 55).

Building the binary

You build the binary in the same way as with the regular SMS, but use xcompkbd.fs instead of xcomp.fs (in arch/z80/sms).

The xcomp is for a keyboard plugged on port A. For port B, replace (ps2kcA) with (ps2kcB).

6.5 PS/2 interface (hw/z80/sms/img/ps2-to-sms.png)



6.6 SMS pad (hw/z80/sms/pad.txt)

There is a driver for getting (key?) input from a SMS pad at B335.

It conveniently exposes an API to read the status of a SMS pad on port A. Moreover, implement a mechanism to input arbitrary characters from it. It goes as follow:

* Direction pad select characters. Up/Down move by one, Left/Right move by 5
* Start acts like Return
* A acts like Backspace
* B changes "character class": lowercase, uppercase, numbers, special chars. The space character is the first among special

```
  chars.
* C confirms letter selection

This module needs CELL! (see doc/grid.txt Page 67) to display
selection
on screen during (key?).
```

_status (-- n)

```
Returns a status bitmask for port A. Bits, from MSB to LSB:
Start - A - C - B - Right - Left - Down - Up
Each bit is high when button is unpressed and low if button is
pressed. When no button is pressed, $ff is returned.
This logic below is for the Genesis controller, which is modal.
TH is an output pin that switches the meaning of TL and TR. When
TH is high (unselected), TL = Button B and TR = Button C. When
TH is low (selected), TL = Button A and TR = Start. )
```

6.7 Building a SPI relay for the SMS (hw/z80/sms/spi.txt)

```
The I/O space on the SMS is, sadly, entirely taken. If you had
the idea of somehow plugging a SPI relay that is similar the one
on the RC2014, you can forget about it. Only A7, A6 and A0 are
considered by the 8 builtin peripherals on the SMS and trying to
do an IN or OUT to any address is going to end up conflicting
with one of them.

What we can do to achieve SPI communication with the SMS is to
use the B controller port. It can already do bit banging. It's
slow, but it works.

One problem we have, however, is that only 2 pins can be set as
output. We need 3. What I did, and it works with SD cards, is
to hard-wire CS to GND so that it's always turned on. The
downside of this is that if you go out-of-sync with the SPI
device, you have to physically disconnect it and reconnect it
to solve the sync problem.

The advantage of using port B is that the connector is really
simple, you don't even need a schematic:

* CLK to TH
* DI to TR
* DO to Up
* CS to GND

Add pull-downs to CLK and DI to avoid messing up with your
device (it's always on, remember).
```

Building the binary

The SPI driver is in B622, which depends on controller port routines at B625-B626. A ready-to-use xcomp unit is at arch/z80/sms/xcompsdc.fs.

The SMS emulator has support for a SPI relay based on the B controller port and can emulate a SD card plugged in it with the "-c" argument. If it works in the emulator, it has good chances of running on the real thing.

6.8 VDP driver (hw/z80/sms/vdp.txt)

The driver for the SMS VDP lives at B330. It requires code from the TMS9918 driver as well as a 7x7 font tied to a ~FNT dict entry.

It takes care of properly initializing Mode 4 and of sending the font to VDP's memory in a way that it will understand. It does so through the _sfont word, which works like this:

Each row in ~FNT is a row of the glyph and there is 7 of them. We insert a blank one at the end of those 7. For each row we set, we need to send 3 zero-bytes because each pixel in the tile is actually 4 bits because it can select among 16 palettes. We use only 2 of them, which is why those bytes always stay zero.

7 Hardware: Other z80 based devices

7.1 Dan's Z80 Single Board Computer (hw/z80/dan.txt)

This single board computer is a project created by Daniel Marks that can be found on the github respository:

http://www.github.com/profdc9/Z80SBC

A copy of its schematic is in img/dan.pdf

It is based on Grant Searle's CP/M Z80SBC and is intended to use components that should be available, along with the Z80, long into the future. Exotic parts such as uncommon LSTTL (e.g. LSTTL670) are eschewed here. The parts needed are:

1. 1 X Z84C00 (7.3728 MHz, 6 MHz may be overclocked, or 8 to 20 MHz versions)
2. 1 X Z80 DART or SIO/2. The SIO/0 or SIO/1 can be adapted as well, however, the pinouts for the DART and SIO/2 are shown.
3. 1 X 82C55 peripheral interface. An extremely common general purpose IO chip used in the IBM PC as well as countless ISA adapter cards.
4. 3 X 74HCT32/74LS32 or gates (probably 74HC32 would work in a

pinch with other CMOS derivative parts)
5. 2 X 74HC00
6. 1 X 74LS138/74HCT138
7. 1 X HM628128 or AS6C1008 128k X 8 static RAM. A 62256 (32k X 8) could be adapted as well if the lower part of memory is set to $8000.
8. 1 X 28C256 32k X 8 flash ROM. 27256 ROMs may be available as scrap BIOS chips from old PCs or ISA cards as well, but if its UV erase you may have to get creative.
9. 1 X 74HC595 serial in, parallel out shift register
10. 1 X 74HC165 parallel in, serial out shift register
11. 1 X 74HC393 ripple counter
12. 1 X MAX232 for logic level to RS232 conversion

At boot time, the 28C256 ROM is mapped as well as 32k/48k of the RAM. When the IO address $38 is written to, the ROM is mapped out and the entire memory address space is 64k RAM. The device includes two serial ports, a SPI port for supporting two SD cards and other SPI hardware, a CompactFlash port, a composite video output port, and a PS/2 keyboard input.

The IO map is:

```
$00-$03  SIO/DART
$10-$17  CompactFlash port
$18-$1B  8255 Port
$20      SPI input/output
$38      Disable ROM
```

There is a PS/2 keyboard port with the clock line of the PS/2 keyboard wired to the SYNCA/RIA input of the SIO/DART. An interrupt is triggered on the falling edge of the PS/2 clock line, and an interrupt routine assembles input byte codes from the keyboard to be passed to the operating system. The PS/2 data line is wired to PC7 on the 8255.

The SPI input/output circuit operates simply by writing a byte to port $20, and reading the input byte also from port $20. The chip select lines for the SD cards are PC0 and PC1 on the 8255. The SPI may be clocked either by the CPU clock or by a second clock, for example, at 16 MHz. The SPI circuit is also used to implement the software composite video output. If it is used for video, the processor and SPI must both be clocked at 7.3728 MHz.

Other pins on the 8255 are left uncommitted and may be used, for example, to implement a ROM programmer or interface to other external hardware.

A makeshift composite video output is implemented through software on the Z80. The MOSI pin of the SPI (74HC165) shifts out pixel data, and the PC3 pin on the 8255 is used to generate

the SYNC signal. The interface to the video is extremely simple being just a couple of resistors. The output video is monochrome NTSC format with 262 lines per frame, 60 frames per second, with 246 lines of picture and 16 lines of vertical blanking at a horizontal scan rate of 15.752 kHz. Because the Z80 is used to scan the video, the video scanning stops whenever the Z80 has to do work. This is like the Sinclair ZX80/ZX81 of old, but like that computer, this video is generated with minimal hardware. The video output is set up to show the video whenever the operating system is waiting for a key to be input.

The operating system may also be configured to use the serial port as the terminal. SIO/DART port B is the port wired to the logic level converter. If connecting to a PC, a null modem adapter is needed. SIO/DART port A may be used with a conventional TTL USB serial adapter. The serial port parameters are 115200 bps, 8 bits, no parity, 1 stop bit. There are jumpers so that CTSA and CTSB may be hardwired to ground or can be controlled through the serial port.

Jumpers JP6 and JP7 control the mapping of the flash memory. If JP6 is set to the 32k setting and JP7 is set to the lower 16k setting, the entire of the 32k is mapped to $0000-$7fff on startup. If JP6 is set to the 16k setting, then if JP7 is set to the lower 16k setting, the lower 16k is mapped to $0000-$3fff, otherwise the upper 16k is mapped to $0000-$3fff. This enables two different ROMS to be swapped out, for example, a conventional loader bios could be placed in the low 16k ($0000-$3fff in the 28C256), and Collapse OS in the upper 16k ($4000-$7fff on the 28C256), and these two may be switched between with a jumper.

I used a TL866II to write the 28C256 ROM. I may work on a ROM writer that can use the 8255 itself to write another 28C256, thus enabling a CollapseOS ROM to write another CollapseOS ROM. This will probably require a couple of 74HC595s to be externally wired to provide address lines to the ROM.

Configuring Collapse OS

When configuring Collapse OS, the xcomp.fs file has a few parts that need to be changed to reconfigure the kernel.

```
VID_WDTH is number of bytes across per scan line (24 is the
  minimum)
VID_SCN is number of display scan lines (246 for full NTSC,
  123 for doubled lines)
VID_VBL is number of vertical blanking interval lines
VID_LN is number of lines to report to the GRID driver
```

```
If memory is constrained, then the scan lines can be doubled,
and the number of bytes across the scan line may be reduced to a
minimum of 24, so the minimum frame buffer size is 123 X 24 =
2952 bytes.  As given the frame buffer size is 246 X 66 = 16236
bytes.

There are 2 (vidfr) implementations, a single and a double.
LOAD the proper blocks in your xcomp unit.

An example configuration lives in /arch/z80/dan.
```

7.2 TRS-80 Model 4p (hw/z80/trs80-4p.txt)

```
The TRS-80 (models 1, 3 and 4) are among the most popular z80
machines. They're very nicely designed and I got my hands on a
4p with two floppy disk drives and a RS-232 port. In this
recipe, we're going to get Collapse OS running on it.
```

Reference documentation

```
These documents are recommended:

* TRS-80 Model 4/4P Technical Reference Manual
* Disk System Owner's Manual - TRS-80 Model 4/4P
* Service Manual - TRS-80 Model 4P, 4P Gate Array
* FDC 1791-02 datasheet from Standard Microsystems Corporation
```

Memory map and interrupts

```
Collapse OS runs on the 4P in memory mode 2: $0000-f3ff is all
RAM, $f400-f7ff maps to the keyboard, $f800-ffff maps to video.

Boot binary begins at $0000, HERESTART begin right after the
binary, PS_ADDR is $f3ff.

$10 bytes are allocated to drivers SYSVARS:

00    KBD input buffer (char)
01    KBD input buffer (shift flags)
02    KBD debouncing flag
03-05 GRID_MEM
06    Floppy drive selection mask
07    Floppy drive "current operation" (rd or wr) alias
09    FD0 Current disk offset
0b    FD1 Current disk offset
0d    Character under the cursor

Except for RTC interrupts, all other interrupts are disabled.
```

Booting

The bootloader, placed in sector 1 of track 0, directly pokes (the 4K boot ROM is not used) FDC ports in order to read tracks 1 and 2 (36 sectors, 9KB) into memory $0000 and then jump to $0000.

It also does a few initializations that are then assumed by the OS:

* 80x24 video mode, page 1
* Memory map 2
* Interrupt enabled, IM 1, with RTC interrupts enabled
* "FAST" mode (4MHz)
* External I/O disabled

In case of an error (CRC error, Lost Data, sector not found), a character corresponding to the error is placed on the screen and we abort (infinite loop).

Keyboard

The 4P doesn't poll its keyboard itself, software has to do it. To do it reliably, we do so during Real Time Clock interrupts (60Hz in 4MHz mode). During each poll, we do this:

1. Decode pressed key (7 first columns)
2. Debounce check. If no key is pressed, reset debounce flag.
3. If debounced, fill input buffer with char and 8th row, which contains shift status.

If you look at the hardware keyboard mapping, you'll see that most of it is straightforward to decode, with exceptions (@ and the , to / range). During the interrupt, we don't care about the exceptions and simply record the first row yielding a nonzero keypress.

Rows 0 to 5 have the particularity of having columns with contiguous ASCII code. This makes them rather straightforward to decode. Row 6 is special because the ASCII codes are heterogenous, so we need a hardcoded map.

Row 7 is for keys that, when pressed, aren't considered a "keypress" (shift keys).

To avoid repeats, we debounce the keyboard after a keypress, that is, when a key is pressed in rows 0 to 6, we wait until we go back to a "no key pressed" state before recording another press.

When (key?) polls for keypress, it checks the input buffer and also applies little shift rules and exceptions to the raw value in the buffer so that it yields the proper character.

The keyboard doesn't yield the whole visible ASCII range in a straightforward manner. To allow a full range, we make left and right shift behave differently.

Left shift is the "regular shift". It yields values on labels (shifted @ yields `). Right shift allows the reaching of chars like [] and {}. These are the yields:

```
, --> [
= --> \
. --> ]
/ --> ^
0 --> _
1 --> {
2 --> |
3 --> }
4 --> ~
5 --> DEL
```

(it makes more sense when looking at the ASCII table.)

You will notice, also, that we take extra step to ensure that when we check, in (key?), whether we have a key press, we only check the LSB. This might seem illogical at first, but this is because the polling interrupt might happen at any time, including during the "0 KBDBUF !" part.

BREAKING away

In Collapse OS, the BREAK key gets a special treatment. It is checked during the polling interrupt and, when pressed, calls QUIT right away. This allows you to escape infinite loops.

Because it's QUIT being called and not ABORT, PS is preserved. Because it can quit at any time (except when interrupts are disabled), you can end up with extra garbage on PS after QUIT.

Video

Video in the 4P is very straightforward: the screen starts at $f800 and is 1 char per byte in memory. We always run in 80 columns mode and use the Grid subsystem (doc/grid.txt Page 67).

We only support the 80x24 mode, which is enabled in the

bootloader.

The cursor is solid and doesn't blink. In CURSOR!, we simply replace the character at target pos with $bf (a solid rectangle) and place old character in UNDERCUR buffer in SYSVARS.

The NEWLN implementation scrolls contents when the bottom of the screen is reached.

Floppy

In our 179X FDC driver, we hardcode for MFM (double density). We seek (with verify) implicitly before each read or write operation and, like TRS-DOS, we enable Write Precompensation for tracks higher than 21.

If an error occurs, "FD err" is raised, with the corresponding status number (which should normally contain the error).

There isn't yet any auto-retry mechanism on error. This results in occasional failures (mostly CRC) which don't occur on TRS-DOS (I suspect it auto-retries on errors).

Collapse OS doesn't yet have any way to format floppies. For now, they need to be formatted through TRS-DOS.

RS-232

The RS-232 driver implements TX> and RX<? which the Remote shell and the XMODEM application use. Before using it, it has to be initialized with CL$, which takes a single bauds argument. This argument is not a direct bauds rating, it's a numerical mapping:

```
00 50          01 75          02 110         03 134.5
04 150         05 300         06 600         07 1200
08 1800        09 2000        0a 2400        0b 3800
0c 4800        0d 7200        0e 9600        0f 19200
```

For example, "$0e CL$" initializes the RS-232 at 9600 bauds.

The boot disk

As already stated, the boot disk has these properties:

* Double Density
* 256b per sector, 18 sectors per track
* Bootloader in sector 1, track 0

* Collapse OS binary in tracks 1 and 2, 9KB max.

If you can produce this floppy through external means, you don't need the instructions below. However, because this can be tricky, the easiest way to proceed is to have a RS-232 equipped TRS-80 4P as well as TRS-DOS 6.x and use DOS to construct that floppy.

Creating the boot disk with TRS-DOS

We need to send sizeable binary programs through the RS-232 port and then run it. The big challenge here is ensuring data integrity. Sure, serial communication has parity check, but it has no helpful way of dealing with parity errors. When parity check is enabled and that a parity error occurs, the byte is simply dropped on the receiving side. Also, a double bit error could be missed by those checks.

What we'll do here is to ping back every received byte back and have the sender do the comparison and report mismatched data.

Another problem is ASCII control characters. When those are sent across serial communication channels, all hell breaks lose. When sending binary data, those characters have to be avoided. We use tools/ttysafe for that.

Does TRSDOS have a way to receive this binary inside these constraints? Not to my knowledge. As far as I know, the COMM program doesn't allow this.

What are we going to do? We're going to punch in a binary program to handle that kind of reception! You're gonna feel real badass about it too...

Testing serial communication

The first step here is ensuring that you have bi-directional serial communication. To do this, first prepare your TRS-80:

```
set *cl to com
setcomm (word=8,parity=no,bauds=9600)
```

The first line loads the communication driver from the COM/DRV file on the TRSDOS disk and binds it to *cl, the name generally used for serial communication devices. The second line sets communication parameters in line with what is generally the default on modern machine.

Then, you can run "COMM *cl" to start a serial communication

console.

Then, on the modern side, use your favorite serial communication program and set the tty to 9600 baud with option "raw". Make sure you have -parenb.

If your line is good, then what you type on either side should echo on the other side. If it does not, something's wrong. Debug.

Building the binaries

You're reaching the point where you need binaries. You can build them with "make" in /arch/z80/trs80, which will yield:

* os.bin: The Collapse OS binary
* boot.bin: The bootloader
* recv.bin: The binary receiver we're going to need to manually punch in the machine.

Punching in the goodie

As stated in the overview, we need a program on the TRS-80 that:

1. Listens to *cl
2. Echoes each character back to *cl
3. Adjusts ttysafe escapes
4. Stores received bytes in memory

You're in luck: that program has already been written and it's in recv.bin. Open it with a hex editor to view its contents. That's what you have to punch in. Not so bad eh?

It can run from any offset (all jumps in it are relative), but it is hardcoded to write to $3000. Make sure you don't place it in a way to be overwritten by its received data.

You're looking at recv.fs and wondering what is that COM_DRV_ADDR constant? That's the DCB handle of your *cl device. You will need to get that address before you continue. Go read the following section and come back here. If your DCB is different from COM_DRV_ADDR, you'll have to change it and run "make" again.

How will you punch that in? The "debug" program! This very useful piece of software is supplied in TRSDOS. To invoke it, first run "debug (on)" and then press the BREAK key. You'll get the debug interface which allows you to punch in any data in any memory address. Let's use $5000 which is the offset it's

designed for (high enough not to be overwritten).

For reference: to go back to the TRSDOS prompt, it's "o<return>".

First, display the $5000-$503f range with the d5000<space> command (I always press Enter by mistake, but it's space you need to press). Then, you can begin punching in with h5000<space>. This will bring up a visual indicator of the address being edited. Punch in the stuff with a space in between each byte and end the edit session with "x".

Getting your DCB address

In the previous step, you need to set COM_DRV_ADDR to your "DCB" address for *cl. That address is your driver "handle". To get it, first get the address where the driver is loaded in memory. You can get this by running "device (b=y)". That address you see next to *cl? that's it. But that's not our DCB.

To get your DBC, go explore that memory area. Right after the part where there's the *cl string, there's the DCB address (little endian). On my setup, the driver was loaded in $0ff4 and the DCB address was 8 bytes after that, with a value of $0238. Don't forget that z80 is little endian. 38 will come before 02.

Saving that program for later

If you want to save yourself typing for later sessions, why not save the program you've painfully typed to disk? TRSDOS enables that easily. Let's say that you typed your program at $5000 and that you want to save it to RECV/CMD on your second floppy drive, you'd do:

```
    dump recv/cmd:1 (start=X'5000',end=X'5030',tra=X'5000')
```

A memory range dumped this way will be re-loaded at the same offset through "load recv/cmd:1". Even better, TRA indicates when to jump after load when using the RUN command. Therefore, you can avoid all this work above in later sessions by simply typing "recv" in the DOS prompt.

Note that you might want to turn "debug" off for these commands to run. I'm not sure why, but when the debugger is on, launching the command triggers the debugger.

Sending binary through the RS-232 port

Once you're finished punching your program in memory, you can run it with g5000<enter> (not space). If you've saved it to disk, run "recv" instead. Because it's an infinite loop, your screen will freeze. You can start sending your data.

To that end, there's the tools/pingpong program. It takes a device and a filename to send. Before you send the binary, make it go through tools/ttysafe first (which just takes input from stdin and spits tty-safe content to stdout):

```
./ttysafe < os.bin > os.ttysafe
```

On OpenBSD, the invocation can look like:

```
doas ./pingpong /dev/ttyU0 os.ttysafe
```

If everything goes well, the program will send your contents, verifying every byte echoed back, and then send a null char to indicate to the receiving end that it's finished sending. This will end the infinite loop on the TRS-80 side and return. That should bring you back to a refreshed debug display and you should see your sent content in memory, at the specified address ($3000 if you didn't change it).

If there was no error during pingpong, the content should be exact. Nevertheless, I recommend that you manually validate a few bytes using TRSDOS debugger before carrying on.

debugging tip: Sometimes, the communication channel can be a bit stubborn and always fail, as if some leftover data was consistently blocking the channel. It would cause a data mismatch at the very beginning of the process, all the time. What I do in these cases is start a "COMM *cl" session on one side and a screen session on the other, type a few characters, and try pingpong again.

Bringing it together

Now that you have all you need to send binary contents to your TRS-80, you're ready to craft your disk! To do so, we'll use DEBUG's low level disk writing capabilities. It is invoked with a command has this signature:

```
driveno,trackno,sector,r/w,addr,sectorcount
```

Example:

```
1,0,1,w,3000,1
```

This writes a single sector at track 0, sector 1 (each sector is 256 bytes) using the contents of memory address $3000.

Drive numbers are 0 and 1.

First, you'll upload and write down boot.bin with this very command. Yes, the boot sector is sector 1, not sector 0. Weird but true.

Then, you'll upload os.bin. It's a bit bigger than the bootloader and spans over multiple tracks, starting with track 1 (the bootloader loads beginning at track 1, sector 0). You might be tempted to write 18 sectors at once (there are 18 sectors per track), but TRS-DOS is a bit tricky for this because it seems to silently drop the write operation sometime. I've found that the sweet spot is to write 6 sectors at once. So, for a binary that is $1a00 bytes big, it would be:

```
1,1,0,w,3000,6
1,1,6,w,3600,6
1,1,c,w,3c00,6
1,2,0,w,4200,6
1,2,6,w,4800,2
```

If everything went well, you have your boot disk! Before you reboot, however, you might want to re-read those sectors in memory (replace "w" with "r") and quickly compare the first bytes of every sector with your reference binary to make sure that everything was written properly (you can zero-out a memory zone with "F". Example: "f3000,5000,0").

You're done! Pop the disk in the first drive, reboot, you should have a Collapse OS prompt.

All this process was a bit inconvenient, but once you have a Collapse OS disk, receiving data and writing them to disk is a bit easier. Read on for details.

Using floopy drives

As it is, your system fully supports reading and writing to both floppy drives. By default, floppy drive 1 is selected. We implement MSPAN (see doc/mspan) with the DRVSEL option, so you can use DRVSEL to change the active drive (FLUSH first).

As it is now, floppy organization is:

```
D0 000-099
```

```
D1 100-199
D2 200-299
D3 300-479
D4 480-560 (unallocated disk. use for extra stuff)
```

Sending blkfs to floppy

Collapse OS has RX<? to read a char from its RS-232 port and TX> to emit to it. That's all you need to have a full Collapse OS with access to disk blocks.

First, make sure your floppies are formatted. Collapse OS is currently hardcoded to single side and double density, which means there's a limit of 180 blocks per disk.

You'll need to send those blocks through RS-232. First, let's initialize the driver with CL$. It is hardcoded to "no parity, 8 bit words" and takes a "baud code" as an argument. It's a 0-15 value with these meanings:

```
00 50          01 75          02 110         03 134.5
04 150         05 300         06 600         07 1200
08 1800        09 2000        0a 2400        0b 3800
0c 4800        0d 7200        0e 9600        0f 19200
```

After CL$ is called, let's have the CL take over the prompt:

```
    ' TX> 'EMIT !
    ' RX<? 'KEY? !
```

"Aliases" in usage.txt[Page 7] for details. Your serial link now has the prompt.

Now, you can use /tools/blkup to send a disk's contents. First, extract the first 180 blocks from blkfs:

```
    dd if=blkfs bs=1024 count=180 > d1
```

Now, insert your formatted disk in drive 1 and push your blocks:

```
    tools/blkup /dev/ttyUSB0 0 d1
```

It takes a while, but you will end up having your first 180 blocks on floppy! Go ahead, LIST around. Then, repeat for other disks.

Once you're done, you will want to go back to local control:

```
    ' (emit) 'EMIT !
    ' (key?) 'KEY? !
```

Alternatively to all this, you can also use Collapse OS' XMODEM implementation at B150. Instead of taking over the prompt, you'd run "0 BLK@" followed by "RX>BLK". On the other side, you'd run your favorite XMODEM app ("rx" probably).

Self-hosting

As it is, your installment of Collapse OS is self-hosting using instructions from /doc/selfhost.txt[Page 70]. The difference is that instead of writing the binary you have in memory to EEPROM, you'll want to write it to disk. To that end, there is the MEM>BLK utility in B2 which allows writing memory spanning multiple sectors to disk.

To write Collapse OS to the boot disk, you have to write your binary to the *half* of the 4th block (18 sectors per track is 4.5K per track, track 1 is there). MEM>BLK doesn't allow writing half blocks, but you can cheat a little bit with something like:

```
XORG $200 - 4 8 MEM>BLK
```

See what I did there? I simply fill the first 2 sectors of block 4 with whatever preceeds my binary. I advise checking of written data by reading it back with BLK>MEM and comparing.

If you need to write the boot sector from within Collapse OS, don't run MEM>BLK because the computer's bootloader is a bit sensible to garbage. What you do is zero-out the whole block 0 like this:

```
0 BLK@ BLK( $400 0 FILL BLK!!
```

Then, you can place the bootloader's content at BLK(+$100 and then call FLUSH to write it out.

7.3 Z80-MBC2 (hw/z80/z80mbc2.txt)

The Z80-MBC2[1] combines a Z80 and an ATMEGA32A to provide a CP/m capable computing environment. It features a SD card bootloader which makes running Collapse OS on it rather simple.

In this recipe, we're going to run Collapse OS on the Z80-MBC2, interfacing through its serial port. We're going to use the MBC's API to implement BLK on the SD card.

Gathering parts

* A Z80-MBC2 computer with its SD card module and a properly

  flashed "IOS" on the ATMega32A.
* A FTDI-to-TTL cable to connect to the serial port.

Building the binary

Running "make" in arch/z80/z80mbc2 will yield "os.bin" which is what we want.

Running on the Z80-MBC2

Mount the SD card on your modern computer and copy "os.bin" as "autoboot.bin", overwriting the binary that was previously there.

We also have to copy the blkfs over. This is done by using IOS' drive system. Each "DSxNyy.DSK" file on the card is a drive, each drive has 512 track of 32 sectors of 512 bytes, so one drive is plenty for our needs. Collapse OS hardcodes drive 0.

Each drive is part of a set. IOS theoretically supports up to 10 sets, but the binary shipped by default only accepts 4. You have to overwrite an existing set. I used set 3. So, copy "blkfs" to file "DS3N00.DSK". If you want, you can change the name of the set by changing the contents of "DS3NAM.DAT".

Put back the SD card in the Z80-MBC2 and power it up by connecting the FTDI adapter to it (red: VCC, black: GND, green: TX, white: RX).

The FTDI adapter will show up as something like "ttyUSB0" (or "ttyU0" on OpenBSD). Connect to it with "screen" or "cu" or whatever you like. Baud rate of the Z80-MBC2 appears to be hardcoded to 115200.

Then, enable IOS program selection by holding RESET and USER at the same time, wait 2 seconds, releasing RESET, wait 2 seconds, releasing USER. You should then be given a 1-8 choice.

You begin by selecting the proper disk set, which is through choice 8, then you select the Autoboot binary through choice 4.

You are now in Collapse OS.

[1]: https://hackaday.io/project/159973-z80-mbc2-a-4-ics-homebrew-z80-computer

7.4 RC2014 (hw/z80/rc2014/intro.txt)

The RC2014[1] is a nice and minimal z80 system that has the

advantage of being available in an assembly kit. Assembling it yourself involves quite a bit of soldering due to the bus system. However, one very nice upside of that bus system is that each component is isolated and simple.

The machine used in this recipe is the "Classic" RC2014 with an 8k ROM module , 32k of RAM, a 7.3728Mhz clock and a serial I/O.

The ROM module being supplied in the assembly kit is an EPROM, not EEPROM, so you can't install Collapse OS on it. You'll have to supply your own.

There are many options around to boot arbitrary sources. What was used in this recipe was a AT28C64B EEPROM module. I chose it because it's compatible with the 8k ROM module which is very convenient. If you do the same, however, don't forget to set the A14 jumper to high because what is the A14 pin on the AT27 ROM module is the WE pin on the AT28! Setting the jumper high will keep is disabled.

The goal is to have the shell running and accessible through the Serial I/O.

You'll need specialized tools to write data to the AT28 EEPROM. There seems to be many devices around made to write in flash and EEPROM modules. If you don't have any but have a Arduino Uno, take a look at doc/hw/arduinouno.

Gathering parts

* A "classic" RC2014 with Serial I/O
* An AT28C64B and a way to write to it.
* A FTDI-to-TTL cable to connect to the Serial I/O module

Build the binary

Building the binary is as simple as running "make" in /arch/z80/rc2014. This will yield "os.bin" which can then be written to EEPROM.

This build is controlled by the xcomp.fs unit, which loads blk/618. That's what you need to modify if you want to customize your build.

Emulate

The Collapse OS project includes a RC2014 emulator suitable for

this image. You can invoke it with "make emul".

Running

Put the AT28 in the ROM module, don't forget to set the A14 jumper high, then power the thing up. Connect a FTDI-to-TTL cable to the Serial I/O module and identify the tty bound to it (in my case, "/dev/ttyUSB0"). Then:

```
    screen /dev/ttyUSB0 115200
```

Press the reset button on the RC2014 and the "ok" prompt should appear.

[1]: https://rc2014.co.uk

7.5 Asynchronous Communications Interface Adapters (hw/z80/rc2014/acia.txt)

The RC2014's Serial I/O module and the Dual Serial module (using Zilog's SIO) both have an important shortcoming: they hard-wire CTS to ground. Considering that these modules' main purpose are to communicate with a modern machine through a USB-to-TTL dongle, this hard-wiring make sense: a modern machine have plenty of power to take whatever is coming on a 115200 bauds channel.

However, this becomes problematic when communicating with the RC2014 through an underpowered machine running Collapse OS: RTS/CTS flow control doesn't work.

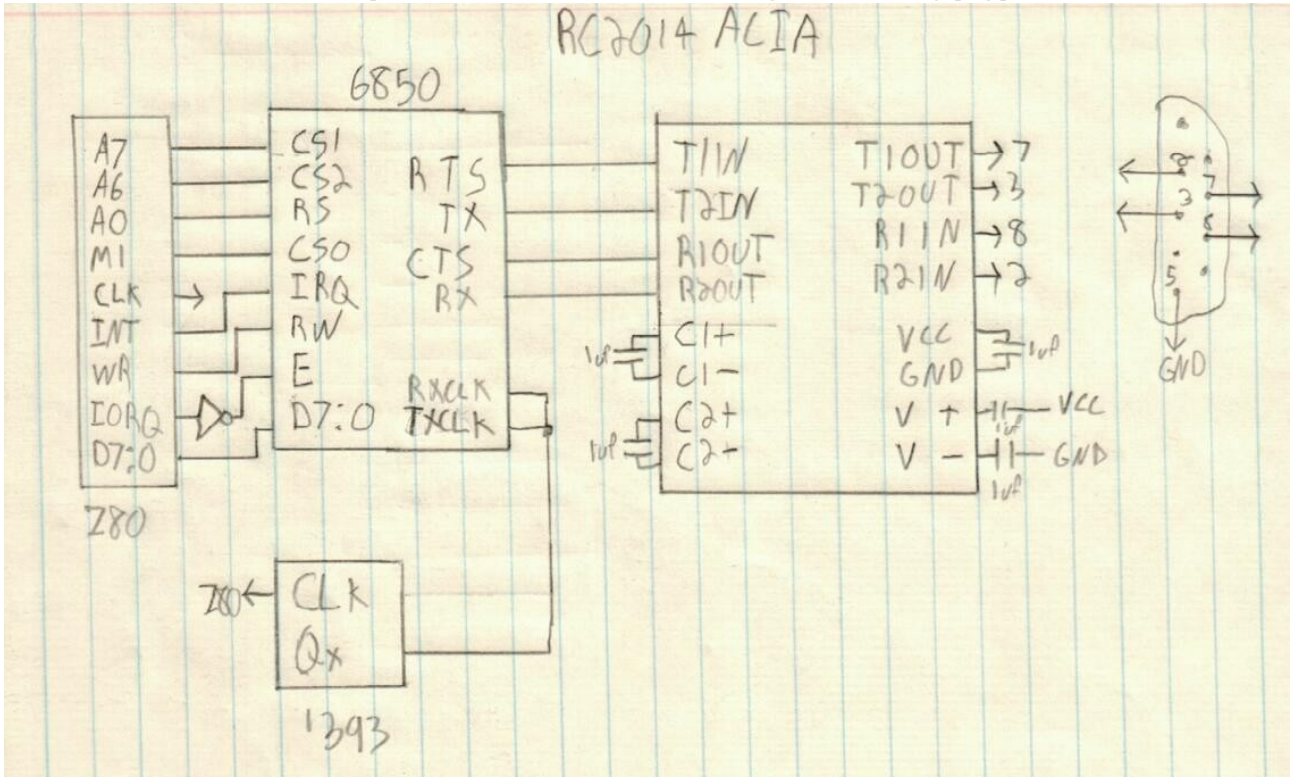
For this reason, I recommend that you build your own ACIA module. A schematic for it is in img/acia.jpg[Page 137]. This module is exactly the same as the "official" Serial I/O module, with two differences:

1. Wire CTS properly
2. Add a '393 counter to allow for lower baud rates.

This design with the '393 has an important limitation: you can't easily fine-select your baud rate. For example, dividing by 12 (for 9600 bauds) is not straightforward with a '393. However, because the '393 is a dual 4-bit counter, it can divide more.

You might want to replace the '393 with a '161 with preset if you want to divide by a more specific number.

7.6 RC2014 ACIA (hw/z80/rc2014/img/acia.jpg)



7.7 TI-84+ (hw/z80/ti84/intro.txt)

```
The TI-84+ is a machine with many advantages, one being that
it's very popular.  It also has a lot of flash memory and RAM.

Its builtin keyboard and screen, however, are hard to use,
especially the screen. With a tiny font, the best we can get is
a 24x10 console.

There is, however, a built-in USB controller that might prove
very handy.

I haven't opened one up yet, but apparently, they have limited
scavenging value because its z80 CPU is packaged in a
TI-specific chip. Due to its sturdy design, and its ample RAM
and flash, we could imagine it becoming a valuable piece of
equipment if found intact.

The best pre-collapse ressource about it is WikiTI[1].

As it is now, with its tiny screen and cumbersome keyboard,
Collapse OS is not really usable on the TI-84+. One could
imagine a scenario where one has a terminal and uses the TI-84+
through the link for its large amount of flash and RAM. But
using it standalone? Nah, not even post-collapse.

Therefore, this recipe is more of a "look, I run!" demo.
```

Gathering parts

* A TI-84+
* A USB cable
* tilp[2]
* mktiupgrade[3]

Build the ROM

Running "make" in arch/z80/ti84 will result in "os.rom" being created.

Upload to the calculator

Background notes

Getting software to run on it is a bit tricky because it needs to be signed with TI-issued private keys. Those keys have long been found and are included in keys/. With the help of the mktiupgrade, an upgrade file can be prepared and then sent through the USB port with the help of tilp.

That, however, requires a modern computing environment. As of now, there is no way of installing Collapse OS on a TI-8X+ calculator from another Collapse OS system.

Because it is not on the roadmap to implement complex cryptography in Collapse OS, the plan is to build a series of pre-signed bootloader images. The bootloader would then receive data through either the Link jack or the USB port and write that to flash (I haven't verified that yet, but I hope that data written to flash this way isn't verified cryptographically by the calculator).

As modern computing fades away, those pre-signed binaries would become opaque, but at least, would allow bootstrapping from post-modern computers.

Instructions

WARNING: the instructions below will wipe all the contents of your calculator, including TI-OS.

To send your ROM to the calculator, you'll need two more tools: mktiupgrade and tilp.

Once you have them, you need to place your calculator in "bootloader mode", that is, in a mode where it's ready to receive a new binary from its USB cable. To do that you need to:

1. Shut down the calculator by removing one of the battery.
2. Hold the DEL key
3. Put the battery back.
4. A "Waiting... Please install operating system now" message will appear.

Once this is done, you can plug the USB cable in your computer and run "make send". This will create an "upgrade file" with mktiupgrade and then push that upgrade file with tilp. tilp will prompt you at some point. Press "1" to continue.

When this is done, you can press the ON button to see Collapse OS' prompt!

Validation errors

Sometimes, when uploading an upgrade file to your calculator, you'll get a validation error. You can always try again, but in my own experience, some specific binaries will simply always be refused by the calculator. Adding random "nop" or reordering lines (when it makes sense, of course) should fix the problem.

I'm not sure whether it's a bug with the calculator or with mktiupgrade.

Soft poweron

Collapse OS can be toggled on/off with the ON button (no 2nd flag, just ON). You have to press it fast enough because debouncing is quick.

It soft-reboots at each power on.

TODO: When the battery is removed, Collapse OS can't be booted again, the calculator has to be reflashed. I don't know why.

Usage

The shell works like a normal Forth shell, but with very tight screen space.

When pressing a "normal" key, it spits the symbol associated to

it depending on the current mode. In normal mode, it spits the digit/symbol. In Alpha mode, it spits the letter. In Alpha+2nd, it spits the uppercase letter.

Special keys are Alpha and 2nd. Pressing them toggles the associated mode. Alpha and 2nd mode don't persist for more than one character. After the character is spit, mode reset to normal.

Pressing 2nd then Alpha will toggle the A-Lock mode, which is a persistent mode. The A-Lock mode makes Alpha enabled all the time. While A-Lock mode is enabled, you have to enable Alpha to spit a digit/symbol.

Simultaneous keypresses have undefined behavior. One of the keys will be registered as pressed. Mode key don't work by simultaneously pressing them with a "normal" key. The presses must be sequential.

Keys that aren't a digit, a letter, a symbol that is part of 7-bit ASCII or one of the two mode key have no effect.

[1]: http://wikiti.brandonw.net/index.php
[2]: http://lpg.ticalc.org/prj_tilp/
[3]: https://github.com/KnightOS/mktiupgrade

7.8 TI-84+ LCD driver (`hw/z80/ti84/lcd.txt`)

Implement (emit) on TI-84+ (for now)'s LCD screen. Lives at B350.

Required config:

* LCD_MEM: 2b area where a that will point to an area allocated to LCD driver memory during LCD$ init.
* LCD_FNTW: Width in pixels of a glyph
* LCD_FNTH: Height in pixels of a glyph

The screen is 96x64 pixels. The 64 rows are addressed directly with CMD_ROW but columns are addressed in chunks of 6 or 8 bits (there are two modes).

In 6-bit mode, there are 16 visible columns. In 8-bit mode, there are 12.

Note that "X-increment" and "Y-increment" work in the opposite way than what most people expect. Y moves left and right, X moves up and down.

Z-Offset

This LCD has a "Z-Offset" parameter, allowing to offset rows on the screen however we wish. This is handy because it allows us to scroll more efficiently. Instead of having to copy the LCD ram around at each linefeed (or instead of having to maintain an in-memory buffer), we can use this feature.

The Z-Offset goes upwards, with wrapping. For example, if we have an 8 pixels high line at row 0 and if our offset is 8, that line will go up 8 pixels, wrapping itself to the bottom of the screen.

The principle is this: The active line is always the bottom one. Therefore, when active row is 0, Z is FNTH+1, when row is 1, Z is (FNTH+1)*2, When row is 8, Z is 0.

6/8 bit columns and smaller fonts

If your glyphs, including padding, are 6 or 8 pixels wide, you're in luck because pushing them to the LCD can be done in a very efficient manner. Unfortunately, this makes the LCD unsuitable for a Collapse OS shell: 6 pixels per glyph gives us only 16 characters per line, which is hardly usable.

This is why we have this buffering system. How it works is that we're always in 8-bit mode and we hold the whole area (8 pixels wide by FNTH high) in memory. When we want to put a glyph to screen, we first read the contents of that area, then add our new glyph, offsetted and masked, to that buffer, then push the buffer back to the LCD. If the glyph is split, move to the next area and finish the job.

That being said, it's important to define clearly what CURX and CURY variable mean. Those variable keep track of the current position *in pixels*, in both axes.

Words descriptions

LCD_BUF: two pixel buffers that are 8 pixels wide (1b) by FNTH pixels high. This is where we compose our resulting pixels blocks when spitting a glyph.

8 Hardware: 6502 based devices

8.1 Apple IIe (hw/6502/appleiie/intro.txt)

The Apple IIe is a computer with many nice features, very good expandability and a rather straightforward design, along with a

very complete documentation.

As it is now, Collapse OS is built upon ProDOS and doesn't directly run the hardware. Maybe some day direct drivers will be written, but the challenge is significant because the floppy controller on the Apple IIe, unlike in many other machines, is very bare. Sector/track detection has to be done entirely in software with precise timing. Maybe one day...

Reference documents

* Apple IIe Reference Manual
* Applesoft BASIC Programming Reference Manual
* Apple II BASIC Programming with ProDOS
* Beneath Apple DOS
* Beneath Apple ProDOS

Installing Collapse OS

I didn't have the luck of having a RS-232 card on the machine I acquired. I could have gone through some hacks (maybe the joystick port?) which would have required the design of some hardware adapter. Another possible route would be to craft a floppy from another machine which could be read from the Apple IIe, but floppy-related tools in Collapse OS are not mature enough yet.

Since I haven't done so yet in any of the recipes, let's go with the long, hard route: typing the whole thing in.

For this recipe, you need:

* An Apple IIe
* A floppy disk drive and some floppies
* A ProDOS disk (mine is ProDOS 8)

The Monitor

We'll be typing in our stuff from Apple's Monitor program which is documented in "Apple IIe Reference Manual". A cheatsheet is available in monitor.txt Page 144.

Things can go wrong and you can lose your work. You are advised to quickly become accustomed to ProDOS BASIC's BSAVE and BLOAD commands to incrementally save your work to floppies.

Typing it in

When you run "make" in /arch/6502/appleiie, in addition to producing os.bin, it also spits the binary contents to the screen in lines of 16 bytes and, at the end of each line, a numerical checksum.

The idea is that with the help of these checksums, if you made a typing error, you'll quickly locate it. The checksum is a simple sum rather than a CRC16 because Applesoft BASIC doesn't support fancy stuff like XOR.

After having typed a few lines (and saved them!), you can type yourself a checksum checker in BASIC:

```
10 A=8192
20 N=0
30 FOR I=A TO A+15
40 N=N+PEEK(A)
50 NEXT I
60 PRINT N
70 INPUT X
80 A=A+16
90 GOTO 20
```

The result of "INPUT X" is ignored, but the pause give you the opportunity to break the loop with Control+C.

You're ready for the real thing. The idea is to type it at its home address, $2000. You'll do so in the Monitor (CALL -151).

Regularly, you'll want to come back to BASIC and save your work with something like "BSAVE COS,A$2000,L$XXXX" with XXXX being the length of the binary you've typed so far. Then, you run "RUN" to do your checksum. Compare numbers you get from BASIC with numbers you got from xcomp.fs. They're supposed to match. The last line doesn't have a checksum, just be extra careful with it.

Once you're ready, you can run the binary with "2000G" in the Monitor.

Alternative to typing: SPI hack through game port

See spihack.txt Page 145

Creating a ProDOS boot disk

With ProDOS, it's easy to create a disk that will directly boot to Collapse OS. To do so, begin with a bootable copy of your ProDOS disk and remove everything from it except the "PRODOS" file.

Then, copy your COS "BIN" file in there and make it into a SYS file. That last part is a bit awkward. Given a BIN file named COS, here's the BASIC commands to copy it to a SYS file:

```
] CREATE COS.SYSTEM,TSYS
] BLOAD COS,A$2000,L$2000
] BSAVE COS.SYSTEM,A$2000,L$2000,TSYS
```

If COS.SYSTEM is the only SYS file besides PRODOS, then it the disk will boot directly to Collapse OS.

8.2 Apple II's system monitor (hw/6502/appleiie/monitor.txt)

The monitor allows peeking and poking memory in a manner that is much more convenient than with BASIC, in hexadecimal notation.

A complete reference is in "Apple IIe Reference Manual". This is a quick reference.

When inside BASIC, we enter the monitor with "CALL -151". We then get a "*" prompt.

Typing an address reads that byte:

```
*1DFC
1DFC- 2A
*
```

We can fetch a range:

```
*1DFC.1E00
1DFC- 2A 2B 2C 2D
1E00- 2E
*
```

We can set memory:

```
*1DFC:01 02 03
*1DFC.1E00
1DFC- 01 02 03 2D
1E00- 2E
*
```

We can "continue" setting memory, omitting address:

```
*1DFC:04 05 06
*:07 08
*1DFC.1E00
1DFC- 04 05 06 07
1E00- 08
*

You can disassemble memory with "L" (for LIST):

*1DFCL
(20 lines of disassembled memory)
```

8.3 Alternative to typing: SPI through game port (hw/6502/appleiie/spihack.txt)

```
Instead of typing, if you have a way to spit SPI slow enough
(for example, with doc/hw/avr/spispit), you can record that data
through the game port.

It's hacking because the game port only has inputs, but if you
perform checksums on both sides, you can end up with good data.

The game port is a DB9 that has the following pinout:

5 4 3 2 1
 9 8 7 6

1 - SW1
2 - +5V
3 - GND
4 - PDL2
5 - PDL0
6 - SW2
7 - SW0
8 - PDL1
9 - PDL3

In my own setup, I used an Arduino because I could power it
directly from the port, which simplifies interfacing (I can wire
directly).

We use only SW0, plugged to SCK and SW1, plugged to MOSI.

SW* are mapped to memory thus:

SW0 - C061
SW1 - C062
SW2 - C063

Only bit 7 is relevant (1=high), the rest is garbage.
```

The idea is that we point to an address in memory, and then run an infinite loop that shifts data in A 8 times, then writes to memory.

The code for doing this is:

```
\ Receive SPI data through game port
\ Plug SCK in SW0 and MOSI in SW1, then run the program
\ ZP+7 must contain the destination's page.
0 # LDY, 6 <> STY, BEGIN,
  1 # LDA, BEGIN, \ 8 times
    BEGIN, $c061 () LDX, ( SW0 ) BR BPL, \ SW0 high!
    CLC, $c062 () LDX, ( SW1 ) FJR BPL, SEC, THEN, ROLA,
    BEGIN, $c061 () LDX, ( SW0 ) BR BMI, \ SW0 low!
    BR BCC,
  6 []Y+ STA, INY, IFZ, 7 <> INC, THEN, BR BCS,
```

You might not have Collapse OS on the IIe yet, so you'll have to type this with the integrated mini assembler. It's position independant, so you can put it anywhere.

The idea is that before you launch the code, you set the destination page in ZP+1 (ZP+0 stays at 0 at all times). Then, you call the code and then spit your SPI. Once it's spit, press CTRL+RESET to come back to AppleSoft prompt. If everything went well, you have your data in memory, do stuff with it.

Note that reading the port with the 6502 at 1MHz represents an important constraint: your SPI spitter has to be pretty slow! At about 30 cycles for the main loop, you can expect to miss data if you spit faster than 30KHz.

Be careful, SW0 is hard-wired to "Open-Apple" key and SW1, to "Solid-Apple". This can have weird effects on warm boot-up, so you might want to disconnect your SPI spitter before you reset.

If you use the Arduino SPI spitter from doc/hw/avr/spispit, you need to ignore the first 3 SCK toggles. You can do that by prepending the above code with:

```
$20 # LDA, BEGIN, \ 3 times
  BEGIN, $c061 () LDX, ( SW0 ) BR BPL, \ SW0 high!
  BEGIN, $c061 () LDX, ( SW0 ) BR BMI, \ SW0 low!
  ROLA, BR BCC,
```

Reboot afterwards

After you got your data in, that you've checked that it's correct and that you've saved it to disk, I recommend that you reboot before launching your binary.

In my tests, the computer exhibited buggy behavior right after a run of the procedure above, which only a clean reboot could fix.

8.4 SPI relay (hw/6502/appleiie/spi.txt)

The Apple IIe can run a SPI relay with the exact same card design as the one described in doc/hw/z80/spi, which you can then plug into one of the expansion slots. You only need to perform these pin mappings:

```
Clock -> 7M
WR -> R/W
RD -> R/W going through a 40106 inverter.
IORQ -> DEVICE SELECT
D0:7 -> D0:7
A0:3 -> A0:3
A4:7 -> manual selection through jumpers
```

With such an adapter (or a card specifically for the IIe, but with modified pin mappings), you can use the Z80 SPI relay in the same way as if you were on a RC2014. Instead of PC! and PC@, you read and write to the proper address. For example, if you select clot 3, then SPI_CTL is going to be $c0b5 and SPI_DATA is going to be $c0b4.

You have to be mindful of the power draw from your data lines, however. The IIe is a bit picky on that front. For example, my own prototype, to simplify my messy wiring, had IORQ go through a diode and happily hop on the "big '138 line". It worked on the RC2014, but on the IIe, the machine wouldn't because (I suspect) too much power was drawn on the DEVICE SELECT line. Be removing the diode and rewiring adequately, the problem was fixed.

9 Hardware: Various other devices

9.1 PC/AT (hw/8086/pcat.txt)

PC-compatible machines need no introduction. They are one of the most popular machines of all time. Collapse OS has a 8086 assembler and has boot code allowing it to run on a PC/AT-compatible machine, using BIOS interrupts in real mode. Collapse OS always runs in real mode.

In this recipe, we will compile Collapse OS and write it to a USB drive that is bootable on a modern PC-compatible machine.

Gathering parts

* A modern PC-compatible machine that can boot from a USB drive.

```
* A USB drive
```

Build the binary

```
Running "make" in /arch/8086/pcat will yield:

* mbr.bin: a 512 byte binary that goes at the beginning of the
           disk
* os.bin: 8086 Collapse OS binary
* disk.bin: a concatenation of the above, with "blkfs" appended
            to it starting at $2000.

disk.bin is what goes on the USB drive.

This binary has BLK and AT-XY support, which means you have disk
I/Os and can run VE.
```

Emulation

```
You can run the built binary in Collapse OS' 8086 emulator using
"make emul".

The 8086 emulator is barbone. If you prefer to use it on a more
realistic setting, use QEMU. The command is:

    qemu-system-i386 -drive file=disk.bin,if=floppy,format=raw
```

Running on a modern PC

```
First, copy disk.bin onto your USB drive. For example, on an
OpenBSD machine, it could look like:

    doas dd if=disk.bin of=/dev/sd1c

Your USB drive is now BIOS-bootable. Boot your computer and
enter your BIOS setup to make sure that "legacy boot" (non-EFI
boot, that is, BIOS boot) is enabled. Configure your boot device
priority to ensure that the USB drive has a chance to boot.

Reboot, you have Collapse OS. Boot is of course instantaneous
(we're not used to this with modern software...).
```

9.2 TRS-80 Color Computer 2 (hw/6809/coco2.txt)

```
The CoCo2 is a nice little 6809 machine featuring 32x16 characters
video output,
a builtin keyboard, a ROM slot, RS-232, I/O ports, more than
enough to have fun
```

with Collapse OS on it.

The most straightforward way to run Collapse OS on it is to build a custom ROM
cart. At first, you would think that you could cannibalize a ROM cart you have
laying around, but the ones I had had some kind of unmovable round plastic chip
on the PCB, so nowhere to place a AT28 on. I built my own from scratch.

Relevant Documents

* M6809 datasheet
* Service Manual - TRS-80 Color Computer 2 NTSC Version
* Color Computer ROM Cartridge Schematic

Gathering parts

* A Coco2. Mine is the 64K RAM version.
* A 40 pin male card edge connector. If you can get a version that has its pins
  pre-bent over 2 rows, you'll save yourself some work.
* A protoboard that is large enough to accomodate 20 pins, narrow enough to fit
  in the ROM card slot, long enough so that you can still comfortably hold it
  while fitting it in the slot.
* A AT28C64B EEPROM
* A socket for it.
* A disposable CoCo2 ROM cart helps when trying to visualize pin placement.

Building the cart

Then, it's only a matter of wiring the proper connector pin to the proper AT28
pin. The CoCo2 ROM cart pinout is this:

7: Q
8: CART/
9: 5V
10: D0
11: D1
12: D2
13: D3
14: D4

```
15: D5
16: D6
17: D7
18: no connect
19: A0
20: A1
21: A2
22: A3
23: A4
24: A5
25: A6
26: A7
27: A8
28: A9
29: A10
30: A11
31: A12
32: CTS/
33: GND
34: GND
```

When you hold the cart with the edge facing you, pin 1 is on the top pane, at
your left. Pin 40 is on the bottom pane, at your right. Pins 1-6, 18 and 35-40
are all no connects.

Q and CART/ are wired together and don't touch the AT28. Data and address lines
are connected to the same AT28 pin. CTS/ is wired to AT28's CE/.

On the AT28, you will want to hard-wire WE/ to 5V and OE/ to GND. If you want
your cart to accomodate bigger EEPROMs, you'll want to hard-wire A13 and A14.

Running Collapse OS

Once you have your cart, run "make" in arch/6809/coco2 and write os.bin onto
your AT28. Then stuff it on your cart, plug it in, and poof! Collapse OS.

ALL CAPS

The CoCo2 has 64 character glyphs builtin and Collapse OS piggy-backs on them.
In those 64 glyphs, there are no lowercase letters. However, every letter can

be displayed with a dark background. This is what we use to indicate a lower-
case letter.

Keyboard input is by default uppercased. Hold shift to type a lowercase.

9.3 Writing to a AT28 EEPROM from a modern environment (hw/avr/at28.txt)

The Arduino Uno is a very popular platform based on the ATMega328p. While Collapse OS doesn't run on AVR MCUs (yet?), the Arduino can be a handy tool, which is why we have recipes for it here.

In this recipe, we'll build ourselves an ad-hoc EEPROM holder which is designed to be driven from an Arduino Uno.

Gathering parts

* An Arduino Uno
* A AT28C64B
* 2 '164 shift registers
* Sockets, header pins, proto board, etc.
* AVRA[1] (will some day rewrite to Collapse OS' ASM)
* avrdude to send program to Arduino

Schema

Schema is at img/at28wr.jpg[Page 152].

This is a rather simple circuit which uses 2 chained '164 shift register to drive the AT28 address pins and connects CE, WE, OE and the data pins directly to the Arduino. Pins have been chosen so that the protoboard can plug directly on the Arduino's right side (except for VCC, which needs to be wired).

PD0 and PD1 are not used because they're used for the UART.

AT28 selection pins are pulled up to avoid accidental writes due to their line floating before Arduino's initialization.

I've put 1uf decoupling caps next to each IC.

Software

The software in code/at28wr.asm listens to the UART and writes every byte it receives to the AT28, starting at address 0. It

expects tty-escaped content (see /tools/ttysafe).

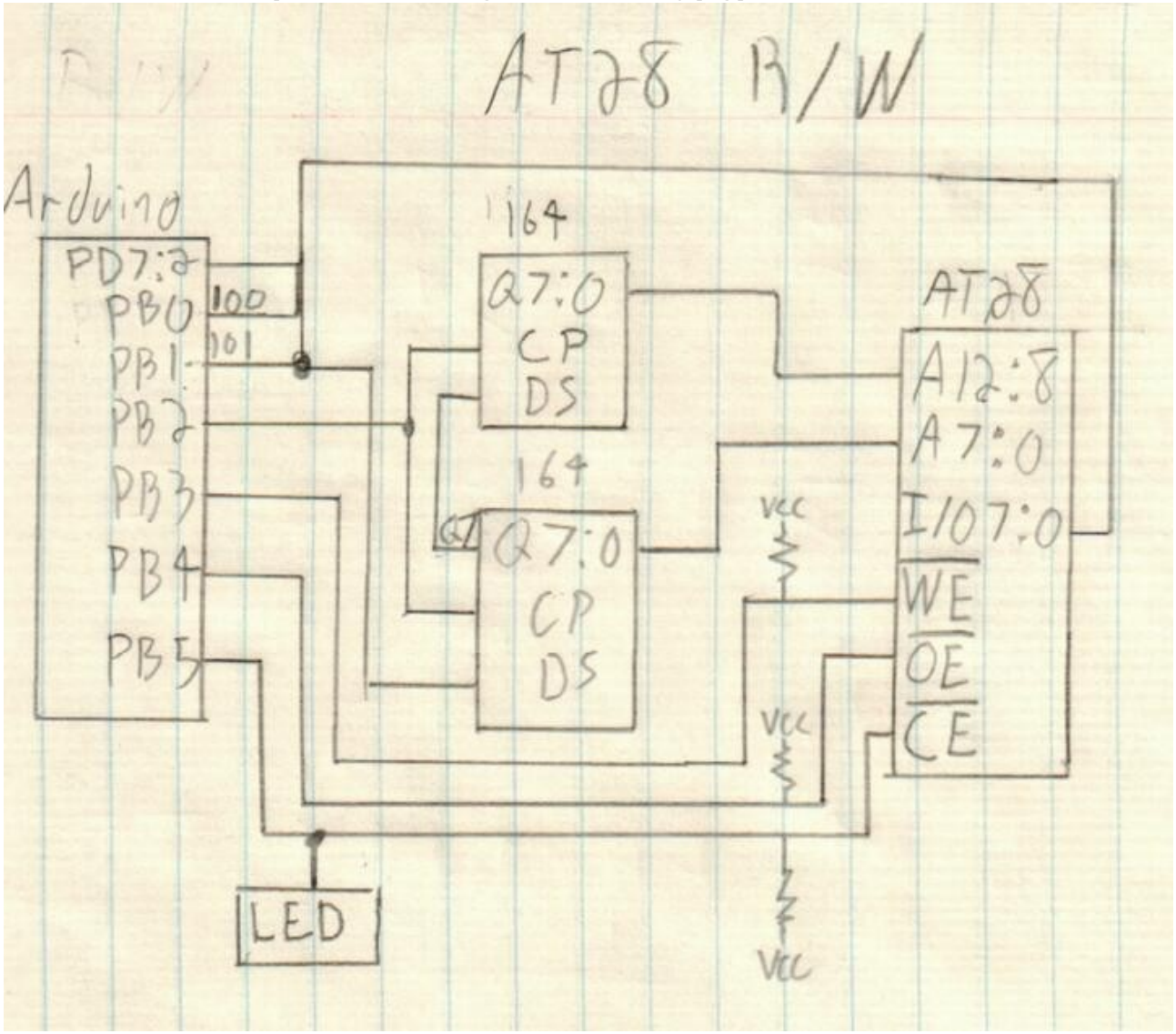
After having written the byte, it re-reads it from the EEPROM
and spits it back to the UART, tty-escaped.

Usage

After you've build and sent your binary to the Arduino with
"make send", you can send your (tty-safe!) content to your
EEPROM using /tools/pingpong.

[1]: http://avra.sourceforge.net/

9.4 AT28 R/W (hw/avr/img/at28wr.jpg)



9.5 Spit bytes through SPI from an Arduino Uno (hw/avr/spispit.txt)

This recipe programs an Arduino Uno to spit data from its Flash

memory through its SPI pins. This allows you to get data into systems with as little as 2 input pins available.

The Arduino acts as an SPI master and assumes the presence of an activated slave.

The use case for which this project was created (Apple IIe reading SPI through its game port) needed SPI to be quite slow because it couldn't keep up otherwise.

This is why we apply a general 16x clock divider through CLKPR and set the SPI speed to f_osc/128. This gives us a rate of about 8KHz, which is plenty slow for just about anything.

It transmits data in chunks of $100 bytes, beginning at address $100. The number of chunks it transmit is read from address $ff.

The program begins transmitting on startup. To control the moment of transmission, you use the Reset button.

While transmitting, it reads the result of the SPI exchange in its SPI data register and spits it to UART. This way, if your destination echoes anything and your arduino's UART is plugged to something, you can control that echo.

Of course, due to the nature of SPI, your first byte will be garbage and you won't get the last byte.

Gathering parts

* An Arduino Uno
* avrdude to send program to Arduino

Programming the Arduino

The program lives in arch/avr/blk.fs and is built using Collapse OS' AVR assembler. A Makefile exists in arch/avr/spispit that takes care of doing this automatically. "make" will yield "spispit.bin"

Data to spit has to be placed in a file named "data" and "make send" will combine spispit.bin and data and place the proper number of blocks at address $ff. It will then send that to the Arduino using avrdude.

At this point, it's ready to use.

Ignore the first 3 SCK toggles

On an Arduino Uno that has its bootloader enabled, SCK is going to toggle 3 times before it begins spitting its payload. The logic reading this payload has to ignore those first 3 toggles.

Check the LED

Because SCK is wired to the builtin LED on the Arduino Uno, you can check whether we're still transmitting by looking at the LED. At 8KHz, its blinking is visible to the naked eye.

Block filesystem

1 Architecture independent

1.1 Master Index: 0

B0

```
MASTER INDEX

001 Useful little words       010 RX/TX tools
020 Block editor              035 Memory Editor
040 AVR SPI programmer        045 Sega ROM signer
050 Virgil's workspace        060-199 unused
200 Cross compilation
210 Core words                230 BLK subsystem
235 RX/TX subsystem           237 Media Span subsystem
240 Grid subsystem
245 PS/2 keyboard subsystem   250 SD Card subsystem
260 Fonts                     280-289 unused
290 Automated tests           300 Arch-specific content
```

1.2 Useful little words: 1-5

B1

```
\ Useful little words. CRC16[] MOVE-
'? CRC16[] [IF] \S [THEN]
\ Compute CRC16 over a memory range
: CRC16[] ( a u -- c ) >R >A 0 BEGIN AC@+ CRC16 NEXT ;
: MOVE- ( a1 a2 u -- ) \ *A* MOVE starting from the end
  ?DUP IF >R OVER - ( a1 diff ) SWAP R@ + >A
    BEGIN ( diff ) A- A> OVER + AC@ SWAP C! NEXT DROP
  ELSE 2DROP THEN ;
```

B2

```
\ Useful little words. MEM>BLK BLK>MEM
\ *A* Copy an area of memory into blocks.
: MEM>BLK ( addr blkno blkcnt )
  >R BEGIN ( a blk )
    DUP BLK@ 1+ SWAP DUP BLK( $400 MOVE BLK!! $400 + SWAP NEXT
  DROP FLUSH ;
\ *A* Copy subsequent blocks in an area of memory
: BLK>MEM ( blkno blkcnt addr )
  ROT> >R BEGIN ( a blk )
    DUP BLK@ 1+ SWAP BLK( OVER $400 MOVE $400 + SWAP NEXT
  DROP ;
```

B3

```
\ Context. Allows multiple concurrent dictionaries.
\ See doc/usage.txt

0 VALUE saveto \ where to save CURRENT in next switch
: context DOER CURRENT , DOES> ( a -- )
  saveto IF CURRENT TO saveto THEN ( a )
  DUP TO saveto ( a )
  @ CURRENT ! ;
```

B4

```
\ string manipulation.
'? >s [IF] \S [THEN]
2 VALUES sa sl
: >s ( sa sl -- ) TO sl TO sa ; : s> sa sl ;
: cutr ( n -- ) sl -^ DUP 0< IF DROP 0 THEN TO sl ;
: cutl ( n -- ) sl SWAP cutr sl - sa + TO sa ;
: prefix? ( sa sl -- f )
  DUP sl > IF 2DROP 0 EXIT THEN sa ROT> []= ;
: suffix? ( sa sl -- f )
  DUP sl > IF 2DROP 0 EXIT THEN sl OVER - ( sa sl off ) sa +
  ( sa sl sa2 ) SWAP []= ;
```

B5

```
\ Word table. See doc/wordtbl
: WORDTBL ( n -- a ) CREATE HERE SWAP << ALLOT0 1 HERE C! ;
: W+ ( a -- a+2? ) 1+ 1+ DUP @ IF DROP THEN ;
: :W ( a -- a+2? ) HERE XTCOMP OVER ! W+ ;
: 'W ( a -- a+2? ) ' OVER ! W+ ;
: WEXEC ( tbl idx -- ) << + @ EXECUTE ;
```

1.3 Pager: 6

B6

```
\ Pager. See doc/pager
4 VALUES ''EMIT ''KEY? chrcnt lncnt
20 VALUE PGSZ
: realKEY BEGIN ''KEY? EXECUTE UNTIL ;
: back ''EMIT 'EMIT ! ''KEY? 'KEY? ! ;
: emit ( c -- )
  chrcnt 1+ TO chrcnt
  DUP CR = chrcnt LNSZ = OR IF
   0 TO chrcnt lncnt 1+ TO lncnt THEN
  ''EMIT EXECUTE lncnt PGSZ = IF
    0 TO lncnt NL> ." Press q to quit, any key otherwise" NL>
    realKEY 'q' = IF back QUIT THEN THEN ;
: key? back KEY? ;
: page 'EMIT @ TO ''EMIT 'KEY? @ TO ''KEY?
  ['] emit 'EMIT ! ['] key? 'KEY? ! ;
```

1.4 Flow words: 7

B7

```
\ Flow words
'? PC NOT [IF] ALIAS HERE PC [THEN]
'? PC2A NOT [IF] : PC2A ; [THEN]
ALIAS PC BEGIN,
: LSET PC TO ;
: BR PC - 2 - _bchk ;
: FJR BEGIN, 1+ 0 ;
: IFZ, FJR JRNZi, ; : IFNZ, FJR JRZi, ;
: IFC, FJR JRNCi, ; : IFNC, FJR JRCi, ;
\ warning: l is a PC value, not a mem addr!
\ also, in 6502, JRi, is 3b instead of 2, hence the hack.
: FMARK ( l -- ) PC2A DUP C@ IF ( hack ) 1+ THEN DUP HERE -^ 1-
  SWAP C! ;
: THEN, FMARK ; : ELSE, FJR JRi, SWAP FMARK ;
```

1.5 RX/TX tools: 10-15

B10

```
\ Communicate blocks with block server. See doc/blksrv.
CREATE h16 '$' C, 4 ALLOT
: RX>h16 ( -- n ) \ *A*
  h16 1+ >A 4 >R BEGIN RX< DUP EMIT SPC> AC!+ NEXT
  h16 5 PARSE NOT IF 0 THEN ;
: csumchk ( c1 c2 ) = NOT IF ABORT" bad csum" THEN ;
: blksrv< ( blkno -- ) \ *A*
  RX<< TX[ 'G' EMIT .X ]TX 0 ( csum ) BLK( >A 1024 >R BEGIN
    RX< DUP AC!+ + NEXT RX>h16 csumchk ;
: blksrv> ( blkno -- ) \ *A*
  RX<< TX[ 'P' EMIT .X ]TX 0 ( csum ) BLK( >A 1024 >R BEGIN
    AC@+ DUP TX> + NEXT TX[ .X ]TX ;
```

B11

```
\ Remote shell. See doc/rxtx
: RX<?? RX<? ?DUP NOT IF 100 TICKS RX<? THEN ;
: _<< \ print everything available from RX<?
  BEGIN RX<?? IF EMIT ELSE EXIT THEN AGAIN ;
: _<<1r RX< EMIT _<< ;
: rsh BEGIN
  KEY? IF DUP EOT = IF DROP EXIT ELSE TX> THEN THEN _<< AGAIN ;
```

B12

```
\ rupload. See doc/rxtx
: CR> CR EMIT ;
: unpack DUP $f0 OR SWAP $0f OR ;
: out unpack TX> TX> ; : out2 L|M out out ;
: rdok \ read RX until after "ok"
    BEGIN RX< WS? NOT UNTIL _<<1r ;
: rupload ( loca rema u -- )
  TX[ ." : in KEY $f0 AND KEY $0f AND OR ;" CR> rdok
      ." : in2 in <<8 in OR ;" CR> rdok
      \ sig: chk -- chk, a and then u are KEYed in
      ." : _ in2 >A in2 >R BEGIN in TUCK + SWAP AC!+ NEXT ;"
      CR> rdok DUP ROT ( loca u u rema )
      ." 0 _" CR> out2 out2 ]TX
  >R >A 0 BEGIN ( chk ) '.' EMIT AC@ out AC@+ + NEXT
  _<<1r TX[ ." .X FORGET in" CR> ]TX rdok .X ;
```

B13

```
\ XMODEM routines. See doc/rxtx
: _<<s BEGIN RX<? IF DROP ELSE EXIT THEN AGAIN ;
: _rx>mem1 ( addr -- f, Receive single packet, f=eot )
  RX< 1 = NOT IF ( EOT ) $6 ( ACK ) TX> 1 EXIT THEN
  '.' EMIT RX< RX< 2DROP ( packet num )
  >A 0 ( crc ) 128 >R BEGIN ( crc )
    RX< DUP ( crc n n ) AC!+ ( crc n ) CRC16 NEXT
  RX< <<8 RX< OR ( sender's CRC )
  = IF $6 ( ACK ) ELSE $15 'N' EMIT ( NACK ) THEN TX> 0 ;
: RX>MEM ( addr --, Receive packets into addr until EOT )
  _<<s 'C' TX> BEGIN ( a )
  DUP _rx>mem1 SWAP 128 + SWAP UNTIL DROP ;
: RX>BLK ( -- )
  _<<s 'C' TX> BLK( BEGIN ( a )
  DUP BLK) = IF DROP BLK( BLK! BLK> 1+ 'BLK> ! THEN
  DUP _rx>mem1 SWAP 128 + SWAP UNTIL 2DROP ;
```

B14

```
: _snd128 ( A:a -- A:a )
    0 128 >R BEGIN ( crc )
      AC@+ DUP TX> ( crc n ) CRC16 ( crc ) NEXT
    L|M TX> TX> ;
: _ack? 0 BEGIN DROP RX< DUP 'C' = NOT UNTIL
 DUP $06 ( ACK ) = IF DROP 1
    ELSE $15 = NOT IF ABORT" out of sync" THEN 0 THEN ;
: _waitC
  ." Waiting for C..." BEGIN RX<? IF 'C' = ELSE 0 THEN UNTIL ;
: _mem>tx ( addr pktstart pktend -- )
  OVER - >R SWAP >A BEGIN ( pkt )
    'P' EMIT DUP . SPC> $01 ( SOH ) TX> ( pkt )
    1+ ( pkt start at 1 ) DUP TX> $ff OVER - TX> ( pkt+1 )
    _snd128 _ack? NOT IF LEAVE THEN NEXT DROP ;
```

B15

```
: MEM>TX ( a u -- Send u bytes to TX )
  _waitC 128 /MOD SWAP IF 1+ THEN ( pktcnt ) 0 SWAP _mem>tx
  $4 ( EOT ) TX> RX< DROP ;
: BLK>TX ( b1 b2 -- )
  _waitC OVER - ( cnt ) >R BEGIN ( blk )
    'B' EMIT DUP . SPC> DUP BLK@ BLK( ( blk a )
    OVER 8 * DUP 8 + ( a pktstart pktend ) _mem>tx 1+ NEXT
  $4 ( EOT ) TX> RX< DROP ;
```

1.6 Block editor: 20-24

B20

```
\ Block editor. see doc/ed.
\ Cursor position in buffer. EDPOS/64 is line number
0 VALUE EDPOS
CREATE IBUF LNSZ 1+ ALLOT0 \ counted string, first byte is len
CREATE FBUF LNSZ 1+ ALLOT0
: L BLK> ." Block " DUP . NL> LIST ;
: B BLK> 1- BLK@ L ; : N BLK> 1+ BLK@ L ;
: IBUF+ IBUF 1+ ; : FBUF+ FBUF 1+ ;
: ILEN IBUF C@ ; : FLEN FBUF C@ ;
: EDPOS! TO EDPOS ; : EDPOS+! EDPOS + EDPOS! ;
: 'pos ( pos -- a, addr of pos in memory ) BLK( + ;
: 'EDPOS EDPOS 'pos ;
```

B21

```
\ Block editor, private helpers
: _lpos ( ln -- a ) LNSZ * 'pos ;
: _pln ( ln -- ) \ print line no ln with pos caret
  DUP _lpos DUP >A LNLEN 1 MAX >R BEGIN ( lno )
    A> 'EDPOS = IF '^' EMIT THEN
    AC@+ SPC MAX EMIT NEXT ( lno ) SPC> 1+ . ;
: _zline ( a -- ) LNSZ SPC FILL ; \ zero-out a line
: _type ( buf -- ) \ *A* type into buf until end of INBUF
  IN<? ?DUP NOT IF DROP EXIT THEN OVER 1+ DUP _zline >A BEGIN
    ( buf c ) AC!+ IN<? ?DUP NOT UNTIL ( buf )
  A> OVER - 1- ( buf len ) SWAP C! ;
```

B22

```
\ Block editor, T P U
\ user-facing lines are 1-based
: T 1- DUP LNSZ * EDPOS! _pln ;
: P IBUF _type IBUF+ 'EDPOS LNSZ MOVE BLK!! ;
: _mvln+ ( ln -- move ln 1 line down )
    DUP 14 > IF DROP EXIT THEN
    _lpos DUP LNSZ + LNSZ MOVE ;
: _U ( U without P, used in VE )
  15 EDPOS LNSZ / - ?DUP IF
    >R 14 BEGIN DUP _mvln+ 1- NEXT DROP THEN ;
: U _U P ;
```

B23

```
\ Block editor, F i
: _F ( F without _type and _pln. used in VE )
  'EDPOS 1+ BEGIN ( a )
    FBUF+ C@ OVER BLK) OVER - ( a c a u ) [C]?
    DUP 0< IF 2DROP EXIT THEN ( a idx ) + ( a )
    DUP FBUF+ FLEN []= IF BLK( - EDPOS! EXIT THEN 1+ AGAIN ;
: F FBUF _type _F EDPOS LNSZ / _pln ;
: _rbufsz ( size of linebuf to the right of curpos )
  EDPOS LNSZ MOD LNSZ -^ ;
: _I ( I without _pln and _type. used in VE )
  _rbufsz ILEN OVER < IF ( rsize )
    ILEN - ( chars-to-move )
    'EDPOS DUP ILEN + ROT ( a a+ilen ctm ) MOVE- ILEN
  THEN ( len-to-insert )
  IBUF+ 'EDPOS ROT MOVE ( ilen ) BLK!! ;
: I IBUF _type _I EDPOS LNSZ / _pln ;
```

B24

```
\ Block editor, X E Y
: icpy ( n -- copy n chars from cursor to IBUF )
  DUP IBUF C! IBUF+ _zline 'EDPOS IBUF+ ( n a buf ) ROT MOVE ;
: _del ( n -- ) ?DUP NOT IF EXIT THEN _rbufsz MIN
  'EDPOS 2DUP + ( n a1 a1+n ) SWAP _rbufsz MOVE ( n )
  \ get to next line - n
  DUP EDPOS $ffc0 AND $40 + -^ 'pos ( n a )
  SWAP SPC FILL BLK!! ;
: _X ( n -- ) ?DUP NOT IF EXIT THEN _rbufsz MIN DUP icpy _del ;
: X _X EDPOS LNSZ / _pln ;
: _E FLEN _X ;
: E FLEN X ;
: Y FBUF IBUF LNSZ 1+ MOVE ;
```

1.7 Visual editor: 25-32

B25

```
\ Visual text editor. VALUEs, lg? width pos@ mode! ...
3 VALUES PREVPOS xoff ACC
LNSZ 3 + VALUE MAXW
10 VALUE MARKCNT
CREATE MARKS MARKCNT << << ALLOT0 \ 4b: blk/edpos
: nspcs ( pos n ) SPC FILLC ;
: lg? COLS MAXW > ; : col- MAXW COLS MIN -^ ;
: width lg? IF LNSZ ELSE COLS THEN ;
: acc@ ACC 1 MAX ; : pos@ ( x y -- ) EDPOS LNSZ /MOD ;
: num ( c -- ) \ c is in range 0-9
  '0' - ACC 10 * + TO ACC ;
: mode! ( c -- ) 4 col- CELL! ;
```

B26

```
\ VE, rfshln contents selblk pos! xoff? setpos
: _ ( ln -- ) \ refresh line ln
  DUP _lpos xoff + SWAP 3 + COLS * lg? IF 3 + THEN
  width CELLS! ;
: rfshln pos@ NIP _ ; \ refresh active line
: contents 16 >R 0 BEGIN DUP _ 1+ NEXT DROP ;
: selblk BLK@ contents ;
: pos! ( newpos -- ) EDPOS TO PREVPOS
    DUP 0< IF DROP 0 THEN 1023 MIN EDPOS! ;
: xoff? pos@ DROP ( x )
  xoff ?DUP IF < IF 0 TO xoff contents THEN ELSE
    width >= IF LNSZ COLS - TO xoff contents THEN THEN ;
: setpos ( -- ) pos@ 3 + ( header ) SWAP ( y x ) xoff -
  lg? IF 3 + ( gutter ) THEN SWAP AT-XY ;
: 'mark ( -- a ) ACC MARKCNT MOD << << MARKS + ;
```

B27

```
\ VE, cmv buftype bufprint bufs
: cmv ( n -- , char movement ) acc@ * EDPOS + pos! ;
: buftype ( buf ln -- ) \ type into buf at ln
  3 OVER AT-XY KEY DUP SPC < IF 2DROP DROP EXIT THEN ( b ln c )
  SWAP COLS * 3 + 3 col- nspcs ( buf c )
  IN( SWAP LNTYPE DROP BEGIN ( buf a ) KEY LNTYPE UNTIL
  IN( - ( buf len ) SWAP C!+ IN( SWAP LNSZ MOVE IN$ ;
: _ ( buf sa sl pos )
  DUP >R STYPEC ( buf ) C@+ ( buf sz ) R> 3 + STYPEC ;
: bufs ( -- ) \ refresh I and F lines
  IBUF S" I: " COLS _ FBUF S" F: " COLS 2 * _ ;
: insl _U EDPOS $3c0 AND DUP pos! 'pos _zline BLK!! contents ;
```

B28

```
\ VE cmds
31 VALUE cmdcnt
CREATE cmdl ," G[]IFnNYEXChlkjHLg@!wWb&mtfROoD"
cmdcnt WORDTBL cmds
:W ( G ) ACC selblk ;
:W ( [ ) BLK> acc@ - selblk ; :W ( ] ) BLK> acc@ + selblk ;
: insert 'I' mode! IBUF 1 buftype _I bufs rfshln ;
'W insert ( I )
:W ( F ) 'F' mode! FBUF 2 buftype _F bufs setpos ;
:W ( n ) _F setpos ;
:W ( N ) EDPOS _F EDPOS = IF 0 EDPOS! acc@ >R BEGIN
    BLK> 1+ BLK@ _F EDPOS IF LEAVE THEN NEXT
    contents setpos THEN ;
:W ( Y ) Y bufs ; :W ( E ) _E bufs rfshln ;
:W ( X ) acc@ _X bufs rfshln ;
:W ( C ) FLEN _del rfshln insert ;
```

B29

```
\ VE cmds
:W ( h ) -1 cmv ; :W ( l ) 1 cmv ;
:W ( k ) -64 cmv ; :W ( j ) 64 cmv ;
: bol EDPOS $3c0 AND pos! ;
'W bol ( H )
:W ( L ) EDPOS DUP $3f OR 2DUP = IF 2DROP EXIT THEN SWAP BEGIN
    ( res p ) 1+ DUP 'pos C@ WS? NOT IF NIP DUP 1+ SWAP THEN
    DUP $3f AND $3f = UNTIL DROP pos! ;
:W ( g ) ACC 1 MAX 1- 64 * pos! ;
:W ( @ ) BLK> BLK( (blk@) 0 BLKDTY ! contents ;
:W ( ! ) BLK> FLUSH 'BLK> ! ;
```

B30

```
\ VE cmds
: C@- DUP 1- SWAP C@ ;
: word>> BEGIN C@+ WS? UNTIL ;
: ws>> BEGIN C@+ WS? NOT UNTIL ;
: word<< BEGIN C@- WS? UNTIL ;
: ws<< BEGIN C@- WS? NOT UNTIL ;
: bpos! BLK( - pos! ;
:W ( w ) 'EDPOS acc@ >R BEGIN word>> ws>> NEXT 1- bpos! ;
:W ( W ) 'EDPOS acc@ >R BEGIN ws>> word>> NEXT 1- bpos! ;
:W ( b ) 'EDPOS acc@ >R BEGIN 1- ws<< word<< NEXT 1+ 1+ bpos! ;
:W ( & ) WIPE contents ;
:W ( m ) BLK> 'mark ! EDPOS 'mark 1+ 1+ ! ;
:W ( t ) 'mark 1+ 1+ @ pos! 'mark @ selblk ;
```

B31

```
\ VE cmds
:W ( f ) EDPOS PREVPOS 2DUP = IF 2DROP EXIT THEN
  2DUP > IF DUP pos! SWAP THEN
  ( p1 p2, p1 < p2 ) OVER - LNSZ MIN ( pos len ) DUP FBUF C!
  FBUF+ _zline SWAP 'pos FBUF+ ( len src dst ) ROT MOVE bufs ;
:W ( R ) 'R' mode! BEGIN
  setpos KEY DUP BS? IF -1 EDPOS+! DROP 0 THEN
  DUP SPC >= IF
  DUP EMIT 'EDPOS C! 1 EDPOS+! BLK!! 0 THEN UNTIL ;
'W insl ( O )
:W ( o ) EDPOS $3c0 < IF EDPOS 64 + EDPOS! insl THEN ;
:W ( D ) bol LNSZ icpy acc@ LNSZ * ( delsz ) BLK) 'EDPOS - MIN
  >R 'EDPOS R@ + 'EDPOS ( src dst )
  BLK) OVER - MOVE BLK) R@ - R> SPC FILL BLK!! bufs contents ;
```

B32

```
\ VE final: status nums gutter handle VE
: status 0 $20 nspcs 0 0 AT-XY ." BLK" SPC> BLK> . SPC> ACC .
  SPC> pos@ 1+ . ',' EMIT . xoff IF '>' EMIT THEN SPC>
  BLKDTY @ IF '*' EMIT THEN SPC mode! ;
: nums 16 >R BEGIN R@ HERE FMTD R@ 2 + COLS * STYPEC NEXT ;
: gutter lg? IF 19 >R BEGIN
  '|' R@ 1- COLS * MAXW + CELL! NEXT THEN ;
: handle ( c -- f )
  DUP '0' '9' =><= IF num 0 EXIT THEN
  DUP cmdl cmdcnt [C]? 1+ ?DUP IF 1- cmds SWAP WEXEC THEN
  0 TO ACC 'q' = ;
: VE BLK> 0< IF 0 BLK@ THEN
  CLRSCR 0 TO ACC 0 TO PREVPOS
  nums bufs contents gutter
  BEGIN xoff? status setpos KEY handle UNTIL 0 19 AT-XY ;
```

1.8 Memory editor: 35-39

B35

```
\ Memory Editor. See doc/me
CREATE CMD '#' C, 0 C,
CREATE BUF '$' C, 4 ALLOT \ always hex
\ POS is relative to ADDR
4 VALUES ADDR POS HALT? ASCII?
16 VALUE AWIDTH
LINES 2 - VALUE AHEIGHT
AHEIGHT AWIDTH * VALUE PAGE
COLS 33 < [IF] 8 TO AWIDTH [THEN]
: addr ADDR POS + ;
CREATE _ ," 0123456789abcdef"
: hex! ( c pos -- )
  OVER 16 / _ + C@ OVER CELL! ( c pos )
  1+ SWAP $f AND _ + C@ SWAP CELL! ;
: bottom 0 LINES 1- AT-XY ;
```

B36

```
\ Memory Editor, line rfshln contents showpos
: line ( ln -- )
  DUP AWIDTH * ADDR + >A 1+ COLS * ( pos )
  ':' OVER CELL! A> <<8 >>8 OVER 1+ hex! 4 + ( pos+4 )
  AWIDTH >> >R A> SWAP BEGIN ( a-old pos )
    AC@+ ( a-old pos c ) OVER hex! ( a-old pos )
    1+ 1+ AC@+ OVER hex! 3 + ( a-old pos+5 ) NEXT
  SWAP >A AWIDTH >R BEGIN ( pos )
    AC@+ DUP SPC - $5e > IF DROP '.' THEN OVER CELL! 1+ NEXT
  DROP ;
: rfshln POS AWIDTH / line ;
: contents LINES 2 - >R BEGIN R@ 1- line NEXT ;
: showpos
  POS AWIDTH /MOD ( r q ) 1+ SWAP ( y r ) ASCII? IF
  AWIDTH >> 5 * + ELSE DUP 1 AND << SWAP >> 5 * + THEN
  4 + ( y x ) SWAP AT-XY ;
```

B37

```
\ Memory Editor, addr! pos! status type typep
: addr! $fff0 AND TO ADDR contents ;
: pos! DUP 0< IF PAGE + THEN DUP PAGE >= IF PAGE - THEN
  TO POS showpos ;
: status 0 COLS SPC FILLC
  0 0 AT-XY ." A: " ADDR .X SPC> ." C: " POS .X SPC> ." S: "
  PSDUMP POS pos! ;
: type ( cnt -- sa sl ) BUF 1+ >A >R BEGIN
  KEY DUP SPC < IF DROP LEAVE ELSE DUP EMIT AC!+ THEN NEXT
  BUF A> BUF - ;
: typep ( cnt -- n? f )
  type ( sa sl ) DUP IF PARSE ELSE NIP THEN ;
```

B38

```
\ Memory Editor, almost all actions
: #] ADDR PAGE + addr! ; : #[ ADDR PAGE - addr! ;
: #J ADDR $10 + addr! POS $10 - pos! ;
: #K ADDR $10 - addr! POS $10 + pos! ;
: #l POS 1+ pos! ; : #h POS 1- pos! ;
: #j POS AWIDTH + pos! ; : #k POS AWIDTH - pos! ;
: #m addr ; : #@ addr @ ; : #! addr ! contents ;
: #g SCNT IF DUP ADDR - PAGE < IF
  ADDR - pos! ELSE DUP addr! $f AND pos! THEN THEN ;
: #G bottom 4 typep IF #g THEN ;
: #a ASCII? NOT TO ASCII? showpos ;
: #f #@ #g ; : #e #m #f ;
: _h SPC> showpos 2 typep ;
: _a showpos KEY DUP SPC < IF DROP 0 ELSE DUP EMIT 1 THEN ;
: #R BEGIN SPC> ASCII? IF _a ELSE _h THEN ( n? f ) IF
    addr C! rfshln #l 0 ELSE 1 THEN UNTIL rfshln ;
```

B39

```
\ Memory Editor, #q handle ME
: #q 1 TO HALT? ;
: handle ( c -- f )
  CMD 1+ C! CMD 2 FIND IF EXECUTE THEN ;
: ME 0 TO HALT? CLRSCR contents 0 pos! BEGIN
    status KEY handle HALT? UNTIL bottom ;
```

1.9 AVR SPI programmer: 40-43

B40

```
\ AVR Programmer, B160-B163. doc/avr.txt
\ page size in words, 64 is default on atmega328P
64 VALUE aspfpgsz
0 VALUE aspprevx
: _x ( a -- b ) DUP TO aspprevx (spix) ;
: _xc ( a -- b ) DUP (spix) ( a b )
    DUP aspprevx = NOT IF ABORT" AVR err" THEN ( a b )
    SWAP TO aspprevx ( b ) ;
: _cmd ( b4 b3 b2 b1 -- r4 ) _xc DROP _xc DROP _xc DROP _x ;
: asprdy ( -- ) BEGIN 0 0 0 $f0 _cmd 1 AND NOT UNTIL ;
: asp$ ( spidevid -- )
    ( RESET pulse ) DUP (spie) 0 (spie) (spie)
    ( wait >20ms ) 220 TICKS
    ( enable prog ) $ac (spix) DROP
    $53 _x DROP 0 _xc DROP 0 _x DROP ;
: asperase 0 0 $80 $ac _cmd asprdy ;
```

B41

```
( fuse access. read/write one byte at a time )
: aspfl@ ( -- lfuse ) 0 0 0 $50 _cmd ;
: aspfh@ ( -- hfuse ) 0 0 $08 $58 _cmd ;
: aspfe@ ( -- efuse ) 0 0 $00 $58 _cmd ;
: aspfl! ( lfuse -- ) 0 $a0 $ac _cmd ;
: aspfh! ( hfuse -- ) 0 $a8 $ac _cmd ;
: aspfe! ( efuse -- ) 0 $a4 $ac _cmd ;
```

B42

```
: aspfb! ( n a --, write word n to flash buffer addr a )
    SWAP L|M SWAP ( a hi lo ) ROT ( hi lo a )
    DUP ROT ( hi a a lo ) SWAP ( hi a lo a )
    0 $40 ( hi a lo a 0 $40 ) _cmd DROP ( hi a )
    0 $48 _cmd DROP ;
: aspfp! ( page --, write buffer to page )
    0 SWAP aspfpgsz * L|M ( 0 lsb msb )
    $4c _cmd DROP asprdy ;
: aspf@ ( page a -- n, read word from flash )
    SWAP aspfpgsz * OR ( addr ) L|M ( lsb msb )
    2DUP 0 ROT> ( lsb msb 0 lsb msb )
    $20 _cmd ( lsb msb low )
    ROT> 0 ROT> ( low 0 lsb msb ) $28 _cmd <<8 OR ;
```

B43

```
: aspe@ ( addr -- byte, read from EEPROM )
    0 SWAP L|M SWAP ( 0 msb lsb )
    $a0 ( 0 msb lsb $a0 ) _cmd ;
: aspe! ( byte addr --, write to EEPROM )
    L|M SWAP ( b msb lsb )
    $c0 ( b msb lsb $c0 ) _cmd DROP asprdy ;
```

1.10 Sega ROM signer: 45

B45

```
( Sega ROM signer. See doc/sega.txt )
: segasig ( addr size -- )
  $2000 OVER LSHIFT ( a sz bytesz ) $10 - >R ( a sz )
  SWAP >A 0 BEGIN ( sz csum ) AC@+ + NEXT ( sz csum )
  'T' AC!+ 'M' AC!+ 'R' AC!+ SPC AC!+ 'S' AC!+
  'E' AC!+ 'G' AC!+ 'A' AC!+ 0 AC!+ 0 AC!+
  ( sum's LSB ) DUP AC!+ ( MSB ) >>8 AC!+
  ( sz ) 0 AC!+ 0 AC!+ 0 AC!+ $4a + AC!+ ;
```

1.11 Virgil's Workspace: 50-51

B50

```
CREATE MSPAN_DISK 0 C,
CREATE (msdsks) 100 C, 100 C, 180 C, 0 C,

: _ ( dsk -- ) DUP MSPAN_DISK C! S" Need disk " STYPE . SPC> ;
: prompt _ KEY DROP ;
: dskchk ( blk -- newblk ) A>R (msdsks) >A BEGIN
    AC@+ - DUP 0< AC@ NOT OR UNTIL A- AC@ + ( newblk )
  A> (msdsks) - ( newblk dsk ) DUP MSPAN_DISK C@ = NOT IF
    prompt ELSE DROP THEN ( blk ) R>A ;
```

B51

```
\ utility to quickly examine freshly written asm words
0 VALUE mark
: see mark >A HERE mark - >R BEGIN
  AC@+ .x SPC> NEXT mark 'HERE ! ;
\ HERE TO mark
```

1.12 Cross compilation: 200-205

B200

```
\ Cross compilation program, generic part. See doc/cross
0 VALUE BIN( \ binary start in target's addr
0 VALUE XORG \ binary start address in host's addr
0 VALUE BIGEND? \ is target big-endian?
3 VALUES L1 L2 L3
: PC HERE XORG - BIN( + ;
: PC2A ( pc -- a ) HERE PC - ( org ) + ;
: XSTART ( bin( -- ) TO BIN( HERE TO XORG ;
: OALLOT ( oa -- ) XORG + HERE - ALLOT0 ;
: |T L|M BIGEND? NOT IF SWAP THEN ;
: T! ( n a -- ) SWAP |T ROT C!+ C! ;
: T, ( n -- ) |T C, C, ;
: T@ C@+ SWAP C@ BIGEND? IF SWAP THEN <<8 OR ;
: XCOMPCL 201 202 LOADR ; : XCOMPCH 203 205 LOADR ;
: XCOMPC XCOMPCL XCOMPCH ; : FONTC 262 263 LOADR ;
```

B201

```
\ Cross compilation program. COS-specific. See doc/cross
: COREL 210 224 LOADR ; : COREH 225 229 LOADR ;
: BLKSUB 230 234 LOADR ; : GRIDSUB 240 241 LOADR ;
: PS2SUB 246 248 LOADR ; : RXTXSUB 235 LOAD ;
: MSPANSUB 237 LOAD ; : SDCSUB 250 258 LOADR ;
'? HERESTART NOT [IF] 0 VALUE HERESTART [THEN]
0 VALUE XCURRENT \ CURRENT in target system, in target's addr
8 VALUES lblnext lblcell lbldoes lblxt lblval
  lblhere lblmain lblboot
'? 'A NOT [IF] SYSVARS $06 + VALUE 'A [THEN]
'? 'N NOT [IF] SYSVARS $08 + VALUE 'N [THEN]
6 VALUES (n)* (b)* (br)* (?br)* EXIT* (next)*
CREATE '~ 2 ALLOT
```

B202

```
\ Cross compilation program
: _xoff ( a -- a ) XORG BIN( - ;
: _wl ( w -- len ) 1- C@ $7f AND ;
: _ws ( w len -- sa ) - 3 - ;
: _xfind ( sa sl -- w? f ) >R >A XCURRENT BEGIN ( w R:sl )
  _xoff + DUP _wl R@ = IF ( w ) DUP R@ _ws A> R@ ( w a1 a2 u )
  []= IF ( w ) R~ 1 EXIT THEN THEN
  3 - ( prev field ) T@ ?DUP NOT UNTIL R~ 0 ( not found ) ;
: XFIND ( sa sl -- w ) _xfind NOT IF (wnf) THEN _xoff - ;
: X' WORD XFIND ;
: '? WORD _xfind DUP IF NIP THEN ;
: ENTRY
  WORD TUCK MOVE, XCURRENT T, C, HERE _xoff - TO XCURRENT ;
```

B203

```
\ Cross compilation program
: ;CODE lblnext JMPi, ;
: ALIAS X' ENTRY JMPi, ; : *ALIAS ENTRY JMP(i), ;
: CONSTANT ENTRY i>, ;CODE ;
: CONSTS >R BEGIN RUN1 CONSTANT NEXT ;
: CONSTS+ ( off n -- )
  >R BEGIN RUN1 OVER + CONSTANT NEXT DROP ;
: *VALUE ENTRY (i)>, ;CODE ; : CREATE ENTRY lblcell CALLi, ;
: _ ( lbl str -- )
  CURWORD S= IF XCURRENT SWAP TO EXECUTE ELSE DROP THEN ;
: CODE ENTRY ['] EXIT* S" EXIT" _ ['] (b)* S" (b)" _
  ['] (n)* S" (n)" _ ['] (br)* S" (br)" _
  ['] (?br)* S" (?br)" _ ['] (next)* S" (next)" _ ;
: LITN DUP $ff > IF (n)* T, T, ELSE (b)* T, C, THEN ;
```

B204

```
\ Cross compilation program
: imm? ( w -- f ) 1- C@ $80 AND ;
: compile BEGIN WORD S" ;" S= IF EXIT* T, EXIT THEN
  CURWORD PARSE IF LITN ELSE CURWORD _xfind IF ( w )
    DUP imm? IF ABORT" immed!" THEN _xoff - T,
  ELSE CURWORD FIND IF ( w )
    DUP imm? IF EXECUTE ELSE (wnf) THEN
    ELSE (wnf) THEN
  THEN ( _xfind ) THEN ( PARSE ) AGAIN ;
: :~ HERE _xoff - '~ ! lblxt CALLi, compile ;
: ~ '~ @ T, ; IMMEDIATE
: _ CODE lblxt CALLi, compile ; \ : can't have its name now
: ?: '? IF S" ;" WAITW ELSE CURWORD WORD! _ THEN ;
: ~DOER ENTRY lbldoes CALLi, [COMPILE] ~ ;
```

B205

```
\ Cross compilation program
: XWRAP COREH XCURRENT lblhere PC2A T!
  HERESTART ?DUP NOT IF PC THEN lblhere PC2A 1+ 1+ T! ;
: ['] WORD XFIND LITN ; IMMEDIATE
: COMPILE [COMPILE] ['] S" ," XFIND T, ; IMMEDIATE
: IF (?br)* T, HERE 1 ALLOT ; IMMEDIATE
: ELSE (br)* T, 1 ALLOT [COMPILE] THEN HERE 1- ; IMMEDIATE
: AGAIN (br)* T, HERE - C, ; IMMEDIATE
: UNTIL (?br)* T, HERE - C, ; IMMEDIATE
: NEXT (next)* T, HERE - C, ; IMMEDIATE
: S" (br)* T, HERE 1 ALLOT HERE ," TUCK HERE -^ SWAP
  [COMPILE] THEN SWAP _xoff - LITN LITN ; IMMEDIATE
: [COMPILE] WORD XFIND T, ; IMMEDIATE
: IMMEDIATE XCURRENT _xoff + 1- DUP C@ $80 OR SWAP C! ;
':' ' _ 4 - C! \ give : its real name now
0 XSTART
```

1.13 Core words: 210-229

B210

```
\ Core Forth words. See doc/cross. SYSVARS
SYSVARS 12 CONSTS+
  $00 IOERR $02 'CURRENT $04 'HERE    $0a NL       $0c LN<
  $0e 'EMIT $10 'KEY?    $12 'CURWORD $16 '(wnf)
  $1c 'IN(  $1e 'IN>     $20 INBUF
SYSVARS $02 + *VALUE CURRENT    SYSVARS $04 + *VALUE HERE
SYSVARS $0e + *ALIAS EMIT       SYSVARS $10 + *ALIAS KEY?
SYSVARS $1c + *VALUE IN(        SYSVARS $1e + *VALUE IN>
$40 CONSTANT LNSZ
CODE NOOP ;CODE
```

B211

```
\ Core words, basic arithmetic and stack management
?: = - NOT ;
?: > SWAP < ;
?: 0< $7fff > ; ?: 0>= $8000 < ; ?: >= < NOT ; ?: <= > NOT ;
?: 1+ 1 + ; ?: 1- 1 - ;
?: 2DROP DROP DROP ;
?: 2DUP OVER OVER ;
?: NIP SWAP DROP ;
?: TUCK SWAP OVER ;
?: ROT> ROT ROT ;
?: =><= ( n l h -- f ) OVER - ROT> ( h n l ) - >= ;
: / /MOD NIP ; : MOD /MOD DROP ;
?: <> ( n n -- l h ) 2DUP > IF SWAP THEN ;
?: MIN <> DROP ; ?: MAX <> NIP ; ?: -^ SWAP - ;
```

B212

```
\ Core words, bit shifting, A register, LEAVE VAL L|M +!
?: << 2 * ;     ?: >> 2 / ;
?: <<8 $100 * ; ?: >>8 $100 / ;
?: RSHIFT ?DUP IF >R BEGIN >> NEXT THEN ;
?: LSHIFT ?DUP IF >R BEGIN << NEXT THEN ;
?: L|M DUP <<8 >>8 SWAP >>8 ;
?: +! ( n a -- ) TUCK @ + SWAP ! ;
?: A> [ 'A LITN ] @ ;    ?: >A [ 'A LITN ] ! ;
?: A>R R> A> >R >R ;     ?: R>A R> R> >A >R ;
?: A+ 1 [ 'A LITN ] +! ; ?: A- -1 [ 'A LITN ] +! ;
?: AC@ A> C@ ;           ?: AC! A> C! ;
: AC@+ AC@ A+ ;          : AC!+ AC! A+ ;
: LEAVE R> R~ 1 >R >R ;
?: TO 1 [ SYSVARS $18 + LITN ] C! ;
```

B213

```
\ Core words, C@+ ALLOT FILL IMMEDIATE , L, M, MOVE MOVE, ..
?: C@+ DUP 1+ SWAP C@ ;
?: C!+ TUCK C! 1+ ;
: ALLOT 'HERE +! ;
?: FILL ( a u b -- ) \ *A*
  ROT> >R >A BEGIN DUP AC!+ NEXT DROP ;
: ALLOT0 ( u -- ) HERE OVER 0 FILL ALLOT ;
: IMMEDIATE CURRENT 1- DUP C@ $80 OR SWAP C! ;
: , HERE ! 2 ALLOT ; : C, HERE C! 1 ALLOT ;
: L, DUP C, >>8 C, ; : M, DUP >>8 C, C, ;
?: MOVE ( src dst u -- ) ?DUP IF
  >R >A BEGIN ( src ) C@+ AC!+ NEXT DROP THEN ;
: MOVE, ( a u -- ) HERE OVER ALLOT SWAP MOVE ;
```

B214

```
\ Core words, [C]? CRC16 []= JMPi! CALLi!
?: JMPi! [ X' NOOP PC2A C@ ( jmp op ) LITN ] SWAP C!+ ! 3 ;
?: CALLi! [ X' MOVE, PC2A C@ ( call op ) LITN ] SWAP C!+ ! 3 ;
?: [C]? ( c a u -- i ) \ Guards A
  ?DUP NOT IF 2DROP -1 EXIT THEN A>R OVER >R >R >A ( c )
  BEGIN DUP AC@+ = IF LEAVE THEN NEXT ( c )
  A- AC@ = IF A> R> - ( i ) ELSE R~ -1 THEN R>A ;
?: []= ( a1 a2 u -- f ) \ Guards A
  ?DUP NOT IF 2DROP 1 EXIT THEN A>R >R >A ( a1 )
  BEGIN AC@+ OVER C@ = NOT IF R~ R>A DROP 0 EXIT THEN 1+ NEXT
  DROP R>A 1 ;
?: CRC16 ( c n -- c )
  <<8 XOR 8 >R BEGIN ( c )
    DUP 0< IF << $1021 XOR ELSE << THEN NEXT ;
```

B215

```
\ Core words, STYPE SPC> NL> STACK? LITN
: STYPE >R >A BEGIN AC@+ EMIT NEXT ;
5 CONSTS $04 EOT $08 BS $0a LF $0d CR $20 SPC
: SPC> SPC EMIT ;
: NL> NL @ L|M ?DUP IF EMIT THEN EMIT ;
: STACK? SCNT 0< IF S" stack underflow" STYPE ABORT THEN ;
: LITN DUP >>8 IF COMPILE (n) , ELSE COMPILE (b) C, THEN ;
```

B216

```
\ Core words, number formatting
: FMTD ( n a -- sa sl ) \ *A*
  6 + >A A>R DUP >R DUP 0< IF 0 -^ THEN BEGIN ( n )
    10 /MOD ( d q ) A- SWAP '0' + AC! ?DUP NOT UNTIL
  R> 0< IF A- '-' AC! THEN R> A> TUCK - ;
PC TO L1 ," 0123456789abcdef"
:~ ( n a 'len -- sa sl ) \ *A*
  C@ DUP >R DUP >R + >A BEGIN ( n ) 16 /MOD ( d q ) A- SWAP
    [ L1 LITN ] + C@ AC! NEXT DROP A> R> ;
~DOER FMTx 2 C, ~DOER FMTX 4 C,
:~ ( n 'w -- sa sl ) @ A>R HERE SWAP EXECUTE STYPE R>A ;
~DOER . X' FMTD T,
~DOER .x X' FMTx T,
~DOER .X X' FMTX T,
```

B217

```
\ Core words, literal parsing
:~ ( sl -- n? f ) \ parse unsigned decimal
  >R 0 BEGIN ( r )
    10 * AC@+ ( r c ) '0' - DUP 9 > IF
      2DROP R~ 0 EXIT THEN + NEXT ( r ) 1 ;
: PARSE ( sa sl -- n? f ) \ *A*
  OVER C@ ''' = IF ( sa sl )
    3 = IF 1+ DUP 1+ C@ ''' = IF C@ 1 EXIT THEN THEN
    DROP 0 EXIT THEN ( sa sl )
  OVER C@ '$' = IF ( sa sl ) 1- >R 1+ >A 0 BEGIN ( r )
    16 * AC@+ ( r c ) $20 OR [ L1 LITN ] ( B216 ) $10 [C]?
    DUP 0< IF 2DROP R~ 0 EXIT THEN + NEXT ( r ) 1 EXIT THEN
  SWAP >A DUP 1 > AC@ '-' = AND IF ( sl )
    A+ 1- ~ IF 0 -^ 1 ELSE 0 THEN ELSE ~ THEN ;
```

B218

```
\ Core words, input buffer
: KEY BEGIN KEY? UNTIL ;
: IN) IN( LNSZ + ;
PC BS C, $7f ( DEL ) C,
: BS? [ ( PC ) LITN ] 2 [C]? 0>= ;
: WS? SPC <= ;
\ type c into ptr inside INBUF. f=true if typing should stop
: LNTYPE ( ptr c -- ptr+-1 f )
  DUP BS? IF ( ptr c )
    DROP DUP IN( > IF 1- BS EMIT THEN SPC> BS EMIT 0
  ELSE ( ptr c ) \ non-BS
    DUP SPC < IF DROP DUP IN) OVER - 0 FILL 1 ELSE
      TUCK EMIT C!+ DUP IN) = THEN THEN ;
```

B219

```
\ Core words, input buffer, ,"
: RDLN ( -- ) \ Read 1 line in IN(
  S"  ok" STYPE NL> IN( BEGIN KEY LNTYPE UNTIL DROP NL> ;
: IN<? ( -- c-or-0 )
  IN> IN) < IF IN> C@+ SWAP 'IN> ! ELSE 0 THEN ;
: IN< ( -- c ) IN<? ?DUP NOT IF
    LN< @ EXECUTE IN( 'IN> ! SPC THEN ;
: IN$ ['] RDLN LN< ! INBUF 'IN( ! IN) 'IN> ! ;
: ," BEGIN IN< DUP '"' = IF DROP EXIT THEN C, AGAIN ;
```

B220

```
\ Core words, WORD parsing
: TOWORD ( -- ) BEGIN IN< WS? NOT UNTIL ;
: CURWORD ( -- sa sl ) 'CURWORD 1+ @ 'CURWORD C@ ;
:~ ( f sa sl -- ) 'CURWORD C!+ TUCK ! 1+ 1+ C! ;
: WORD ( -- sa sl )
  'CURWORD 3 + C@ IF CURWORD ELSE
    TOWORD IN> 1- 0 ( sa sl ) BEGIN 1+ IN<? WS? UNTIL THEN
  ( sa sl ) 2DUP 0 ROT> ~ ;
: WORD! 1 ROT> ~ ;
```

B221

```
\ Core words, FIND (wnf) RUN1 INTERPRET nC,
?: FIND ( sa sl -- w? f ) \ Guards A
  A>R >R >A CURRENT BEGIN ( w R:sl )
  DUP 1- C@ $7f AND ( wlen ) R@ = IF ( w )
    DUP R@ - 3 - A> R@ ( w a1 a2 u )
    []= IF ( w ) R~ 1 R>A EXIT THEN THEN
  3 - ( prev field ) @ ?DUP NOT UNTIL R~ 0 R>A ( not found ) ;
: (wnf) CURWORD STYPE S"  word not found" STYPE ABORT ;
: RUN1 ( -- ) \ interpret next word
  WORD PARSE NOT IF
    CURWORD FIND NOT IF '(wnf) @ THEN EXECUTE STACK? THEN ;
: INTERPRET BEGIN RUN1 AGAIN ;
: nC, ( n -- ) >R BEGIN RUN1 C, NEXT ;
```

B222

```
\ Core words, CODE '? ' TO FORGET
: CODE WORD TUCK MOVE, ( len )
  CURRENT , C, \ write prev value and size
  HERE 'CURRENT ! ;
: '? WORD FIND DUP IF NIP THEN ;
: ' WORD FIND NOT IF (wnf) THEN ;
: FORGET
  ' DUP ( w w )
  \ HERE must be at the end of prev's word, that is, at the
  \ beginning of w.
  DUP 1- C@ ( len ) $7f AND ( rm IMMEDIATE )
  3 + ( fixed header len ) - 'HERE ! ( w )
  ( get prev addr ) 3 - @ 'CURRENT ! ;
```

B223

```
\ Core words, S= WAITW [IF] _bchk
: S= ( sa1 sl1 sa2 sl2 -- f )
  ROT OVER = IF ( same len, s2 s1 l ) []=
  ELSE DROP 2DROP 0 THEN ;
: WAITW ( sa sl -- ) BEGIN 2DUP WORD S= UNTIL 2DROP ;
: [IF] NOT IF S" [THEN]" WAITW THEN ;
ALIAS NOOP [THEN]
: _bchk DUP $80 + $ff > IF S" br ovfl" STYPE ABORT THEN ;
```

B224

```
\ Core words, DUMP .S
: DUMP ( n a -- ) \ *A*
  >A 8 /MOD SWAP IF 1+ THEN >R BEGIN
    ':' EMIT A> DUP .x SPC> ( a )
    4 >R BEGIN AC@+ .x AC@+ .x SPC> NEXT ( a ) >A
    8 >R BEGIN AC@+ DUP SPC - $5e > IF DROP '.' THEN EMIT NEXT
  NL> NEXT ;
: PSDUMP SCNT NOT IF EXIT THEN
  SCNT >A BEGIN DUP .X SPC> >R SCNT NOT UNTIL
  BEGIN R> SCNT A> = UNTIL ;
: .S ( -- )
  S" SP " STYPE SCNT .x SPC> S" RS " STYPE RCNT .x SPC>
  S" -- " STYPE STACK? PSDUMP ;
```

B225

```
\ Core high, CREATE DOER DOES> CODE ALIAS VALUE
: ;CODE [ lblnext LITN ] HERE JMPi! ALLOT ;
: CREATE CODE [ lblcell LITN ] HERE CALLi! ALLOT ;
: DOER CODE [ lbldoes LITN ] HERE CALLi! 1+ 1+ ALLOT ;
: _ R> CURRENT 3 + ! ; \ Popping RS makes us EXIT from parent
: DOES> COMPILE _ [ lblxt LITN ] HERE CALLi! ALLOT ; IMMEDIATE
: ALIAS ' CODE HERE JMPi! ALLOT ;
: VALUE CODE [ lblval LITN ] HERE CALLi! ALLOT , ;
: VALUES >R BEGIN 0 VALUE NEXT ;
: CONSTS >R BEGIN RUN1 VALUE NEXT ;
```

B226

```
\ Core high, BOOT
:~ IN$ INTERPRET BYE ;
'~ @ lblmain PC2A T! \ set jump in QUIT
PC TO lblhere 4 ALLOT \ CURRENT, HERESTART
: BOOT [ lblhere LITN ] 'CURRENT 4 MOVE
  ['] (emit) 'EMIT ! ['] (key?) 'KEY? ! ['] (wnf) '(wnf) !
  0 'CURWORD 3 + C!
  0 IOERR ! $0d0a ( CR/LF ) NL !
  0 [ SYSVARS $18 ( TO? ) + LITN ] C!
  INIT S" Collapse OS" STYPE ABORT ;
XCURRENT lblboot PC2A T! \ initial jump to BOOT
```

B227

```
\ Core high, :
: XTCOMP [ lblxt LITN ] HERE CALLi! ALLOT BEGIN
    WORD S" ;" S= IF COMPILE EXIT EXIT THEN
    CURWORD PARSE IF LITN ELSE CURWORD FIND IF
      DUP 1- C@ $80 AND ( imm? ) IF EXECUTE ELSE , THEN
    ELSE '(wnf) @ EXECUTE THEN THEN
  AGAIN ;
: : CODE XTCOMP ;
```

B228

```
\ Core high, IF..ELSE..THEN ( \
: IF ( -- a | a: br cell addr )
  COMPILE (?br) HERE 1 ALLOT ( br cell allot ) ; IMMEDIATE
: THEN ( a -- | a: br cell addr )
  DUP HERE -^ _bchk SWAP ( a-H a ) C! ; IMMEDIATE
: ELSE ( a1 -- a2 | a1: IF cell a2: ELSE cell )
  COMPILE (br) 1 ALLOT [COMPILE] THEN
  HERE 1- ( push a. 1- for allot offset ) ; IMMEDIATE
: ( S" )" WAITW ; IMMEDIATE
: \ IN) 'IN> ! ; IMMEDIATE
: S"
  COMPILE (br) HERE 1 ALLOT HERE ," TUCK HERE -^ SWAP
  [COMPILE] THEN SWAP LITN LITN ; IMMEDIATE
```

B229

```
\ Core high, .", ABORT", BEGIN..AGAIN..UNTIL, many others.
: ." [COMPILE] S" COMPILE STYPE ; IMMEDIATE
: ABORT" [COMPILE] ." COMPILE ABORT ; IMMEDIATE
: BEGIN HERE ; IMMEDIATE
: AGAIN COMPILE (br) HERE - _bchk C, ; IMMEDIATE
: UNTIL COMPILE (?br) HERE - _bchk C, ; IMMEDIATE
: NEXT COMPILE (next) HERE - _bchk C, ; IMMEDIATE
: [ INTERPRET ; IMMEDIATE
: ] R~ R~ ; \ INTERPRET+RUN1
: COMPILE ' LITN ['] , , ; IMMEDIATE
: [COMPILE] ' , ; IMMEDIATE
: ['] ' LITN ; IMMEDIATE
```

1.14 BLK subsystem: 230-234

B230

```
\ BLK subsystem. See doc/blk
BLK_MEM CONSTANT BLK( \ $400 + "\S "
BLK_MEM $400 + CONSTANT BLK)
\ Current blk pointer -1 means "invalid"
BLK_MEM $403 + DUP CONSTANT 'BLK> *VALUE BLK>
\ Whether buffer is dirty
BLK_MEM $405 + CONSTANT BLKDTY
BLK_MEM $407 + CONSTANT BLKIN>
: BLK$ 0 BLKDTY ! -1 'BLK> ! S" \S " BLK) SWAP MOVE ;
```

B231

```
: BLK! ( -- ) BLK> BLK( (blk!) 0 BLKDTY ! ;
: FLUSH BLKDTY @ IF BLK! THEN -1 'BLK> ! ;
: BLK@ ( n -- )
  DUP BLK> = IF DROP EXIT THEN
  FLUSH DUP 'BLK> ! BLK( (blk@) ;
: BLK!! 1 BLKDTY ! ;
: WIPE BLK( 1024 SPC FILL BLK!! ;
: COPY ( src dst -- ) FLUSH SWAP BLK@ 'BLK> ! BLK! ;
```

B232

```
: LNLEN ( a -- len ) \ len based on last visible char in line
  1- LNSZ >R BEGIN
    DUP R@ + C@ SPC > IF DROP R> EXIT THEN NEXT DROP 0 ;
: EMITLN ( a -- ) \ emit LNSZ chars from a or stop at CR
  DUP LNLEN ?DUP IF
    >R >A BEGIN AC@+ EMIT NEXT ELSE DROP THEN NL> ;
: LIST ( n -- ) \ print contents of BLK n
  BLK@ 16 >R 0 BEGIN ( n )
    DUP 1+ DUP 10 < IF SPC> THEN . SPC>
    DUP LNSZ * BLK( + EMITLN 1+ NEXT DROP ;
: INDEX ( b1 b2 -- ) \ print first line of blocks b1 through b2
  OVER - 1+ >R BEGIN
    DUP . SPC> DUP BLK@ BLK( EMITLN 1+ NEXT DROP ;
```

B233

```
: \S BLK) 'IN( ! IN( 'IN> ! ;
:~ ( -- ) IN) 'IN( ! ;
: LOAD
  IN> BLKIN> ! [ '~ @ LITN ] LN< ! BLK@ BLK( 'IN( ! IN( 'IN> !
  BEGIN RUN1 IN( BLK) = UNTIL IN$ BLKIN> @ 'IN> ! ;
\ >R R> around LOAD is to avoid bad blocks messing PS up
: LOADR OVER - 1+ >R BEGIN
  DUP . SPC> DUP >R LOAD R> 1+ NEXT DROP ;
```

B234

```
\ Application loader, to include in boot binary
: ED 1 LOAD ( MOVE- ) 20 24 LOADR ;
: VE 5 LOAD ( wordtbl ) ED 25 32 LOADR ;
: ME 35 39 LOADR ;
: ARCHM 301 LOAD ;
: RXTX 10 15 LOADR ;
: XCOMP 200 LOAD ;
```

1.15 RX/TX subsystem: 235

B235

```
\ RX/TX subsystem. See doc/rxtx
RXTX_MEM CONSTANT _emit
RXTX_MEM 2 + CONSTANT _key
: RX< BEGIN RX<? UNTIL ;
: RX<< 0 BEGIN DROP RX<? NOT UNTIL ;
: TX[ 'EMIT @ _emit ! [’] TX> 'EMIT ! ;
: ]TX _emit @ 'EMIT ! ;
: RX[ 'KEY? @ _key ! [’] RX<? 'KEY? ! ;
: ]RX _key @ 'KEY? ! ;
```

1.16 Media Span subsystem: 237

B237

```
\ Media Spanning subsystem. see doc/mspan
MSPAN_MEM CONSTANT MSPAN_DISK

?: DRVSEL ( drv -- ) DROP ;
: prompt ( dsk -- )
  DUP MSPAN_DISK C! S" Need disk " STYPE . SPC> KEY '0' -
  DUP 10 < IF DRVSEL ELSE DROP THEN ;
: MSPAN$ 0 MSPAN_DISK C! ;
: dskchk ( blk -- newblk ) A>R (msdsks) >A BEGIN
    AC@+ - DUP 0< AC@ NOT OR UNTIL A- AC@ + ( newblk )
  A> (msdsks) - ( newblk dsk ) DUP MSPAN_DISK C@ = NOT IF
    prompt ELSE DROP THEN ( blk ) R>A ;
:~ ( blk dest 'w -- ) ROT dskchk ROT> @ EXECUTE ;
~DOER (blk@) X' (ms@) T,
~DOER (blk!) X' (ms!) T,
```

1.17 Grid subsystem: 240-241

B240

```
\ Grid subsystem. See doc/grid.
GRID_MEM DUP CONSTANT 'XYPOS *VALUE XYPOS
?: CURSOR! 2DROP ;
: XYPOS! COLS LINES * MOD DUP XYPOS CURSOR! 'XYPOS ! ;
: AT-XY ( x y -- ) COLS * + XYPOS! ;
?: NEWLN ( oldln -- newln )
  1+ LINES MOD DUP COLS * ( pos )
  COLS >R BEGIN SPC OVER CELL! 1+ NEXT DROP ;
?: CELLS! ( a pos u -- )
  ?DUP IF >R SWAP >A BEGIN ( pos ) AC@+ OVER CELL! 1+ NEXT
    ELSE DROP THEN DROP ;
: STYPEC ( sa sl pos -- ) SWAP CELLS! ;
?: FILLC ( pos n c )
  SWAP >R SWAP BEGIN ( b pos ) 2DUP CELL! 1+ NEXT 2DROP ;
: CLRSCR 0 COLS LINES * SPC FILLC 0 XYPOS! ;
```

B241

```
:~ ( line feed ) XYPOS COLS / NEWLN COLS * XYPOS! ;
?: (emit)
    DUP BS? IF
      DROP SPC XYPOS TUCK CELL! ( pos ) 1- XYPOS! EXIT THEN
    DUP CR = IF DROP SPC XYPOS CELL! ~ EXIT THEN
    DUP SPC < IF DROP EXIT THEN
    XYPOS CELL!
    XYPOS 1+ DUP COLS MOD IF XYPOS! ELSE DROP ~ THEN ;
: GRID$ 0 'XYPOS ! ;
```

1.18 PS/2 keyboard subsystem: 245-248

B245

```
PS/2 keyboard subsystem

Provides (key?) from a driver providing the PS/2 protocol. That
is, for a driver taking care of providing all key codes emanat-
ing from a PS/2 keyboard, this subsystem takes care of mapping
those keystrokes to ASCII characters. This code is designed to
be cross-compiled and loaded with drivers.

Requires PS2_MEM to be defined.

Load range: 246-249
```

B246

```
: PS2_SHIFT [ PS2_MEM LITN ] ; : PS2$ 0 PS2_SHIFT C! ;
\ A list of the values associated with the $80 possible scan
\ codes of the set 2 of the PS/2 keyboard specs. 0 means no
\ value. That value is a character that can be read in (key?)
\ No make code in the PS/2 set 2 reaches $80.
\ TODO: I don't know why, but the key 2 is sent as $1f by 2 of
\ my keyboards. Is it a timing problem on the ATtiny?
CREATE PS2_CODES $80 nC,
0   0   0   0   0   0   0   0 0 0   0   0   0   9   '`' 0
0   0   0   0   0   'q' '1' 0 0 0   'z' 's' 'a' 'w' '2' '2'
0   'c' 'x' 'd' 'e' '4' '3' 0 0 32  'v' 'f' 't' 'r' '5' 0
0   'n' 'b' 'h' 'g' 'y' '6' 0 0 0   'm' 'j' 'u' '7' '8' 0
0   ',' 'k' 'i' 'o' '0' '9' 0 0 '.' '/' 'l' ';' 'p' '-' 0
0   0   ''' 0   '[' '=' 0   0 0 0   13  ']' 0   '\' 0   0
0   0   0   0   0   0   8   0 0 '1' 0   '4' '7' 0   0   0
'0' '.' '2' '5' '6' '8' 27  0 0 0   '3' 0   0   '9' 0   0
```

B247

```
( Same values, but shifted ) $80 nC,
0   0   0   0   0   0   0   0 0 0   0   0   0   9   '~' 0
0   0   0   0   0   'Q' '!' 0 0 0   'Z' 'S' 'A' 'W' '@' '@'
0   'C' 'X' 'D' 'E' '$' '#' 0 0 32  'V' 'F' 'T' 'R' '%' 0
0   'N' 'B' 'H' 'G' 'Y' '^' 0 0 0   'M' 'J' 'U' '&' '*' 0
0   '<' 'K' 'I' 'O' ')' '(' 0 0 '>' '?' 'L' ':' 'P' '_' 0
0   0   '"' 0   '{' '+' 0   0 0 0   13  '}' 0   '|' 0   0
0   0   0   0   0   0   8   0 0 0   0   0   0   0   0   0
0   0   0   0   0   0   27  0 0 0   0   0   0   0   0   0
```

B248

```
: _shift? ( kc -- f ) DUP $12 = SWAP $59 = OR ;
: (key?) ( -- c? f )
    (ps2kc) DUP NOT IF EXIT THEN ( kc )
    DUP $e0 ( extended ) = IF ( ignore ) DROP 0 EXIT THEN
    DUP $f0 ( break ) = IF DROP ( )
        ( get next kc and see if it's a shift )
        BEGIN (ps2kc) ?DUP UNTIL ( kc )
        _shift? IF ( drop shift ) 0 PS2_SHIFT C! THEN
        ( whether we had a shift or not, we return the next )
        0 EXIT THEN
    DUP $7f > IF DROP 0 EXIT THEN
    DUP _shift? IF DROP 1 PS2_SHIFT C! 0 EXIT THEN
    ( ah, finally, we have a gentle run-of-the-mill KC )
    PS2_CODES PS2_SHIFT C@ IF $80 + THEN + C@ ( c, maybe 0 )
    ?DUP ( c? f ) ;
```

1.19 SD Card subsystem: 250-258

B250

```
\ SD Card subsystem Load range: B250-B258
SDC_MEM CONSTANT SDC_SDHC
: _idle ( -- n ) $ff (spix) ;

( spix $ff until the response is something else than $ff
  for a maximum of 20 times. Returns $ff if no response. )
: _wait ( -- n )
  0 ( dummy ) 20 >R BEGIN
    DROP _idle DUP $ff = NOT IF LEAVE THEN NEXT ;

( adjust block for LBA for SD/SDHC )
: _badj ( arg1 arg2 -- arg1 arg2 )
  SDC_SDHC @ IF 0 SWAP ELSE DUP 128 / SWAP <<8 << THEN ;
```

B251

```
( The opposite of sdcWaitResp: we wait until response is $ff.
  After a successful read or write operation, the card will be
  busy for a while. We need to give it time before interacting
  with it again. Technically, we could continue processing on
  our side while the card it busy, and maybe we will one day,
  but at the moment, I'm having random write errors if I don't
  do this right after a write, so I prefer to stay cautious
  for now. )
: _ready ( -- ) BEGIN _idle $ff = UNTIL ;
```

B252

```
( Computes n into crc c with polynomial $09
  Note that the result is "left aligned", that is, that 8th
  bit to the "right" is insignificant (will be stop bit). )
: _crc7 ( c n -- c )
  XOR 8 >R BEGIN ( c )
    << ( c<<1 ) DUP >>8 IF
      ( MSB was set, apply polynomial )
      <<8 >>8 $12 XOR ( $09 << 1, we apply CRC on high bits )
    THEN NEXT ;
( send-and-crc7 )
: _s+crc ( n c -- c ) SWAP DUP (spix) DROP _crc7 ;
```

B253

```
( cmd arg1 arg2 -- resp )
( Sends a command to the SD card, along with arguments and
  specified CRC fields. (CRC is only needed in initial commands
  though). This does *not* handle CS. You have to
  select/deselect the card outside this routine. )
: _cmd
    _wait DROP ROT    ( a1 a2 cmd )
    0 _s+crc          ( a1 a2 crc )
    ROT L|M ROT       ( a2 h l crc )
    _s+crc _s+crc     ( a2 crc )
    SWAP L|M ROT      ( h l crc )
    _s+crc _s+crc     ( crc )
    1 OR              ( ensure stop bit )
    (spix) DROP       ( send CRC )
    _wait  ( wait for a valid response... ) ;
```

B254

```
( cmd arg1 arg2 -- r )
( Send a command that expects a R1 response, handling CS. )
: SDCMDR1 [ SDC_DEVID LITN ] (spie) _cmd 0 (spie) ;

( cmd arg1 arg2 -- r arg1 arg2 )
( Send a command that expects a R7 response, handling CS. A R7
  is a R1 followed by 4 bytes. arg1 contains bytes 0:1, arg2
  has 2:3 )
: SDCMDR7
    [ SDC_DEVID LITN ] (spie)
    _cmd                ( r )
    _idle <<8 _idle +  ( r arg1 )
    _idle <<8 _idle +  ( r arg1 arg2 )
    0 (spie) ;
: _rdsdhc ( -- ) $7A ( CMD58 ) 0 0 SDCMDR7 DROP $4000
  AND SDC_SDHC ! DROP ;
```

B255

```
: _err 0 (spie) S" SDerr" STYPE ABORT ;

( Tight definition ahead, pre-comment.

  Initialize a SD card. This should be called at least 1ms
  after the powering up of the card. We begin by waking up the
  SD card. After power up, a SD card has to receive at least
  74 dummy clocks with CS and DI high. We send 80.
  Then send cmd0 for a maximum of 10 times, success is when
  we get $01. Then comes the CMD8. We send it with a $01aa
  argument and expect a $01aa argument back, along with a
  $01 R1 response. After that, we need to repeatedly run
  CMD55+CMD41 ($40000000) until the card goes out of idle
  mode, that is, when it stops sending us $01 response and
  send us $00 instead. Any other response means that
  initialization failed. )
```

B256

```
: SDC$
  10 >R BEGIN _idle DROP NEXT
  0 ( dummy ) 10 >R BEGIN  ( r )
    DROP $40 0 0 SDCMDR1  ( CMD0 )
    1 = DUP IF LEAVE THEN
  NEXT NOT IF _err THEN
  $48 0 $1aa ( CMD8 ) SDCMDR7 ( r arg1 arg2 )
  ( expected 1 0 $1aa )
  $1aa = ROT ( arg1 f r ) 1 = AND SWAP ( f&f arg1 )
  NOT ( 0 expected ) AND ( f&f&f ) NOT IF _err THEN
  BEGIN
    $77 0 0 SDCMDR1  ( CMD55 )
    1 = NOT IF _err THEN
    $69 $4000 0 SDCMDR1  ( CMD41 )
    DUP 1 > IF _err THEN
  NOT UNTIL _rdsdhc ; ( out of idle mode, success! )
```

B257

```
:~ ( dstaddr blkno -- )
  [ SDC_DEVID LITN ] (spie)
  $51 ( CMD17 ) SWAP _badj ( a cmd arg1 arg2 ) _cmd IF _err THEN
  _wait $fe = NOT IF _err THEN
  >A 512 >R 0 BEGIN ( crc1 )
    _idle ( crc1 b ) DUP AC!+ ( crc1 b ) CRC16 NEXT ( crc1 )
    _idle <<8 _idle + ( crc1 crc2 )
    _wait DROP 0 (spie) = NOT IF _err THEN ;
: SDC@ ( blkno blk( -- )
  SWAP << ( 2x ) 2DUP ( a b a b ) ~
  ( a b ) 1+ SWAP 512 + SWAP ~ ;
```

B258

```
:~ ( srcaddr blkno -- )
  [ SDC_DEVID LITN ] (spie)
  $58 ( CMD24 ) SWAP _badj ( a cmd arg1 arg2 ) _cmd IF _err THEN
  _idle DROP $fe (spix) DROP
  >A 512 >R 0 BEGIN ( crc )
    AC@+ ( crc b ) DUP (spix) DROP CRC16 NEXT ( crc )
    DUP >>8 ( crc msb ) (spix) DROP (spix) DROP
    _wait $1f AND 5 = NOT IF _err THEN _ready 0 (spie) ;
: SDC! ( blkno blk( -- )
  SWAP << ( 2x ) 2DUP ( a b a b ) ~
  ( a b ) 1+ SWAP 512 + SWAP ~ ;
```

1.20 Fonts: 260-276

B260

```
Fonts

Fonts are kept in "source" form in the following blocks and
then compiled to binary bitmasks by the following code. In
source form, fonts are a simple sequence of '.' and 'X'. '.'
means empty, 'X' means filled. Glyphs are entered one after the
other, starting at $21 and ending at $7e. To be space
efficient in blocks, we align glyphs horizontally in the blocks
to fit as many character as we can. For example, a 5x7 font
would mean that we would have 12x2 glyphs per block.

261 Font compiler              265 3x5 font
267 5x7 font                   271 7x7 font
```

B261

```
\ Converts "dot-X" fonts to binary "glyph rows". One byte for
\ each row. In a 5x7 font, each glyph thus use 7 bytes.
\ Resulting bytes are aligned to the left of the byte.
\ Therefore, for a 5-bit wide char, "X.X.X" translates to
\ 10101000. Left-aligned bytes are easier to work with when
\ compositing glyphs.
```

B262

```
2 VALUES _w _h
: _g ( given a top-left of dot-X in BLK(, spit H bin lines )
  DUP >A _h >R BEGIN _w >R 0 BEGIN ( a r )
      << AC@+ 'X' = IF 1+ THEN NEXT
    8 _w - LSHIFT C, 64 + DUP >A NEXT DROP ;
: _l ( a u -- a,  spit a line of u glyphs )
  >R DUP BEGIN ( a ) DUP _g _w + NEXT DROP ;
```

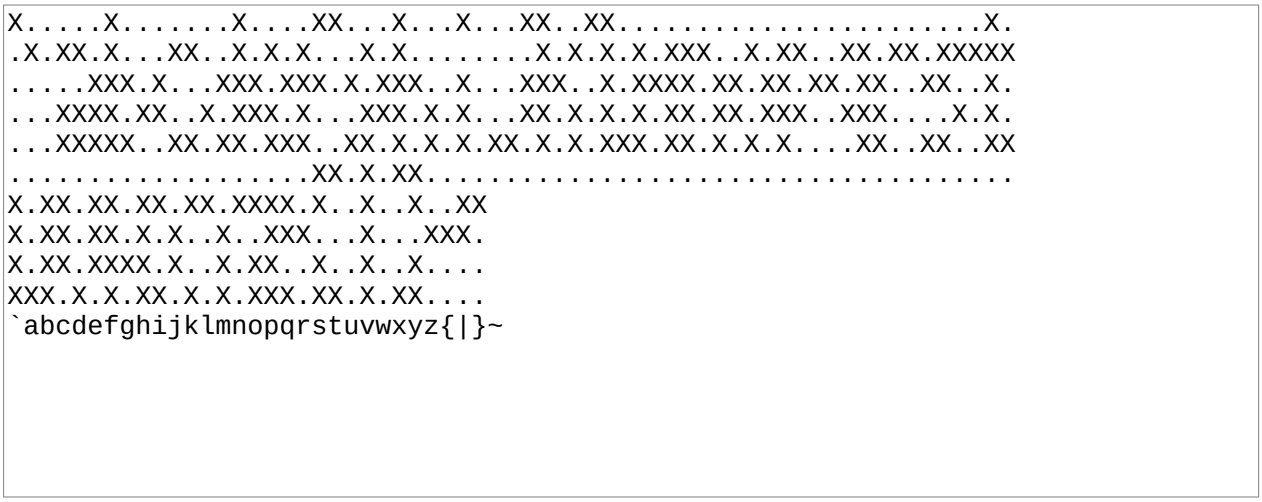
B263

```
: CPFNT3x5 3 TO _w 5 TO _h
  _h ALLOT0 ( space char )
  265 BLK@ BLK( 21 _l 320 + 21 _l 320 + 21 _l DROP ( 63 )
  266 BLK@ BLK( 21 _l 320 + 10 _l DROP ( 94! ) ;
: CPFNT5x7 5 TO _w 7 TO _h
  _h ALLOT0 ( space char )
  3 >R 267 BEGIN ( b )
    DUP BLK@ BLK( 12 _l 448 + 12 _l DROP 1+ NEXT ( 72 )
  ( 270 ) BLK@ BLK( 12 _l 448 + 10 _l DROP ( 94! ) ;
: CPFNT7x7 7 TO _w 7 TO _h
  _h ALLOT0 ( space char )
  5 >R 271 BEGIN ( b )
    DUP BLK@ BLK( 9 _l 448 + 9 _l DROP 1+ NEXT ( 90 )
  ( 276 ) BLK@ BLK( 4 _l DROP ( 94! ) ;
```

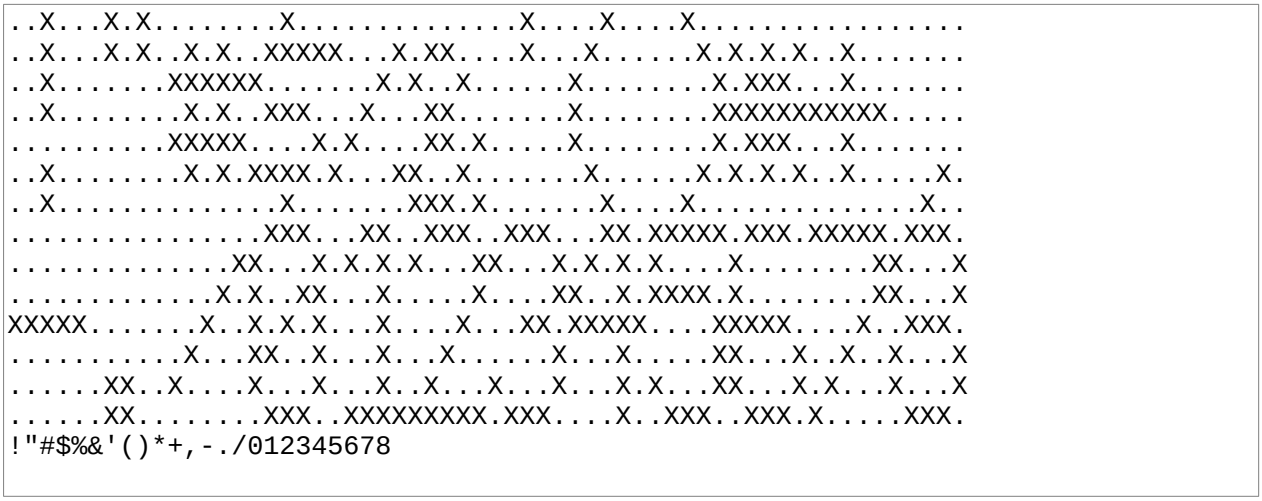
B265

```
.X.X.XX.X.XXX...X..X...XX...X..............X.X..X.XX.XX.X.XXXX
.X.X.XXXXXX...XX.X.X..X..X.XXX.X...........XX.XXX...X..XX.XX..
.X........XX.X..X.....X..X..X.XXX...XXX....X.X.X.X..X.XX.XXXXX.
......XXXXX.X..X.X....X..X.X.X.X..X.......X..X.X.X.X....X..X..X
.X....X.X.X...X.XX.....XX........X......X.X...X.XXXXXXX...XXX.
.XXXXXXXXXX.........X...X..XX..X..X.XX..XXXX.XXXXXX.XXX.XXXXXXX
X....XX.XX.X.X..X..X.XXX.X...XXXXX.XX.XX..X.XX..X..X..X.X.X...X
XXX.X.XXXXXX......X.......X.X.XXXXXXXX.X..X.XXX.XX.X.XXXX.X...X
X.XX..X.X..X.X..X..X.XXX.X....X..X.XX.XX..X.XX..X..X.XX.X.X...X
XXXX..XXXXX....X....X...X...X..XXX.XXX..XXXX.XXXX...XXX.XXXXXX.
X.XX..X.XXX.XXXXX.XXXXX..XXXXXX.XX.XX.XX.XX.XXXXXXXX..XXX.X....
XX.X..XXXX.XX.XX.XX.XX.XX...X.X.XX.XX.XX.XX.X..XX..X....XX.X...
X..X..XXXX.XX.XXX.X.XXX..X..X.X.XX.XXXX.X..X..X.X...X...X......
XX.X..X.XX.XX.XX..XXXX.X..X.X.X.XX.XXXXX.X.X.X..X....X..X......
X.XXXXX.XX.XXXXX...XXX.XXX..X.XXX.X.X.XX.X.X.XXXXXX..XXXX...XXX
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
```

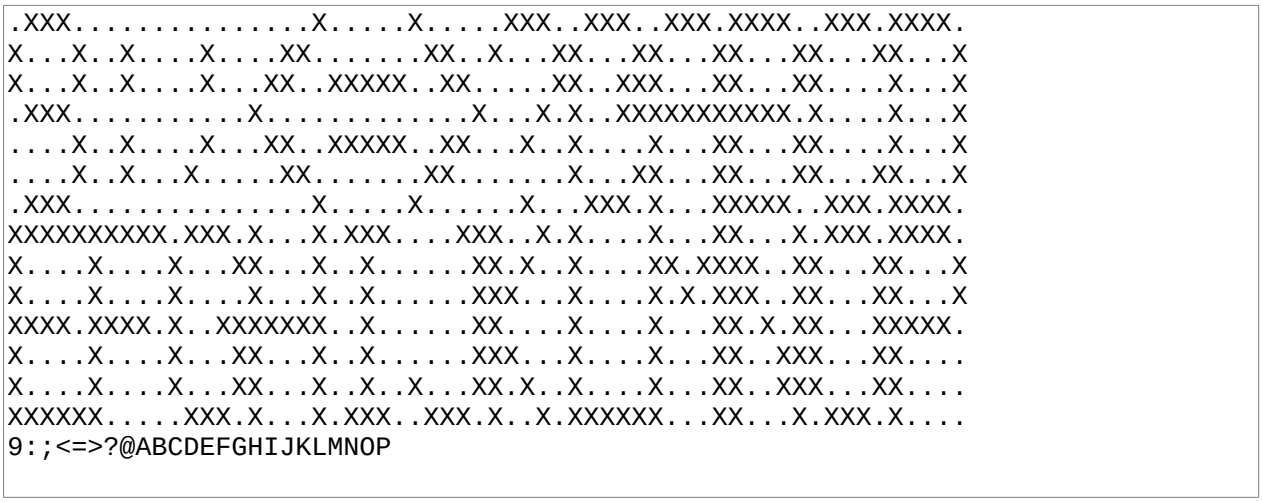
B266



B267



B268



B269

```
.XXX.XXXX..XXX.XXXXXX...XX...XX...XX...XX...XXXXXXXXX.......
X...XX...XX...X..X..X...XX...XX...XX...XX...XX...XX....X....
X...XX...XX......X..X...XX...XX...X.X.X..X.X....X.X.....X...
X...XXXXX..XXX...X..X...XX...XX...X..X....X....X..X......X..
X.X.XX.X......X..X..X...XX...XX.X.X.X.X...X...X...X.......X.
X..XXX..X.X...X..X..X...X.X.X.X.X.XX...X..X..X...XX........X
.XXXXX...X.XXX...X...XXX...X...X.X.X...X..X..XXXXXXXX.......
..XXX..X.........X.....................................
....X.X.X.........X.....................................
....XX...X...........XXX.X.....XXX.....X.XXX..XX....XXXX....
....X..................XX....X...X....XX...XX..X..X..XX....
....X..............XXXXXXX..X......XXXXXXXXX......XXXXXX..
....X..............X...XX..X.X...X.X..XX....XXX......XX..X.
..XXX.....XXXXX......XXXXXXX...XXX...XXX.XXXXX......XX.X..X.
QRSTUVWXYZ[\]^_`abcdefgh
```

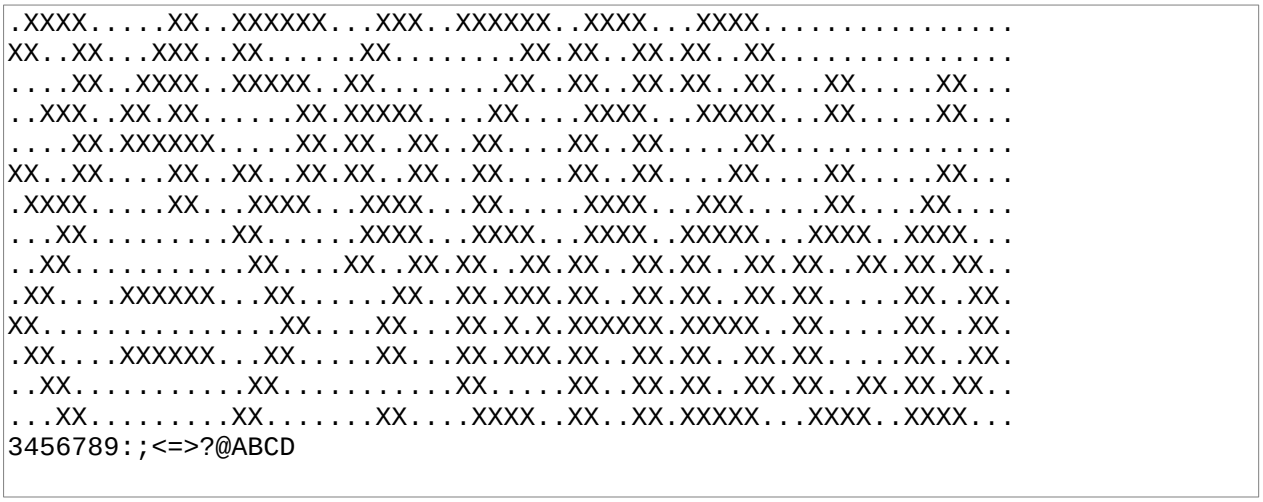
B270

```
..........................................................
..........................................................
..X......XX..X..XX...X.X.XXX...XXX.XXX....XXXX.XX..XXX..X...
..........X.X....X..X.X.XX..X.X...XX..X..X..XXX...X....XXX..
..X......XXX.....X..X...XX...XX...XXXX....XXXX.....XXX..X...
..X...X..XX.X....X..X...XX...XX...XX.......XX........X.X...
..X....XX.X..X...XX.X...XX...X.XXX.X.......XX.....XXX...XX.
.................................XX...X...XX.......
.................................X.....X.....X......
X...XX...XX...XX...XX...XXXXXX.X.....X.....X..X.X.
X...XX...XX...X.X.X..X.X....X.X......X......XX.X..
X...XX...XX...X..X....X....X...X.....X.....X......
X...X.X.X.X.X.X.X..X....X....X.....X.....X......
.XXX...X...X.X.X...XX....XXXXX..XX...X...XX.......
ijklmnopqrstuvwxyz{|}~
```

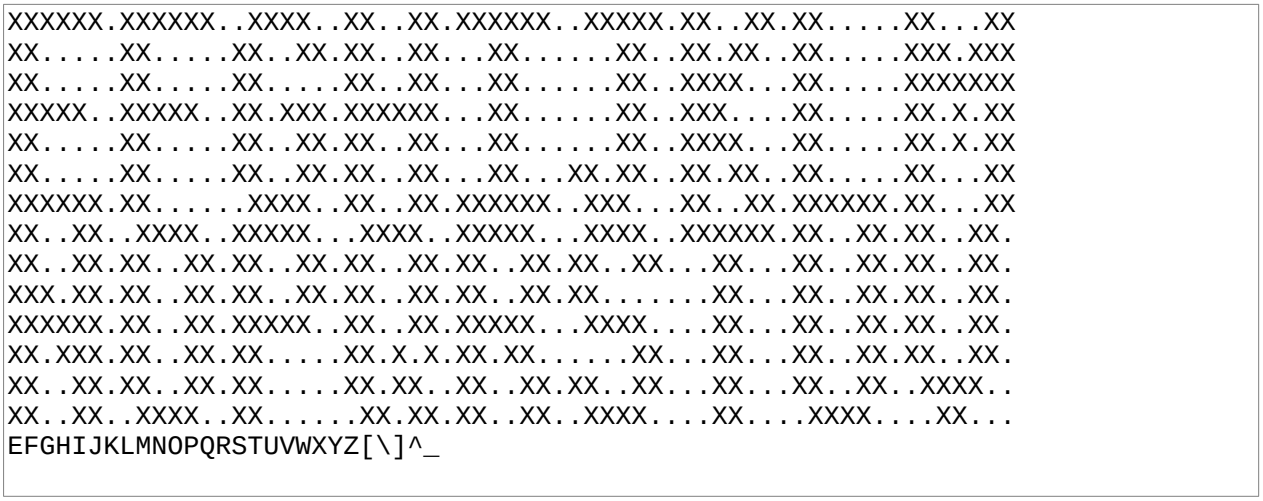
B271

```
..XX....XX.XX..XX.XX....XX..XX......XXX......XX.....XX...XX....
..XX....XX.XX..XX.XX..XXXXXXXX..XX.XX.XX....XX.....XX.....XX...
..XX....XX.XX.XXXXXXXXX.X......XX..XX.XX...XX.....XX.......XX..
..XX...........XX.XX..XXXXX...XX....XXX...........XX.......XX..
..XX..........XXXXXXX...X.XX.XX....XX.XX.X........XX.......XX..
................XX.XX.XXXXXX.XX..XX.XX..XX..........XX.....XX...
..XX...........XX.XX...XX.......XX..XXX.XX..........XX...XX....
.............................................XXXX....XX....XXXX..
..XX.....XX.............................XX.XX..XX..XXX...XX..XX.
XXXXXX...XX.............................XX..XX.XXX...XX.......XX.
.XXXX..XXXXXX........XXXXXX...........XX...XXXXXX...XX......XX..
XXXXXX...XX.........................XX....XXX.XX...XX.....XX...
..XX.....XX.....XX.............XX...XX.....XX..XX...XX....XX....
...............XX.............XX...........XXXX..XXXXXX.XXXXXX.
!"#$%&'()*+,-./012
```

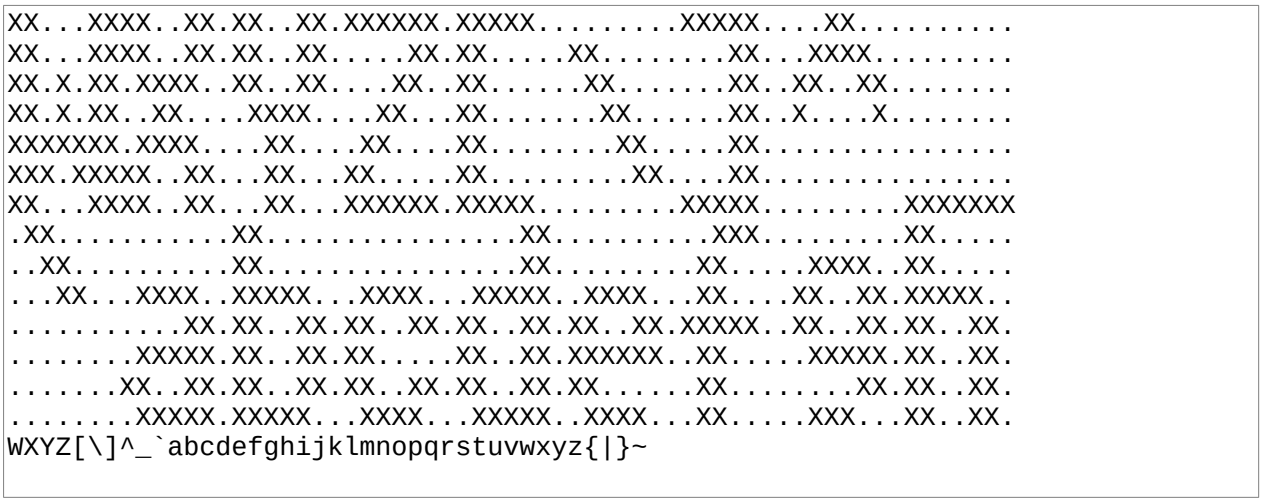
B272



B273



B274



B275

```
..XX.....XX...XX......XXX.....................................
..............XX.......XX.....................................
.XXX....XXX...XX..XX...XX....XX.XX.XXXXX...XXXX..XXXXX...XXXXX.
..XX.....XX...XX.XX....XX...XXXXXXXXX..XX.XX..XX.XX..XX.XX..XX.
..XX.....XX...XXXX.....XX...XX.X.XXXX..XX.XX..XX.XX..XX.XX..XX.
..XX.....XX...XX.XX....XX...XX.X.XXXX..XX.XX..XX.XXXXX...XXXXX.
.XXXX..XX.....XX..XX..XXXX..XX...XXXX..XX..XXXX..XX.........XX.
..............XX.............................................
..............XX.............................................
XX.XX...XXXXX.XXXXX..XX..XX.XX..XX.XX...XXXX..XX.XX..XX.XXXXXX.
XXX.XX.XX......XX....XX..XX.XX..XX.XX.X.XX.XXXX..XX..XX....XX..
XX......XXXX...XX....XX..XX.XX..XX.XX.X.XX..XX...XX..XX...XX...
XX.........XX..XX....XX..XX..XXXX..XXXXXXX.XXXX...XXXXX..XX....
XX.....XXXXX....XXX...XXXXX...XX....XX.XX.XX..XX.....XX.XXXXXX.
ijklmnopqrstuvwxyz{|}~
```

B276

```
...XX....XX...XX......XX...X
..XX.....XX....XX....XX.X.XX
..XX.....XX....XX....X...XX.
XXX......XX.....XXX.........
..XX.....XX....XX...........
..XX.....XX....XX...........
...XX....XX...XX............
{|}~
```

1.21 Automated tests: 290-296

B290

```
\ Automated tests. "1 LOAD 290 296 LOADR" to run.
\ "#" means "assert". We ABORT on failure.
: fail SPC> ABORT" failed" ;
: # IF SPC> ." pass" NL> ELSE fail THEN ;
: #eq 2DUP SWAP . SPC> '=' EMIT SPC> . '?' EMIT = # ;
```

B291

```
\ Arithmetics
48 13 + 61 #eq
48 13 - 35 #eq
48 13 * 624 #eq
48 13 / 3 #eq
48 13 MOD 9 #eq
5 3 LSHIFT 40 #eq
155 5 RSHIFT 4 #eq
```

B292

```
\ Comparisons
$22 $8065 < #
-1 0 > #
-1 0< #
```

B293

```
\ Memory
42 C, 43 C, 44 C,
HERE 3 - HERE 3 MOVE
HERE C@ 42 #eq HERE 1+ C@ 43 #eq HERE 2 + C@ 44 #eq
HERE HERE 1+ 3 MOVE ( demonstrate MOVE's problem )
HERE 1+ C@ 42 #eq HERE 2 + C@ 42 #eq HERE 3 + C@ 42 #eq
HERE 3 - HERE 3 MOVE
HERE HERE 1+ 3 MOVE- ( see? better )
HERE 1+ C@ 42 #eq HERE 2 + C@ 43 #eq HERE 3 + C@ 44 #eq

HERE ( ref )
HERE 3 - 3 MOVE,
( ref ) HERE 3 - #eq
HERE 3 - C@ 42 #eq HERE 2 - C@ 43 #eq HERE 1- C@ 44 #eq
```

B294

```
\ Parse
'b' $62 #eq
```

B295

```
\ Stack
42 43 44 ROT
42 #eq 44 #eq 43 #eq
42 43 44 ROT>
43 #eq 42 #eq 44 #eq
```

B296

```
\ CRC
$0000 $00 CRC16 $0000 #eq
$0000 $01 CRC16 $1021 #eq
$5678 $34 CRC16 $34e4 #eq
```

2 Z80

2.1 Architecture index: 300

B300

```
Z80 MASTER INDEX

301 Z80 boot code              310 Z80 HAL
320 Z80 assembler
330 AT28 EEPROM                332 SPI relay
335 TMS9918
340 MC6850 driver              345 Zilog SIO driver
350 Sega Master System VDP     355 SMS PAD
360 SMS KBD                    367 SMS SPI relay
368 SMS Ports
370 TI-84+ LCD                 375 TI-84+ Keyboard
380 TI-84+ Boot code & macros
390 TRS-80 4P drivers          405 Dan SBC drivers
```

2.2 Z80 boot code: 301-314

B301

```
\ Z80 port's Macros and constants. See doc/code/z80.txt
: Z80A 5 LOAD ( wordtbl ) 320 329 LOADR 7 LOAD ( Flow words ) ;
: Z80C 302 314 LOADR ;
: TRS804PM 390 LOAD ;
: TI84M 380 LOAD ; : TI84BOOT 381 382 LOADR ;
\ see comment at TICKS' definition
\ 7.373MHz target: 737t. outer: 37t inner: 16t
\ tickfactor = (737 - 37) / 16
44 VALUE tickfactor
0 VALUE L4 \ we need a 4th temp label in core routines
```

B302

```
\ Z80 port, core routines
FJR jr, TO L1 $10 OALLOT LSET lblxt ( RST 10 )
  IX inc, IX inc, 0 ix+) E ld, 1 ix+) D ld,
  HL pop, ldDE(HL), HL inc, DE HL ex, jp(HL), \ 17 bytes
$28 OALLOT LSET lblcell ( RST 28 )
  HL pop, BC push, HL>BC, FJR jr, TO L2 ( next ) $30 OALLOT
LSET lblval ( RST 30 ) A SYSVARS $18 + m) ld, A A or,
  FJR CZ jrc, TO L3 ( read ) FJR jr, TO L4 ( write ) \ 8 bytes
0 jp, ( RST 38 ) $66 OALLOT retn,
L1 FMARK
 di, SP PS_ADDR i) ld, IX RS_ADDR i) ld, 0 jp, PC 2 - TO lblboot
L3 FMARK ( val read ) HL pop, BC push, ldBC(HL), \ to lblnext
LSET lblnext L2 FMARK
DE HL ex, LSET L1 ( EXIT ) ldDE(HL), HL inc, DE HL ex, jp(HL),
L4 FMARK ( val write ) clrA, SYSVARS $18 + m) A ld, HL pop,
  (HL) C ld, HL inc, (HL) B ld, BC pop, lblnext BR jr,
```

B303

```
\ Z80 port, lbldoes EXIT QUIT ABORT BYE RCNT SCNT
LSET lbldoes HL pop, BC push, HL>BC, BC inc, BC inc, ldHL(HL),
  jp(HL),
CODE EXIT \ put new IP in HL instead of DE for speed
  L 0 ix+) ld, H 1 ix+) ld, IX dec, IX dec, L1 jp,
CODE QUIT LSET L1 \ used in ABORT
  IX RS_ADDR i) ld, 0 jp, PC 2 - TO lblmain
CODE ABORT SP PS_ADDR i) ld, L1 BR jr,
CODE BYE halt,
CODE RCNT BC push, IX push, HL pop, BC RS_ADDR i) ld,
  BC subHL, HL>BC, ;CODE
CODE SCNT HL 0 i) ld, HL SP add, BC push, HL>BC,
  HL PS_ADDR i) ld, BC subHL, HL>BC, ;CODE
```

B304

```
\ Z80 port, TICKS
\ The word below is designed to wait the proper 100us per tick
\ at 500kHz when tickfactor is 1. If the CPU runs faster,
\ tickfactor has to be adjusted accordingly. "t" in comments
\ below means "T-cycle", which at 500kHz is worth 2us.
CODE TICKS
  \ we pre-dec to compensate for initialization
  BEGIN,
    BC dec, ( 6t )
    IFZ, ( 12t ) BC pop, ;CODE THEN,
    A tickfactor i) ld, ( 7t )
    BEGIN, A dec, ( 4t ) BR CNZ jrc, ( 12t )
  BR jr, ( 12t ) ( outer: 37t inner: 16t )
```

B305

```
\ Z80 port, PC! PC@ []= [C]? (im1)
CODE PC! HL pop, (C) L out, BC pop, ;CODE
CODE PC@ C (C) in, B 0 i) ld, ;CODE
CODE []= BC push, exx, ( protect DE ) BC pop, DE pop, HL pop,
  LSET L1 ( loop )
    A (DE) ld, DE inc, cpi,
    IFNZ, exx, BC 0 i) ld, ;CODE THEN,
    L1 CPE jpc, ( BC not zero? loop )
  exx, BC 1 i) ld, ;CODE
CODE [C]? BCZ, IFZ, BC dec, HL pop, HL pop, ;CODE THEN,
  BC push, exx, BC pop, HL pop, DE pop, A E ld, D H ld,
  E L ld, \ HL=a DE=a BC=u A=c
  cpir, IFZ, DE subHL, HL dec, ELSE, HL -1 i) ld, THEN,
  HL push, exx, BC pop, ;CODE
CODE (im1) im1, ei, ;CODE
```

B306

```
\ Z80 port, /MOD *
CODE * HL pop, DE push, DE HL ex, ( DE * BC -> HL )
  HL 0 i) ld, A $10 i) ld, BEGIN,
    HL HL add, E rl, D rl,
    IFC, HL BC add, THEN,
    A dec, BR CNZ jrc,
  HL>BC, DE pop, ;CODE
\ Divides AC by DE. quotient in AC remainder in HL
CODE /MOD BC>HL, BC pop, DE push, DE HL ex,
  A B ld, B 16 i) ld, HL 0 i) ld, BEGIN,
    scf, C rl, rla, HL HL adc, HL DE sbc,
    IFC, HL DE add, C dec, THEN,
  BR djnz,
  DE pop, HL push, B A ld, ;CODE
```

B307

```
\ Z80 port, FIND
CODE FIND ( sa sl -- w? f ) HL pop,
  HL BC add, \ HL points to after last char of s
  'N m) HL ld, HL SYSVARS $02 ( CURRENT ) + m) ld, BEGIN,
    HL dec, A (HL) ld, A $7f i) and, ( imm ) A C cp, IFZ,
      HL push, DE push, BC push, DE 'N m) ld,
      HL dec, HL dec, HL dec, \ Skip prev field
      LSET L1 ( loop )
        DE dec, A (DE) ld, cpd, IFZ, TO L2 ( break! )
      L1 CPE jpc, ( BC not zero? loop ) L2 FMARK
      BC pop, DE pop, HL pop, THEN,
    IFZ, ( match ) HL inc, HL push, BC 1 i) ld, ;CODE THEN,
    \ no match, go to prev and continue
    HL dec, A (HL) ld, HL dec, L (HL) ld, H A ld,
    A L or, IFZ, ( end of dict ) BC 0 i) ld, ;CODE THEN,
  BR jr,
```

B308

```
\ Z80 port, (b) (n) (br) (?br) (next)
CODE (b) ( -- c ) BC push, A (DE) ld, A>BC, DE inc, ;CODE
CODE (n) ( -- n ) BC push,
  DE HL ex, ldBC(HL), HL inc, DE HL ex, ;CODE
CODE (br) LSET L1 ( used in ?br and next )
  A (DE) ld, ( sign extend A into HL )
  L A ld, A A add, ( sign in carry ) A A sbc, ( FF if neg )
  H A ld, HL DE add, ( HL --> new IP ) DE HL ex, ;CODE
CODE (?br) BCZ, BC pop, L1 BR CZ jrc, DE inc, ;CODE
CODE (next)
  0 ix+) dec, IFNZ,
    A $ff i) ld, A 0 ix+) cp, IFZ, 1 ix+) dec, THEN,
    L1 BR jr, THEN,
  A A xor, A 1 ix+) cp, L1 BR CNZ jrc,
  IX dec, IX dec, DE inc, ;CODE
```

B309

```
\ Z80 port, >R I C@ @ C! ! 1+ 1- + -
CODE >R IX inc, IX inc, 0 ix+) C ld, 1 ix+) B ld, BC pop, ;CODE
CODE R@ BC push, C 0 ix+) ld, B 1 ix+) ld, ;CODE
CODE R~ IX dec, IX dec, ;CODE
CODE R> BC push, C 0 ix+) ld, B 1 ix+) ld,
  IX dec, IX dec, ;CODE
CODE C@ A (BC) ld, A>BC, ;CODE
CODE @ BC>HL, ldBC(HL), ;CODE
CODE C! BC>HL, BC pop, (HL) C ld, BC pop, ;CODE
CODE ! BC>HL, BC pop,
  (HL) C ld, HL inc, (HL) B ld, BC pop, ;CODE
CODE 1+ BC inc, ;CODE
CODE 1- BC dec, ;CODE
CODE + HL pop, HL BC add, HL>BC, ;CODE
CODE - HL pop, BC subHL, HL>BC, ;CODE
```

B310

```
\ Z80 port, AND OR XOR >> << >>8 <<8
CODE AND HL pop,
  A C ld, A L and, C A ld, A B ld, A H and, B A ld, ;CODE
CODE OR HL pop,
  A C ld, A L or, C A ld, A B ld, A H or, B A ld, ;CODE
CODE XOR HL pop,
  A C ld, A L xor, C A ld, A B ld, A H xor, B A ld, ;CODE
CODE NOT BCZ, BC 0 i) ld, IFZ, C inc, THEN, ;CODE
CODE >> B srl, C rr, ;CODE
CODE << C sla, B rl, ;CODE
CODE >>8 C B ld, B 0 i) ld, ;CODE
CODE <<8 B C ld, C 0 i) ld, ;CODE
```

B311

```
\ Z80 port, ROT ROT> DUP DROP SWAP OVER EXECUTE
CODE ROT ( a b c -- b c a ) ( BC=c )
  HL pop, ( b ) (SP) HL ex, ( a<>b ) BC push, ( c ) HL>BC, ;CODE
CODE ROT> ( a b c -- c a b ) ( BC=c )
  BC>HL, BC pop, ( b ) (SP) HL ex, ( a<>c ) HL push, ;CODE
CODE DUP ( a -- a a ) LSET L1 BC push, ;CODE
CODE ?DUP BCZ, L1 BR CNZ jrc, ;CODE
CODE DROP ( a -- ) BC pop, ;CODE
CODE SWAP ( a b -- b a ) HL pop, BC push, HL>BC, ;CODE
CODE OVER ( a b -- a b a )
  HL pop, HL push, BC push, HL>BC, ;CODE
CODE EXECUTE BC>HL, BC pop, jp(HL),
```

B312

```
\ Z80 port, JMPi! CALLi!
CODE JMPi! ( pc a -- len ) BC>HL, BC pop,
  A $c3 i) ld, LSET L1 (HL) A ld, HL inc,
  (HL) C ld, HL inc, (HL) B ld, BC 3 i) ld, ;CODE
CODE CALLi! ( pc a -- len ) BC>HL, BC pop,
  A B ld, A A or, IFZ, A C ld, A $c7 i) and, IFZ, \ RST
   A C ld, A $c7 i) or, (HL) A ld, BC 1 i) ld, ;CODE THEN, THEN,
  ( not RST ) A $cd i) ld, L1 BR jr,
```

B313

```
\ Z80 port speedups
CODE TUCK ( a b -- b a b ) HL pop, BC push, HL push, ;CODE
CODE NIP ( a b -- b ) HL pop, ;CODE
CODE +! ( n a -- ) BC>HL, ldBC(HL), HL dec, (SP) HL ex,
  HL BC add, HL>BC, HL pop, (HL) C ld, HL inc, (HL) B ld,
  BC pop, ;CODE
CODE A> BC push, IY push, BC pop, ;CODE
CODE >A BC push, IY pop, BC pop, ;CODE
CODE A>R IY push, HL pop,
  IX inc, IX inc, 0 ix+) L ld, 1 ix+) H ld, ;CODE
CODE R>A L 0 ix+) ld, H 1 ix+) ld, IX dec, IX dec,
  HL push, IY pop, ;CODE
CODE A+ IY inc, ;CODE
CODE A- IY dec, ;CODE
CODE AC@ BC push, C 0 iy+) ld, B 0 i) ld, ;CODE
CODE AC! 0 iy+) C ld, BC pop, ;CODE
```

B314

```
\ Z80 port speedups
CODE MOVE ( src dst u -- ) HL pop, DE HL ex, (SP) HL ex,
  BCZ, IFNZ, ldir, THEN, DE pop, BC pop, ;CODE
CODE = HL pop, BC subHL, BC 0 i) ld, IFZ, BC inc, THEN, ;CODE
CODE < HL pop, BC subHL, BC 0 i) ld, IFC, BC inc, THEN, ;CODE
CODE CRC16 ( c n -- c ) BC push, exx, ( protect DE )
  HL pop, ( n ) DE pop, ( c ) A L ld, A D xor, D A ld,
  B 8 i) ld, BEGIN,
    E sla, D rl, IFC, ( msb is set, apply polynomial )
      A D ld, A $10 i) xor, D A ld,
      A E ld, A $21 i) xor, E A ld, THEN,
  BR djnz,
  DE push, exx, ( unprotect DE ) BC pop, ;CODE
```

2.3 Z80 assembler: 320-329

B320

```
\ Z80 Assembler. Operands. See doc/asm. Requires B5
: >>3 >> >> >> ; : <<3 << << << ; : <<4 <<3 << ;
: opreg 7 AND ; : optype >>3 3 AND ;
CREATE nbank $10 ALLOT
0 VALUE nbank>
: nbank@ ( op -- n ) opreg << nbank + @ ;
: nbank! ( n -- idx )
  nbank> TUCK << nbank + ! DUP 1+ opreg TO nbank> ;
28 CONSTS
  $00 B  $01 C  $02 D  $03 E  $04 H  $05 L  $06 (HL)  $07 A
  $08 BC $09 DE $0a HL $0b AF $0b SP
  $20 (BC) $21 (DE) $22 (SP) $23 AF' $24 I $25 R $26 (C)
  $00 CNZ $01 CZ $02 CNC $03 CC $04 CPO $05 CPE $06 CP $07 CM
: i) nbank! $10 OR ;          : m) nbank! $18 OR ;
: ix, $dd C, ; : iy, $fd C, ; : IX ix, HL ; : IY iy, HL ;
: _ <<8 (HL) OR $40 OR ; : ix+) ix, _ ; : iy+) iy, _ ;
```

B321

```
\ Z80 Assembler. Checks, asserts, util
: err ABORT" argument error" ;
: # ( f -- ) NOT IF err THEN ;
: HL# HL = # ; : A# A = # ;
: 8b? optype 0 = ; : 16b? optype 1 = ; : ixy+? $40 AND ;
: special? $20 AND ;
: 8b# 8b? # ;
: opexec ( op tbl -- ) SWAP optype WEXEC ;
: opcode, ( opcode -- ) DUP >>8 ?DUP IF C, THEN C, ;
: ?ixy+, ( op -- ) DUP ixy+? IF >>8 C, ELSE DROP THEN ;
```

B322

```
\ Z80 Assembler. sub, and, or, xor, cp,
: _reg8, OVER opreg OR opcode, ?ixy+, ;
: _imm, $46 OR opcode, nbank@ C, ;
4 WORDTBL _ ( op code -- )
  'W _reg8, 'W err 'W _imm, 'W err
: 8bari, ( A op code -- ) ROT A# OVER _ opexec ;
: op DOER , DOES> ( A op 'code -- ) @ 8bari, ;
$a0 op and,               $b8 op cp,
$b0 op or,                $90 op sub,
$a8 op xor,
```

B323

```
\ Z80 Assembler. rl, rr, rlc, rrc, sla, srl, bit, set, res,
4 WORDTBL _ ( op code -- )
'W _reg8, 'W err 'W err 'W err
: op DOER , DOES> ( op 'code ) @ OVER _ opexec ;
$cb10 op rl,   $cb18 op rr,   $cb00 op rlc,  $cb08 op rrc,
$cb20 op sla,  $cb38 op srl,
: op DOER , DOES> ( op b 'code ) @ SWAP <<3 OR OVER _ opexec ;
$cbc0 op set,      $cb80 op res,     $cb40 op bit,
```

B324

```
\ Z80 Assembler. inc, dec, add, adc, sbc,
: _reg8<<, @ OVER opreg <<3 OR C, ?ixy+, ;
: _reg16<<, 2 + @ SWAP opreg <<4 OR opcode, ;
: _ixy+<<, C, (HL) SWAP _reg8<<, nbank@ C, ;
4 WORDTBL _ ( op 'codes -- )
  'W _reg8<<, 'W _reg16<<, 'W err 'W err
: op DOER ( 8b ) , ( 16b ) , DOES> ( op 'codes ) OVER _ opexec ;
$03 04 op inc,      $0b 05 op dec,
: op DOER ( 8b ) , ( 16b ) , DOES> ( dst src 'codes -- )
  OVER 16b? IF ROT HL# _reg16<<, ELSE @ 8bari, THEN ;
$09 $80 op add,   $ed4a $88 op adc,   $ed42 $98 op sbc,
```

B325

```
\ Z80 Assembler. push, pop, in, out, rst,
4 WORDTBL _ ( op 'codes -- )
'W err 'W _reg16<<, 'W err 'W err
: op DOER 0 , , DOES> ( op 'code -- ) OVER _ opexec ;
$c5 op push,                $c1 op pop,
: _A, ( n in? ) <<3 $d3 OR C, nbank@ C, ;
: _C, ( reg in? ) NOT $ed40 OR SWAP <<3 OR opcode, ;
: _inout, ( op n-or-C in? )
  OVER (C) = IF NIP _C, ELSE ROT DROP _A, THEN ;
: in, 1 _inout, ;  : out, SWAP 0 _inout, ;
: rst, ( n ) $c7 OR C, ;
CREATE _ 9 nC, AF DE (SP) AF' HL HL $08 $eb $e3
: ex, ( op1 op2 -- ) SWAP _ 3 [C]? DUP 0>= #
  3 + _ + DUP C@ ROT = # 3 + C@ C, ;
```

B326

```
\ Z80 Assembler. Inherent ops
: op DOER , DOES> @ opcode, ;
$f3 op di,     $fb op ei,     $d9 op exx,    $76 op halt,
$00 op nop,    $37 op scf,    $3f op ccf,    $c9 op ret,
$17 op rla,    $07 op rlca,   $1f op rra,    $0f op rrca,
$eda1 op cpi,  $edb1 op cpir, $eda9 op cpd,  $edb9 op cpdr,
$ed46 op im0,  $ed56 op im1,  $ed5e op im2,  $eda0 op ldi,
$edb0 op ldir, $eda8 op ldd,  $edb8 op lddr, $ed44 op neg,
$ed4d op reti, $ed45 op retn, $eda2 op ini,  $edaa op ind,
$eda3 op outi,
```

B327

```
\ Z80 Assembler. ld,
CREATE _s1 $0a , $1a , 0 , 0 , $ed57 , $ed5f , 0 , 0 ,
CREATE _s2 $02 , $12 , 0 , 0 , $ed47 , $ed4f , 0 , 0 ,
: _r8 OVER opreg <<3 OVER opreg OR $40 OR C, OR ?ixy+, ;
: _sp DUP special? IF NIP _s1 ELSE DROP _s2 THEN
      SWAP opreg << + @ opcode, ;
: _n ( dst src -- i mask 16b? )
  nbank@ SWAP DUP 16b? IF opreg <<4 1 ELSE opreg <<3 0 THEN ;
4 WORDTBL _ ( dst src -- ) \ sel on src. dst should be a reg
:W 2DUP OR special? IF _sp ELSE _r8 THEN ;
:W HL# SP = # $f9 C, ;
:W _n IF $01 OR C, L, ELSE $06 OR C, C, THEN ;
:W 2DUP < <<3 ROT> <> _n IF
    DUP $20 = IF $02 ELSE $ed43 THEN OR ROT OR
    ELSE $38 = # SWAP $32 OR THEN opcode, L, ;
: ld, ( dst src -- ) OVER optype OVER optype MAX _ SWAP WEXEC ;
```

B328

```
\ Z80 Assembler. Macros
: clrA, A A xor, ;
: subHL, A A or, HL SWAP sbc, ;
: pushA, B 0 i) ld, C A ld, BC push, ;
: HLZ, A H ld, A L or, ;
: DEZ, A D ld, A E or, ;
: BCZ, A B ld, A C or, ;
: ldDE(HL), E (HL) ld, HL inc, D (HL) ld, ;
: ldBC(HL), C (HL) ld, HL inc, B (HL) ld, ;
: ldHL(HL), A (HL) ld, HL inc, H (HL) ld, L A ld, ;
: outHL, A H ld, DUP A out, A L ld, A out, ;
: outDE, A D ld, DUP A out, A E ld, A out, ;
: HL>BC, B H ld, C L ld, ;
: BC>HL, H B ld, L C ld, ;
: A>BC, C A ld, B 0 i) ld, ;
: A>HL, L A ld, H 0 i) ld, ;
```

B329

```
\ Z80 Assembler. Jumps, calls and HAL
: cond ( cond opcode -- opcode ) SWAP <<3 OR ;
: br8, ( n opcode -- ) C, C, ;
: jr, $18 br8, ; : djnz, $10 br8, ; : jrc, $20 cond br8, ;
: br16, ( n opcode -- ) C, L, ;
: jp, $c3 br16, ;            : call, $cd br16, ;
: jpc, $c2 cond br16, ;      : callc, $c4 cond br16, ;
: retc, $c0 cond C, ;        : jp(HL), $e9 C, ;
: jp(IX), IX DROP jp(HL), ; : jp(IY), IY DROP jp(HL), ;
ALIAS jp, JMPi, ALIAS jr, JRi,
: JMP(i), m) HL SWAP ld, jp(HL), ;
: CALLi, DUP $38 AND OVER = IF rst, ELSE call, THEN ;
: JRZi, CZ jrc, ; : JRNZi, CNZ jrc, ;
: JRCi, CC jrc, ; : JRNCi, CNC jrc, ;
: i>, BC push, i) BC SWAP ld, ;
: (i)>, BC push, m) BC SWAP ld, ;
```

2.4 AT28 EEPROM: 330

B330

```
CODE AT28C! ( c a -- )
  BC>HL, BC pop,
  (HL) C ld, A C ld, ( orig ) B C ld, ( save )
  C (HL) ld, ( poll ) BEGIN,
    A (HL) ld, ( poll ) A C cp, ( same as old? )
    C A ld, ( save old poll, Z preserved )
  BR CNZ jrc,
\ equal to written? SUB instead of CP to ensure IOERR is NZ
  A B sub, IFNZ, SYSVARS ( IOERR ) m) A ld, THEN, BC pop, ;CODE
: AT28! ( n a -- ) 2DUP AT28C! 1+ SWAP >>8 SWAP AT28C! ;
```

2.5 SPI relay: 332

B332

```
( SPI relay driver. See doc/hw/z80/spi.txt )
CODE (spix) ( n -- n )
  A C ld,
  SPI_DATA i) A out,
  \ wait until xchg is done
  BEGIN, A SPI_CTL i) in, A 1 i) and, BR CNZ jrc,
  A SPI_DATA i) in,
  C A ld, ;CODE
CODE (spie) ( n -- ) A C ld, SPI_CTL i) A out, BC pop, ;CODE
```

2.6 TMS9918: 335-337

B335

```
( Z80 driver for TMS9918. Implements grid protocol. Requires
TMS_CTLPORT, TMS_DATAPORT and ~FNT from the Font compiler at
B520. Patterns are at addr $0000, Names are at $3800.
Load range B315-317 )
CODE _ctl ( a -- sends LSB then MSB )
  A C ld, TMS_CTLPORT i) A out, A B ld, TMS_CTLPORT i) A out,
  BC pop, ;CODE
CODE _data
  A C ld, TMS_DATAPORT i) A out, BC pop, ;CODE
```

B336

```
: _zero ( x -- send 0 _data x times )
  ( x ) >R BEGIN 0 _data NEXT ;
( Each row in ~FNT is a row of the glyph and there is 7 of
them.  We insert a blank one at the end of those 7. )
: _sfont ( a -- a+7, Send font to TMS )
  7 >R BEGIN C@+ _data NEXT ( blank row ) 0 _data ;
: _sfont^ ( a -- a+7, Send inverted font to TMS )
  7 >R BEGIN C@+ $ff XOR _data NEXT ( blank row ) $ff _data ;
: CELL! ( c pos )
    $7800 OR _ctl ( tilenum )
    SPC - ( glyph ) $5f MOD _data ;
```

B337

```
: CURSOR! ( new old -- )
    DUP $3800 OR _ctl [ TMS_DATAPORT LITN ] PC@
    $7f AND ( new old glyph ) SWAP $7800 OR _ctl _data
    DUP $3800 OR _ctl [ TMS_DATAPORT LITN ] PC@
    $80 OR ( new glyph ) SWAP $7800 OR _ctl _data ;
: COLS 40 ; : LINES 24 ;
: TMS$
    $8100 _ctl ( blank screen )
    $7800 _ctl COLS LINES * _zero
    $4000 _ctl $5f >R ~FNT BEGIN _sfont NEXT DROP
    $4400 _ctl $5f >R ~FNT BEGIN _sfont^ NEXT DROP
    $820e _ctl ( name table $3800 )
    $8400 _ctl ( pattern table $0000 )
    $87f0 _ctl ( colors 0 and 1 )
    $8000 _ctl $81d0 _ctl ( text mode, display on ) ;
```

2.7 MC6850 driver: 340-342

B340

```
( MC6850 Driver. Load range B320-B322. Requires:
  6850_CTL for control register
  6850_IO for data register.
  CTL numbers used: $16 = no interrupt, 8bit words, 1 stop bit
  64x divide. $56 = RTS high )
CODE 6850>
  BEGIN,
    A 6850_CTL i) in, A $02 i) and, ( are we transmitting? )
  BR CZ jrc, ( yes, loop )
  A C ld, 6850_IO i) A out, BC pop, ;CODE
```

B341

```
CODE 6850<? BC push,
  clrA, ( 256x ) A $16 i) ( RTS lo ) ld, 6850_CTL i) A out,
  BC 0 i) ld, ( pre-push a failure )
  BEGIN, AF AF' ex, ( preserve cnt )
    A 6850_CTL i) in, A $1 i) and, ( rcv buff full? )
    IFNZ, ( full )
      A 6850_IO i) in, pushA, C 1 i) ld, clrA, ( end loop )
    ELSE, AF AF' ex, ( recall cnt ) A dec, THEN,
  BR CNZ jrc,
  A $56 i) ( RTS hi ) ld, 6850_CTL i) A out, ;CODE
```

B342

```
ALIAS 6850<? RX<? ALIAS 6850<? (key?)
ALIAS 6850> TX> ALIAS 6850> (emit)
: 6850$ $56 ( RTS high ) [ 6850_CTL LITN ] PC! ;
```

2.8 Zilog SIO driver: 345-348

B345

```
( Zilog SIO driver. Load range B325-328. Requires:
  SIOA_CTL for ch A control register SIOA_DATA for data
  SIOB_CTL for ch B control register SIOB_DATA for data )
CODE SIOA<? BC push,
  clrA, ( 256x ) BC 0 i) ld, ( pre-push a failure )
  A 5 i) ( PTR5 ) ld, SIOA_CTL i) A out,
  A $68 i) ( RTS low ) ld, SIOA_CTL i) A out,
  BEGIN, AF AF' ex, ( preserve cnt )
    A SIOA_CTL i) in, A $1 i) and, ( rcv buff full? )
    IFNZ, ( full )
      A SIOA_DATA i) in, pushA, C 1 i) ld, clrA, ( end loop )
    ELSE, AF AF' ex, ( recall cnt ) A dec, THEN,
  BR CNZ jrc,
  A 5 i) ( PTR5 ) ld, SIOA_CTL i) A out,
  A $6a i) ( RTS high ) ld, SIOA_CTL i) A out, ;CODE
```

B346

```
CODE SIOA>
  BEGIN,
    A SIOA_CTL i) in, A $04 i) and, ( are we transmitting? )
  BR CZ jrc, ( yes, loop )
  A C ld, SIOA_DATA i) A out, BC pop, ;CODE
CREATE _ ( init data ) $18 C, ( CMD3 )
    $24 C, ( CMD2/PTR4 ) $c4 C, ( WR4/64x/1stop/nopar )
    $03 C, ( PTR3 ) $c1 C, ( WR3/RXen/8char )
    $05 C, ( PTR5 ) $6a C, ( WR5/TXen/8char/RTS )
    $21 C, ( CMD2/PTR1 ) 0 C, ( WR1/Rx no INT )
: SIOA$ _ >A 9 >R BEGIN AC@+ [ SIOA_CTL LITN ] PC! NEXT ;
```

B347

```
CODE SIOB<? BC push, ( copy/paste of SIOA<? )
  clrA, ( 256x ) BC 0 i) ld, ( pre-push a failure )
  A 5 i) ( PTR5 ) ld, SIOB_CTL i) A out,
  A $68 i) ( RTS low ) ld, SIOB_CTL i) A out,
  BEGIN, AF AF' ex, ( preserve cnt )
    A SIOB_CTL i) in, A $1 i) and, ( rcv buff full? )
    IFNZ, ( full )
      A SIOB_DATA i) in, pushA, C 1 i) ld, clrA, ( end loop )
    ELSE, AF AF' ex, ( recall cnt ) A dec, THEN,
  BR CNZ jrc,
  A 5 i) ( PTR5 ) ld, SIOB_CTL i) A out,
  A $6a i) ( RTS high ) ld, SIOB_CTL i) A out, ;CODE
```

B348

```
CODE SIOB>
  BEGIN,
    A SIOB_CTL i) in, A $04 i) and, ( are we transmitting? )
  BR CZ jrc, ( yes, loop )
  A C ld, SIOB_DATA i) A out, BC pop, ;CODE
: SIOB$ _ >A 9 >R BEGIN AC@+ [ SIOB_CTL LITN ] PC! NEXT ;
```

2.9 Sega Master System VDP: 350-352

B350

```
\ VDP Driver. see doc/hw/sms/vdp. Load range B330-B332.
CREATE _idat
$04 C, $80 C, \ Bit 2: Select mode 4
$00 C, $81 C,
$0f C, $82 C, \ Name table: $3800, *B0 must be 1*
$ff C, $85 C, \ Sprite table: $3f00
$ff C, $86 C, \ sprite use tiles from $2000
$ff C, $87 C, \ Border uses palette $f
$00 C, $88 C, \ BG X scroll
$00 C, $89 C, \ BG Y scroll
$ff C, $8a C, \ Line counter (why have this?)
```

B351

```
\ VDP driver
: _sfont ( a -- a+7, Send font to VDP )
  7 >R BEGIN C@+ _data 3 _zero NEXT ( blank row ) 4 _zero ;
: CELL! ( c pos )
  2 * $7800 OR _ctl ( c )
  $20 - ( glyph ) $5f MOD _data ;
```

B352

```
\ VDP driver
: CURSOR! ( new old -- )
  ( unset palette bit in old tile )
  2 * 1+ $7800 OR _ctl 0 _data
  ( set palette bit for at specified pos )
  2 * 1+ $7800 OR _ctl $8 _data ;
: VDP$
  9 >R _idat BEGIN DUP @ _ctl 1+ 1+ NEXT DROP
  ( blank screen ) $7800 _ctl COLS LINES * 2 * _zero
  ( palettes )
  $c000 _ctl
  ( BG ) 1 _zero $3f _data 14 _zero
  ( sprite, inverted colors ) $3f _data 15 _zero
  $4000 _ctl $5f >R ~FNT BEGIN _sfont NEXT DROP
  ( bit 6, enable display, bit 7, ?? ) $81c0 _ctl ;
: COLS 32 ; : LINES 24 ;
```

2.10 SMS PAD: 355-358

B355

```
\ SMS pad driver. See doc/hw/z80/sms/pad. Load range: 355-358
: _prevstat [ PAD_MEM LITN ] ;
: _sel [ PAD_MEM 1+ LITN ] ;
: _next [ PAD_MEM 2 + LITN ] ;
: _sel+! ( n -- ) _sel C@ + _sel C! ;
: _status ( -- n, see doc )
  1 _THA! ( output, high/unselected )
  _D1@ $3f AND ( low 6 bits are good )
( Start and A are returned when TH is selected, in bits 5 and
  4. Well get them, left-shift them and integrate them to B. )
  0 _THA! ( output, low/selected )
  _D1@ $30 AND << << OR ;
```

B356

```
: _chk ( c --, check _sel range )
  _sel C@ DUP $7f > IF $20 _sel C! THEN
  $20 < IF $7f _sel C! THEN ;
CREATE _ '0' C, ':' C, 'A' C, '[' C, 'a' C, $ff C,
: _nxtcls
  _sel @ >R _ BEGIN ( a R:c ) C@+ R@ > UNTIL ( a R:c ) R~
  1- C@ _sel ! ;
```

B357

```
: _updsel ( -- f, has an action button been pressed? )
  _status _prevstat C@ OVER = IF DROP 0 EXIT THEN
  DUP _prevstat C! ( changed, update ) ( s )
  $01 ( UP ) OVER AND NOT IF 1 _sel+! THEN
  $02 ( DOWN ) OVER AND NOT IF -1 _sel+! THEN
  $04 ( LEFT ) OVER AND NOT IF -5 _sel+! THEN
  $08 ( RIGHT ) OVER AND NOT IF 5 _sel+! THEN
  $10 ( BUTB ) OVER AND NOT IF _nxtcls THEN
  ( update sel in VDP )
  _chk _sel C@ XYPOS CELL!
  ( return whether any of the high 3 bits is low )
  $e0 AND $e0 < ;
```

B358

```
: (key?) ( -- c? f )
  _next C@ IF _next C@ 0 _next C! 1 EXIT THEN
  _updsel IF
    _prevstat C@
    $20 ( BUTC ) OVER AND NOT IF DROP _sel C@ 1 EXIT THEN
    $40 ( BUTA ) AND NOT IF $8 ( BS ) 1 EXIT THEN
    ( If not BUTC or BUTA, it has to be START )
    $d _next C! _sel C@ 1
    ELSE 0 ( f ) THEN ;
: PAD$ $ff _prevstat C! 'a' _sel C! 0 _next C! ;
```

2.11 SMS KBD: 360-361

B360

```
( kbd - implement (ps2kc) for SMS PS/2 adapter )
: (ps2kcA) ( for port A )
( Before reading a character, we must first verify that there
is something to read. When the adapter is finished filling its
'164 up, it resets the latch, which output's is connected to
TL. When the '164 is full, TL is low. Port A TL is bit 4 )
  _D1@ $10 AND IF 0 EXIT ( nothing ) THEN
  0 _THA! ( Port A TH output, low )
  _D1@ ( bit 3:0 go in 3:0 ) $0f AND ( n )
  1 _THA! ( Port A TH output, high )
  _D1@ ( bit 3:0 go in 7:4 ) $0f AND << << << << OR ( n )
  2 _THA! ( TH input ) ;
```

B361

```
: (ps2kcB) ( for port B )
  ( Port B TL is bit 2 )
  _D2@ $04 AND IF 0 EXIT ( nothing ) THEN
  0 _THB! ( Port B TH output, low )
  _D1@ ( bit 7:6 go in 1:0 ) >> >> >> >> >> >> ( n )
  _D2@ ( bit 1:0 go in 3:2 ) $03 AND << << OR ( n )
  1 _THB! ( Port B TH output, high )
  _D1@ ( bit 7:6 go in 5:4 ) $c0 AND >> >> OR ( n )
  _D2@ ( bit 1:0 go in 7:6 ) $03 AND <<8 >> >> OR ( n )
  2 _THB! ( TH input ) ;
```

2.12 SMS SPI relay: 367

B367

```
: (spie) DROP ; ( always enabled )
CODE (spix) ( x -- x, for port B )
  \ TR = DATA TH = CLK
  A CPORT_MEM m) ld, A $f3 i) and, ( TR/TH output )
  B 8 i) ld, BEGIN,
    A $bf i) and, ( TR lo ) C rl,
    IFC, A $40 i) or, ( TR hi ) THEN,
    CPORT_CTL i) A out, ( clic! ) A $80 i) or, ( TH hi )
    CPORT_CTL i) A out, ( clac! )
    AF AF' ex, A CPORT_D1 i) in, ( Up Btn is B6 ) rla, rla,
      L rl, AF AF' ex,
    A $7f i) and, ( TH lo ) CPORT_CTL i) A out, ( cloc! )
  BR djnz, CPORT_MEM m) A ld, C L ld, ;CODE
```

2.13 SMS Ports: 368-369

B368

```
\ Routines for interacting with SMS controller ports.
\ Requires CPORT_MEM, CPORT_CTL, CPORT_D1 and CPORT_D2 to be
\ defined. CPORT_MEM is a 1 byte buffer for CPORT_CTL. The last
\ 3 consts will usually be $3f, $dc, $dd.
\ mode -- set TR pin on mode a on:
\ 0= output low 1=output high 2=input
CODE _TRA! ( B0 -> B4, B1 -> B0 )
  C rr, rla, rla, rla, rla, B rr, rla,
  A $11 i) and, C A ld, A CPORT_MEM m) ld,
  A $ee i) and, A C or, CPORT_CTL i) A out, CPORT_MEM m) A ld,
  BC pop, ;CODE
CODE _THA! ( B0 -> B5, B1 -> B1 )
  C rr, rla, rla, rla, rla, C rr, rla, rla,
  A $22 i) and, C A ld, A CPORT_MEM m) ld,
  A $dd i) and, A C or, CPORT_CTL i) A out, CPORT_MEM m) A ld,
  BC pop, ;CODE
```

B369

```
CODE _TRB! ( B0 -> B6, B1 -> B2 )
  C rr, rla, rla, rla, rla, C rr, rla, rla, rla,
  A $44 i) and, C A ld, A CPORT_MEM m) ld,
  A $bb i) and, A C or, CPORT_CTL i) A out, CPORT_MEM m) A ld,
  BC pop, ;CODE
CODE _THB! ( B0 -> B7, B1 -> B3 )
  C rr, rla, rla, rla, rla, C rr, rla, rla, rla, rla,
  A $88 i) and, C A ld, A CPORT_MEM m) ld,
  A $77 i) and, A C or, CPORT_CTL i) A out, CPORT_MEM m) A ld,
  BC pop, ;CODE
CODE _D1@ BC push, A CPORT_D1 i) in, C A ld, B 0 i) ld, ;CODE
CODE _D2@ BC push, A CPORT_D2 i) in, C A ld, B 0 i) ld, ;CODE
```

2.14 TI-84+ LCD: 370-373

B370

```
( TI-84+ LCD driver. See doc/hw/z80/ti84/lcd.txt
  Load range: 350-353 )
: _mem+ [ LCD_MEM LITN ] @ + ;
: FNTW [ LCD_FNTW LITN ] ; : FNTH [ LCD_FNTH LITN ] ;
: COLS 96 FNTW 1+ / ; : LINES 64 FNTH 1+ / ;
( Wait until the lcd is ready to receive a command. It's a bit
  weird to implement a waiting routine in asm, but the forth
  version is a bit heavy and we don't want to wait longer than
  we have to. )
CODE _wait
  BEGIN,
    A $10 i) ( CMD ) in,
    rla, ( When 7th bit is clr, we can send a new cmd )
  BR CC jrc, ;CODE
```

B371

```
: LCD_BUF 0 _mem+ ;
: _cmd $10 ( CMD ) PC! _wait ;
: _data! $11 ( DATA ) PC! _wait ;
: _data@ $11 ( DATA ) PC@ _wait ;
: LCDOFF $02 ( CMD_DISABLE ) _cmd ;
: LCDON $03 ( CMD_ENABLE ) _cmd $17 ( power on ) _cmd
  $f0 _cmd ( some contrast ) ;
: _yinc $07 _cmd ; : _xinc $05 _cmd ;
: _zoff! ( off -- ) $40 + _cmd ;
: _col! ( col -- ) $20 + _cmd ;
: _row! ( row -- ) $80 + _cmd ;
: LCD$ HERE [ LCD_MEM LITN ] ! FNTH 2 * ALLOT
  LCDON $01 ( 8-bit mode ) _cmd FNTH 1+ _zoff! ;
```

B372

```
: _clrrows ( n u -- Clears u rows starting at n )
  >R _row! BEGIN
    _yinc 0 _col! 11 >R BEGIN 0 _data! NEXT
    _xinc 0 _data! NEXT ;
: NEWLN ( oldln -- newln )
  1+ DUP 1+ FNTH 1+ * _zoff! ( ln )
  DUP FNTH 1+ * FNTH 1+ _clrrows ( newln ) ;
: LCDCLR 0 64 _clrrows ;
```

B373

```
: _atrow! ( pos -- ) COLS / FNTH 1+ * _row! ;
: _tocol ( pos -- col off ) COLS MOD FNTW 1+ * 8 /MOD ;
: CELL! ( c pos -- )
  DUP _atrow! DUP _tocol _col! ROT ( pos coff c )
  $20 - FNTH * ~FNT + ( pos coff a )
  _xinc _data@ DROP
  A> >R LCD_BUF >A FNTH >R BEGIN ( pos coff a )
    OVER 8 -^ SWAP C@+ ( pos coff 8-coff a+1 c ) ROT LSHIFT
    _data@ <<8 OR ( pos coff a+1 c )
    DUP A> FNTH + C! >>8 AC!+
  NEXT 2DROP ( pos )
  DUP _atrow!
  LCD_BUF >A FNTH >R BEGIN AC@+ _data! NEXT
  DUP _atrow! _tocol NIP 1+ _col!
  FNTH >R BEGIN AC@+ _data! NEXT R> >A ;
```

2.15 TI-84+ Keyboard: 375-379

B375

```
\ Requires KBD_MEM, KBD_PORT and nC, from B120.
\ Load range: 355-359

\ gm -- pm, get pressed keys mask for group mask gm
CODE _get
  di,
    A $ff i) ld,
    KBD_PORT i) A out,
    A C ld,
    KBD_PORT i) A out,
    A KBD_PORT i) in,
  ei,
  C A ld,
;CODE
```

B376

```
\ wait until all keys are de-pressed. To avoid repeat keys, we
\ require 64 subsequent polls to indicate all depressed keys.
\ all keys are considered depressed when the 0 group returns
\ $ff.
: _wait 64 BEGIN 0 _get $ff = NOT IF DROP 64 THEN
    1- DUP NOT UNTIL DROP ;
\ digits table. each row represents a group. 0 means unsupported
\ no group 7 because it has no key. $80 = alpha, $81 = 2nd
CREATE _dtbl 7 8 * nC,
  0   0   0   0   0   0    0 0
  $d  '+' '-' '*' '/' '^'  0 0
  0   '3' '6' '9' ')' 0    0 0
  '.' '2' '5' '8' '(' 0    0 0
  '0' '1' '4' '7' ',' 0    0 0
  0   0   0   0   0   0    0 $80
  0   0   0   0   0   $81  0 $7f
```

B377

```
\ alpha table. same as _dtbl, for when we're in alpha mode.
CREATE _atbl 7 8 * nC,
  0   0   0   0   0   0   0   0
  $d  '"' 'W' 'R' 'M' 'H' 0   0
  '?' 0   'V' 'Q' 'L' 'G' 0   0
  ':' 'Z' 'U' 'P' 'K' 'F' 'C' 0
  32 'Y'  'T' 'O' 'J' 'E' 'B' 0
  0  'X'  'S' 'N' 'I' 'D' 'A' $80
  0   0   0   0   0    $81 0  $7f
: _@ [ KBD_MEM LITN ] C@ ; : _! [ KBD_MEM LITN ] C! ;
: _2nd@ _@ 1 AND ; : _2nd! _@ $fe AND + _! ;
: _alpha@ _@ 2 AND ; : _alpha! 2 * _@ $fd AND + _! ;
: _alock@ _@ 4 AND ; : _alock^ _@ 4 XOR _! ;
```

B378

```
: _gti ( -- tindex, that it, index in _dtbl or _atbl )
  7 >R 0 BEGIN ( gid )
    1 OVER LSHIFT $ff -^ ( gid dmask ) _get
    DUP $ff = IF DROP 1+ ELSE R~ 1 >R THEN
  NEXT ( gid dmask )
  _wait $ff XOR ( dpos ) 0 ( dindex )
  BEGIN 1+ 2DUP RSHIFT NOT UNTIL 1-
  ( gid dpos dindex ) NIP
  ( gid dindex ) SWAP 8 * + ;
```

B379

```
CODE HALT 0 rst,
: (key?) ( -- c? f )
  [ ONPRESSED LITN ] C@ IF HALT THEN
  0 _get $ff = IF ( no key pressed ) 0 EXIT THEN
  _alpha@ _alock@ IF NOT THEN IF _atbl ELSE _dtbl THEN
  _gti + C@ ( c )
  DUP $80 = IF _2nd@ IF _alock^ ELSE 1 _alpha! THEN THEN
  DUP $81 = _2nd!
  DUP 1 $7f =><= IF ( we have something )
  ( lower? ) _2nd@ IF DUP 'A' 'Z' =><= IF $20 OR THEN THEN
      0 _2nd! 0 _alpha! 1 ( c f )
  ELSE ( nothing ) DROP 0 THEN ;
: KBD$ 0 [ KBD_MEM LITN ] C! ;
```

2.16 TI-84+ Boot code & macros: 380-382

B380

```
\ TI-84+ XCOMP macros
: debounce,
  HL 0 i) ld, DE 1 i) ld,
  BEGIN, HL DE add, BR CNC jrc, ;
```

B381

```
\ TI-84+ Boot code, RST 0 and INT handler
$59 jp, $18 OALLOT
$59 jp, ( reboot ) $38 OALLOT
\ handleInterrupt acknowledge
di,
AF push,
A $08 i) ld,
$03 i) ( PORT_INT_MASK ) A out,
ONPRESSED m) A ld,
AF pop,
ei, reti,

$53 OALLOT
$59 jp, ( $56 ) $ff C, $a5 C, $ff C, ( $59 )
```

B382

```
\ TI-84+ Boot code, offset $59
di, im1, SP RS_ADDR i) ld,
A A xor, $57 i) A out, ( disable USB interrupts )
$04 i) A out, \ memory mode 0
A $81 i) ld, ( bits 0-2 are RAM page, for addr $8000 )
$07 i) ( PORT_BANKB ) A out,
A $02 i) ( LCD_CMD_DISABLE ) ld,
$10 i) ( LCD_PORT_CMD ) A out,
debounce,
A $01 i) ld, ( enable ON key, low power mode )
ei, $03 i) ( PORT_INT_MASK ) A out,
halt, di, debounce,
A A xor, ONPRESSED m) A ld,
A $09 i) ld, ( enable ON key )
$03 i) ( PORT_INT_MASK ) A out,
ei, $100 jp, $100 OALLOT
```

2.17 TRS-80 4P drivers: 390-401

B390

```
\ TRS-80 drivers declarations and macros
\ FDMEM 3b: FDSEL 1b FDOP 2b
: TRS804P 391 399 LOADR ;
$f800 VALUE VIDMEM $bf VALUE CURCHAR
0 VALUE lblflush
: fdstat A $f0 i) in, ;
: fdcmd ( i )
  A SWAP i) ld, B $18 i) ld, $f0 i) A out, BEGIN, BR djnz, ;
: fdwait BEGIN, fdstat rrca, BR CC jrc, rlca, ;
: vid+, ( reg -- ) HL VIDMEM i) ld, HL SWAP add, ;
```

B391

```
\ TRS-80 4P video driver
24 CONSTANT LINES 80 CONSTANT COLS
CODE CELL! ( c pos -- ) HL pop,
  A L ld, BC vid+, (HL) A ld, BC pop, ;CODE
CODE CELLS! ( a pos u -- ) BC push, exx, BC pop, DE pop,
  DE vid+, DE HL ex, HL pop, BCZ, IFNZ, ldir, THEN, exx, BC pop,
;CODE
CODE CURSOR! ( new old -- ) BC vid+, A (HL) ld, A CURCHAR i) cp,
  IFZ, A UNDERCUR m) ld, (HL) A ld, THEN,
  BC pop, BC vid+, A (HL) ld, UNDERCUR m) A ld, A CURCHAR i) ld,
  (HL) A ld, BC pop, ;CODE
CODE SCROLL ( -- )
  exx, HL VIDMEM 80 + i) ld, DE VIDMEM i) ld, BC 1840 i) ld,
  ldir, H D ld, L E ld, DE inc, A SPC i) ld, (HL) A ld,
  BC 79 i) ld, ldir, exx, ;CODE
: NEWLN ( old -- new ) 1+ DUP LINES = IF 1- SCROLL THEN ;
```

B392

```
LSET L2 ( seek, B=trk ) A 21 i) ld, A B cp, A FDMEM m) ld,
  IFC, A $20 i) or, ( WP ) THEN,
  A $80 i) or, $f4 i) A out, \ FD sel
  A B ld, ( trk ) $f3 i) A out, $1c fdcmd ret,
CODE FDRD ( trksec addr -- st ) BC>HL, BC pop,
  L2 call, fdwait A $98 i) and, IFZ, di,
    A C ld, $f2 i) A out, ( sec ) C $f3 i) ld, $84 fdcmd \ read
    BEGIN, BEGIN, fdstat A $b6 i) and, BR CZ jrc, \ DRQ
      A $b4 i) and, IFZ, TO L3 ( error ) ini, BR CNZ jrc, THEN,
  fdwait A $3c i) and, L3 FMARK A>BC, ei, ;CODE
CODE FDWR ( trksec addr -- st ) BC>HL, BC pop,
  L2 call, fdwait A $98 i) and, IFZ, di,
    A C ld, $f2 i) A out, ( sec ) C $f3 i) ld, $a4 fdcmd \ read
    BEGIN, BEGIN, fdstat A $f6 i) and, BR CZ jrc, \ DRQ
      A $f4 i) and, IFZ, TO L3 ( error ) outi, BR CNZ jrc, THEN,
  fdwait A $3c i) and, L3 FMARK A>BC, ei, ;CODE
```

B393

```
CODE _dsel ( fdmask -- )
  A C ld, FDMEM m) A ld, A $80 i) or, $f4 i) A out,
  0 fdcmd ( restore ) fdwait BC pop, ;CODE
: DRVSEL ( drv -- ) 1 SWAP LSHIFT [ FDMEM LITN ] C@ OVER = NOT
  IF _dsel ELSE DROP THEN ;
: FD$ 1 DRVSEL ;
FDMEM 1+ DUP CONSTANT 'FDOP *ALIAS FDOP
: _err S" FDerr " STYPE .X ABORT ;
: _trksec ( sec -- trksec )
\ 4 256b sectors per block, 18 sec per trk, 40 trk max
  18 /MOD ( sec trk ) DUP 39 > IF $ffff _err THEN <<8 + ;
```

B394

```
: FD@! ( blk blk( -- )
  A> >R SWAP << << ( blk*4=sec ) >A 4 >R BEGIN ( dest )
    A> A+ _trksec OVER ( dest trksec dest )
    FDOP ( dest ) ?DUP IF _err THEN $100 +
  NEXT DROP R> >A ;
: FD@ ['] FDRD 'FDOP ! FD@! ;
: FD! ['] FDWR 'FDOP ! FD@! ;
```

B395

```
: CL$ ( baudcode -- )
  $02 $e8 PC! ( UART RST ) DUP 16 * OR $e9 PC! ( bauds )
  $6d $ea PC! ( word8 no parity no-RTS ) ;
CODE TX> BEGIN,
    A $ea i) in, A $40 i) and, IFNZ, ( TX reg empty )
      A $e8 i) in, A $80 i) and, IFZ, ( CTS low )
        A C ld, $eb i) A out, ( send byte ) BC pop, ;CODE
  THEN, THEN, BR jr,
```

B396

```
CODE RX<? BC push,
  clrA, ( 256x ) BC 0 i) ld, ( pre-push a failure )
  A $6c i) ( RTS low ) ld, $ea i) A out,
  BEGIN, AF AF' ex, ( preserve cnt )
    A $ea i) in, A $80 i) and, ( rcv buff full? )
    IFNZ, ( full )
      A $eb i) in, A>HL, HL push, C inc, clrA, ( end loop )
    ELSE, AF AF' ex, ( recall cnt ) A dec, THEN,
  BR CNZ jrc,
  A $6d i) ( RTS high ) ld, $ea i) A out, ;CODE
```

B397

```
LSET L1 6 nC, '`' 'h' 'p' 'x' '0' '8'
LSET L2 8 nC, $0d 0 $ff 0 0 $08 0 $20
PC XORG $39 + T! ( RST 38 )
AF push, HL push, DE push, BC push,
A $ec i) in, ( RTC INT ack )
A $f440 m) ld, A A or, IFNZ, \ 7th row is special
  HL L2 1- i) ld, BEGIN, HL inc, rra, BR CNC jrc,
  A (HL) ld, ELSE, \ not 7th row
  HL L1 i) ld, DE $f401 i) ld, BC $600 i) ld, BEGIN,
    A (DE) ld, A A or, IFNZ,
      C (HL) ld, BEGIN, C inc, rra, BR CNC jrc,
      C dec, THEN,
    E sla, HL inc, BR djnz,
  A C ld, THEN, \ cont.
```

B398

```
\ A=char or zero if no keypress. Now let's debounce
HL KBD_MEM 2 + i) ld, A A or, IFZ, \ no keypress, debounce
  (HL) A ld, ELSE, \ keypress, is it debounced?
  A (HL) cp, IFNZ, \ != debounce buffer
    C A ld, (HL) C ld, A $ff i) cp, IFZ, \ BREAK!
      HL pop, HL pop, HL pop, HL pop, HL pop, ei,
      X' QUIT jp, THEN,
    HL dec, A $f480 m) ld, A 3 i) and, (HL) A ld, HL dec,
    (HL) C ld, THEN, THEN,
BC pop, DE pop, HL pop, AF pop, ei, ret,
```

B399

```
KBD_MEM CONSTANT KBDBUF \ LSB=char MSB=shift
: KBD$ 0 KBDBUF ! $04 $e0 PC! ( enable RTC INT ) (im1) ;
: (key?) KBDBUF @ DUP <<8 >>8 NOT IF DROP 0 EXIT THEN
  0 KBDBUF ! L|M ( char flags )
  OVER '<' '`' =><= IF 1 XOR THEN \ invert shift
  TUCK 1 AND IF \ lshift  ( flags char )
    DUP '@' < IF $ef ELSE $df THEN AND THEN
  SWAP 2 AND IF \ rshift ( char )
    DUP '1' < IF $2f ELSE $4a THEN + THEN
  1 ( success ) ;
```

B401

```
\ TRS-80 4P bootloader. Loads sectors 2-17 to addr 0.
di, A $86 i) ld, $84 i) A out, \ mode 2, 80 chars, page 1
A $81 i) ld, $f4 i) A out, \ DRVSEL DD, drv0
A $40 i) ld, $ec i) A out, \ MODOUT 4MHZ, no EXTIO
HL 0 i) ld, ( dest addr ) clrA, $e4 i) A out, ( no NMI )
A inc, ( trk1 ) BEGIN,
  $f3 i) A out, AF AF' ex, ( save ) $18 ( seek ) fdcmd fdwait
  clrA, $f2 i) A out, C $f3 i) ld, BEGIN,
    $80 ( read sector ) fdcmd ( B=0 )
    BEGIN, fdstat rra, rra, BR CNC jrc, ( DRQ )
    ini, A $c1 i) ld, BEGIN, $f4 i) A out, ini, BR CNZ jrc,
    fdwait A $1c i) ( error mask ) and, IFNZ,
      A SPC i) add, VIDMEM m) A ld, BEGIN, BR jr, THEN,
    A $f2 i) in, A inc, $f2 i) A out, A 18 i) cp, BR CC jrc,
  AF AF' ex, ( restore ) A inc, A 3 i) cp, BR CC jrc, 0 rst,
```

2.18 Dan SBC drivers: 405-419

B405

```
\ Dan SBC drivers. See doc/hw/z80/dan.txt
\ Macros
: OUTii, ( val port -- ) A ROT i) ld, i) A out, ;
: repeat ( n -- ) >R ' BEGIN ( w ) DUP EXECUTE NEXT DROP ;
```

B406

```
\ SPI relay driver
CODE (spix) ( n -- n )
  A C ld,
  SPI_DATA i) A out,
  ( wait until xchg is done )
  nop, nop, nop, nop,
  A SPI_DATA i) in,
  C A ld, ;CODE
CODE (spie) ( n -- )
  $9A CTL8255 OUTii, $3 CTL8255 OUTii,
  A C ld, A 1 i) xor, A 1 i) and, CTL8255 i) A out,
  BC pop, ;CODE
```

B407

```
\ software framebuffer subsystem
VID_MEM CONSTANT VD_DECFR
VID_MEM $02 + CONSTANT VD_DECTL
VID_MEM $04 + CONSTANT VD_CURCL
VID_MEM $06 + CONSTANT VD_FRMST
VID_MEM $08 + CONSTANT VD_COLS
VID_MEM $0A + CONSTANT VD_LINES
VID_MEM $0C + CONSTANT VD_FRB
VID_MEM $0E + CONSTANT VD_OFS
\ Clear Framebuffer
CODE (vidclr) ( -- ) BC push,
  $9A CTL8255 OUTii, $3 CTL8255 OUTii, $1 CTL8255 OUTii,
  BC VID_MEM $10 + i) ld, HL VID_WDTH VID_SCN * i) ld,
  BEGIN, clrA, (BC) A ld, BC inc, HL dec, HLZ, BR CNZ jrc,
  BC pop, ;CODE
```

B408

```
: VID_OFS
  [ VID_WDTH 8 * LITN ] * + VD_FRB @ + VD_OFS ! (vidclr) ;
: VID$ ( -- )
  1 VD_DECFR ! 0 VD_DECTL ! 0 VD_CURCL ! 0
  VD_FRMST ! [ VID_WDTH 1 - LITN ] VD_COLS !
  [ VID_LN 1 - LITN ] VD_LINES !
  [ VID_MEM $10 + LITN ] VD_FRB ! 1 4 VID_OFS ;
```

B409

```
: COLS VD_COLS @ ;
: LINES VD_LINES @ ;
: VID_LOC VD_COLS @ /MOD
  [ VID_WDTH 8 * LITN ] * VD_OFS @ + ;
: CELL! VID_LOC + SWAP SPC - DUP 96 < IF
  DUP DUP << + << + ~FNT + 7 >R BEGIN
  2DUP C@ >> SWAP C! 1+ SWAP
  [ VID_WDTH LITN ] + SWAP NEXT
  DROP 0 SWAP C! ELSE 2DROP THEN ;
```

B410

```
: VID_LCR VID_LOC SWAP DUP
  DUP 12 < IF DROP 0 ELSE 12 -
  DUP [ VID_WDTH 24 - LITN ] > IF DROP [ VID_WDTH 24 - LITN ]
  THEN THEN VD_CURCL ! ;
: CURSOR! 0 SWAP VID_LOC + [ VID_WDTH 7 * LITN ] + C!
  255 SWAP VID_LCR + [ VID_WDTH 7 * LITN ] + C! ;
CODE (vidscr) BC push, exx,
 BC VID_SCN 8 - VID_WDTH * i) ld, DE VID_MEM $10 + i) ld,
 HL VID_MEM $10 + VID_WDTH 8 * + i) ld,
 ldir, HL VID_WDTH 8 * i) ld,
 BEGIN, clrA, (DE) A ld, DE inc, HL dec, HLZ,
 BR CNZ jrc, exx, BC pop, ;CODE
: NEWLN DUP 1+ VD_LINES @ = IF (vidscr) ELSE 1+ THEN ;
```

B411

```
\ Stream video frames, single scan
CODE (vidfr) ( -- ) BC push, exx,
  C SPI_DATA i) ld, DE VID_MEM $04 + m) ld,
  HL VID_MEM 40 + VID_WDTH - i) ld, HL DE add,
  VID_MEM $06 + m) HL ld, DE VID_WDTH 24 - i) ld,
  B VID_SCN i) ld,
  LSET L1 BEGIN,
    6 CTL8255 OUTii, HL DE add, 7 CTL8255 OUTii,
    A B ld, 4 repeat nop, 24 repeat outi,
    B A ld, BR djnz,
  B 0 i) ld, B 0 i) ld, B 0 i) ld, B VID_VBL 1 - i) ld, FJR jr,
  LSET L2 A VID_VBL 1 - i) ld, FJR jr, FMARK FMARK
    A B ld, B 28 i) ld, BEGIN, BR djnz, HL inc, B A ld,
    7 CTL8255 OUTii, 5 repeat nop, 6 CTL8255 OUTii,
  L2 BR djnz,
```

B412

```
  A VID_MEM $02 + m) ld, B A ld, A VID_MEM m) ld,
  A B sub, IFNZ,
    VID_MEM m) A ld, B 23 i) ld, HL inc, B 23 i) ld,
    BEGIN, BR djnz,
    HL VID_MEM $06 + m) ld, B VID_SCN i) ld, 7 CTL8255 OUTii,
    5 repeat nop, 6 CTL8255 OUTii, L1 jp,
  THEN, exx, BC pop, ;CODE
```

B413

```
\ Stream video frames, double scan
CODE (vidfr) ( -- ) BC push, exx,
  C SPI_DATA i) ld, DE VID_MEM $04 + m) ld,
  HL VID_MEM 40 + VID_WDTH - i) ld, HL DE add,
  VID_MEM $06 + m) HL ld, DE VID_WDTH 24 - i) ld,
  B VID_SCN i) ld,
  LSET L1 BEGIN,
    6 CTL8255 OUTii, HL DE add, 7 CTL8255 OUTii, A B ld,
    DE dec, DE -25 i) ld, 24 repeat outi, AF push, DE inc,
    6 CTL8255 OUTii, HL DE add, 7 CTL8255 OUTii, AF pop,
    DE VID_WDTH 24 - i) ld, 24 repeat outi, B A ld, BR djnz,
  B 0 i) ld, B 0 i) ld, B 0 i) ld, B VID_VBL 1 - i) ld, FJR jr,
  LSET L2 A VID_VBL 1 - i) ld, FJR jr, FMARK FMARK
    A B ld, B 28 i) ld, BEGIN, BR djnz, HL inc, B A ld,
    7 CTL8255 OUTii, 5 repeat nop, 6 CTL8255 OUTii,
  L2 BR djnz,
```

B414

```
  A VID_MEM $02 + m) ld, B A ld, A VID_MEM m) ld, A B sub,
  IFNZ,
    VID_MEM m) A ld, B 23 i) ld, HL inc, B 23 i) ld,
    BEGIN, BR djnz, HL VID_MEM $06 + m) ld, B VID_SCN i) ld,
    7 CTL8255 OUTii, 5 repeat nop, 6 CTL8255 OUTii, L1 jp,
  THEN, exx, BC pop, ;CODE
```

B415

```
\ PS2 keyboard driver subsystem
PSK_MEM CONSTANT PSK_STAT
PSK_MEM $02 + CONSTANT PSK_CC
PSK_MEM $04 + CONSTANT PSK_BUFI
PSK_MEM $06 + CONSTANT PSK_BUFO
PSK_MEM $08 + CONSTANT PSK_BUF
PC XORG $39 + T! ( RST 38 )
di, AF push, $10 SIOA_CTL OUTii, A SIOA_CTL i) in,
A 4 bit, IFZ, AF pop, ei, reti, THEN,  ( I1 - T1 )
A PSK_MEM m) ld, A A or,
IFZ, A PTC8255 i) in, A 7 bit,         ( I1 - )
IFZ, A 1 i) ld, PSK_MEM m) A ld, THEN, ( I2 - T2 )
```

B416

```
AF pop, ei, reti, THEN,             ( - T1 )
A $9 i) cp, FJR CNZ jrc, TO L3
HL push, HL PSK_MEM $02 + m) ld, H 8 i) ld, clrA,
BEGIN, L rrc, A 0 i) adc, H dec, BR CNZ jrc,
H A ld, A PTC8255 i) in, A H ld, A 0 i) adc, A $1 i) and,
FJR CZ jrc, TO L1 clrA, VID_MEM m) A ld, VID_MEM $02 + m) A ld,
A PSK_MEM $04 + m) ld, L A ld, A PSK_MEM $06 + m) ld,
A inc, A PS2_BMSK i) and, A L cp, FJR CZ jrc, TO L1
PSK_MEM $06 + m) A ld, L A ld,
A PSK_MEM $08 + <<8 >>8 i) ld, A L add, L A ld,
A PSK_MEM $08 + >>8 i) ld, A 0 i) adc,
```

B417

```
H A ld, A PSK_MEM $02 + m) ld, (HL) A ld,
L1 FMARK clrA, PSK_MEM m) A ld, HL pop, AF pop, ei, reti,
L3 FMARK A PTC8255 i) in, rlca, A PSK_MEM $02 + m) ld,
rra, PSK_MEM $02 + m) A ld,
A PSK_MEM m) ld, A inc, PSK_MEM m) A ld,
AF pop, ei, reti,
```

B418

```
CODE (pskset)
  di, $11 SIOA_CTL OUTii, $19 SIOA_CTL OUTii, im1, ei, ;CODE
: PSK< ( -- n )
  PSK_BUFI @ PSK_BUFO @ = IF 0 ELSE PSK_BUFI @
  1+ [ PS2_BMSK LITN ] AND DUP PSK_BUF + C@
  SWAP PSK_BUFI ! THEN ;
: PSKV< ( -- n )
  PSK_BUFI @ PSK_BUFO @ = IF
  BEGIN 1 VD_DECFR ! (vidfr)
  PSK_BUFI @ PSK_BUFO @ = NOT UNTIL THEN
  PSK_BUFI @ 1+ [ PS2_BMSK LITN ] AND DUP
  PSK_BUF + C@ SWAP PSK_BUFI ! ;
: PSK$ ( -- )
  0 PSK_BUFO ! 0 PSK_BUFI ! 0 PSK_STAT ! (pskset) ;
```

B419

```
: (ps2kc) 0 BEGIN DROP PSKV<
  DUP 5 = IF 0 VD_CURCL ! DROP 0 THEN
  DUP 6 = IF VD_CURCL @ 4 < IF 0 ELSE VD_CURCL @ 4 - THEN
    VD_CURCL ! DROP 0 THEN
  DUP 4 = IF VD_CURCL @ [ VID_WDTH 28 - LITN ] > IF
    [ VID_WDTH 24 - LITN ] ELSE VD_CURCL @ 4 + THEN
    VD_CURCL ! DROP 0 THEN DUP UNTIL ;
```

3 AVR

3.1 Architecture index: 300

B300

```
AVR MASTER INDEX

301 AVR macros                  302 AVR assembler
320 SMS PS/2 controller         345 Arduino blinker
350 Arduino SPI spitter
```

3.2 AVR macros: 301

B301

```
: AVRA 302 312 LOADR ;
: ATMEGA328P 315 LOAD ;
```

3.3 AVR assembler: 302-312

B302

```
\ AVR assembler. See doc/asm/avr.txt.
\ We divide by 2 because each PC represents a word.
: PC HERE XORG - >> ;
: <<3 << << << ; : <<4 <<3 << ;
: _oor ." arg out of range: " .X SPC> ." PC " PC .X NL> ABORT ;
: _r8c DUP 7 > IF _oor THEN ;
: _r32c DUP 31 > IF _oor THEN ;
: _r16+c _r32c DUP 16 < IF _oor THEN ;
: _r64c DUP 63 > IF _oor THEN ;
: _r256c DUP 255 > IF _oor THEN ;
: _Rdp ( op rd -- op', place Rd ) <<4 OR ;
```

B303

```
( 0000 000d dddd 0000 )
: OPRd DOER , DOES> @ SWAP _r32c _Rdp L, ;
$9405 OPRd ASR,   $9400 OPRd COM,
$940a OPRd DEC,   $9403 OPRd INC,
$9206 OPRd LAC,   $9205 OPRd LAS,
$9207 OPRd LAT,
$9406 OPRd LSR,   $9401 OPRd NEG,
$900f OPRd POP,   $920f OPRd PUSH,
$9407 OPRd ROR,   $9402 OPRd SWAP,
$9204 OPRd XCH,

$9200 OPRd _ : STS, ( k16 rd ) _ L, ;
$9000 OPRd _ : LDS, ( rd k16 ) SWAP _ L, ;
```

B304

```
( 0000 00rd dddd rrrr )
: OPRdRr DOER C, DOES> C@ ( rd rr op )
    OVER _r32c $10 AND >> >> >> OR ( rd rr op' )
    <<8 OR $ff0f AND ( rd op' )
    SWAP _r32c _Rdp L, ;
$1c OPRdRr ADC,   $0c OPRdRr ADD,    $20 OPRdRr AND,
$14 OPRdRr CP,    $04 OPRdRr CPC,    $10 OPRdRr CPSE,
$24 OPRdRr EOR,   $2c OPRdRr MOV,    $9c OPRdRr MUL,
$28 OPRdRr OR,    $08 OPRdRr SBC,    $18 OPRdRr SUB,

( 0000 0AAd dddd AAAA )
: OPRdA DOER C, DOES> C@ ( rd A op )
    OVER _r64c $30 AND >> >> >> OR ( rd A op' )
    <<8 OR $ff0f AND ( rd op' ) SWAP _r32c _Rdp L, ;
$b0 OPRdA IN,     $b8 OPRdA _ : OUT, SWAP _ ;
```

B305

```
( 0000 KKKK dddd KKKK )
: OPRdK DOER C, DOES> C@ ( rd K op )
    OVER _r256c $f0 AND >> >> >> >> OR ( rd K op' )
    ROT _r16+c <<4 ROT $0f AND OR ( op' rdK ) C, C, ;
$70 OPRdK ANDI,   $30 OPRdK CPI,     $e0 OPRdK LDI,
$60 OPRdK ORI,    $40 OPRdK SBCI,    $60 OPRdK SBR,
$50 OPRdK SUBI,

( 0000 0000 AAAA Abbb )
: OPAb DOER C, DOES> C@ ( A b op )
    ROT _r32c <<3 ROT _r8c OR C, C, ;
$98 OPAb CBI,     $9a OPAb SBI,      $99 OPAb SBIC,
$9b OPAb SBIS,
```

B306

```
: OPNA DOER , DOES> @ L, ;
$9598 OPNA BREAK, $9488 OPNA CLC,    $94d8 OPNA CLH,
$94f8 OPNA CLI,   $94a8 OPNA CLN,    $94c8 OPNA CLS,
$94e8 OPNA CLT,   $94b8 OPNA CLV,    $9498 OPNA CLZ,
$9419 OPNA EIJMP, $9509 OPNA ICALL,  $9519 OPNA EICALL,
$9409 OPNA IJMP,  $0000 OPNA NOP,    $9508 OPNA RET,
$9518 OPNA RETI,  $9408 OPNA SEC,    $9458 OPNA SEH,
$9478 OPNA SEI,   $9428 OPNA SEN,    $9448 OPNA SES,
$9468 OPNA SET,   $9438 OPNA SEV,    $9418 OPNA SEZ,
$9588 OPNA SLEEP, $95a8 OPNA WDR,
```

B307

```
( 0000 0000 0sss 0000 )
: OPb DOER , DOES> @ ( b op )
    SWAP _r8c _Rdp L, ;
$9488 OPb BCLR,   $9408 OPb BSET,

( 0000 000d dddd 0bbb )
: OPRdb DOER , DOES> @ ( rd b op )
    ROT _r32c _Rdp SWAP _r8c OR L, ;
$f800 OPRdb BLD,  $fa00 OPRdb BST,
$fc00 OPRdb SBRC, $fe00 OPRdb SBRS,

( special cases )
: CLR, DUP EOR, ;  : TST, DUP AND, ; : LSL, DUP ADD, ;
```

B308

```
( a -- k12, absolute addr a, relative to PC in a k12 addr )
: _r7ffc DUP $7ff > IF _oor THEN ;
: _raddr12
    PC - DUP 0< IF $800 + _r7ffc $800 OR ELSE _r7ffc THEN ;
: RJMP _raddr12 $c000 OR ;
: RCALL _raddr12 $d000 OR ;
: RJMP, RJMP L, ; : RCALL, RCALL L, ;
```

B309

```
( a -- k7, absolute addr a, relative to PC in a k7 addr )
: _r3fc DUP $3f > IF _oor THEN ;
: _raddr7
    PC - DUP 0< IF $40 + _r3fc $40 OR ELSE _r3fc THEN ;
: _brbx ( a b op -- a ) OR SWAP _raddr7 <<3 OR ;
: BRBC $f400 _brbx ; : BRBS $f000 _brbx ; : BRCC 0 BRBC ;
: BRCS 0 BRBS ; : BREQ 1 BRBS ; : BRNE 1 BRBC ; : BRGE 4 BRBC ;
: BRHC 5 BRBC ; : BRHS 5 BRBS ; : BRID 7 BRBC ; : BRIE 7 BRBS ;
: BRLO BRCS ; : BRLT 4 BRBS ; : BRMI 2 BRBS ; : BRPL 2 BRBC ;
: BRSH BRCC ; : BRTC 6 BRBC ; : BRTS 6 BRBS ; : BRVC 3 BRBC ;
: BRVS 3 BRBS ;
```

B310

```
9 CONSTS $100c X  $0008 Y  $0000 Z
         $100d X+ $1009 Y+ $1001 Z+
         $100e -X $100a -Y $1002 -Z
: _ ( Rd XYZ op ) OR ( Rd op' ) SWAP _Rdp L, ;
: LD, $8000 _ ; : ST, SWAP $8200 _ ;
: LPM, $9004 _ ;
```

B311

```
\ LBL! L1 .. L1 ' RJMP LBL,
: LBL! ( -- ) PC TO ;
: LBL, ( opw pc -- ) 1- SWAP EXECUTE L, ;
: SKIP, PC 0 L, ;
: TO, ( opw pc )
  \ warning: pc is a PC offset, not a mem addr!
  << XORG + PC 1- HERE ( opw addr tgt hbkp )
  ROT 'HERE ! ( opw tgt hbkp )
  SWAP ROT EXECUTE HERE ! ( hbkp ) 'HERE ! ;
\ FLBL, L1 .. ' RJMP L1 TO,
: FLBL, LBL! 0 L, ;
: BEGIN, PC ; : AGAIN?, ( pc op ) SWAP LBL, ;
: AGAIN, ['] RJMP AGAIN?, ;
: IF, ['] BREQ SKIP, ; : THEN, TO, ;
```

B312

```
\ Constant common to all AVR models
38 CONSTS 0 R0 1 R1 2 R2 3 R3 4 R4 5 R5 6 R6 7 R7 8 R8 9 R9
  10 R10 11 R11 12 R12 13 R13 14 R14 15 R15 16 R16 17 R17
  18 R18 19 R19 20 R20 21 R21 22 R22 23 R23 24 R24 25 R25
  26 R26 27 R27 28 R28 29 R29 30 R30 31 R31
  26 XL 27 XH 28 YL 29 YH 30 ZL 31 ZH
```

3.4 ATmega328P definitions: 315

B315

```
( ATmega328P definitions ) 87 CONSTS
$c6 UDR0 $c4 UBRR0L $c5 UBRR0H $c2 UCSR0C $c1 UCSR0B $c0 UCSR0A
$bd TWAMR $bc TWCR $bb TWDR $ba TWAR $b9 TWSR $b8 TWBR $b6 ASSR
$b4 OCR2B $b3 OCR2A $b2 TCNT2 $b1 TCCR2B $b0 TCCR2A $8a OCR1BL
$8b OCR1BH $88 OCR1AL $89 OCR1AH $86 ICR1L $87 ICR1H $84 TCNT1L
$85 TCNT1H $82 TCCR1C $81 TCCR1B $80 TCCR1A $7f DIDR1 $7e DIDR0
$7c ADMUX $7b ADCSRB $7a ADCSRA $79 ADCH $78 ADCL $70 TIMSK2
$6f TIMSK1 $6e TIMSK0 $6c PCMSK1 $6d PCMSK2 $6b PCMSK0 $69 EICRA
$68 PCICR $66 OSCCAL $64 PRR $61 CLKPR $60 WDTCSR $3f SREG
$3d SPL $3e SPH $37 SPMCSR $35 MCUCR $34 MCUSR $33 SMCR $30 ACSR
$2e SPDR $2d SPSR $2c SPCR $2b GPIOR2 $2a GPIOR1 $28 OCR0B
$27 OCR0A $26 TCNT0 $25 TCCR0B $24 TCCR0A $23 GTCCR $22 EEARH
$21 EEARL $20 EEDR $1f EECR $1e GPIOR0 $1d EIMSK $1c EIFR
$1b PCIFR $17 TIFR2 $16 TIFR1 $15 TIFR0 $0b PORTD $0a DDRD
$09 PIND $08 PORTC $07 DDRC $06 PINC $05 PORTB $04 DDRB $03 PINB
```

3.5 SMS PS/2 controller: 320-342

B320

```
SMS PS/2 controller (doc/hw/z80/sms)

To assemble, load the AVR assembler with AVRA, then
"324 342 LOADR".

Receives keystrokes from PS/2 keyboard and send them to the
'164. On the PS/2 side, it works the same way as the controller
in the rc2014/ps2 recipe.  However, in this case, what we have
on the other side isn't a z80 bus, it's the one of the two
controller ports of the SMS through a DB9 connector.

The PS/2 related code is copied from rc2014/ps2 without much
change. The only differences are that it pushes its data to a
'164 instead of a '595 and that it synchronizes with the SMS
with a SR latch, so we don't need PCINT. We can also afford to
run at 1MHz instead of 8.                                  cont.
```

B321

```
Register Usage

GPIOR0 flags:
0 - when set, indicates that the DATA pin was high when we
    received a bit through INT0. When we receive a bit, we set
    flag T to indicate it.

R16: tmp stuff
R17: recv buffer. Whenever we receive a bit, we push it in
     there.
R18: recv step:
     - 0: idle
     - 1: receiving data
     - 2: awaiting parity bit
     - 3: awaiting stop bit                                cont.
```

B322

```
R19: Register used for parity computations and tmp value in
     some other places
R20: data being sent to the '164
Y: pointer to the memory location where the next scan code from
   ps/2 will be written.
Z: pointer to the next scan code to push to the 595
```

B324

```
18 CONSTS $0060 SRAM_START $015f RAMEND $3d SPL $3e SPH
          $11 GPIOR0 $35 MCUCR $33 TCCR0B $3b GIMSK
          $38 TIFR $32 TCNT0 $16 PINB $17 DDRB $18 PORTB
          2 CLK 1 DATA 3 CP 0 LQ 4 LR
$100 100 - VALUE TIMER_INITVAL
\ We need a lot of labels in this program...
5 VALUES L4 L5 L6 L7 L8
```

B325

```
FLBL, L1 \ main
FLBL, L2 \ hdlINT0
\ Read DATA and set GPIOR0/0 if high. Then, set flag T.
\ no SREG fiddling because no SREG-modifying instruction
' RJMP L2 TO, \ hdlINT0
PINB DATA SBIC,
GPIOR0 0 SBI,
SET,
RETI,
```

B326

```
' RJMP L1 TO, \ main
R16 RAMEND <<8 >>8 LDI, SPL R16 OUT,
R16 RAMEND >>8 LDI, SPH R16 OUT,
R18 CLR, GPIOR0 R18 OUT, \ init variables
R16 $02 ( ISC01 ) LDI, MCUCR R16 OUT, \ INT0, falling edge
R16 $40 ( INT0 ) LDI, GIMSK R16 OUT, \ Enable INT0
YH CLR, YL SRAM_START LDI, \ Setup buffer
ZH CLR, ZL SRAM_START LDI,
\ Setup timer. We use the timer to clear up "processbit"
\ registers after 100us without a clock. This allows us to start
\ the next frame in a fresh state. at 1MHZ, no prescaling is
\ necessary. Each TCNT0 tick is already 1us long.
R16 $01 ( CS00 ) LDI, \ no prescaler
TCCR0B R16 OUT,
DDRB CP SBI, PORTB LR CBI, DDRB LR SBI, SEI,
```

B327

```
LBL! L1 \ loop
FLBL, L2 \ BRTS processbit. flag T set? we have a bit to process
YL ZL CP, \ if YL == ZL, buf is empty
FLBL, L3 \ BRNE sendTo164. YL != ZL? buf has data
\ nothing to do. Before looping, let's check if our
\ communication timer overflowed.
R16 TIFR IN,
R16 1 ( TOV0 ) SBRC,
FLBL, L4 \ RJMP processbitReset, timer0 overflow? reset
\ Nothing to do for real.
' RJMP L1 LBL, \ loop
```

B328

```
\ Process the data bit received in INT0 handler.
' BRTS L2 TO, \ processbit
R19 GPIOR0 IN, \ backup GPIOR0 before we reset T
R19 $1 ANDI, \ only keep the first flag
GPIOR0 0 CBI,
CLT, \ ready to receive another bit
\ We've received a bit. reset timer
FLBL, L2 \ RCALL resetTimer
\ Which step are we at?
R18 TST, FLBL, L5 \ BREQ processbits0
R18 1 CPI, FLBL, L6 \ BREQ processbits1
R18 2 CPI, FLBL, L7 \ BREQ processbits2
```

B329

```
\ step 3: stop bit
R18 CLR, \ happens in all cases
\ DATA has to be set
R19 TST, \ was DATA set?
' BREQ L1 LBL, \ loop, not set? error, don't push to buf
\ push r17 to the buffer
Y+ R17 ST,
FLBL, L8 \ RCALL checkBoundsY
' RJMP L1 LBL, \ loop
```

B330

```
' BREQ L5 TO, \ processbits0
\ step 0 - start bit
\ DATA has to be cleared
R19 TST, \ was DATA set?
' BRNE L1 LBL, \ loop. set? error. no need to do anything. keep
               \ r18 as-is.
\ DATA is cleared. prepare r17 and r18 for step 1
R18 INC,
R17 $80 LDI,
' RJMP L1 LBL, \ loop
```

B331

```
' BREQ L6 TO, \ processbits1
\ step 1 - receive bit
\ We're about to rotate the carry flag into r17. Let's set it
\ first depending on whether DATA is set.
CLC,
R19 0 SBRC, \ skip if DATA is cleared
SEC,
\ Carry flag is set
R17 ROR,
\ Good. now, are we finished rotating? If carry flag is set,
\ it means that we've rotated in 8 bits.
' BRCC L1 LBL, \ loop
\ We're finished, go to step 2
R18 INC,
' RJMP L1 LBL, \ loop
```

B332

```
' BREQ L7 TO, \ processbits2
\ step 2 - parity bit
R1 R19 MOV,
R19 R17 MOV,
FLBL, L5 \ RCALL checkParity
R1 R16 CP,
FLBL, L6 \ BRNE processBitError, r1 != r16? wrong parity
R18 INC,
' RJMP L1 LBL, \ loop
```

B333

```
' BRNE L6 TO, \ processBitError
R18 CLR,
R19 $fe LDI,
FLBL, L6 \ RCALL sendToPS2
' RJMP L1 LBL, \ loop

' RJMP L4 TO, \ processbitReset
R18 CLR,
FLBL, L4 \ RCALL resetTimer
' RJMP L1 LBL, \ loop
```

B334

```
' BRNE L3 TO, \ sendTo164
\ Send the value of r20 to the '164
PINB LQ SBIS, \ LQ is set? we can send the next byte
' RJMP L1 LBL, \ loop, even if we have something in the
               \ buffer, we can't: the SMS hasn't read our
               \ previous buffer yet.
\ We disable any interrupt handling during this routine.
\ Whatever it is, it has no meaning to us at this point in time
\ and processing it might mess things up.
CLI,
DDRB DATA SBI,
R20 Z+ LD,
FLBL, L3 \ RCALL checkBoundsZ
R16 R8 LDI,
```

B335

```
BEGIN,
    PORTB DATA CBI,
    R20 7 SBRC, \ if leftmost bit isn't cleared, set DATA high
    PORTB DATA SBI,
    \ toggle CP
    PORTB CP CBI, R20 LSL, PORTB CP SBI,
    R16 DEC,
' BRNE AGAIN?, \ not zero yet? loop
\ release PS/2
DDRB DATA CBI,
SEI,
\ Reset the latch to indicate that the next number is ready
PORTB LR SBI,
PORTB LR CBI,
' RJMP L1 LBL, \ loop
```

B336

```
' RCALL L2 TO, ' RCALL L4 TO, LBL! L2 \ resetTimer
R16 TIMER_INITVAL LDI,
TCNT0 R16 OUT,
R16 $02 ( TOV0 ) LDI,
TIFR R16 OUT,
RET,
```

B337

```
' RCALL L6 TO, \ sendToPS2
\ Send the value of r19 to the PS/2 keyboard
CLI,
\ First, indicate our request to send by holding both Clock low
\ for 100us, then pull Data low lines low for 100us.
PORTB CLK CBI,
DDRB CLK SBI,
' RCALL L2 LBL, \ resetTimer
\ Wait until the timer overflows
BEGIN, R16 TIFR IN, R16 1 ( TOV0 ) SBRS, AGAIN,
\ Good, 100us passed.
\ Pull Data low, that's our start bit.
PORTB DATA CBI,
DDRB DATA SBI,
```

B338

```
\ Now, let's release the clock. At the next raising edge, we'll
\ be expected to have set up our first bit (LSB). We set up
\ when CLK is low.
DDRB CLK CBI, \ Should be starting high now.
R16 8 LDI, \ We will do the next loop 8 times
R1 R19 MOV, \ Let's remember initial r19 for parity
BEGIN,
    BEGIN, PINB CLK SBIC, AGAIN, \ Wait for CLK to go low
    PORTB DATA CBI, \ set up DATA
    R19 0 SBRC, \ skip if LSB is clear
    PORTB DATA SBI,
    R19 LSR,
 \ Wait for CLK to go high
    BEGIN, PINB CLK SBIS, AGAIN,
    16 DEC,
' BRNE AGAIN?, \ not zero? loop
```

B339

```
\ Data was sent, CLK is high. Let's send parity
R19 R1 MOV, \ recall saved value
FLBL, L6 \ RCALL checkParity
BEGIN, PINB CLK SBIC, AGAIN, \ Wait for CLK to go low
\ set parity bit
PORTB DATA CBI,
R16 0 SBRC, \ parity bit in r16
PORTB DATA SBI,
BEGIN, PINB CLK SBIS, AGAIN, \ Wait for CLK to go high
BEGIN, PINB CLK SBIC, AGAIN, \ Wait for CLK to go low
\ We can now release the DATA line
DDRB DATA CBI,
\ Wait for DATA to go low, that's our ACK
BEGIN, PINB DATA SBIC, AGAIN,
BEGIN, PINB CLK SBIC, AGAIN, \ Wait for CLK to go low
```

B340

```
\ We're finished! Enable INT0, reset timer, everything back to
\ normal!
' RCALL L2 LBL, \ resetTimer
CLT, \ also, make sure T isn't mistakely set.
SEI,
RET,
```

B341

```
' RCALL L8 TO, \ checkBoundsY
\ Check that Y is within bounds, reset to SRAM_START if not.
YL TST,
IF, RET, ( not zero, nothing to do ) THEN,
\ YL is zero. Reset Z
YH CLR, YL SRAM_START <<8 >>8 LDI,
RET,
' RCALL L3 TO, \ checkBoundsZ
\ Check that Z is within bounds, reset to SRAM_START if not.
ZL TST,
IF, RET, ( not zero, nothing to do ) THEN,
\ ZL is zero. Reset Z
ZH CLR, ZL SRAM_START <<8 >>8 LDI,
RET,
```

B342

```
' RCALL L5 TO, ' RCALL L6 TO, \ checkParity
\ Counts the number of 1s in r19 and set r16 to 1 if there's an
\ even number of 1s, 0 if they're odd.
R16 1 LDI,
BEGIN,
    R19 LSR,
    ' BRCC SKIP, R16 INC, ( carry set? we had a 1 ) TO,
    R19 TST, \ is r19 zero yet?
' BRNE AGAIN?, \ no? loop
R16 $1 ANDI,
RET,
```

3.6 Arduino blinker: 345

B345

```
\ A simple LED blinker on the Arduino Uno
\ To test the assembler mechanism. Requires ATMEGA328P.
DDRB 5 SBI, PORTB 5 CBI,
R16 $05 LDI, \ 1024 prescaler, CS00+CS02
TCCR0B R16 OUT,
R1 CLR, \ initialize overflow counter
BEGIN,
  R16 TIFR0 IN,
  R16 0 ( TOV0 ) SBRS, DUP AGAIN, \ no overflow
  R16 $01 LDI, TIFR0 R16 OUT,
  R1 INC,
  PORTB 5 CBI,
  R1 7 SBRS, PORTB 5 SBI, \ LED is on
AGAIN,
```

3.7 Arduino SPI spitter: 350-351

B350

```
\ Arduino SPI Spitter. See doc/hw/avr/spispit
103 VALUE BAUD_PRESCALE \ 9600 bauds at 16 MHz
R16 $80 LDI, R17 $04 LDI, CLKPR R16 STS, CLKPR R17 STS, \ x16
R16 BAUD_PRESCALE >>8 LDI, UBRR0H R16 STS,
R16 BAUD_PRESCALE <<8 >>8 LDI, UBRR0L R16 STS,
R16 $08 LDI, UCSR0B R16 STS, \ TXEN0
R16 CLR, PORTB R16 OUT,
R16 $2c LDI, DDRB R16 OUT, \ MOSI+SCK+SS/PB5+PB3+PB2
R16 $53 LDI, SPCR R16 OUT, \ SPE+MSTR+f_osc/128
ZH 0 LDI, ZL $ff LDI,
R1 Z+ LPM, \ number of 0x100 bytes blocks
```

B351

```
BEGIN, \ main loop
  R16 Z+ LPM, SPDR R16 OUT,
  BEGIN, R16 SPSR IN, R16 7 ( SPIF ) SBRS, AGAIN,
  BEGIN, R16 UCSR0A LDS, R16 5 ( UDRE0 ) SBRS, AGAIN,
  R16 SPDR IN, UDR0 R16 STS,
  ZL TST, ' BRNE SKIP, R1 DEC, TO,
  R1 TST, ' BRNE AGAIN?, \ end main
R16 $00 LDI, UCSR0B R16 STS, \ Disable UART
BEGIN, AGAIN, \ end program
```

4 8086

4.1 Architecture index: 300

B300

```
8086 MASTER INDEX

301 8086 boot code              306 8086 HAL
311 8086 assembler              320 8086 drivers
```

4.2 8086 boot code: 301-309

B301

```
\ 8086 macros
: 8086A 5 LOAD ( wordtbl ) 311 318 LOADR 7 LOAD ( Flow ) ;
: 8086C 302 309 LOADR ;
```

B302

```
\ 8086 boot code. PS=SP, RS=BP, IP=DX, TOS=BX
FJR JRi, TO L1 ( main ) 4 OALLOT ( 3=boot driveno )
LSET lblboot 2 ALLOT0 LSET lblmain 2 ALLOT0
L1 FMARK ( main ) DX POPx, ( boot drive no ) $03 DL MOVmr,
  SP PS_ADDR MOVxI, BP RS_ADDR MOVxI,
  DI $04 ( BOOT ) MOVxm, DI JMPr,
LSET lblval AL SYSVARS $18 ( TO? ) + MOVrm, AL AL ORrr, IFZ,
  DI POPx, BX PUSHx, BX [DI] x[] MOV[], ELSE,
  AL AL XORrr, SYSVARS $18 + AL MOVmr, DI POPx,
  [DI] BX []x MOV[], BX POPx, THEN, \ to next
LSET lblnext DI DX MOVxx, ( <-- IP ) DX INCx, DX INCx,
  DI [DI] x[] MOV[], DI JMPr,
LSET lblcell AX POPx, BX PUSHx, BX AX MOVxx, lblnext BR JRi,
LSET lblxt BP INCx, BP INCx, [BP] 0 DX []+x MOV[], ( pushRS )
  DX POPx, lblnext BR JRi,
```

B303

```
LSET lbldoes DI POPx, BX PUSHx, BX DI MOVxx,  BX INCx, BX INCx,
  DI [DI] x[] MOV[], DI JMPr,
CODE EXIT DX [BP] 0 x[]+ MOV[], BP DECx, BP DECx, ;CODE
CODE []= ( a1 a2 u -- f ) CX BX MOVxx, SI POPx, DI POPx,
  CLD, REPZ, CMPSB, BX 0 MOVxI, IFZ, BX INCx, THEN, ;CODE
CODE [C]? ( c a u -- i ) CX BX MOVxx, DI POPx, AX POPx,
  CLD, REPNZ, SCASB, IFNZ, CX BX MOVxx, THEN,
  BX CX SUBxx, BX DECx, ;CODE
CODE QUIT LSET L1 ( used in ABORT )
  BP RS_ADDR MOVxI, DI $06 ( main ) MOVxm, DI JMPr,
CODE ABORT SP PS_ADDR MOVxI, L1 BR JRi,
CODE BYE HLT, BEGIN, BR JRi,
```

B304

```
CODE FIND ( sa sl -- w? f ) CX BX MOVxx, SI POPx,
  DI SYSVARS $2 ( CURRENT ) + MOVxm,
  BEGIN, ( loop )
    AL [DI] -1 r[]+ MOV[], $7f ANDALi, ( strlen )
    CL AL CMPrr, IFZ, ( same len )
      SI PUSHx, DI PUSHx, CX PUSHx, ( --> )
        3 ADDALi, ( header ) AH AH XORrr, DI AX SUBxx,
        CLD, REPZ, CMPSB,
      CX POPx, DI POPx, SI POPx, ( <-- )
      IFZ, DI PUSHx, BX 1 MOVxI, ;CODE THEN,
    THEN,
    DI [x] 3 SUB[]i, DI [DI] x[] MOV[], ( prev ) DI DI ORxx,
  BR JRNZi, ( loop ) BX BX XORxx, ;CODE
```

B305

```
CODE * AX POPx, DX PUSHx, ( protect from MUL ) BX MULx, DX POPx,
  BX AX MOVxx, ;CODE
CODE /MOD AX POPx, DX PUSHx, ( protect )
  DX DX XORxx, BX DIVx,
  BX DX MOVxx, DX POPx, ( unprotect )
  BX PUSHx, ( modulo ) BX AX MOVxx, ( division ) ;CODE
CODE RCNT
  BX PUSHx, BX BP MOVxx, AX RS_ADDR MOVxI, BX AX SUBxx, ;CODE
CODE SCNT
  AX PS_ADDR MOVxI, AX SP SUBxx, BX PUSHx, BX AX MOVxx, ;CODE
CODE TICKS ( n=100us ) BX PUSHx,
    SI DX MOVxx, ( protect IP )
    AX POPx, BX 100 MOVxI, BX MULx,
    CX DX MOVxx, ( high ) DX AX MOVxx, ( low )
    AX $8600 MOVxI, ( 86h, WAIT ) $15 INT,
    DX SI MOVxx, ( restore IP ) BX POPx, ;CODE
```

B306

```
CODE (n)
  BX PUSHx, DI DX MOVxx, BX [DI] x[] MOV[],
  DX INCx, DX INCx, ;CODE
CODE (b)
  BX PUSHx, DI DX MOVxx, BH BH XORrr, BL [DI] r[] MOV[],
  DX INCx, ;CODE
CODE (br) LSET L1 ( used in ?br )
  DI DX MOVxx, AL [DI] r[] MOV[], AH AH XORrr, CBW,
  DX AX ADDxx, ;CODE
CODE (?br)
  BX BX ORxx, BX POPx, L1 BR JRZi, DX INCx, ;CODE
CODE (next)
  [BP] 0 [w]+ DEC[], L1 BR JRNZi,
  BP DECx, BP DECx, DX INCx, ;CODE
```

B307

```
CODE + AX POPx, BX AX ADDxx, ;CODE
CODE - AX POPx, AX BX SUBxx, BX AX MOVxx, ;CODE
CODE < AX POPx, CX CX XORxx, AX BX CMPxx, IFC, CX INCx, THEN,
  BX CX MOVxx, ;CODE
CODE 1+ BX INCx, ;CODE
CODE 1- BX DECx, ;CODE
CODE AND AX POPx, BX AX ANDxx, ;CODE
CODE OR AX POPx, BX AX ORxx, ;CODE
CODE XOR AX POPx, BX AX XORxx, ;CODE
CODE NOT BX BX ORxx, BX 0 MOVxI, IFZ, BX INCx, THEN, ;CODE
CODE >> BX SHRx1, ;CODE
CODE << BX SHLx1, ;CODE
CODE >>8 BL BH MOVrr, BH BH XORrr, ;CODE
CODE <<8 BH BL MOVrr, BL BL XORrr, ;CODE
```

B308

```
CODE R@ BX PUSHx, BX [BP] 0 x[]+ MOV[], ;CODE
CODE R~ BP DECx, BP DECx, ;CODE
CODE R> BX PUSHx, BX [BP] 0 x[]+ MOV[], BP DECx, BP DECx, ;CODE
CODE >R BP INCx, BP INCx, [BP] 0 BX []+x MOV[], BX POPx, ;CODE
CODE ROT ( a b c -- b c a ) ( BX=c ) CX POPx, ( b ) AX POPx, \ a
  CX PUSHx, BX PUSHx, BX AX MOVxx, ;CODE
CODE ROT> ( a b c -- c a b ) CX POPx, AX POPx,
  BX PUSHx, AX PUSHx, BX CX MOVxx, ;CODE
CODE DUP LSET L1 BX PUSHx, ;CODE
CODE ?DUP AX BX MOVxx, AX AX ORxx, L1 BR JRNZi, ;CODE
CODE OVER ( a b -- a b a )
  AX POPx, AX PUSHx, BX PUSHx, BX AX MOVxx, ;CODE
CODE SWAP AX BX MOVxx, BX POPx, AX PUSHx, ;CODE
CODE DROP BX POPx, ;CODE
CODE EXECUTE AX BX MOVxx, BX POPx, AX JMPr,
```

B309

```
CODE C@ DI BX MOVxx, BH BH XORrr, BL [DI] r[] MOV[], ;CODE
CODE @ DI BX MOVxx, BX [DI] x[] MOV[], ;CODE
CODE C! DI BX MOVxx, CX POPx, [DI] CL []r MOV[], BX POPx, ;CODE
CODE ! DI BX MOVxx, CX POPx, [DI] CX []x MOV[], BX POPx, ;CODE
CODE JMPi! ( pc a -- len ) DI BX MOVxx, AX POPx,
  CL $e9 MOVri, LSET L1 [DI] CL []r MOV[],
  CX SYSVARS $4 ( HOME ) + MOVxm, AX CX SUBxx, AX DECx, AX DECx,
  AX DECx, [DI] 1 AX []+x MOV[], BX 3 MOVxI, ;CODE
CODE CALLi! ( pc a -- len ) DI BX MOVxx, AX POPx,
  CL $e8 MOVri, L1 BR JRi,
```

4.3 8086 assembler: 311-318

B311

```
\ 8086 assembler. See doc/asm
28 CONSTS 0 AL 1 CL 2 DL 3 BL
          4 AH 5 CH 6 DH 7 BH
          0 AX 1 CX 2 DX 3 BX
          4 SP 5 BP 6 SI 7 DI
          0 ES 1 CS 2 SS 3 DS
          0 [BX+SI] 1 [BX+DI] 2 [BP+SI] 3 [BP+DI]
          4 [SI] 5 [DI] 6 [BP] 7 [BX]
: <<3 << << << ;
```

B312

```
: OP1 DOER C, DOES> C@ C, ;
$c3 OP1 RET,         $fa OP1 CLI,        $fb OP1 STI,
$f4 OP1 HLT,         $fc OP1 CLD,        $fd OP1 STD,
$90 OP1 NOP,         $98 OP1 CBW,
$f3 OP1 REPZ,        $f2 OP1 REPNZ,      $ac OP1 LODSB,
$ad OP1 LODSW,       $a6 OP1 CMPSB,      $a7 OP1 CMPSW,
$a4 OP1 MOVSB,       $a5 OP1 MOVSW,      $ae OP1 SCASB,
$af OP1 SCASW,       $aa OP1 STOSB,      $ab OP1 STOSW,

: OP1r DOER C, DOES> C@ + C, ;
$40 OP1r INCx,       $48 OP1r DECx,
$58 OP1r POPx,       $50 OP1r PUSHx,
```

B313

```
: OPr0 ( reg op ) DOER C, C, DOES>
    C@+ C, C@ <<3 OR $c0 OR C, ;
0 $d0 OPr0 ROLr1,   0 $d1 OPr0 ROLx1,  4 $f6 OPr0 MULr,
1 $d0 OPr0 RORr1,   1 $d1 OPr0 RORx1,  4 $f7 OPr0 MULx,
4 $d0 OPr0 SHLr1,   4 $d1 OPr0 SHLx1,  6 $f6 OPr0 DIVr,
5 $d0 OPr0 SHRr1,   5 $d1 OPr0 SHRx1,  6 $f7 OPr0 DIVx,
0 $d2 OPr0 ROLrCL,  0 $d3 OPr0 ROLxCL, 1 $fe OPr0 DECr,
1 $d2 OPr0 RORrCL,  1 $d3 OPr0 RORxCL, 0 $fe OPr0 INCr,
4 $d2 OPr0 SHLrCL,  4 $d3 OPr0 SHLxCL,
5 $d2 OPr0 SHRrCL,  5 $d3 OPr0 SHRxCL,
```

B314

```
: OPrr DOER C, DOES> C@ C, <<3 OR $c0 OR C, ;
$31 OPrr XORxx,     $30 OPrr XORrr,
$88 OPrr MOVrr,     $89 OPrr MOVxx,    $28 OPrr SUBrr,
$29 OPrr SUBxx,     $08 OPrr ORrr,     $09 OPrr ORxx,
$38 OPrr CMPrr,     $39 OPrr CMPxx,    $00 OPrr ADDrr,
$01 OPrr ADDxx,     $12 OPrr ADCrr,    $13 OPrr ADCxx,
$20 OPrr ANDrr,     $21 OPrr ANDxx,
```

B315

```
4 WORDTBL mods 'W NOOP 'W C, 'W L, 'W NOOP
: modrm ( disp? modrm -- )
  DUP C, DUP $c7 AND 6 = IF DROP $80 THEN 64 / mods SWAP WEXEC ;
: OP[] ( opbase+modrmbase ) DOER , DOES>
  @ L|M ( disp? modrm opoff modrmbase op ) ROT + C, + modrm ;
( -- disp? modrm opoff )
: [b] ( r/m ) 0 ; : [w] ( r/m ) 1 ;
: [m] ( a ) 6 0 ; : [M] [m] 1+ ;
: [r] ( r ) $c0 OR 0 ; : [x] [r] 1+ ;
: [b]+ ( r/m disp8 ) SWAP $40 OR 0 ; : [w]+ [b]+ 1+ ;
: r[] ( r r/m ) SWAP <<3 OR 2 ; : x[] r[] 1+ ;
: []r ( r/m r ) <<3 OR 0 ; : []x []r 1+ ;
: r[]+ ( r r/m disp8 )
    ROT <<3 ROT OR $40 OR 2 ; : x[]+ r[]+ 1+ ;
: []+r ( r/m disp8 r ) <<3 ROT OR $40 OR 0 ; : []+x []+r 1+ ;
```

B316

```
$fe00 OP[] INC[],        $fe08 OP[] DEC[],
$fe30 OP[] PUSH[],       $8e00 OP[] POP[],
$8800 OP[] MOV[],        $3800 OP[] CMP[],

: OP[]i ( opbase+modrmbase ) DOER , DOES> SWAP >R ( i )
  SWAP ( opoff ) DUP IF R@ >>8 NOT IF 2 + THEN THEN >R
  @ L|M ( disp? modrm modrmbase op )
  R@ + C, + modrm R> 1 = IF R> L, ELSE R> C, THEN ;
$8000 OP[]i ADD[]i,      $8010 OP[]i ADC[]i,
$8038 OP[]i CMP[]i,      $8028 OP[]i SUB[]i,

: OPI DOER C, DOES> C@ C, L, ;
$05 OPI ADDAXI,     $15 OPI ADCALI,    $25 OPI ANDAXI,
$2d OPI SUBAXI,     $a1 OPI MOVAXm,    $a3 OPI MOVmAX,
```

B317

```
: OPi DOER C, DOES> C@ C, C, ;
$04 OPi ADDALi,     $14 OPi ADCALi,    $24 OPi ANDALi,
$2c OPi SUBALi,     $cd OPi INT,
$eb OPi JRi,        $74 OPi JRZi,
$75 OPi JRNZi,      $72 OPi JRCi,      $73 OPi JRNCi,
$a0 OPi MOVALm,     $a2 OPi MOVmAL,
: MOVri, SWAP $b0 OR C, C, ; : MOVxI, SWAP $b8 OR C, L, ;
: MOVsx, $8e C, SWAP <<3 OR $c0 OR C, ;
: MOVrm, $8a C, SWAP <<3 $6 OR C, L, ;
: MOVxm, $8b C, SWAP <<3 $6 OR C, L, ;
: MOVmr, $88 C, <<3 $6 OR C, L, ;
: MOVmx, $89 C, <<3 $6 OR C, L, ;
: PUSHs, <<3 $06 OR C, ; : POPs, <<3 $07 OR C, ;
: JMPr, $ff C, 7 AND $e0 OR C, ;
: JMPf, ( seg off ) $ea C, L, L, ;
```

B318

```
: JMPi, $e9 C, ( jmp near ) PC - 2 - L, ;
: CALLi, $e8 C, ( jmp near ) PC - 2 - L, ;
: i>, BX PUSHx, BX SWAP MOVxI, ;
: JMP(i), MOVAXm, AX JMPr, ;
: (i)>, BX PUSHx, BX SWAP MOVxm, ;
```

4.4 8086 drivers: 320-324

B320

```
( PC/AT drivers. Load range: 320-326 )
CODE (key?)
  BX PUSHx, BX BX XORxx, AH 1 MOVri, $16 INT, IFNZ,
    AH AH XORrr, $16 INT, AH AH XORrr, BX INCx, AX PUSHx, THEN,
;CODE
```

B321

```
CODE 13H08H ( driveno -- cx dx )
  DX PUSHx, ( protect ) DX BX MOVxx, AX $800 MOVxI,
  ES PUSHs, DI DI XORxx, ES DI MOVsx,
  $13 INT, BX DX MOVxx, ES POPs, DX POPx, ( unprotect )
  CX PUSHx, ;CODE
CODE 13H ( ax bx cx dx -- ax bx cx dx )
  SI BX MOVxx, ( DX ) CX POPx, BX POPx, AX POPx,
  DX PUSHx, ( protect ) DX SI MOVxx, DI DI XORxx,
  $13 INT, SI DX MOVxx, DX POPx, ( unprotect )
  AX PUSHx, BX PUSHx, CX PUSHx, BX SI MOVxx, ;CODE
```

B322

```
DRV_ADDR CONSTANT FDSPT
DRV_ADDR 1+ CONSTANT FDHEADS
:~ ( AX BX sec )
  ( AH=read sectors, AL=1 sector, BX=dest,
    CH=trackno CL=secno DH=head DL=drive )
  FDSPT C@ /MOD ( AX BX sec trk )
  FDHEADS C@ /MOD ( AX BX sec head trk )
  <<8 ROT OR 1+ ( AX BX head CX )
  SWAP <<8 $03 C@ ( boot drive ) OR ( AX BX CX DX )
  13H 2DROP 2DROP ;
```

B323

```
\ Sectors are 512b, so blk numbers are all x2. We add 16 to
\ this because blkfs starts at sector 16.
: FD@ ( blkno blk( -- )
  SWAP << ( 2* ) 16 + 2DUP ( a b a b )
  $0201 ROT> ( a b c a b ) ~ ( a b )
  1+ SWAP $200 + SWAP $0201 ROT> ( c a b ) ~ ;
: FD! ( blkno blk( -- )
  SWAP << ( 2* ) 16 + 2DUP ( a b a b )
  $0301 ROT> ( a b c a b ) ~ ( a b )
  1+ SWAP $200 + SWAP $0301 ROT> ( c a b ) ~ ;
: FD$
\ get number of sectors per track with command 08H.
  $03 ( boot drive ) C@ 13H08H
  >>8 1+ FDHEADS C!
  $3f AND FDSPT C! ;
```

B324

```
2 CONSTS 80 COLS 25 LINES
CODE CURSOR! ( new old ) AX POPx, ( new ) DX PUSHx, ( protect )
  BX 80 MOVxI, DX DX XORxx, BX DIVx, ( col in DL, row in AL )
  DH AL MOVrr, AH 2 MOVri,
  $10 INT, DX POPx, ( unprotect ) BX POPx, ;CODE
CODE _ ( c -- ) \ char out
  AL BL MOVrr, BX POPx, AH $0e MOVri, $10 INT, ;CODE
: CELL! ( c pos -- ) 0 CURSOR! _ ;
: NEWLN ( old -- new ) 1+ DUP LINES = IF 1- CR _ LF _ THEN ;
```

5 6809

5.1 Architecture index: 300

B300

```
6809 MASTER INDEX

301 6809 macros                302 6809 boot code
310 6809 assembler
320 TRS-80 Color Computer 2
325 6809 disassembler          340 6809 emulator
360 Virgil's workspace
```

5.2 6809 macros: 301

B301

```
( 6809 declarations )
: 6809A 310 318 LOADR 7 LOAD ( flow ) ;
: 6809C 302 308 LOADR ;
: 6809D 325 336 LOADR ; : 6809E 340 354 LOADR ;
: COCO2 320 LOAD 322 324 LOADR ;
: DGN32 321 LOAD 322 324 LOADR ;
```

5.3 6809 boot code: 302-305

B302

```
\ 6809 Boot code. IP=Y, PS=S, RS=U
PS_ADDR # LDS, RS_ADDR # LDU, 0 () JMP, PC 2 - TO lblboot
LSET lblval SYSVARS $18 ( TO? ) + () TST, FJR BNE, TO L1
  ( val rd ) [S+0] LDD, S+0 STD, \ to next
LSET lblcell LSET lblnext Y++ LDX, X+0 JMP,
L1 FMARK ( val wr ) SYSVARS $18 + () CLR, 2 S+N LDD,
  [S++] STD, S++ TST, lblnext BR BRA,
LSET lblxt U++ STY, ( IP->RS ) PULS, Y lblnext BR JRi,
LSET lbldoes [S+0] LDX, 2 # LDD, S+0 ADDD, S+0 STD, X+0 JMP,
CODE QUIT LSET L1 ( for ABORT )
  RS_ADDR # LDU, 0 () JMP, PC 2 - TO lblmain
CODE ABORT PS_ADDR # LDS, L1 BR JRi,
CODE BYE BEGIN, BR JRi,
CODE EXIT --U LDY, ;CODE
CODE EXECUTE PULS, X X+0 JMP,
```

B303

```
CODE SCNT PS_ADDR # LDD, 0 <> STS, 0 <> SUBD, PSHS, D ;CODE
CODE RCNT
  RS_ADDR # LDD, 0 <> STD, U D TFR, 0 <> SUBD, PSHS, D ;CODE
CODE @ [S+0] LDD, S+0 STD, ;CODE
CODE C@ [S+0] LDB, CLRA, S+0 STD, ;CODE
CODE ! PULS, X PULS, D X+0 STD, ;CODE
CODE C! PULS, X PULS, D X+0 STB, ;CODE
LSET L1 ( PUSH Z ) CCR B TFR, LSRB, LSRB,
  1 # ANDB, CLRA, S+0 STD, ;CODE
CODE = PULS, D S+0 CMPD, L1 BR BRA, ( PUSH Z )
CODE NOT S+0 LDB, 1 S+N ORB, L1 BR BRA, ( PUSH Z )
CODE <
  2 S+N LDD, S++ CMPD, CCR B TFR, 1 # ANDB, CLRA, S+0 STD, ;CODE
```

B304

```
CODE /MOD ( a b -- a/b a%b )
  16 # LDA, 0 <> STA, CLRA, CLRB, ( D=running rem ) BEGIN,
    1 # ORCC, 3 S+N ROL, ( a lsb ) 2 S+N ROL, ( a msb )
    ROLB, ROLA, S+0 SUBD,
    FJR BHS, ( if < ) S+0 ADDD, 3 S+N DEC, ( a lsb ) THEN,
  0 <> DEC, BR JRNZi,
  2 S+N LDX, 2 S+N STD, ( rem ) S+0 STX, ( quotient ) ;CODE
CODE * ( a b -- a*b )
  S+0 ( bm ) LDA, 3 S+N ( al ) LDB, MUL, S+0 ( bm ) STB,
  2 S+N ( am ) LDA, 1 S+N ( bl ) LDB, MUL,
    S+0 ( bm ) ADDB, S+0 STB,
  1 S+N ( al ) LDA, 3 S+N ( bl ) LDB, MUL,
  S++ ADDA, S+0 STD, ;CODE
```

B305

```
LSET L1 ( X=s1 Y=s2 B=cnt ) BEGIN,
  X+ LDA, Y+ CMPA, IFNZ, RTS, THEN, DECB, BR JRNZi, RTS,
CODE []= ( a1 a2 u -- f TODO: allow u>$ff )
  0 <> STY, PULS, DXY ( B=u, X=a2, Y=a1 ) L1 () JSR,
  IFZ, 1 # LDD, ELSE, CLRA, CLRB, THEN, PSHS, D 0 <> LDY, ;CODE
CODE FIND ( sa sl -- w? f )
  SYSVARS $02 + ( CURRENT ) () LDX,
  0 <> STY, PULS, D 2 <> STB, BEGIN,
    -X LDB, $7f # ANDB, --X TST, 2 <> CMPB, IFZ,
      3 <> STX, S+0 LDY, NEGB, X+B LEAX, NEGB, L1 () JSR,
      IFZ, ( match ) 0 <> LDY, 3 <> LDD, 3 # ADDD, S+0 STD,
        1 # LDD, PSHS, D ;CODE THEN,
      3 <> LDX, THEN, \ nomatch, X=prev
  X+0 LDX, BR JRNZi, \ not zero, loop
  ( end of dict ) 0 <> LDY, S+0 STX, ( X=0 ) ;CODE
```

5.4 6809 HAL: 306-310

B306

```
CODE AND PULS, D S+0 ANDA, 1 S+N ANDB, S+0 STD, ;CODE
CODE OR PULS, D S+0 ORA, 1 S+N ORB, S+0 STD, ;CODE
CODE XOR PULS, D S+0 EORA, 1 S+N EORB, S+0 STD, ;CODE
CODE + PULS, D S+0 ADDD, S+0 STD, ;CODE
CODE - 2 S+N LDD, S++ SUBD, S+0 STD, ;CODE
CODE 1+ 1 S+N INC, IFZ, S+0 INC, THEN, ;CODE
CODE 1- 1 S+N TST, IFZ, S+0 DEC, THEN, 1 S+N DEC, ;CODE
CODE << 1 S+N LSL, S+0 ROL, ;CODE
CODE >> S+0 LSR, 1 S+N ROR, ;CODE
CODE <<8 1 S+N LDA, S+0 STA, 1 S+N CLR, ;CODE
CODE >>8 S+0 LDA, 1 S+N STA, S+0 CLR, ;CODE
```

B307

```
CODE R@ -2 U+N LDD, PSHS, D ;CODE
CODE R~ --U TST, ;CODE
CODE R> --U LDD, PSHS, D ;CODE
CODE >R PULS, D U++ STD, ;CODE
CODE DROP 2 S+N LEAS, ;CODE
CODE DUP ( a -- a a ) S+0 LDD, PSHS, D ;CODE
CODE ?DUP ( a -- a? a ) S+0 LDD, IFNZ, PSHS, D THEN, ;CODE
CODE SWAP ( a b -- b a )
  S+0 LDD, 2 S+N LDX, S+0 STX, 2 S+N STD, ;CODE
CODE OVER ( a b -- a b a ) 2 S+N LDD, PSHS, D ;CODE
CODE ROT ( a b c -- b c a )
  4 S+N LDX, ( a ) 2 S+N LDD, ( b ) 4 S+N STD, S+0 LDD, ( c )
  2 S+N STD, S+0 STX, ;CODE
CODE ROT> ( a b c -- c a b )
  S+0 LDX, ( c ) 2 S+N LDD, ( b ) S+0 STD, 4 S+N LDD, ( a )
  2 S+N STD, 4 S+N STX, ;CODE
```

B308

```
CODE (b) Y+ LDB, CLRA, PSHS, D ;CODE
CODE (n) Y++ LDD, PSHS, D ;CODE
CODE (br) LSET L1 Y+0 LDA, Y+A LEAY, ;CODE
CODE (?br) S+ LDA, S+ ORA, L1 BR JRZi, Y+ TST, ;CODE
CODE (next) --U LDD, 1 # SUBD, IFNZ,
  U++ STD, L1 BR JRi, THEN, Y+ TST, ;CODE
```

B310

```
\ 6809 assembler. See doc/asm.txt.
'? BIGEND? [IF] 1 TO BIGEND? [THEN]
: <<3 << << << ; : <<4 <<3 << ;
\ For TFR/EXG
10 CONSTS 0 D 1 X 2 Y 3 U 4 S 5 PCR 8 A 9 B 10 CCR 11 DPR
\ Addressing modes. output: n3? n2? n1 nc opoff
: # ( n ) 1 0 ; \ Immediate
: <> ( n ) 1 $10 ; \ Direct
: () ( n ) L|M 2 $30 ; \ Extended
: [] ( n ) L|M $9f 3 $20 ; \ Extended Indirect
\ Offset Indexed. We auto-detect 0, 5-bit, 8-bit, 16-bit
: _0? ?DUP IF 1 ELSE $84 1 0 THEN ;
: _5? DUP $10 + $1f > IF 1 ELSE $1f AND 1 0 THEN ;
: _8? DUP $80 + $ff > IF 1 ELSE <<8 >>8 $88 2 0 THEN ;
: _16 L|M $89 3 ;
```

5.5 6809 assembler: 311-318

B311

```
: R+N DOER C, DOES> C@ ( roff ) >R
    _0? IF _5? IF _8? IF _16 THEN THEN THEN
    SWAP R> ( roff ) OR SWAP $20 ;
: R+K DOER C, DOES> C@ 1 $20 ;
: PCR+N ( n ) _8? IF _16 THEN SWAP $8c OR SWAP $20 ;
: [R+N] DOER C, DOES> C@ $10 OR ( roff ) >R
    _0? IF _8? IF _16 THEN THEN SWAP R> OR SWAP $20 ;
: [PCR+N] ( n ) _8? IF _16 THEN SWAP $9c OR SWAP $20 ;
0 R+N X+N   $20 R+N Y+N  $40 R+N U+N   $60 R+N S+N
: X+0 0 X+N ; : Y+0 0 Y+N ; : S+0 0 S+N ; : U+0 0 S+N ;
0 [R+N] [X+N] $20 [R+N] [Y+N]
$40 [R+N] [U+N] $60 [R+N] [S+N]
: [X+0] 0 [X+N] ; : [Y+0] 0 [Y+N] ;
: [S+0] 0 [S+N] ; : [U+0] 0 [U+N] ;
```

B312

```
$86 R+K X+A   $85 R+K X+B   $8b R+K X+D
$a6 R+K Y+A   $a5 R+K Y+B   $ab R+K Y+D
$c6 R+K U+A   $c5 R+K U+B   $cb R+K U+D
$e6 R+K S+A   $e5 R+K S+B   $eb R+K S+D
$96 R+K [X+A] $95 R+K [X+B] $9b R+K [X+D]
$b6 R+K [Y+A] $b5 R+K [Y+B] $bb R+K [Y+D]
$d6 R+K [U+A] $d5 R+K [U+B] $db R+K [U+D]
$f6 R+K [S+A] $f5 R+K [S+B] $fb R+K [S+D]
$80 R+K X+  $81 R+K X++  $82 R+K -X  $83 R+K --X
$a0 R+K Y+  $a1 R+K Y++  $a2 R+K -Y  $a3 R+K --Y
$c0 R+K U+  $c1 R+K U++  $c2 R+K -U  $c3 R+K --U
$e0 R+K S+  $e1 R+K S++  $e2 R+K -S  $e3 R+K --S
$91 R+K [X++] $93 R+K [--X] $b1 R+K [Y++] $b3 R+K [--Y]
$d1 R+K [U++] $d3 R+K [--U] $f1 R+K [S++] $f3 R+K [--S]
```

B313

```
: ,? DUP $ff > IF M, ELSE C, THEN ;
: ,N ( cnt ) >R BEGIN C, NEXT ;
: OPINH ( inherent ) DOER , DOES> @ ,? ;
( Targets A or B )
: OP1 DOER , DOES> @ ( n2? n1 nc opoff op ) + ,? ,N ;
( Targets D/X/Y/S/U. Same as OP1, but spit 2b immediate )
: OP2 DOER , DOES> @ OVER + ,? IF ,N ELSE DROP M, THEN ;
( Targets memory only. opoff scheme is different than OP1/2 )
: OPMT DOER , DOES> @
    SWAP $10 - ?DUP IF $50 + + THEN ,? ,N ;
( Targets 2 regs )
: OPRR ( src tgt -- ) DOER C, DOES> C@ C, SWAP <<4 OR C, ;
: OPBR ( op1 -- ) DOER C, DOES> ( off -- ) C@ C, C, ;
: OPLBR ( op? -- ) DOER , DOES> ( off -- ) @ ,? M, ;
```

B314

```
$89 OP1 ADCA,          $c9 OP1 ADCB,
$8b OP1 ADDA,          $cb OP1 ADDB,        $c3 OP2 ADDD,
$84 OP1 ANDA,          $c4 OP1 ANDB,        $1c OP1 ANDCC,
$48 OPINH ASLA,        $58 OPINH ASLB,      $08 OPMT ASL,
$47 OPINH ASRA,        $57 OPINH ASRB,      $07 OPMT ASR,
$4f OPINH CLRA,        $5f OPINH CLRB,      $0f OPMT CLR,
$81 OP1 CMPA,          $c1 OP1 CMPB,        $1083 OP2 CMPD,
$118c OP2 CMPS,        $1183 OP2 CMPU,      $8c OP2 CMPX,
$108c OP2 CMPY,
$43 OPINH COMA,        $53 OPINH COMB,      $03 OPMT COM,
$3c OP1 CWAI,          $19 OPINH DAA,
$4a OPINH DECA,        $5a OPINH DECB,      $0a OPMT DEC,
$88 OP1 EORA,          $c8 OP1 EORB,        $1e OPRR EXG,
$4c OPINH INCA,        $5c OPINH INCB,      $0c OPMT INC,
$0e OPMT JMP,          $8d OP1 JSR,
```

B315

```
$86 OP1 LDA,           $c6 OP1 LDB,         $cc OP2 LDD,
$10ce OP2 LDS,         $ce OP2 LDU,         $8e OP2 LDX,
$108e OP2 LDY,
$12 OP1 LEAS,          $13 OP1 LEAU,        $10 OP1 LEAX,
$11 OP1 LEAY,
$48 OPINH LSLA,        $58 OPINH LSLB,      $08 OPMT LSL,
$44 OPINH LSRA,        $54 OPINH LSRB,      $04 OPMT LSR,
$3d OPINH MUL,
$40 OPINH NEGA,        $50 OPINH NEGB,      $00 OPMT NEG,
$12 OPINH NOP,
$8a OP1 ORA,           $ca OP1 ORB,         $1a OP1 ORCC,
$49 OPINH ROLA,        $59 OPINH ROLB,      $09 OPMT ROL,
$46 OPINH RORA,        $56 OPINH RORB,      $06 OPMT ROR,
$3b OPINH RTI,         $39 OPINH RTS,
$82 OP1 SBCA,          $c2 OP1 SBCB,
$1d OPINH SEX,
```

B316

```
$87 OP1 STA,           $c7 OP1 STB,         $cd OP2 STD,
$10cf OP2 STS,         $cf OP2 STU,         $8f OP2 STX,
$108f OP2 STY,
$80 OP1 SUBA,          $c0 OP1 SUBB,        $83 OP2 SUBD,
$3f OPINH SWI,         $103f OPINH SWI2,    $113f OPINH SWI3,
$13 OPINH SYNC,        $1f OPRR TFR,
$4d OPINH TSTA,        $5d OPINH TSTB,      $0d OPMT TST,

$24 OPBR BCC,          $1024 OPLBR LBCC,    $25 OPBR BCS,
$1025 OPLBR LBCS,      $27 OPBR BEQ,        $1027 OPLBR LBEQ,
$2c OPBR BGE,          $102c OPLBR LBGE,    $2e OPBR BGT,
$102e OPLBR LBGT,      $22 OPBR BHI,        $1022 OPLBR LBHI,
$24 OPBR BHS,          $1024 OPLBR LBHS,    $2f OPBR BLE,
$102f OPLBR LBLE,      $25 OPBR BLO,        $1025 OPLBR LBLO,
$23 OPBR BLS,          $1023 OPLBR LBLS,    $2d OPBR BLT,
$102d OPLBR LBLT,      $2b OPBR BMI,        $102b OPLBR LBMI,
```

B317

```
$26 OPBR BNE,         $1026 OPLBR LBNE,  $2a OPBR BPL,
$102a OPLBR LBPL,     $20 OPBR BRA,      $16 OPLBR LBRA,
$21 OPBR BRN,         $1021 OPLBR LBRN,  $8d OPBR BSR,
$17 OPLBR LBSR,       $28 OPBR BVC,      $1028 OPLBR LBVC,
$29 OPBR BVS,         $1029 OPLBR LBVS,

: _ ( r c cref mask -- r c ) ROT> OVER = ( r mask c f )
    IF ROT> OR SWAP ELSE NIP THEN ;
: OPP DOER C, DOES> C@ C, 0 TOWORD IN> 1- C@ BEGIN ( r c )
    '$' $80 _ 'S' $40 _ 'U' $40 _ 'Y' $20 _ 'X' $10 _
    '%' $08 _ 'B' $04 _ 'A' $02 _ 'C' $01 _ 'D' $06 _
    '@' $ff _ DROP IN< DUP WS? UNTIL DROP C, ;
$34 OPP PSHS, $36 OPP PSHU, $35 OPP PULS, $37 OPP PULU,
```

B318

```
\ 6809 HAL, flow words. Also used in 6809A
: JMPi, () JMP, ; : CALLi, () JSR, ; : JMP(i), [] JMP, ;
ALIAS BRA, JRi,
ALIAS BEQ, JRZi, ALIAS BNE, JRNZi,
ALIAS BCS, JRCi, ALIAS BCC, JRNCi,
: i>, # LDD, $3406 M, ( pshs d ) ;
: (i)>, () LDD, $3406 M, ( pshs d ) ;
```

5.6 TRS-80 Color Computer 2: 320-324

B320

```
\ CoCo2 keyboard layout
PC ," @HPX08" CR C, ," AIQY19" 0 C,
   ," BJRZ2:" 0 C,  ," CKS_3;" 0 C,
   ," DLT_4," 0 C,  ," EMU" BS C, ," 5-" 0 C,
   ," FNV_6." 0 C,  ," GOW 7/" 0 C,
   ," @hpx0(" CR C, ," aiqy!)" 0 C,
   ," bjrz" '"' C, '*' C, 0 C, ," cks_#+" 0 C,
   ," dlt_$<" 0 C,  ," emu" BS C, ," %=" 0 C,
   ," fnv_&>" 0 C,  ," gow '?" 0 C,
```

B321

```
\ Dragon32 keyboard layout
PC ," 08@HPX" CR C, ," 19AIQY" 0 C,
   ," 2:BJRZ" 0 C,  ," 3;CKS_" 0 C,
   ," 4,DLT_" 0 C,  ," 5-EMU" BS C, 0 C,
   ," 6.FNV_" 0 C,  ," 7/GOW " 0 C,
   ," 0(@hpx" CR C, ," !)aiqy" 0 C,
   '"' C, '*' C, ," bjrz" 0 C, ," #+cks_" 0 C,
   ," $<dlt_" 0 C,  ," %=emu" BS C, 0 C,
   ," &>fnv_" 0 C,  ," '?gow " 0 C,
```

B322

```
\ Coco2 keyboard driver
LSET L1 ( PC ) # LDX, $fe # LDA, BEGIN, ( 8 times )
  $ff02 () STA, ( set col ) $ff00 () LDB, ( read row )
  ( ignore 8th row ) $80 # ORB, $7f # CMPA, IFZ,
    ( ignore shift row ) $40 # ORB, THEN,
  INCB, IFNZ, ( key pressed ) DECB, RTS, THEN,
  ( inc col ) 7 X+N LEAX, 1 # ORCC, ROLA, BR JRCi,
  ( no key ) CLRB, RTS,
```

B323

```
\ Coco2 keyboard driver
CODE (key?) ( -- c? f ) CLRA, CLRB, PSHS, D L1 () JSR,
  IFNZ, ( key! row mask in B col ptr in X )
    ( is shift pressed? ) $7f # LDA, $ff02 () STA,
    $ff00 () LDA, $40 # ANDA, IFZ, ( shift! )
      56 X+N LEAX, THEN,
    BEGIN, X+ LDA, LSRB, BR JRCi,
    ( A = our char ) 1 S+N STA, TSTA, IFNZ, ( valid key )
      1 # LDD, ( f ) PSHS, D ( wait for keyup )
      BEGIN, L1 () JSR, BR JRNZi, THEN,
  THEN, ;CODE
```

B324

```
\ Coco2 grid driver
32 CONSTANT COLS 16 CONSTANT LINES
: CELL! ( c pos -- )
  SWAP $20 - DUP $5f < IF
    DUP $20 < IF $60 + ELSE DUP $40 < IF $20 + ELSE $40 -
      THEN THEN ( pos glyph )
    SWAP $400 + C! ELSE 2DROP THEN ;
: CURSOR! ( new old -- )
  DROP $400 + DUP C@ $40 XOR SWAP C! ;
```

5.7 6809 disassembler: 325-336

B325

```
\ 6809 disassembler
\ order below represent opid, alpha order, branches last
CREATE OPNAME ," ABXADCADDANDASLASRBITCLRCMPCOMCWADAA" \ x12
  ," DECEOREXGINCJMPJSR LDLEALSRMULNEGNOP ORPSHPULROL" \ x16
  ," RORRTIRTSSBCSEX STSUBSWISYNTFRTSTBSR" \ x12
  ," BRABRNBHIBLSBCCBCSBNEBEQBVCBVSBPLBMIBGEBLTBGTBLE" \ x16
  ," ???"
3 CONSTS 56 OPCNT $ff NUL 40 BRIDX
10 VALUES addr oplen page opcode opid modeid arg indid indR
  tgtid
: >>4 >> >> >> >> ; : n, ( n -- ) >R BEGIN RUN1 , NEXT ;
: signext ( b -- n ) DUP $7f > IF $ff00 OR THEN ;
: M@ ( a -- n ) C@+ <<8 SWAP C@ OR ; : M@+ DUP 1+ 1+ SWAP M@ ;
: WORDTBL ( n -- ) CREATE >R BEGIN ' , NEXT ;
: opname. ( -- ) opid BRIDX >= page 1 = AND IF ( Lbranch )
  'L' EMIT THEN opid OPCNT MIN 3 * OPNAME + 3 STYPE ;
```

B326

```
\ NEG, COM, LSR, ...
CREATE GRP0 $10 nC, 22 NUL NUL 9   20 NUL 28 5
                     4 27  12  NUL 15 38  16 7
\ NOP, SYNC, DAA, ...
CREATE GRP1 $10 nC, NUL NUL 23  36 NUL NUL NUL NUL
                    NUL  11 24 NUL   3  32  14  37
\ branches
CREATE GRP2 $10 nC, 40 41 42 43 44 45 46 47
                    48 49 50 51 52 53 54 55
\ LEA, PSH, PUL, ...
CREATE GRP3 $10 nC, 19  19 19 19 25 26 25  26
                    NUL 30  0 29 10 21 NUL 35
\ SUB, CMP, SBC, ...
CREATE GRP8 $10 nC, 34 8 31 34  3  6 18 NUL
                    13 1 24  2  8 39 18 NUL
```

B327

```
\ GRP8 + ST + JSR
CREATE GRP9 $10 nC, 34 8 31 34  3  6 18 33
                    13 1 24  2  8 17 18 33
\ SUB, CMP, ADD, ...
CREATE GRPC $10 nC, 34 8 31  2  3   6 18 NUL
                    13 1 24  2 18 NUL 18 NUL
\ GRPC + ST
CREATE GRPD $10 nC, 34 8 31  2  3   6 18 33
                    13 1 24  2 18  33 18 33
CREATE _ $10 n, GRP0 GRP1 GRP2 GRP3 GRP0 GRP0 GRP0 GRP0
                GRP8 GRP9 GRP9 GRP9 GRPC GRPD GRPD GRPD
: opid@ opcode DUP >>4 << _ + @ SWAP $f AND + C@ TO opid ;
```

B328

```
\ tgt id is the same as in TFR/EXG. 2b for each name. $6 is for
\ memory or inherent targets. $c and $d are for SWI
CREATE TGTNAME ," D X Y U S PC  ??A B CCDP2 3 ??"
CREATE GRP0 $10 nC, 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
CREATE GRP1 $10 nC, 6 6 6 6 6 6 6 6 6 6 10 6 10 6 6 6
CREATE GRP3 $10 nC, 1 2 4 3 4 4 3 3 6 6 6 6 6 6 6 6
CREATE GRP4 $10 nC, 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
CREATE GRP5 $10 nC, 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
CREATE GRP8 $10 nC, 8 8 8 0 8 8 8 8 8 8 8 1 6 1 1
CREATE GRPC $10 nC, 9 9 9 0 9 9 9 9 9 9 9 0 0 3 3
CREATE _ $10 n, GRP0 GRP1 GRP0 GRP3 GRP4 GRP5 GRP0 GRP0
                GRP8 GRP8 GRP8 GRP8 GRPC GRPC GRPC GRPC
: tgtid@ opcode DUP >>4 << _ + @ SWAP $f AND + C@ TO tgtid ;
: tgtwide? ( tgtid -- f ) 6 < ;
: tgtname ( tgtid -- 's ) $e MIN << TGTNAME + ;
```

B329

```
\ 6809D, index modes
\ 0=n,R 1=R,R 2=,R+ 3=,R++ 4=,-R 5=,--R 6=n,PC
\ 8=[n,R] 9=[R,R] 11=[,R++] 13=[,--R] 14=[n,PC] 15=iext
CREATE _ 8 nC, $02 $03 $04 $05 $00 $81 $91 $ff
: _1RRi0xxx ( n&7 -- off indid ) _ + C@ DUP >>4 SWAP $f AND ;
: _nul ( a -- a+n off indid ) 0 $ff ;
: _b,R C@+ 0 ; : _n,R M@+ 0 ;
: _D,R 0 1 ;
: _b,PC C@+ 6 ; : _n,PC M@+ 6 ;
8 WORDTBL _ _b,R _n,R _nul _D,R _b,PC _n,PC _nul _nul
: _1RRi1xxx ( a n&f -- a+n off indid )
  7 AND << _ + @ EXECUTE ;
```

B330

```
\ 6809D, index modes
: _R@ ( n -- R ) $60 AND >>4 >> 1+ ;
0 VALUE _i
\ R is a tgtid, off in R,R is a tgtid, an abs EA in iext.
: ind@ ( a -- a+n R off indid )
  C@+ DUP TO _i ( a n ) $9e AND $9c = IF ( iext )
    _i 1 AND IF M@+ ELSE C@+ THEN 0 SWAP 15 EXIT THEN ( a )
  _i $80 AND NOT IF ( 5b,R ) _i _R@ _i $1f AND 0 EXIT THEN ( a )
  _i $f AND DUP 7 > IF _1RRi1xxx ELSE _1RRi0xxx THEN
  _i $10 AND >> OR \ apply indirect bit to indid
  ( a off indid ) _i _R@ ROT> ;
```

B331

```
\ 6809D, addr modes
\ 0=inh 1=acc 2=imm 3=rel 4=zp 5=regTFR 6=regPSH 7=ext 8=ind
: nop ( a -- a+n arg ) 0 ;
: imm tgtid tgtwide? IF M@+ ELSE C@+ THEN ;
: rel page 1 = IF ( lbranch ) M@+ ELSE C@+ signext THEN ;
: zp C@+ ;
: ext M@+ ;
: ind ind@ TO indid SWAP TO indR ( a+n arg ) ;
9 WORDTBL _ nop nop imm rel zp zp zp ext ind
: arg@+ ( a -- a+n ) modeid << _ + @ EXECUTE TO arg ;
```

B332

```
\ 6809D, addr modes
\ 0=inh 1=acc 2=imm 3=rel 4=zp 5=regTFR 6=regPSH 7=ext 8=ind
\ $10 regular modes followed by 1x and 3x special modes
CREATE modes $30 nC, 4 $10 3 $20 1 1 8 7 2 4 8 7 2 4 8 7
  0 0 0 0 0 0 3 3 0 0 2 0 2 0 5 5
  8 8 8 8 6 6 6 6 0 0 0 0 2 0 0 0
: mode@+ ( a -- a+n )
  opcode >>4 modes + C@ DUP $10 >= IF ( offset )
    modes + opcode $f AND + C@ THEN TO modeid
  opcode $8d = IF ( BSR is special ) 3 TO modeid THEN arg@+ ;
```

B333

```
CREATE s 9 ALLOT
: s$ s 9 SPC FILL ;
: nul. s 9 '?' FILL ; ALIAS NOOP inh. ALIAS NOOP acc.
: imm. '#' s C! arg s 1+
  tgtid tgtwide? IF FMTX ELSE FMTx THEN 2DROP ;
: zp. '$' s C! arg s 1+ FMTx 2DROP ;
: ext. '$' s C! arg s 1+ FMTX 2DROP ;
: rel. page 1 = IF ext. ELSE zp. THEN ;
: regTFR. arg >>4 tgtname s 2 MOVE
  arg $f AND tgtname s 1+ 1+ 2 MOVE ;
CREATE _ ," $SYX%BAC"
: regPSH. A>R s >A _ s 8 MOVE arg $80 BEGIN
  2DUP AND NOT IF SPC AC! THEN A+ >> DUP NOT UNTIL 2DROP R>A ;
```

B334

```
\ 6809D, index modes printing
: fmt ( n -- ) DUP $ff > IF A> FMTX ELSE A> FMTx THEN 2DROP ;
: Rfmt ( tgtid -- ) tgtname C@ AC! A+ ;
: +> '+' AC! A+ ; : -> '-' AC! A+ ;
: _n,R indR Rfmt +> arg fmt ;
: _R,R indR Rfmt +> arg Rfmt ;
: _,R+ indR Rfmt +> ; : _,R++ _,R+ +> ;
: _,-R -> indR Rfmt ; : _,--R -> _,-R ;
: _n,PC S" PC+" A> SWAP MOVE A+ A+ A+ arg fmt ;
8 WORDTBL _ _n,R _R,R _,R+ _,R++ _,-R _,--R _n,PC nul.
: ind. A>R s >A indid 7 > IF '[' AC! A+ ']' A> 7 + C! THEN
  indid 7 AND << _ + @ EXECUTE R>A ;
```

B335

```
$10 WORDTBL _ inh. acc. imm. rel. zp. regTFR. regPSH. ext.
  ind. nul. nul. nul. nul. nul. nul. nul.
: mode. s$ modeid << _ + @ EXECUTE s 9 STYPE ;
: repl ( val tbl -- nval-or-ff *A* ) >A BEGIN
  AC@+ 2DUP = IF 2DROP AC@+ EXIT THEN A+ $ff = UNTIL DROP $ff ;
CREATE _ops 11 nC, 35 35 34 8 8 8 18 18 33 33 $ff
CREATE _tgt 9 nC, 6 12 0 0 1 2 3 4 $ff
: _op10 opid DUP BRIDX < IF _ops repl TO opid THEN
  tgtid _tgt repl TO tgtid ;
```

B336

```
CREATE _ops 7 nC, 35 35 34 8 8 8 $ff
CREATE _tgt 7 nC, 6 13 0 3 1 4 $ff
: _op11 opid _ops repl TO opid tgtid _tgt repl TO tgtid ;
: bin. oplen >R addr BEGIN C@+ .x SPC> NEXT DROP ;
: op@+ ( a -- a+n ) 0 TO page DUP TO addr C@+
  DUP $fe AND $10 = IF 1 AND 1+ TO page C@+ THEN ( a+n opc )
  TO opcode opid@ tgtid@ mode@+ ( a+n ) DUP addr - TO oplen
  page 1 = IF _op10 THEN page 2 = IF _op11 THEN ( a+n ) ;
: op. opname. tgtid tgtname 2 STYPE mode. SPC> bin. NL> ;
20 VALUE DISCNT
: dis ( a -- ) DISCNT >R BEGIN op@+ op. NEXT DROP ;
```

5.8 6809 emulator: 340-354

B340

```
\ mapping: D X Y U S PC CC/DP
CREATE 'D 14 ALLOT
'D VALUE 'A 'A 1+ VALUE 'B 'D 2 + VALUE 'X
'X 2 + VALUE 'Y 'Y 2 + VALUE 'U 'U 2 + VALUE 'S
'S 2 + VALUE 'PC 'PC 2 + VALUE 'CC 'CC 1+ VALUE 'DP
CREATE MEM $800 ALLOT \ 2K ought to be enough for anybody
\ EA is in *target* addr
4 VALUES EA HALT? VERBOSE 'BRK?
: BRK? 'BRK? DUP IF EXECUTE THEN ;
: M! ( n a -- ) OVER >>8 OVER C! 1+ C! ;
: MEM+ ( off -- addr ) MEM + ;
```

B341

```
\ tgtid is from 6809D
CREATE _ $10 n, 'D 'X 'Y 'U 'S 'PC 0 0 'A 'B 'CC 'DP 0 0 0 0
: tgtreg ( tgtid -- regaddr )
  $f AND << _ + @ DUP NOT IF ABORT" invalid tgt" THEN ;
: 'tgt ( tgtid -- a ) \ host addr for tgtid
  DUP 6 = IF DROP EA MEM+ ELSE tgtreg THEN ;
: CC@ 'CC C@ ; : CC! 'CC C! ;
: neg? ( n -- f ) tgtid tgtwide? NOT IF <<8 THEN 0< ;
: ZNV! ( old new -- ) <<8 >>8
  ( Z? ) DUP NOT ( old new z ) ( N? ) SWAP neg? ( old z n )
  ( V? n != oldn ) ROT neg? OVER = NOT ( z n v )
  << SWAP <<3 OR ( z f ) SWAP << << OR ( f=0000NZV0 )
  CC@ $f1 AND OR CC! ;
```

B342

```
: W?@ tgtid tgtwide? IF M@ ELSE C@ THEN ;
: W?! tgtid tgtwide? IF M! ELSE C! THEN ;
: PC@ 'PC M@ ; : PC! 'PC M! ;
: EA@@ EA MEM+ W?@ ; : EA@! EA MEM+ W?! ;
: tgt@ ( tgtid -- n )
  DUP 'tgt SWAP tgtwide? IF M@ ELSE C@ THEN ;
: tgt! ( n tgtid -- )
  DUP 'tgt SWAP tgtwide? IF M! ELSE C! THEN ;
```

B343

```
: push8 'S M@ 1- DUP 'S M! MEM+ C! ;
: push16 DUP push8 >>8 push8 ;
: pull8 'S M@ DUP 1+ 'S M! MEM+ C@ ;
: pull16 pull8 pull8 <<8 OR ;
: carry> ( -- f ) CC@ $01 AND ;
: >carry ( f -- ) CC@ $fe AND OR CC! ;
: >>CC ( b -- b>>1 ) DUP 1 AND >carry >> ;
: <<CC ( b -- b<<1 ) << DUP $ff > >carry ;
: cpu. ." A B  X    Y    U    S    PC   CC DP" NL>
  'D 6 >R BEGIN DUP M@ .X SPC> 1+ 1+ NEXT DROP
  CC@ .x SPC> 'DP C@ .x NL> ;
: NIL ." invalid opcode " NL> cpu. ABORT ;
```

B344

```
\ all same signature
: n,R ( -- ea ) indR tgt@ arg + ;
: R,R indR tgt@ arg tgt@ + ;
: ,R+ indR tgt@ DUP 1+ indR tgt! ;
: ,R++ indR tgt@ DUP 1+ 1+ indR tgt! ;
: ,-R indR tgt@ 1- DUP indR tgt! ;
: ,--R indR tgt@ 1- 1- DUP indR tgt! ;
: n,PC 'PC M@ arg + ;
: indirect DOER ' , DOES> @ EXECUTE MEM+ M@ ;
indirect [n,R] n,R              indirect [R,R] R,R
indirect [,R++] ,R++            indirect [,--R] ,--R
indirect [n,PC] n,PC            indirect [n] arg
```

B345

```
\ maps indid from 6809D
16 WORDTBL IMODES
  n,R   R,R   ,R+ ,R++   ,-R ,--R   n,PC   NIL
  [n,R] [R,R] NIL [,R++] NIL [,--R] [n,PC] [n]
: indexed indid << IMODES + @ EXECUTE ( ea ) TO EA ;
```

B346

```
: imm addr MEM - oplen + 1- tgtid tgtwide? IF 1- THEN TO EA ;
: rel PC@ oplen + arg + TO EA ;
: zp arg 'DP C@ <<8 + TO EA ;
: ext arg TO EA ;
9 WORDTBL ADDRS NOOP NOOP imm rel zp NOOP NOOP ext indexed
: setEA ( -- ) modeid << ADDRS + @ EXECUTE ;
: TGT!ZNV ( n -- ) \ write n to TGT and update flags
  tgtid tgt@ OVER ( n old n ) ZNV! ( n ) tgtid tgt! ;
```

B347

```
\ ops all have a neutral sig and expect EA and tgt to be set.
\ INH and special words
: TODO ABORT" TODO" ; : RTS TODO ; : ABX TODO ; : RTI TODO ;
: CWAI TODO ; : SWI TODO ;
ALIAS NOOP NOP
: CLR 0 tgtid tgt! CC@ $f0 AND $04 OR CC! ;
: JMP EA PC! ;
: JSR PC@ push16 EA PC! ; ALIAS JSR BSR
: MUL 'A C@ 'B C@ * DUP 'D M! ( n )
  DUP NOT << << ( n z ) SWAP $80 AND << >>8 ( z c ) OR
  ( f = 00000Z0C ) CC@ $fa AND OR CC! ;
```

B348

```
: SYNC 1 TO HALT? ;
: DAA TODO ;
: SEX 'B C@ signext DUP 'D M! DUP ZNV! ;
: src arg >>4 ; : dst arg $f AND ;
: EXG src tgt@ dst tgt@ >R dst tgt! R> dst tgt! ;
: TFR src tgt@ dst tgt! ;
```

B349

```
: CBRA 0 AND ; : CBHI 5 AND ; : CBCC 1 AND ; : CBNE 4 AND ;
: CBVC 2 AND ; : CBPL 8 AND ;
: CBGE DUP CBVC SWAP CBPL = NOT ;
: CBGT DUP CBGE SWAP CBNE OR ;
8 WORDTBL BRCOND CBRA CBHI CBCC CBNE CBVC CBPL CBGE CBGT
: BROP ( -- ) CC@ opcode >> 7 AND << BRCOND + @ EXECUTE ( f )
  opcode 1 AND XOR NOT IF JMP THEN ;
```

B350

```
\ TGTOP: Read TGT value, operate, then write to TGT and flags
: TGTOP ' DOER , DOES> @ tgtid tgt@ SWAP EXECUTE TGT!ZNV ;
: asr ( n -- n ) DUP >>CC SWAP $80 AND OR ; TGTOP asr ASR
: com $ff XOR ; TGTOP com COM
TGTOP 1- DEC TGTOP 1+ INC
TGTOP <<CC ASL TGTOP >>CC LSR
: neg 0 -^ ; TGTOP neg NEG
: rol carry> SWAP <<CC OR ; TGTOP rol ROL
: ror carry> <<8 >> OR >>CC ; TGTOP ror ROR
: lea DROP EA ; TGTOP lea LEA
```

B351

```
\ EAOP: Read TGT and EA, apply op, write to TGT and flags
: EAOP ' DOER , DOES>
  @ tgtid tgt@ EA@@ ROT EXECUTE TGT!ZNV ;
: +c ( a b -- a+b carry ) <> ( l h ) OVER + ( h sum ) TUCK > ;
: adc ( a b -- n ) tgtid tgtwide? IF +c SWAP carry> +c ROT OR
  ELSE + carry> + DUP >>8 THEN >carry ;
EAOP adc ADC
: add 0 >carry adc ; EAOP add ADD
: -c ( a b -- a-b carry ) 2DUP < ROT> - SWAP ;
: sbc -c SWAP carry> -c ROT OR >carry ; EAOP sbc SBC
: sub 0 >carry sbc ; EAOP sub SUB
EAOP AND AND_ EAOP XOR EOR EAOP OR OR_ EAOP NIP LD
```

B352

```
\ FLAGOP: Reads TGT, perform op, then update NVZ flags only
\ op sig: n -- n
: FLAGOP ' DOER , DOES>
  @ tgtid tgt@ DUP ROT EXECUTE ( old new ) ZNV! ;
FLAGOP NOOP TST
: cmp EA@@ sub ; FLAGOP cmp CMP
: st DUP EA@! ; FLAGOP st ST
: bit EA@@ AND ; FLAGOP bit BIT
```

B353

```
CREATE _ 8 nC, 10 8 9 11 1 2 4 5
: PSHS _ 7 + arg BEGIN ( a flags )
    DUP $80 AND IF OVER C@ DUP tgtreg ( a flags tgtid reg )
      SWAP tgtwide? IF M@ push16 ELSE C@ push8 THEN THEN
    << $ff AND SWAP 1- SWAP ?DUP NOT UNTIL DROP ;
: PULS _ arg BEGIN ( a flags )
    DUP 1 AND IF OVER C@ DUP tgtreg ( a flags tgtid reg )
      SWAP tgtwide? IF pull16 M! ELSE pull8 C! THEN THEN
    >> SWAP 1+ SWAP ?DUP NOT UNTIL DROP ;
: U<>S 'U @ 'S @ SWAP 'S ! 'U ! ;
: PSHU U<>S PSHS U<>S ; : PULU U<>S PULS U<>S ;
: PSH tgtid 3 = IF PSHU ELSE PSHS THEN ;
: PUL tgtid 3 = IF PULU ELSE PULS THEN ;
```

B354

```
BRIDX WORDTBL OPS ABX ADC ADD AND_ ASL ASR BIT CLR CMP COM CWAI
  DAA DEC EOR EXG INC JMP JSR LD LEA LSR MUL NEG NOP OR PSH PUL
  ROL ROR RTI RTS SBC SEX ST SUB SWI SYNC TFR TST BSR
: rdop PC@ MEM+ op@+ DROP ; : peekop rdop op. ;
: run1
  HALT? IF ABORT" CPU halted" THEN
  rdop VERBOSE IF op. THEN setEA PC@ oplen + PC! opid
  DUP OPCNT < IF DUP BRIDX < IF << OPS + @ EXECUTE
    ELSE DROP BROP THEN ELSE NIL THEN
  VERBOSE IF cpu. THEN
  BRK? IF ABORT" breakpoint reached" THEN ;
: runN >R BEGIN run1 NEXT ; : run BEGIN run1 AGAIN ;
: 6809E$ 0 TO HALT? $100 PC! 0 'DP C! ;
```

5.9 Virgil's workspace: 360-361

B360

```
\ test a few ops in 6809E
6809E$ 1 TO VERBOSE
HERE MEM $100 + 'HERE !
$42 # LDA, $56 # ADDA, $12 <> STA, $12 <> SUBB, SYNC,
'HERE !
```

B361

```
\ test new lblval semantics
$200 VALUE TO?
6809E$ 1 TO VERBOSE
HERE MEM $100 + 'HERE ! $100 XSTART
FJR BRA, TO L1
LSET L2 ( lblval ) TO? () TST, IFZ, ( read )
  [S+0] LDD, S+0 STD, ELSE, ( write )
  2 S+N LDD, [S++] STD, S++ TST, THEN, SYNC,
LSET L3 ( VALUE ) L2 () JSR, $1234 M,
L1 FMARK $ff # LDS,
( read ) \ TO? () STS,
( write ) 42 # LDD, PSHS, D TO? () STB,
L3 BR BRA,
'HERE !
```

6 6502

6.1 Architecture index: 300

B300

```
6502 MASTER INDEX

301 6502 macros and consts     302 6502 assembler
310 6502 boot code             330 6502 disassembler
335 6502 emulator              350 Virgil's workspace
360 Apple IIe drivers
```

6.2 6502 macros and consts: 301

B301

```
\ 6502 macros and constants. See doc/code/6502.txt
: 6502A 302 305 LOADR 7 LOAD ( flow ) ;
: 6502M 309 LOAD ;
: 6502C 310 321 LOADR ;
: 6502D 330 334 LOADR ;
: 6502E 335 342 LOADR ;
\ ZP assignments
$06 VALUE 'A
$08 VALUE 'N
0 VALUE IPL 2 VALUE INDJ
: IPH IPL 1+ ; : INDL INDJ 1+ ; : INDH INDL 1+ ;
```

6.3 6502 assembler: 302-307

B302

```
\ 6502 assembler, Addressing modes.
\ output: n n-is-2b opoff
: # ( n ) 0 $09 ; \ Immediate
: <> ( n ) 0 $05 ; \ ZeroPage
: <X+> ( n ) 0 $15 ; \ ZeroPage+X
: <Y+> ( n ) 0 $15 ; \ Only for LDX
: () ( n ) 1 $0d ; \ Absolute
: (X+) ( n ) 1 $1d ; \ Absolute+X
: (Y+) ( n ) 1 $19 ; \ Absolute+Y
: [X+] ( n ) 0 $01 ; \ Indirect+X
: []Y+ ( n ) 0 $11 ; \ Indirect+Y
: ?, ( n n-is-2b -- ) IF L, ELSE C, THEN ;
```

B303

```
\ 6502 asm, Groups 1 and 2, 3-with-AM
: OPG1 DOER C, DOES> C@ OR C, ?, ;
$60 OPG1 ADC,  $20 OPG1 AND,  $c0 OPG1 CMP,  $40 OPG1 EOR,
$a0 OPG1 LDA,  $00 OPG1 ORA,  $e0 OPG1 SBC,  $80 OPG1 STA,

: _09repl DUP $09 = IF DROP 1 THEN ;
: OPG2 DOER C, DOES> C@ SWAP _09repl OR 1+ C, ?, ;
$00 OPG2 ASL,  $c0 OPG2 DEC,  $e0 OPG2 INC,  $a0 OPG2 LDX,
$40 OPG2 LSR,  $20 OPG2 ROL,  $60 OPG2 ROR,  $80 OPG2 STX,

: OPG3 DOER C, DOES> C@ SWAP _09repl OR 1- C, ?, ;
$20 OPG3 BIT,  $e0 OPG3 CPX,  $c0 OPG3 CPY,  $a0 OPG3 LDY,
$80 OPG3 STY,
```

B304

```
\ 6502 asm, implied, branching
: OP DOER C, DOES> C@ C, ;
$0a OP ASLA, $00 OP BRK,  $18 OP CLC,  $d8 OP CLD,  $58 OP CLI,
$b8 OP CLV,  $ca OP DEX,  $88 OP DEY,  $e8 OP INX,  $c8 OP INY,
$4a OP LSRA, $ea OP NOP,  $48 OP PHA,  $08 OP PHP,  $68 OP PLA,
$28 OP PLP,  $2a OP ROLA, $6a OP RORA, $40 OP RTI,  $60 OP RTS,
$38 OP SEC,  $f8 OP SED,  $78 OP SEI,  $aa OP TAX,  $a8 OP TAY,
$98 OP TYA,  $ba OP TSX,  $8a OP TXA,  $9a OP TXS,

: OPBR DOER C, DOES> C@ C, C, ;
$90 OPBR BCC, $b0 OPBR BCS, $f0 OPBR BEQ, $30 OPBR BMI,
$d0 OPBR BNE, $10 OPBR BPL, $50 OPBR BVC, $70 OPBR BVS,

: OPBR2 DOER C, DOES> C@ C, L, ;
$20 OPBR2 JSR, $4c OPBR2 JMP, $6c OPBR2 JMP[],
```

B305

```
\ 6502 HAL
ALIAS JMP, JMPi, ALIAS JMP[], JMP(i), ALIAS JSR, CALLi,
: JRi, CLV, BVC, ; \ no BRA!
ALIAS BEQ, JRZi, ALIAS BNE, JRNZi,
ALIAS BCS, JRCi, ALIAS BCC, JRNCi,
: i>, DEX, DEX, DUP # LDA, 0 <X+> STA, >>8 # LDA, 1 <X+> STA, ;
: (i)>,
  DEX, DEX, DUP () LDA, 0 <X+> STA, 1+ () LDA, 1 <X+> STA, ;
```

6.4 6502 port macros: 309

B309

```
\ 6502 port macros

\ helpers
: PS<>, ( src dst ) SWAP <X+> LDA, <X+> STA, ;
: PSCLR16, 0 # LDA, DUP <X+> STA, 1+ <X+> STA, ;
: A>IND+, INDL []Y+ STA, INY, ;
: PS>A, <X+> LDA, ;
: A>PS, <X+> STA, ;
: PSINC, 0 <X+> INC, IFZ, 1 <X+> INC, THEN, ;
: IP+, IPL <> INC, 2 BNE, IPH <> INC, ;
```

6.5 6502 boot code: 310-321

B310

```
\ 6502 boot code PS=X RS=S
$6c # LDA, INDJ <> STA, $ff # LDX, TXS,
0 JMP, PC 2 - TO lblboot
LSET lblcell DEX, DEX, PLA, 0 A>PS, PLA, 1 A>PS, PSINC, \ next
LSET lblnext IPH <> LDY, IPL <> LDA, INDH <> STY, INDL <> STA,
LSET L1 CLC, 2 # ADC, IFC, INY, THEN, IPL <> STA, IPH <> STY,
  INDJ JMP,
LSET lblxt PLA, INDL <> STA, PLA, INDH <> STA,
  IPH <> LDA, PHA, IPL <> LDA, PHA,
  INDL <> INC, IFZ, INDH <> INC, THEN,
  INDL <> LDA, INDH <> LDY, L1 JMP,
LSET lbldoes CLC, PLA, TAY, PLA, INY, IFZ, 1 # ADC, THEN,
  INDL <> STY, INDH <> STA, DEX, DEX, 1 <X+> STA, TYA, 2 # ADC,
  IFC, 1 <X+> INC, THEN, 0 <X+> STA, INDJ JMP,
```

B311

```
CODE BYE BRK,
CODE QUIT
  TXA, $ff # LDX, TXS, TAX, 0 JMP, PC 2 - TO lblmain
CODE ABORT $ff # LDX, X' QUIT BR BNE,
CODE EXIT PLA, IPL <> STA, PLA, IPH <> STA, ;CODE
CODE EXECUTE 0 <X+> LDA, INDL <> STA, 1 <X+> LDA, INDH <> STA,
  INX, INX, INDL JMP[],
CODE SCNT INDL <> STX, DEX, DEX, 0 # LDA, 1 <X+> STA,
  $ff # LDA, SEC, INDL <> SBC, 0 <X+> STA,
  FJR BPL, 1 <X+> DEC, THEN, ;CODE
CODE RCNT TXA, TSX, INDL <> STX, TAX, DEX, DEX, 0 # LDA,
  1 <X+> STA, $ff # LDA, SEC, INDL <> SBC, 0 <X+> STA, ;CODE
```

B312

```
CODE (b) 0 # LDY, IPL []Y+ LDA, DEX, DEX, 0 A>PS, 0 # LDA,
  1 A>PS, IP+, ;CODE
CODE (n) 0 # LDY, IPL []Y+ LDA, DEX, DEX, 0 A>PS, INY,
  IPL []Y+ LDA, 1 A>PS, IP+, IP+, ;CODE
CODE (br) 0 # LDY, IPL []Y+ LDA, FJR BPL, IPH <> DEC, THEN,
  CLC, IPL <> ADC, IFC, IPH <> INC, THEN, IPL <> STA, ;CODE
CODE (?br) 0 <X+> LDA, 1 <X+> ORA, INX, INX,
  0 # ORA, X' (br) BR BEQ, IP+, ;CODE
LSET L1 ( ovfl, always branch, C is clear )
  PLA, 0 # SBC, PHA, $ff # LDA, PHA, X' (br) JMP,
LSET L2 PHA, 0 # LDA, PHA, X' (br) JMP,
CODE (next) PLA, SEC, 1 # SBC, L1 BR BCC,
  PHA, X' (br) BR BNE, \ branch if nonzero
  PLA, PLA, L2 BR BNE, ( finished ) IP+, ;CODE
```

B313

```
CODE C@ 0 [X+] LDA, 0 <X+> STA, 0 # LDA, 1 <X+> STA, ;CODE
CODE @ LSET L1 0 [X+] LDA, TAY, PSINC, 0 [X+] LDA,
  0 <X+> STY, 1 <X+> STA, ;CODE
CODE C! 2 <X+> LDA, 0 [X+] STA, INX, INX, INX, INX, ;CODE
CODE ! LSET L2
  2 <X+> LDA, 0 [X+] STA, PSINC, 3 <X+> LDA, 0 [X+] STA,
  INX, INX, INX, INX, ;CODE
LSET lblval DEX, DEX, PLA, 0 A>PS, PLA, 1 A>PS, PSINC,
  SYSVARS $18 + () LSR, L2 BR BCS, L1 JMP,
CODE 1+ PSINC, ;CODE
CODE 1- 0 <X+> LDA, IFZ, 1 <X+> DEC, THEN, 0 <X+> DEC, ;CODE
CODE + CLC, 2 <X+> LDA, 0 <X+> ADC, 2 <X+> STA, 3 <X+> LDA,
  1 <X+> ADC, 3 <X+> STA, INX, INX, ;CODE
CODE - 2 <X+> LDA, SEC, 0 <X+> SBC, 2 <X+> STA, 3 <X+> LDA,
  1 <X+> SBC, 3 <X+> STA, INX, INX, ;CODE
```

B314

```
CODE < 3 PS>A, 1 <X+> CMP, IFZ, 2 PS>A, 0 <X+> CMP, THEN,
  INX, INX, 0 # LDA, 1 A>PS, 0 # ADC, 1 # EOR, 0 A>PS, ;CODE
CODE << 0 <X+> ASL, 1 <X+> ROL, ;CODE
CODE >> 1 <X+> LSR, 0 <X+> ROR, ;CODE
CODE <<8 0 1 PS<>, 0 # LDA, 0 <X+> STA, ;CODE
CODE >>8 1 0 PS<>, 0 # LDA, 1 <X+> STA, ;CODE
CODE AND 0 <X+> LDA, 2 <X+> AND, 2 <X+> STA, 1 <X+> LDA,
  3 <X+> AND, 3 <X+> STA, INX, INX, ;CODE
CODE OR 0 <X+> LDA, 2 <X+> ORA, 2 <X+> STA, 1 <X+> LDA,
  3 <X+> ORA, 3 <X+> STA, INX, INX, ;CODE
CODE XOR 0 <X+> LDA, 2 <X+> EOR, 2 <X+> STA, 1 <X+> LDA,
  3 <X+> EOR, 3 <X+> STA, INX, INX, ;CODE
CODE NOT 0 # LDY, 0 <X+> LDA, 1 <X+> ORA, 1 <X+> STY,
  IFZ, INY, THEN, 0 <X+> STY, ;CODE
```

B315

```
CODE * DEX, DEX, 16 # LDY, 0 PSCLR16,
  BEGIN, 0 <X+> ASL, 1 <X+> ROL, 4 <X+> ASL, 5 <X+> ROL,
    IFC, CLC, 2 <X+> LDA, 0 <X+> ADC, 0 <X+> STA, 3 <X+> LDA,
      1 <X+> ADC, 1 <X+> STA, THEN, DEY, BR BNE,
  0 4 PS<>, 1 5 PS<>, INX, INX, INX, INX, ;CODE
CODE /MOD \ a b -- r q
  DEX, DEX, DEX, 16 # LDA, 0 <X+> STA, ( cnt )
  1 PSCLR16, ( remaining )
  \ 3-4 = divisor 5-6 = dividend
  BEGIN, 5 <X+> ASL, 6 <X+> ROL, 1 <X+> ROL, 2 <X+> ROL,
    1 <X+> LDA, SEC, 3 <X+> SBC, TAY, 2 <X+> LDA, 4 <X+> SBC,
    IFC, 2 <X+> STA, 1 <X+> STY, 5 <X+> INC, THEN,
    0 <X+> DEC, BR BNE,
  5 3 PS<>, 6 4 PS<>, 1 5 PS<>, 2 6 PS<>, INX, INX, INX, ;CODE
```

B316

```
CODE DUP LSET L1 DEX, DEX, 2 0 PS<>, 3 1 PS<>, ;CODE
CODE ?DUP 0 <X+> LDA, 1 <X+> ORA, L1 BR BNE, ;CODE
CODE DROP INX, INX, ;CODE
CODE SWAP 0 <X+> LDA, 2 <X+> LDY, 0 <X+> STY, 2 <X+> STA,
  1 <X+> LDA, 3 <X+> LDY, 1 <X+> STY, 3 <X+> STA, ;CODE
CODE OVER DEX, DEX, 4 0 PS<>, 5 1 PS<>, ;CODE
CODE ROT ( a b c -- b c a ) 5 <X+> LDY, 3 5 PS<>, 1 3 PS<>,
  1 <X+> STY, 4 <X+> LDY, 2 4 PS<>, 0 2 PS<>, 0 <X+> STY, ;CODE
CODE ROT> ( a b c -- c a b ) 1 <X+> LDY, 3 1 PS<>, 5 3 PS<>,
  5 <X+> STY, 0 <X+> LDY, 2 0 PS<>, 4 2 PS<>, 4 <X+> STY, ;CODE
CODE R@ DEX, DEX, PLA, 0 <X+> STA, TAY, PLA, 1 <X+> STA, PHA,
  TYA, PHA, ;CODE
CODE >R 1 <X+> LDA, PHA, 0 <X+> LDA, PHA, INX, INX, ;CODE
CODE R> DEX, DEX, PLA, 0 <X+> STA, PLA, 1 <X+> STA, ;CODE
CODE R~ PLA, PLA, ;CODE
```

B317

```
CODE MOVE ( src dst u -- )
  4 PS>A, INDL <> STA, 5 PS>A, INDH <> STA, 2 PS>A, 'N <> STA,
  3 PS>A, 'N 1+ <> STA, 1 PS>A, 5 A>PS, 0 PS>A, 4 A>PS, INX,
  INX, INX, INX, 1 <X+> INC, 0 # LDY, FJR BEQ, TO L1 BEGIN,
    INDL []Y+ LDA, 'N []Y+ STA, INY,
    IFZ, INDH <> INC, 'N 1+ <> INC, THEN,
    L1 FMARK ( entry ) TYA, 0 <X+> CMP,
    DUP BR BNE, 1 <X+> DEC, BR BNE,
  INX, INX, ;CODE
```

B318

```
LSET L1 \ cmp strs at [INDL] and ['N] with cnt <X+0>
  0 # LDY, BEGIN,
    INDL []Y+ LDA, 'N []Y+ CMP, IFNZ, RTS, THEN,
    INY, 0 <X+> DEC, BR BNE, RTS,
CODE []= ( a1 a2 u -- f )
  2 <X+> LDA, INDL <> STA, 3 <X+> LDA, INDH <> STA,
  4 <X+> LDA, 'N <> STA, 5 <X+> LDA, 'N 1+ <> STA,
  0 4 PS<>, 1 <X+> LDY, INY, 5 <X+> STY, INX, INX, INX, INX,
  BEGIN,
    L1 JSR, IFNZ, ( fail ) 0 PSCLR16, ;CODE THEN,
    1 <X+> DEC, BR BNE,
  ( success ) 0 <X+> INC, ;CODE
```

B319

```
CODE FIND ( sa sl -- w? f ) \ 0=cnt 1=sl 2-3=curword N=sa
  2 <X+> LDA, 'N <> STA, 3 <X+> LDA, 'N 1+ <> STA, 0 1 PS<>,
  SYSVARS $02 + DUP () LDA, 2 <X+> STA, 1+ () LDA, 3 <X+> STA,
  BEGIN,
    3 <X+> LDA, INDH <> STA, 2 <X+> LDA,
    SEC, 3 # SBC, IFNC, INDH <> DEC, THEN, INDL <> STA,
    0 # LDY, INDL []Y+ LDA, PHA, INY, INDL []Y+ LDA, PHA, \ prev
    INY, INDL []Y+ LDA, $7f # AND, 1 <X+> CMP, IFZ,
      0 <X+> STA, INDL <> LDA, SEC, 0 <X+> SBC, INDL <> STA,
      IFNC, INDH <> DEC, THEN, L1 JSR, IFZ, \ match
        PLA, PLA, 0 # LDY, 1 <X+> STY, INY, 0 <X+> STY, ;CODE
      THEN, THEN,
    PLA, 3 <X+> STA, PLA, 2 <X+> STA, 3 <X+> ORA, IFZ, \ end
      INX, INX, 0 <X+> STA, 1 <X+> STA, ;CODE THEN,
  JMP,
```

B320

```
CODE [C]? ( c a u -- i ) \ X+0=c X+1=iH
  2 PS>A, INDL <> STA, 3 PS>A, INDH <> STA,
  0 PS>A, 'N <> STA, 1 PS>A, 'N 1+ <> STA,
  INX, INX, INX, INX, 'N <> ORA, IFNZ, ( u!=0 )
    0 # LDY, 1 <X+> STY, BEGIN, 0 PS>A, ( A=c ) BEGIN,
        INDL []Y+ CMP, IFZ, ( match ) 0 <X+> STY, ;CODE THEN,
        INY, IFZ, 1 <X+> INC, INDH <> INC, THEN,
        'N <> CPY, BR BNE,
      1 PS>A, 'N 1+ <> CMP, BR BNE, ( no match ) THEN,
  $ff # LDA, 0 A>PS, 1 A>PS, ;CODE
```

B321

```
CODE A> DEX, DEX, 'A <> LDA, 0 A>PS, 'A 1+ <> LDA, 1 A>PS, ;CODE
CODE >A 0 PS>A, 'A <> STA, 1 PS>A, 'A 1+ <> STA, INX, INX, ;CODE
CODE A>R 'A 1+ <> LDA, PHA, 'A <> LDA, PHA, ;CODE
CODE R>A PLA, 'A <> STA, PLA, 'A 1+ <> STA, ;CODE
CODE A+ 'A <> INC, IFZ, 'A 1+ <> INC, THEN, ;CODE
CODE A- 'A <> LDA, IFZ, 'A 1+ <> DEC, THEN, 'A <> DEC, ;CODE
CODE AC@
  DEX, DEX, 0 # LDY, 1 <X+> STY, 'A []Y+ LDA, 0 A>PS, ;CODE
CODE AC!  0 # LDY, 0 PS>A, 'A []Y+ STA, INX, INX, ;CODE
```

6.6 6502 disassembler: 330-334

B330

```
\ 6502 disassembler
\ order below represent "opid", also used in emulator
CREATE OPNAME ," ORAANDEORADCSTALDACMPSBC" \ 1/5/9/d x8
," ASLROLLSRRORSTXLDXDECINC" \ 6/a/e x8
," BITJMPSTYLDYCPYCPX" \ 4/c x6
," BRKBPLJSRBMIRTIBVCRTSBVSBCCLDYBCSCPYBNECPXBEQ" \ 0 x15
," PHPCLCPLPSECPHACLIPLASEIDEYTYATAYCLVINYCLDINXSED" \ 8 x16
," TXATXSTAXTSXDEXNOP" \ a x6
59 VALUE OPCNT $ff VALUE NUL 20 VALUE DISCNT
: >>4 >> >> >> >> ;
: opid. DUP OPCNT < IF
  3 * OPNAME + 3 STYPE ELSE DROP ." ???" THEN ;
: WORDTBL ( n -- ) CREATE >R BEGIN ' , NEXT ;
: spcs ( n -- ) >R BEGIN SPC> NEXT ;
```

B331

```
: id159d ( opcode -- opid )
  DUP $89 = IF DROP NUL ELSE >>4 >> THEN ;
CREATE _ 24 nC, $c $c $d $d $e $e $f $f
                $35 $36 $37 $38 $39 NUL $3a NUL
                $c NUL $d $d $e $e $f $f
: id6ae DUP $80 < IF ( ASL/ROL/LSR/ROR )
    DUP $1f AND $1a = IF DROP NUL EXIT THEN >>4 >> 8 + EXIT THEN
  DUP >> >> 1- 3 AND 8 * _ + ( op tbl ) SWAP >>4 7 AND + C@ ;
CREATE _ 32 nC,
NUL NUL $10 NUL NUL NUL NUL NUL $12 $12 $13 $13 $14 $14 $15 NUL
NUL NUL $10 NUL $11 NUL $11 NUL $12 NUL $13 $13 $14 NUL $15 NUL
: id4c _ OVER $8 AND IF $10 + THEN SWAP >>4 + C@ ;
: idnul DROP NUL ;
: id0 >>4 DUP 8 = IF DROP NUL EXIT THEN
  DUP 8 > IF 1- THEN 22 + ;
: id8 >>4 37 + ;
```

B332

```
: id2 $a2 = IF $0d ELSE NUL THEN ;
16 WORDTBL _ id0 id159d id2 idnul id4c
  id159d id6ae idnul id8 id159d
  id6ae idnul id4c id159d id6ae idnul
: opid DUP $f AND << _ + @ EXECUTE ;
\ 0=inh 1=imm 2=acc 3=zp 4=zp,X 5=zp,Y 6=abs 7=abs,X 8=abs,Y
\ 9=ind 10=ind,X 11=ind,Y 12=rel
CREATE _ $40 nC, 0  10 1 0 3 3 3 0 0 1 2 0 6 6 6 0
                 12 11 0 0 4 4 4 0 0 8 0 0 7 7 7 0
                 1  10 1 0 3 3 3 0 0 1 0 0 6 6 6 0
                 12 11 0 0 4 4 4 0 0 8 0 0 7 7 7 0
: modeid ( opcode -- id )
  DUP $20 = IF DROP 6 EXIT THEN DUP $6c = IF DROP 9 EXIT THEN
  DUP $be = IF DROP 8 EXIT THEN
  DUP $80 AND >> >> SWAP $1f AND OR _ + C@ ;
```

B333

```
: inh. ( a -- a ) 7 spcs ; : byte. C@+ .x ;
: $. '$' EMIT byte. ; : zp. $. 4 spcs ; ALIAS zp. rel.
: imm. '#' EMIT byte. 4 spcs ;
: $$. '$' EMIT C@+ SWAP C@+ .x SWAP .x ; : abs. $$. 2 spcs ;
: ind. '(' EMIT $$. ')' EMIT ;
: acc. 'A' EMIT 6 spcs ;
: ,X. ',' EMIT 'X' EMIT ;
: ,Y. ',' EMIT 'Y' EMIT ;
: zp,X. $. ,X. 2 spcs ; : zp,Y. $. ,Y. 2 spcs ;
: abs,X. $$. ,X. ; : abs,Y. $$. ,Y. ;
: ind,X. '(' EMIT $. ,X. ')' EMIT ;
: ind,Y. '(' EMIT $. ')' EMIT ,Y. ;
13 WORDTBL _ inh. imm. acc. zp. zp,X. zp,Y. abs. abs,X. abs,Y.
  ind. ind,X. ind,Y. rel.
```

B334

```
: mode. ( a opcode -- a ) modeid << _ + @ EXECUTE ;
: op. ( a -- a ) C@+ DUP opid DUP opid. SPC>
  OPCNT < IF mode. ELSE DROP THEN ;
: dump ( a u -- ) >R BEGIN C@+ .x SPC> NEXT DROP ;
HERE PC - VALUE OFFSET
: dis ( a -- ) DISCNT >R BEGIN
  DUP OFFSET - .X SPC> DUP op. SPC>
  TUCK OVER - dump NL> NEXT DROP ;
```

6.7 6502 emulator: 335-342

B335

```
\ 6502 emulator
CREATE 'A 7 ALLOT
'A 1+ VALUE 'X 'X 1+ VALUE 'Y 'Y 1+ VALUE 'S
'S 1+ VALUE 'P 'P 1+ VALUE 'PC
0 VALUE EA \ effective addr in *target*. ffff means accumulator
$800 VALUE MEMSZ \ 2k ought to be enough for anybody
CREATE MEM MEMSZ ALLOT
: 6502E$ 0 'P C! $200 'PC ! ;
: oor? ( pc -- pc ) DUP MEMSZ >= IF
  .X ABORT"  addr out of range" THEN ;
: mem+ oor? MEM + ; : ea@ EA $ffff = IF 'A ELSE EA mem+ THEN ;
: mc@ mem+ C@ ; : mc@+ DUP mc@ SWAP 1+ SWAP ; : mc! mem+ C! ;
: m@ mem+ @ ; : m@+ DUP m@ SWAP 1+ 1+ SWAP ;
: X+ 'X C@ + ; : Y+ 'Y C@ + ; : a@ 'A C@ ; : a! 'A C! ;
: pc@ 'PC @ ; : mpc@ pc@ mem+ ;
```

B336

```
: ea! ( pc -- ) oor? [TO] EA ;
: inh ( pc -- pc+? ) 0 ea! ; ALIAS inh acc
: zp mc@+ ea! ;
: imm DUP ea! 1+ ;
: abs m@+ ea! ;
: ind m@+ m@ ea! ;
: zp,X mc@+ X+ <<8 >>8 ea! ;
: zp,Y mc@+ Y+ <<8 >>8 ea! ;
: abs,X m@+ X+ oor? ea! ;
: abs,Y m@+ Y+ oor? ea! ;
: ind,X mc@+ X+ m@ ea! ;
: ind,Y mc@+ m@ Y+ ea! ;
13 WORDTBL _ inh imm acc zp zp,X zp,Y abs abs,X abs,Y ind ind,X
  ind,Y imm
: eard ( pc opcode -- pc+? ) modeid << _ + @ EXECUTE ;
```

B337

```
: p! ( n mask -- ) 'P C@ AND OR 'P C! ;
: carry! ( n -- n ) L|M NOT NOT ( n cf ) $fe p! ;
: carry? ( -- f ) 'P C@ 1 AND ;
: nz! ( n -- ) DUP NOT << SWAP $80 AND OR $7d p! ;
: v! ( n -- ) $80 AND a@ $80 AND XOR >> $bf p! ;
: a!nz DUP a! nz! ; : a!nzv DUP v! a!nz ;
: ora EA mc@ a@ OR a!nz ;
: and EA mc@ a@ AND a!nz ;
: eor EA mc@ a@ XOR a!nz ;
: adc EA mc@ carry? + a@ + carry! a!nzv ;
: sbc a@ EA mc@ carry? + - carry! a!nzv ;
: asl ea@ DUP C@ << carry! DUP nz! SWAP C! ;
: rol ea@ DUP C@ << carry? OR carry! DUP nz! SWAP C! ;
```

B338

```
: lsr ea@ DUP C@ DUP 1 AND $fe p! >> DUP nz! SWAP C! ;
: ror
  ea@ DUP C@ carry? <<8 OR DUP 1 AND $fe p! >> DUP nz! SWAP C! ;
: _ DOER , DOES> @ C@ ea@ C! ;
'A _ sta 'X _ stx 'Y _ sty
: _ DOER , DOES> @ EA mc@ DUP nz! SWAP C! ;
'A _ lda 'X _ ldx 'Y _ ldy
: _ DOER , DOES> @ C@ EA mc@ - carry! DUP v! nz! ;
'A _ cmp 'X _ cpx 'Y _ cpy
: pc+ea EA mc@ DUP $7f > IF $ff00 OR THEN 'PC @ + 'PC ! ;
: _ DOER C, DOES> C@ 'P C@ AND IF pc+ea THEN ;
$01 _ bcs $02 _ beq $40 _ bvs $80 _ bmi
: _ DOER C, DOES> C@ 'P C@ AND NOT IF pc+ea THEN ;
$01 _ bcc $02 _ bne $40 _ bvc $80 _ bpl
```

B339

```
: _ DOER C, DOES> C@ 'P C@ OR 'P C! ;
$01 _ sec $08 _ sed $04 _ sei $10 _ brk
: _ DOER C, DOES> C@ 'P C@ AND 'P C! ;
$fe _ clc $f7 _ cld $fb _ cli $bf _ clv
: pull ( -- b ) 'S C@ $100 OR mc@+ SWAP 'S C! ;
: push ( b -- ) 'S C@ 1- <<8 >>8 DUP 'S C! $100 OR mc! ;
: pla pull 'A C! ; : plp pull 'P C! ;
: pha a@ push ; : php 'P C@ push ;
: rti plp pull pull <<8 OR 'PC ! ;
: rts pull pull <<8 OR 1+ 'PC ! ;
: jmp EA 'PC ! ;
: jsr pc@ 1- L|M push push jmp ;
: bit EA mc@ DUP a@ AND NOT << OR $cd p! ;
```

B340

```
: inc EA mc@ 1+ DUP nz! EA mc! ;
: dec EA mc@ 1- DUP nz! EA mc! ;
: dex 'X C@ 1- DUP nz! 'X C! ;
: dey 'Y C@ 1- DUP nz! 'Y C! ;
: inx 'X C@ 1+ DUP nz! 'X C! ;
: iny 'Y C@ 1+ DUP nz! 'Y C! ;
: txa 'X C@ 'A C! ;
: tax 'A C@ 'X C! ;
: tya 'Y C@ 'A C! ;
: tay 'A C@ 'Y C! ;
: txs 'X C@ 'S C! ;
: tsx 'S C@ 'X C! ;
ALIAS NOOP nop
```

B341

```
\ opid same as in disassembler
OPCNT WORDTBL _ ora and eor adc sta lda cmp sbc asl rol lsr ror
  stx ldx dec inc bit jmp sty ldy cpy cpx brk bpl jsr bmi rti
  bvc rts bvs bcc ldy bcs cpy bne cpx beq php clc plp sec pha
  cli pla sei dey tya tay clv iny cld inx sed txa txs tax tsx
  dex nop
: nulop ( op -- ) .x ABORT"  invalid opcode" ;
: oprun ( opcode -- ) opid DUP OPCNT < IF
  << _ + @ EXECUTE ELSE nulop THEN ;
CREATE _ ," AXYSP"
: cpu. _ >A 'A >B 5 >R BEGIN
  AC@+ EMIT SPC> B> C@ .x B+ SPC> NEXT ." PC " 'PC @ .X NL> ;
```

B342

```
2 VALUES VERBOSE 'BRK?
: BRK? 'BRK? DUP IF EXECUTE THEN ;
: run1 ( -- )
  'P C@ $10 AND IF ABORT" CPU halted" THEN
  'PC @ mc@+ TUCK eard 'PC ! oprun
  VERBOSE IF cpu. THEN
  BRK? IF ABORT" breakpoint reached" THEN ;
: runN >R BEGIN run1 NEXT ; : run BEGIN run1 AGAIN ;
```

6.8 Virgil's workspace: 348-353

B348

```
: key BEGIN $c000 C@ $80 AND UNTIL
  $c000 C@ $7f AND DUP $c010 C! ;
```

B349

```
2 CONSTS 80 COLS 24 LINES
: pos2yx ( pos -- yx ) COLS /MOD ( x y ) <<8 OR ;
CODE yx2a ( yx -- a )
  $c054 () STA, 1 <X+> LDA, CLC, RORA, 0 <X+> ROR,
  IFNC, $c055 () STA, THEN, PHA, 3 # AND, 4 # ORA, 1 <X+> STA,
  PLA, $0c # AND, IFNZ, 8 # CMP, 40 # LDA, IFC, CLC, ROLA,
    THEN, 0 <X+> ADC, 0 <X+> STA, THEN, ;CODE
CODE~ ( c pos ) 2 <X+> LDA, CLC, $80 # ADC, 2 <X+> STA, ;CODE
: cell! ( c pos ) pos2yx yx2a ~ C! ;
: cursor! ( new old )
  posyx yx2a DUP C@ DUP $80 < IF $80 + SWAP C! ELSE 2DROP THEN
  posyx yx2a DUP C@ $80 XOR SWAP C! ;
```

B350

```
\ APPLE IIE xcomp, constants and macros
$300 VALUE SYSVARS
$91f0 VALUE BLK_MEM
SYSVARS $60 + VALUE GRID_MEM
GRID_MEM 2 + VALUE MSPAN_MEM
MSPAN_MEM 1+ VALUE SDC_MEM
4 CONSTS 100 MSPAN_SZ $c0b4 SPI_DATA $c0b5 SPI_CTL
  2 SDC_DEVID
\ ARCM ( D3-01 ) XCOMP ( D2-00 )
\ 6502A ( D3-02-05 D0-07 ) 6502M ( D3-09 )
\ XCOMPC ( D2-01-05 ) $2000 XSTART 6502C ( D3-10-20 )
\ COREL ( D2-10-24 ) Drivers ( D3-60-63 )
```

B351

```
\ Apple IIe: xwrap D2 in drive
\ ALIAS FD@ (ms@)
\ ALIAS FD! (ms!)
\ MSPANSUB ( D2-37 )
\ BLKSUB ( D2-30-34 ) GRIDSUB ( D2-40-41 )
: INIT CR NL ! 80col GRID$ BLK$ MSPAN$ ;
\ XWRAP
```

B352

```
\ Apple IIe, SPI
CODE (spie) ( n -- )
  0 <X+> LDA, INX, INX, SPI_CTL () STA, ;CODE
CODE (spix) ( n -- n )
  0 <X+> LDA, SPI_DATA () STA, 0 # LDA, 1 <X+> STA,
  SPI_DATA () LDA, 0 <X+> STA, ;CODE
```

B353

```
: _ ( a -- ) DUP 40 + SWAP ( src dst ) LINES 1- 40 * MOVE ;
: scroll 0 pos2a _ 1 pos2a _ ;
: scroll LINES >R BEGIN LINES R@ - COLS * DUP COLS + ( dst src )
    2DUP pos2a SWAP pos2a 40 MOVE ( dst src ) 1+ pos2a SWAP 1+
    pos2a 40 MOVE NEXT ;
```

6.9 Apple IIe drivers: 360-365

B360

```
\ Apple IIe drivers, (key?)
CODE (key?) ( -- c? f )
  DEX, DEX, 0 # LDA, 0 A>PS, 1 A>PS, $c000 () LDA, FJR BPL,
    $7f # AND, 0 A>PS, DEX, DEX, 0 # LDA, 1 A>PS,
    1 # LDA, 0 A>PS, $c010 () STA,
  THEN, ;CODE
```

B361

```
\ Apple IIe drivers, grid
CODE pos2a ( pos -- a )
  ( div by 80 ) 80 # LDA, INDL <> STA, 1 <X+> LDA, 8 # LDY,
  0 <X+> ASL, BEGIN, ROLA,
    IFNC, INDL <> CMP, FJR BCC, TO L1 THEN,
    INDL <> SBC, SEC, L1 FMARK 0 <X+> ROL, DEY, BR BNE,
  ( 0=y A=x ) TAY, 0 PS>A, 0 <X+> STY, ( 0=x A=y )
  $c054 () STA, CLC, RORA, 0 <X+> ROR,
  IFNC, $c055 () STA, THEN, TAY, 3 # AND, 4 # ORA, 1 A>PS, TYA,
  $0c # AND, IFNZ, 8 # CMP, 40 # LDA, IFC, CLC, ROLA, THEN,
    0 <X+> ADC, 0 A>PS, THEN, ;CODE
```

B362

```
\ Apple IIe drivers, grid
2 CONSTS 80 COLS 24 LINES
CODE 80col $c300 JSR, ;CODE
CODE hi ( c pos ) 2 PS>A, $7f # AND, $60 # CMP,
  IFNC, $3f # AND, THEN, 2 A>PS, ;CODE
CODE lo ( c pos ) 2 PS>A, $80 # ORA, 2 A>PS, ;CODE
: CELL! ( c pos ) pos2a lo C! ;
: CURSOR! ( new old -- )
  pos2a DUP C@ SWAP lo C! pos2a DUP C@ SWAP hi C! ;
```

B363

```
\ Apple IIe drivers, grid
CODE _ ( a -- src dst u ) \ prepare line at a for MOVE
\ doesn't work with line 0.
  DEX, DEX, DEX, DEX, 5 <X+> LDA, 3 <X+> STA, 4 <X+> LDA,
  SEC, $80 # SBC, 2 <X+> STA, IFNC, 3 <X+> DEC, THEN,
  3 <X+> LDA, 4 # CMP, IFNC, 7 # LDA, 3 <X+> STA, 2 <X+> LDA,
  $d8 # ADC, 2 <X+> STA, THEN,
  0 # LDA, 1 <X+> STA, 40 # LDA, 0 <X+> STA, ;CODE
: scroll A>R LINES 1- >R 80 BEGIN ( pos )
  DUP pos2a _ MOVE DUP 1+ pos2a _ MOVE 80 + NEXT DROP
  1840 ( 23*80 ) pos2a 40 $a0 FILL 1841 pos2a 40 $a0 FILL R>A ;
: NEWLN ( old -- new )
  DUP 23 = IF scroll ELSE 1+ THEN ;
```

B364

```
\ Apple IIe drivers, Floppy Drive
\ NOTE: this write 3 bytes over allocated space after N. This
\ might be a problem depending on how variables are arranged.
CODE _p ( blkno addr -- ) \ blkno = ProDOS 512b blk!
  3 # LDA, 'N <> STA, $60 # LDA, 'N 1+ <> STA, 0 <X+> LDA,
  'N 2 + <> STA, 1 <X+> LDA, 'N 3 + <> STA, 2 <X+> LDA,
  'N 4 + <> STA, 3 <X+> LDA, 'N 5 + <> STA,
  INX, INX, INX, INX, ;CODE
: _e S" FDErr " STYPE .x ABORT ;
LSET L1 DEX, DEX, 0 <X+> STA, 0 # LDA, 1 <X+> STA, X' _e JMP,
CODE _r $bf00 JSR, $80 C, 'N L, L1 BR BCS, ;CODE
CODE _w $bf00 JSR, $81 C, 'N L, L1 BR BCS, ;CODE
: FD@ ( blk blk( -- ) SWAP << TUCK 1+ OVER $200 + _p _r _p _r ;
: FD! ( blk blk( -- ) SWAP << TUCK 1+ OVER $200 + _p _w _p _w ;
```

B365

```
\ Apple IIe, SPI
CODE (spie) ( n -- )
  0 <X+> LDA, INX, INX, SPI_CTL () STA, ;CODE
CODE (spix) ( n -- n )
  0 <X+> LDA, SPI_DATA () STA, 0 # LDA, 1 <X+> STA,
  SPI_DATA () LDA, 0 <X+> STA, ;CODE
```