Collapse OS Documentation

This document was created at 2021-09-29 18:00 from documentation in CollapseOS snapshot 2021-09-24.

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 (theoretically thus far) 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 don't despair. There's a Forth primer in primer.txt Page 6.

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.

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 usage.txt Page 12 and impl.txt Page 18.

When you're ready to move to real hardware, read hw/intro.txt Page 80.

Other topics in this documentation

1.2 Forth Primer (`primer.txt`)

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.

```
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).

Dictionary

Forth's dictionary link words to code. On boot, this dictionary contains the system's words (look in dict.txt Page 25 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 defined *before* the overshadowing took place will still use the old word.

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

Cell size

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.

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).

Parameter Stack

Unlike most programming languages, Forth execute 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 number 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 definitions. You can't use IF directly in the interpreter loop.

Example usage:

```
: FOO IF 42 ELSE 43 THEN . ;
0 FOO --> 43
1 FOO --> 42
```

Loops

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

```
BEGIN..AGAIN --> Loop forever
BEGIN..UNTIL --> Loop conditionally
DO..LOOP --> Loop X times
```

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

DO pops the lower, then the higher bounds of the loop to be executed, then pushes them on RS. Then, each time LOOP is encountered, RS' TOS is increased. As long as the 2 numbers at RS' TOS aren't equal, we jump back to DO.

The word "I" copies RS' TOS to PS, which can be used to get our "loop counter".

Beware: the bounds arguments for DO are unintuitive. We begin with the upper bound. Example:

```
42 0 DO I . SPC> LOOP
```

Will print numbers 0 to 41, separated by a space.

You can use the word "LEAVE" to exit a DO..LOOP early. When used, it will finish the current loop and then stop looping when LOOP is reached.

```
: foo 5 0 DO I 3 = IF LEAVE THEN I . LOOP ;
foo \ prints 123
```

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 DO..LOOP, you have to
call UNLOOP 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. Explain-
ing the "culture of HERE" is beyond the scope of this primer,
but know that it's a very important concept in Forth. For examp-
le, new word definitions are written to HERE.

There is also "H", which is the address of "HERE" in memory.
Therefore, HERE's definition is "H @".
```

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-but 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 (or [TO] if you're inside a definition). 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. It is also significantly faster.

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
foo \ prints 42
```

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

We approach the end of our primer. 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 "DO" 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 (dict.txtPage 25) and try to understand
why this or that word is immediate. Good luck!

1.3 Collapse OS usage guide (usage.txt)

If you already know Forth, start here. Otherwise, read primer.txtPage 6 first.

We begin with a few oddities in Collapse OS compared to traditional forths, then cover higher level operations.

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.

Cell size and memory map

Cell size is hardcoded to 16-bit. Endian-ness is arch-dependent and core words dealing with words will read-write according to native endian-ness.

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 $80 bytes in size. If drivers need

   more memory, it's bigger. See impl.txt[Page 18] for details.
4. PS+RS: Typically around $100 bytes in size, PSP starts at
   the top and RSP starts at the bottom and both stacks "grow"
   against each other. When they meet, it's a stack overflow.

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

Number Literals

Whenever a word is parsed in the interpreter loop, we first try
parsing the word as a number literal. There are 3 literal types.

1. A 100% digits number is parsed as a decimal.
2. A string starting with $ is parsed as hexadecimal ($ab12).
3. A character inside quotes is parsed as that character ('A').

Strings and lines

Strings in Collapse OS are an array of characters in memory
associated with a 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 LIT" 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.

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 >".

Branching

Branching in Collapse OS is limited to 8-bit. This represents 64 word references 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.

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 generally a 64 character 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<? *TO 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 INPUF, 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.

Native words

Native words can be assembled in two ways.

With the proper assembler loaded in memory, you can compile words that directly execute native code. See doc/asm/intro.txt Page 60.

Otherwise, without anything loaded, you can use the HAL to generate native code. See doc/hal.txt Page 43.

Native words are created with CODE:

```
CODE foo 42 i>w, PUSHp, ;CODE \ same as : foo 42 ;
```

Native code can also be inlined in a regular word:

```
: foo 42 CODE[ INCw, ]CODE . ; \ prints 43
```

VALUE and TO

Cell access with @ becomes heavy in cases where a cell is read at many places in the code and seldom written to. It is also inefficient.

Collapse OS has a special "value" word type which is very similar to a cell, but instead of pushing the cell's address to PS, it reads the value at that address and pushes it to PS in a much faster and lighter way than "MYVAR @". You create such word with VALUE:

```
42 VALUE FOO
FOO . \ prints 42
```

Modifying that value is a bit less straightforward than with a regular cell, but can be done with TO:

```
43 TO FOO
FOO . \ prints 43
```

Traditional forths have CONSTANT, modern forths have CONSTANT and VALUE, Collapse OS only has VALUE.

There's an additional word that facilitates the declaration of multiple values: VALUES. You call it with the number of values to declare an then type down the associations, like this:

```
3 VALUES FOO 42 BAR $55 BAZ $1234
```

Values can be either a literal value or a single word yielding a single value on PS. If you want to assign a complex result to a value, you have to use VALUE.

To set a value in a compiled word, use [TO] instead of TO.

There is also *VALUE, which is an indirect version of VALUE. It contains an address, and calling it returns the value at that address. It has companion *TO and [*TO] words. To change the indirect value. Using TO on it changes the address.

Aliases

A common pattern in Forth is to add an indirection layer with a pointer word. For example, if you have a word "FOO" for which you would like to add an indirection layer, you would rename "FOO" to "_FOO", add a variable "FOO*" pointing to "_FOO" and re-defining "FOO" as ": FOO FOO* @ EXECUTE".

This is all well and good, but it is resource intensive and verbose, which make us want to avoid this pattern for words that are often used.

For this purpose, Collapse OS has two special word types: ALIAS and *ALIAS (indirect ALIAS).

An alias is a variable that contains a pointer to another word. When invoked, we invoke the specified pointer with minimal overhead. Using our FOO example above, we would create an alias with "' _FOO ALIAS FOO". Invoking FOO will then invoke "_FOO".

An alias is structured exactly like a VALUE and you can change the alias' pointer with TO like this:

' BAR TO FOO \ FOO now invokes BAR.

Aliases execution takes place in native code and is much, much faster than the "@ EXECUTE" form.

A ialias is like an alias, but with a second level of indirection, exactly like a *VALUE.

*ALIAS and *VALUE are used by core code which point to hardcoded addresses in RAM (because the core code is designed to run from ROM, we can't have regular variables). You are unlikely to need *ALIAS or *VALUE in regular code.

Mass storage through disk blocks

Collapse OS can access mass storage through its BLK subsystem. See doc/blk.txt Page 32 for more information.

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 B120 and you'll probably want to load some of them quite regularly because they make the system more usable.

Contexts

B122 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.

DOER and DOES>

In traditional forths, DOES> is often used with CREATE. Not in Collapse OS. To use the DOES> word, you must pair it with DOER. See doc/primer.txt Page 6 for details.

1.4 Implementation notes (`impl.txt`)

Execution model

At the end of BOOT, we call ABORT which triggers our main loop, which is in (main). 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). not null-terminated
- 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 ofer 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: set W to hardcoded DOES> addr, then CALL lbldoes, which pushes IP to RS, set IP to W and then JMP to next. The CALL to lbldoes had the same effect to PS as with cell.

value: set W to hardcoded value, then JMP to lblpush, which pushes W, then JMP to lblnext.

*value: same as value, but with an added indirection: hardcoded value is the address containing the value.

alias: directly JMP to hardcoded value.

*alias: alias with indirection.

Executing a XT (eXecution Tokens) word

At its core, executing a word is pushing the wordref on PS and calling EXECUTE. 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 wordref 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 wordrefs, but not all those wordrefs are 2 bytes in length. Some wordrefs 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 (s), (br), (?br) and (loop).

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...

Endian-ness

Unless explicitly noted, all 2 bytes numbers are written in the CPU's native endian-ness. For example, the Z80 and 8086 are little-endian, so they will write the low order byte before the high order one, but the 6809 is big-endian, so it will do the opposite.

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).

Those stacks are contiguous and grow in opposite directions. PS grows "down", RS grows "up".

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 the PSP is bigger than PS_ADDR, 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 RSP and PSP meet somewhere in the middle. 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 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. In fact, they're mostly referred to directly as their numerical offset along with a comment indicating what this offset refers to.

SYSVARS occupy $80 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       RESERVED
+08       LN< *ALIAS
+0a       NL characters
+0c       IP
+0e       EMIT *ALIAS
+10       KEY? *ALIAS
+12       CURWORD
+16       HAL ?JROP buffer
+18       PAD
+2e       IN(*
+30       IN>
+32       RESERVED
+34       RESERVED
+36       RESERVED
+38       BLK>
+3a       BLKDTY
+3c       RESERVED
+40       INBUF
+80       DRIVERS
```

CURRENT points to the last dict entry.

HERE points to current write offset.

IP is the Interpreter Pointer. Most CPU dedicate a register to it, but not all.

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

LN< *ALIAS 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 3 bytes buffer containing a reference to the word last read with WORD. First byte is length, the 2 others are the address to the character string.

The PAD is a space reserved in SYSVARS used for temporary storage at different places. Before you use it, make sure the words you use don't also use the same PAD space.

BLK* "Disk blocks" in usage.txt Page 12.

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 manually 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 ialiases default to (key?) and (emit) but can be overwritten to other routines.

DRIVERS section is reserved for recipe-specific drivers.

RESERVED sections are unused for now.

Initialization sequence

The first thing we do on powerup is to jump past the stable ABI and into the "early init" routine, written in native code. This does 2 things: initialize PSP and RSP and then jump to BOOT using the word offset recorded in the stable ABI.

Then, BOOT does this:

1. Initialize CURRENT and HERE from stable ABI.
2. Initialize system alises 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.

(main) is separate from BOOT because this word is also called by QUIT. This way, when we ABORT during a LOAD (for example), we go back to a usable prompt instead of being stuck in an input nightmare maze.

Stable ABI

The Stable ABI lives at the beginning of the binary and provides a way for Collapse OS code to access values that would otherwise be difficult to access. Here's the complete list of these references:

```
04 BOOT addr          06 CURRENT          08 LATEST
0a (main) addr
```

BOOT and (main) exist because they are referred to before those words are defined (in core words).

CURRENT and LATEST are initial values for CURRENT and HERE in SYSVARS.

All Collapse OS binaries, regardless of architecture, have those values at those offsets of them. Some binaries are built to run at offset different than zero. This stable ABI lives at that offset, not 0.

Input buffer (INBUF)

As indicated above, the Input Buffer lives in SYSVARS and is $40 bytes in length (configured by 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 RDLN<. 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 C<. If you type LNSZ characters without typing CR, an additional CR will be fed to C< after INBUF has gone through.

1.5 Dictionary (`dict.txt`)

List of words defined in arch-specific boot code (for example, B280 for Z80) and Core words (B210).

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.

Word references (wordref): When we say we have a "word reference", it's a pointer to a word's *entry type field*. For example, the address that "' DUP" puts on the stack is a wordref, that is, a reference to the entry type field of the word DUP. See impl.txt Page 18 for details.

```
PF: Parameter field. The area following the entry type field of
a word. For example, "' H@ 1+" points to the PF of the word H@.

Words between "()" are "support words" that aren't really meant
to be used directly, but as part of another word.

"*I*" in description indicates an IMMEDIATE word.
"*F*" in description indicates an word that sets Flags.
```

Symbols

```
Throughout words, different symbols are used in different
contexts, but we try to been 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 ...?

Placement of those symbols is important. To the left of the
words, it refers to inputs and to the right, the output.
```

Values and aliases

Values and aliases (see usage.txt Page 12) can be handled with those words:

```
TO x     n --         Write n to value or alias x.
[TO] x   n --         *I*. Same as TO.
*TO x    n --         Write n to ivalue or ialias x.
[*TO] x  n --         *I*. Same as *TO.
VAL!     n a --       Write n to value or alias a.
*VAL!    n a --       Write n to ivalue or ialias a.

Collapse OS' core has a few *indirect* values and aliases which
can be manipulated with *TO, [*TO] and *VAL!.

Values:

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.
NL       1 or 2 chars to spit during NL>, MSB first. If MSB is
         0, it's ignored.

Aliases (see further below for description):

BLK@*
BLK!*
LN<
EMIT
KEY?
```

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     a --   Define a new alias with a starting value of a
*ALIAS x    a --   Define a new ialias where the indirection
                   points to a (actual alias in *a).
CREATE x    --     Create cell named x. Doesn't allocate a PF.
[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.
VALUE x     n --   Creates cell x that when called pushes its
                   value.
VALUES x    n --   Create a serie of x values. See usage.txt Page 12
DOER        --     See primer.txt Page 6
DOES>       --     See primer.txt Page 6
IMMEDIATE   --     Flag the latest defined word as immediate.
LITN        n --   Write number n as a literal.
```

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.
x y DO .. LOOP: LOOP increments y. if y != x, branch to DO.

(          --   *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" x"  --   *I* Compiles a ." followed by a ABORT.
EXECUTE    a -- Execute wordref at addr a
INTERPRET  --   Main interpret loop.
LEAVE      --   In a DO..LOOP, exit at the next LOOP call.
UNLOOP     --   Remove loop counters from RS before an early
                EXIT.
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
2>R         x y -- R:x y    Equivalent to SWAP >R >R
R>          R:n -- n        Pops RS and push to PS
2R>         R:x y -- x y    Equivalent to R> R> SWAP
I           -- n            Copy RS TOS to PS
```

Stacks meta

```
'S -- n Current address of PSP
'R -- n Current address of RSP
.S --   Prints stack information as well as the contents of PS.
S0 -- n Start address of PS
R0 -- n Start address of RS
```

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.
[]=      a1 a2 u -- f Compare u bytes between a1 and a2. Returns
                      true if equal.
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.
ALLOT    n --         Move HERE by n bytes.
ALLOT0   n --         ALLOT and fill with zero.
FILL     a n b --     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 nig-endian regardless of
                      native endianess (M=MSB first)
MOVE     a1 a2 u --   Copy u bytes from a1 to a2, starting
                      with a1, going up.
MOVE,    a u --       Copy u bytes from a to HERE.
PAD      -- a         a is a safe space for tmp stuff.
RANGE    a u -- ah al Get low/high bounds of an addr+length
                      pair. Useful for DO..LOOP.
```

Arithmetic / Bits

```
+           a b -- a+b
-           a b -- a-b
-^          a b -- b-a
*           a b -- a*b
/           a b -- a/b
<<          a -- a<<1
<<8         a -- a<<8
>>          a -- a>>1
>>8         a -- a>>8
L|M         n -- lsb msb    Split n word in 2 bytes, MSB on TOS
1+          n -- n+1
1-          n -- n-1
```

```
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      a u -- a<<u
RSHIFT      a u -- a>>u
```

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
=><= n l h -- f Push true if l <= n <= h
NOT  f -- f     Push the logical opposite of f
```

Strings and lines

See doc/usage.txt Page 12 for the concepts of strings and lines.

```
LIT" x" --     Read following characters and write to HERE as a
               string literal.
LNLEN   a -- n Return length of line at a.
S=      sa1 sl1 sa2 sl2 -- f
        Returns whether string s1 == s2.
```

I/O

```
.        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
," xxx" --        Write xxx to HERE
." xxx" --        *I* Compiles string literal xxx 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 --      EMIT line at addr a
IN<     -- c      Read one char from buffered input.
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
        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 --  EMIT all chars of string.
WORD    -- sa sl
        Read one word from buffered input and push it.
        That word is a string (begins with a length byte).

These ASCII consts are defined:
EOT BS CR LF SPC

KEY? and EMIT are *ALIAS to (key?) and (emit) (see TTY protocol
in protocol.txt Page 54). 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 18.
```

BLK subsystem (see doc/blk.txt)

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

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 --    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 0.1 millisecond. Don't
                  use with n=0.
```

Loaders

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

ED     Block Editor
VE     Visual Editor
ASM    Assembler common words
Z80A   Z80 assembler
8086A  8086 assembler
AVRA   AVR assembler
8086A  8086 assembler
RSH    Remote shell and XMODEM implementation
AVRP   AVR programmer
XCOMPL Cross-compilation "low" part. XCOMPH (and many others) is
       defined in XCOMP itself.
```

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".
```

1.6 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.

Exploring blocks

Blocks 0 and 1 in Collapse OS are text blocks 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. You can work your way around following these instructions, or you can take the easy way: application loaders. The BLK subsystem has convenience words for loading applications at B234.

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.

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 and B1 are for a master index of blocks.
2. B2-B100 are for assemblers.
3. B100-B199 are for programs loaded at runtime.
4. B200-B299 are for arch-independent cross-compiled code, including xcomp tools.
5. B300+ is for arch-specific code.

In the POSIX package of Collapse OS, arch-specific code is kept in separate ".blk" 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 XCOMPH 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.). See doc/usage.txt Page 12 for usage.

Including the BLK subsystem in a kernel

Before assembling, this requires 3 words:

BLK_ADDR: where the 1024 bytes block buffer will live. The xcomp unit defines it as SYSVARS - 1024 by default.

(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.

1.7 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 B100.

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 ubsystem. 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.

Why "i" and not "I"? Because "I" is already used and we don't want to overshadow it.

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 inserted 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.
```

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.txt Page 55)

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". 't' opens the previously opened block.

'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. 'B', backwards.

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

'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.

'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 combination 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.

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

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

'!' writes the current block to disk.

'q' quits VE

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.

1.8 Memory Editor (me.txt)

The Memory Editor at B115, 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.

1.9 Remote Shell (rsh.txt)

You can control a remote system from Collapse OS using the Remote Shell application at B150. All you need is a ACIA driver (see doc/hw/acia.txt Page 81).

With a proper driver in place, you can load the Remote Shell at B150.

Then, it's only a matter of running "rsh". This 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 temporarily) 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.

1.10 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 71)

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 )
```

1.11 Harmonized Assembler Layer (`hal.txt`)

```
The HAL is a set of macros whose goal is to make it possible to
make some parts of high-level code faster without having to
write code for each platform. The secondary goal of it is to
reduce the size of the native part of each port by moving words
that aren't critical into HAL code.

The idea is that each supported CPU has a set of code-emitting
macros that all follow the same API.

The HAL is *not* for general purposes. It is specially designed
for Collapse OS itself. Trying to generate another kind of
binary with it is likely a bad idea.
```

Ops targets

```
This API revolves around PS, RS, IP, as with the rest of Forth,
but also give more control over what goes to W, the Working
register and how the stacks are managed.

Operation names always specify the elements they affect with a
list of suffixes, which are:

w: Working Register. The most important element. Most
   computation happens there.
p: Top Of PS. It is used, of course, for push/pop/transfer
   operations, but also for arithmetics. It is always the
   "secondary" element of the op. The result goes in w.
r: Top of RS. Rarely used for things other than push/pop/
   transfer. The RS is often assigned to a more "awkward"
   register in the CPU, making operations with it more
   expensive.
f: (for "far") Second element of PS. Often used for complex
   stack juggling.
i: immediate number which will be written next to the op. When
   an op has this signature, it means you have to supply it
   with a number at compile time.

Operations that work on a single element will work on W, not PS
or RS. Therefore, you have to POP or copy from PS or RS, do your
thing, push/copy back.

Some operations work on more than one elements at once. They
will then work on W and PS/RS/IP explicitly. Their name and
```

description will tell it.

There are also some exceptions, such as INCp, which works directly on PS, without W. The idea is to take advantage of the TOS register on CPUs that have it, or CPUs that have more flexible access to their stacks in memory.

Ops costs

The HAL hides the cost of underlying operations, which is different from CPU to CPU. The HAL has been designed to work well with Collapse OS' darling CPU, the z80, but is also designed to avoid ridiculously expensive operations.

It is useful to be aware of costs saving operations however. For example, POPp, (do stuff) PUSHp, can work very well and might be fast on some CPUs, but on the z80 which uses BC as its TOS register, it's faster to copy BC to HL and back again. For this reason, there are the p>w, and w>p, ops (which mean "copy from p to w" and "copy from p to w") and if you use them instead of POPp,/PUSHp, when you don't need them, your code can be significantly faster.

Macro arguments

HAL macros generally require no argument. It's only when they have a "i" signature (immediate) that they need to be supplied with a number to hardcode in their instructions.

Jumps

Jump words all require a numerical argument which will be written next to them. For absolute jumps, it's easy: You use labels. For example, you use "LSET mylabel" to mark, then "mylabel JMPi," to jump.

Relative jump offsets are computed with "BR" and "FMARK". You can use either labels or BEGIN, markers. Examples:

```
LSET L1 INCw, L1 BR JRi, \ backward jump with label
BEGIN, INCw, BR JRi, \ backward jump with BEGIN,
FJR JRi, INCw, FMARK \ forward jump with FJR
FJR JRi, TO L1 INCw, L1 FMARK \ forward jump with labels
```

The HAL convenience layer also has structures jumps helpers:

```
IFZ, INCw, ELSE, DECw, THEN, \ also: IFNZ, IFC, IFNC,
```

```
42 i>w, BEGIN, DECw, Z?w, BR JRNZi, \ loop 42 times
```

Jump arguments

Wrapping one's head around jumping can be a challenge, especially with a cross-CPU API. For JMPw and JMPi, it's rather easy: the immediate or w are expected to contain an absolute address to jump to.

For relative jump words (JRi, JRNZi, etc.), things must be clarified: The expected argument is a CPU-specific offset. You will seldom determine that offset yourself, but will instead use BR and FMARK, which will use the CPU-specific JROFF and JROPLEN constants to compute a proper offset.

Conditional jumps

The HAL only allows relative conditional jumps through the ?JRi, word. The jump can be conditional to the value of two flags: Zero and Carry. A condition must be selected before ?JRi, is used. The words to select the condition are Z? (jump if Z is set) C? (jump if C is set) and ^? (invert jump condition).

On "good old" register based CPUs, these words don't emit anything, they simply select the opcode that ?JRi, will emit. On stack-based CPUs, things are different because the conditional jump is generally based on a flag placed on the stack, so these words will emit the code that will place the proper value on the stack.

The flag selection word has to be called before BR/FJR. The proper form looks like this:

```
Z? L1 BR ?JRi,
C^? FJR ?JRi,
```

Convenience words like IFZ, IFNC, etc. take care of selecting the proper flag.

HAL and high level flow words

The HAL implements all words required for ASMH, which implements high level flow words such as IFZ, IFC, THEN, etc.

Those words are often bundled together, and alone, in the same block because they're often re-used by assemblers.

See doc/asm/intro.txt[Page 60] for details.

Important assumptions

Some things are considered the same across all supported CPUs:

* JMPi always yield 3 bytes.
* i>w yields a single opcode, followed by a 2-byte number.
* (i)>w yields a single opcode, followed by a 2-byte address.

Macros

The "," suffix is omitted below, but it's always there to indicate that the effect of these word is to write (,) stuff to HERE.

Affected elements: as a general rule, w's value is always modified after an op. It's generally affected to the accumulator register of the CPU which is pretty much always needed to do meaningful stuff.

Except of course simple copy ops such as "w>p" and friends. They never change w.

Some ops explicitely save w though. Those are mentioned in the description with "Saves w".

In the same way, p and r are generally preserved by ops. Some ops, however, can't really achieve this feat without making the op expensive, so it destroys p and/or r. When that happens, the description mentions it with "Destroys p/r".

On many CPUs, Z and C flags are set after a large part of the arithmetic ops, but that can't be assumed in the HAL. C and Z flags are set only after an op that specifies it, and it can't be assumed anymore after running another op.

```
## stack
POPp      Pop PS TOS to w
POPr      Pop RS TOS to w
POPf      Pop PS 2nd element to w
PUSHp     Push w to PS TOS
PUSHr     Push w to RS TOS
PUSHf     PUSH w to PS 2nd element
DUPp      Push PS TOS to PS. Saves w
DROPp     Pop PS TOS. Saves w
SWAPwp    Swap w and PS TOS
SWAPwf    Swap w and PS 2nd element
```

```
## transfer
p>w       Copy PS TOS to w
w>p       Copy w to PS TOS
i>w       Load immediate value to w
C@w       w being an address, load its byte in w
@w        w being an address, load its word in w, native endian
C!wp      w being an address, write p's LSB to memory. Saves w
!wp       w being an address, write p to memory, native endian
w>IP      Copy w to IP (Interpreter Pointer)
IP>w      Copy IP to w
IP+       Increase IP by 1. Saves w
IP+off    IP pointing to a signed byte, increase IP by this byte
          Saves w.

## CPU flags
w>Z       Set CPU's Z flag according to w's value. Saves w
p>Z       Set CPU's Z flag according to p's value. Saves w
Z>w       Set w to 1 if Z is set, 0 otherwise
C>w       Set w to 1 if C is set, 0 otherwise

## Jump

Jumps never affect w.

JMPw      Inconditional jump to PC w
JMPi      Inconditional jump to PC i
CALLi     Push PC+X to PS, then JMPi. X=length of CALL op
JRi       Inconditional jump with offset i
JRZi      Jump with offset i if Z is set
JRNZi     Jump with offset i if Z is unset
JRCi      Jump with offset i if C is set
JRNCi     Jump with offset i if C is unset

## Arithmetic

INCw      Increase w by 1
INCp      Increase p by 1. Saves w
DECw      Decrease w by 1
DECp      Decrease p by 1. Saves w
CMPwp     Set Z if p==w. Set C if w<p. Saves w
ANDwp     Make w the bitwise AND of w and p
ORwp      Make w the bitwise OR of w and p
XORwp     Make w the bitwise XOR of w and p
XORwi     Make w the bitwise XOR of w and i
+wp       Add p to w. Sets Z/C flags.
-wp       Subtract p from w. Sets Z/C flags.
>>w       Shift w right by 1 bit. Sets C flag.
<<w       Shift w left by 1 bit. Sets C flag.
>>8w      Shift w right by 1 bit.
<<8w      Shift w left by 1 bit.
```

1.12 Cross-compilation (`cross.txt`)

There is two general types of Forth code in Collapse OS: regular code and code designed for cross-compilation. Both types are very similar in form, but they're not always interchangeable.

All blocks before B200 contain regular code. It's regular stuff, nothing special about it. The rest, however, is "xcomp" code and things can get tricky in there. Let's dive in...

What is xcomp for?

When Forth words are compiled, they are compiled for the system currently running. Those compiled words are tricky to relocate because their wordrefs reference addresses within the running system.

If you want to deploy to a new system, you need tricks, and those tricks are located at B200, the cross-compilation toolset.

The mechanism is simple: override defining words (:, CREATE, VALUE, etc.) so that it adds an offset to every wordrefs it compiles.

What should that offset be? the beginning of the binary being built. That offset is the value in the ORG variable, supplied by the assembler. It's logical: every binary begins with a bit of assembler, which makes every following Forth word aligned with this value.

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 our cross binary begins at offset $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.

xcomp unit

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

The xcomp toolset at B200 alters core words in a deep way, so ordering is important. First, we load our tools. Assembler, xcomp toolset, etc.

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 xcomp overrides. From this point on. every defining word is messed up and will produce offsetted binaries.

At this point, it's critical to set ORG before spitting anything. Boot binaries will usually take care of this, so you don't have to do it yourself. You just have to make sure that you load the boot binary right after loading B200.

If your binary is not located at 0 on the target machine, you have to set BIN(. For example, if your target OS is designed to run from offset $3000, you'll add "$3000 TO BIN(" to your xcomp unit.

Then, you spit your content following instructions in bootstrap.txt Page 51.

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 guest, 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 guest 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 .".

How to know whether a word can be used?

1. If it's not an immediate compiling word, it's fine.
2. If its overriden in B200-B205, it's fine.
3. Otherwise, you can't cross-compile it.

List of words that are known to *not* work in code having to be cross-compiled:

* DOES>: words with DOES> in it can be cross-compiled, but not used. ": FOO CREATE DOES> ;" is fine, but *not* "FOO BAR" afterwards!

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, all assemblers define these words:

|T: Split word into 2 bytes, using Target's endian-ness. Calls |L or |M
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 B200 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!).

1.13 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!

1.14 Bootstrap guide (bootstrap.txt)

You want to deploy Collapse OS on a new system? Read usage.txt[Page 12], impl.txt[Page 18], cross.txt[Page 48], then continue here.

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 boot words (B304 for Z80)
3. Arch-independant core words (low) (B210)
4. Runtime HAL
5. Drivers, might contain arch-specific code
6. Arch-independant core words (high) (B226)

Boot code

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

1. A jump to the early initialization routine

2. The stable ABI (see doc/impl.txtPage 18)
3. Core routines. At this point, lblnext, lblxt, lblcell, lblpush and lbldoes are set.
4. The early initialization routines which initialized PSP and RSP and then executed BOOT from its address in the stable ABI.

Boot words

Then come the implementation of core Forth words in native assembly. This is a very limited set of words that aren't written in HAL either for speed reasons, or because they do things that the HAL doesn't allow (PSP/RSP manipulations mostly). The list of boot words is:

QUIT ABORT BYE RCNT SCNT FIND * /MOD []= TICKS

On CPUs having ports, PC! and PC@ are also native.

Sometimes, these words contain HAL words for convenience. Jumps and flow words in particular are very often done using the HAL, which yields the same binary code as "real" native jumps but is more convenient to write.

All other words are written in cross-CPU HAL or Forth.

Core words (low)

Then comes the part where we begin defining words in Forth. Core words are designed to be cross-compiled (B200), from a full Forth interpreter. This means that it has access to more than boot words. This comes with tricky limitations.

See doc/cross.txt Page 48

Some of those words contain native code, but written in cross-CPU HAL. See doc/hal.txt Page 43

Runtime HAL

We usually want to include a runtime HAL in our binary so that we can compile HAL code without needing to load the HAL first. Many runtime applications (it's actually not true at the time of this writing, but will soon be) have HAL optimizations, so they need a HAL to be loaded.

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 code we want to include in the binary.

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.

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.

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 impl.txt Page 18.

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

RS_ADDR is where RSP starts and PS_ADDR is where PSP starts. RSP and PSP are designed to be contiguous. RSP goes up and PSP goes down. If they meet, we know we have a stack overflow.

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

First come the assembler (example, Z80A). There are core loader words for all supported assemblers.

Then comes XCOMPL (xcomp "low"), which is the early part of the xcomp toolkit and doesn't override important words. So, at this point, we're still in "host" mode. What we need in XCOMPL are

all xcomp-related loader words such as Z80M, Z80C, etc.

Then comes CPU-specific macros, constants further loader words such as the "H" and "C" units. They always live in B301 (see doc/blk.txt Page 32). So, "301 LOAD". It's important that it's loaded before XCOMPH because it's being executed during xcomp, not included in the target binary.

The comes the CPU-specific HAL, the "H" unit (example: Z80H). This comes before XCOMPH, will override the host's "native" HAL. This is not the runtime HAL.

Now comes the real deal: XCOMPH. It's the "high" 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.txt Page 48).

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)".

At this point, things are weird: we load the target's HAL again! that's because this time, it's the runtime HAL.

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 ORG 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!

1.15 Protocols (`protocol.txt`)

Some subsystems (and in the case of KEY and EMIT, the core) require drivers to implement certain words in a certain way. For example, the core requires drivers to implement (key?) and (emit) or else it won't know how to provide a console.

These protocols are described here.

TTY protocol

```
(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.
```

PS/2 protocol

This protocol enables communication with a device that spits PS/2 keycodes.

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

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)
```

Grid protocol

See grid.txt Page 55

1.16 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:
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 -- Update u contiguous cells, starting at pos,
                     using characters starting at address a.
```

"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.

1.17 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 12), so I/O indirections will work.

1.18 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.

TWith 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.txt Page 51.

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 *TO HERE
```

Then, you can continue the process normally.

1.19 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.txt Page 12 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"). After that, you have to set it up before spitting opcodes. More specifically, you might have to set ORG and BIN(variables.

ORG, defaulting to 0, specifies where the binary begins in memory. It allows the PC word to return the proper value. Generally, when you're ready to spit upcodes, you run "HERE TO ORG" so that PC is set to 0.

BIN(, defaulting to 0, specifies where the resulting binary lives in memory. If all you spit are relative jumps, it doesn't matter, but if you need to jump to an absolute address, BIN(needs to be correct. Note that ;CODE spits an absolute jump in many arches, so BIN(often needs to be correct.

If you compile for a "live" target (the computer running Collapse OS), you don't need to set ORG and BIN(.

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 have to main options.

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 INCd, ;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 ENTRY head.

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

Another option is "inline assembler". When you're in a tight
spot inside a word that you'd like to be faster, but that
creating a whole word is too much, you can use CODE[. Example:

; foo 42 CODE[ BC INCd, BC INCd, ]CODE . ; \ prints 44

This example above would be significantly faster than "2 +".
At runtime, the overhead, in terms of speed, is the same as a
regular CODE word, but in terms of binary size, it's better.
```

Usage

```
To spit binary code, use opcode words such as "LDrr," in the
Z80 assembler which spits LD in its "r1, r2" form. Unlike
typical assemblers, operation arguments go before the opcode
word, not after it. Therefore, the "LD A, B" you would write in
a regular assembler becomes "A B LDrr,"

Those opcode words, of which there is a complete list in each
arch-specific documentation, end with "," to indicate that their
effect is to write (,) the corresponding opcode.

The "argtype" suffix after each mnemonic is needed because the
assembler doesn't auto-detect the op's form based on arguments.
It has to be explicitly specified.

Although efforts are made to keep those argtypes consistent
across arches, there are differences. Arch-specific doc has
precise definitions for those argtypes.

For example, in Z80 asm, "r" is for 8-bit registers, "d" for
16-bit ones, "i" for immediate, "c" is for conditions.
```

Labels and flow

```
All assemblers and HALs implement standard flow words:

JRi, ( off -- ) \ relative unconditional jump
?JRi, ( off -- ) \ relative conditional jump
Z? \ make Z the condition
C? \ make C the condition
^? \ invert condition
JMPi, ( addr -- ) \ unconditional absolute jump
```

```
CALLi, ( addr -- ) \ unconditional absolute call
```

See doc/hal.txt[Page 43] for details about those words.

```
The ASMH (asm common words, "high" part) builds upon those words
to implement useful structured flow words:

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 Z? ?JRi, \ loop if Z is set
FJR JRi, .. THEN, \ unconditional forward jump

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

To avoid using dict memory in compilation targets, we pre-
declare label variables here, which means we have a limited
number of it. We have 3: L1, L2, L3.

You can define your own labels with a simple "0 VALUE lblname",
but you have to do so before you begin spitting opcodes.
```

Endian-ness

```
As explained in cross.txt[Page 48], all assembler supply words allowing
to write 16bit numbers in a target's endian-ness. Common words
at B2 already supply these words and they're all dependent on
the BIGEND? variable which defaults to 0. Assemblers for big-
endian architectures have to set this to 1.
```

2.2 Z80 assembler specificities (asm/z80.txt)

```
Load with "Z80A".

Mnemonics having only a single form, such as PUSH and POP,
don't have argtype suffixes.

Be aware that "SP" and "AF" refer to the same value: some 16-
bit ops can affect SP, others, AF. If you use the wrong argu-
```

```
ment on the wrong op, you will affect the wrong register.
```

Flow examples

```
IFZ, NOP, ELSE, NOP, THEN,
BEGIN, NOP, JR, AGAIN, ( unconditional )
BEGIN, NOP, JRZ, AGAIN, ( conditional )
LSET L1 NOP, L1 BR JRi, ( backward jump )
FJR JRi, TO L1 NOP, L1 FMARK ( forward jump )
```

IX+, IY+

```
IX/IY instructions are a bit complicated.  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 LDdi,
HL ADDIXd,

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 LDIXY, ( ld e, (ix+0) )
-2 IY+ INC(IXY+), ( inc (iy-2) )
```

Instructions list

```
Letters in [] brackets indicate "argtype" variants. When the
bracket starts with ",", it means that a "plain" mnemonic is
available. For example, "RET," and "RETc," exist.

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
(i) => memory reference (both 8b and 16b)

LD  [rr, ri, di, (i)HL, HL(i), d(i), (i)d, rIXY, IXYr,
    (DE)A, A(DE), (i)A, A(i)]
ADD [r, i, HLd, IXd, IXIX, IYd, IYIY]
ADC [r, HLd]
```

```
CP  [r, i, (IXY+)]
SBC [r, HLd]
SUB [r, i]
INC [r, d, (IXY+)]
DEC [r, d, (IXY+)]
AND [r, i]
OR  [r, i]
XOR [r, i]
OUT [iA, (C)r]
IN  [Ai, r(C)]
JP  [, c, (HL), (IX), (IY)]
JR  [, Z, NZ, C, NC]
CALL[, c]
RET [, c]

PUSH       POP
SET        RES         BIT
RL         RLC         SLA         RLA         RLCA
RR         RRC         SRL         RRA         RRCA
RST        DJNZ
DI         EI          EXDEHL      EXX         HALT
NOP        RETI        RETN        SCF
CPI        CPIR        CPD         CPDR        IM0
IM1        IM2         INI         LDI         LDIR
LDD        LDDR        NEG         OUTI

Macros:

SUBHLd     Clear carry + SBCHLd
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
```

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[],". List of modrm arguments:

[b]    Indirect byte
[w]    Indirect word
[b]+   Indirect byte + displacement (8b)
[w]+   Indirect word + displacement (8b)
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)

NOT ALL COMBINATIONS ARE LEGAL. The assembler will happily
assemble mod/rm for instructions that don't support it. Only use
mod/rm arguments with instructions that support them.

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

The instructions list below specify available mod/rm forms for
each mnemonic.
```

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 43
```

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
[] -> [SI] [DI] [BP] [BX] [BX+SI] [BX+DI] [BP+SI] [BP+DI]

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,[w],[b],[w]+,[b]+]
DEC[r,x,[w],[b],[w]+,[b]+]
POP[x,[w],[w]+]
PUSH[x,[w],[w]+,s]
MUL[r,x]
DIV[r,x]
XOR[rr,xx]
OR[rr,xx]
AND[rr,xx,ALi,AXI]
ADD[rr,xx,ALi,AXI,xi]
SUB[rr,xx,ALi,AXI,xi]
INT

CMP[rr,xx,ri,xi,xI,r[],x[],r[]+,x[]+]
MOV[rr,xx,r[],x[],[]r,[]x,r[]+,x[]+,[]+r,[]+x,ri,xI,sx,rm,xm
    mr,mx]

("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.

Any flow word with implicit branching type (such as IF,) only supports 8-bit branching. However, any time you specify a "L" type of branching op (LBRA, for example), the following "offset word" (BWR, FWR, or AGAIN,) will do the right thing and work with 16-bit. Therefore, all of these are correct flows:

```
LSET L1 NOP, L1 BR BRAi,
LSET L1 NOP, L1 LBRAi,
FJR BRAi, TO L1 NOP, L1 FMARK
BEGIN, NOP, BR BRAi,
BEGIN, NOP, LBRAi,
```

BR, LSET, FMARK come from the HAL convenience layer, see doc/hal.txt Page 43

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 (B53)
ASR  COM   DEC   INC   LAC   LAS   LAT   LSR   NEG   POP   PUSH
ROR  SWAP  XCH

OPRdRr (B54)
ADC  ADD   AND   CP    CPC   CPSE  EOR   MOV   MUL   OR    SBC
SUB

OPRdA (B54)
```

```
IN   OUT

OPRdK (B55)
ANDI CPI   LDI   ORI   SBCI   SBR   SUBI

OPAb (B55)
CBI   SBI   SBIC  SBIS

OPNA (B56)
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 (B57)
BCLR  BSET

OPRdb (B57)
BLD   BST   SBRC  SBRS

Special (B57,B60)
CLR   TST   LSL   LD    ST

Flow (B58)
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 (B61)
LBL! LBL, SKIP, TO, FLBL, FLBL! BEGIN, AGAIN? AGAIN, IF, THEN,
```

3 How to read the code

3.1 How to read this code (`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.

Core routines

At the heart of every port are the "core routines" of Collapse OS. These are called all the time and their optimization is paramount. They are, however, very small in scope and all fit in a single block.

When you see labels lblnext, lblcell and friends being defined, you're in core routines territory. The goal of these core

routines is to support all word types as described in doc/impl.txtPage 18.

HAL and Reserved registers

Most of Collapse OS' native code is written using the HAL. See doc/hal.txtPage 43 first.

If you are reading real CPU-specific native code, you should be aware of that CPU's register roles. See doc/impl.txtPage 18 and the the
CPU-specific document of doc/code/.

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" it points to the word's type byte.
* "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.
* Before a DO, when range arguments come from PS, we often add
  a comment describing which "variable" is used for range.
  Example: "( cnt ) 0 DO ... LOOP". We can also see this
  pattern sometimes in assembler code when writing an hardcoded
  address put on PS at compile-time.
```

Driver code

```
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. There-
fore, 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.

## 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 51 for details.

RESERVED REGISTERS:

* HL is the Work register
* SP points to PSP
* IX points to RSP
* DE hold IP (Interpreter Pointer)
* BC holds PSP's Top Of Stack value

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 18]), 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 18].

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 18].
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.

(c): That's a native code literal for CODE[. The byte next to it
has the same meaning as in (br) except it's unsigned (it always
goes forward). What we do it that we increase DE by that amount,
and then jump to the next byte, where the native code begins.
Then the native code will call the next routine, DE will already
be set to where we want to continue.

*: 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 51 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 51 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 18), 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.

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.txt Page 51 for details, but the short version is:

When instructions tell you to declare constant XXX, then load drivers from block BYYY and then add word ZZZ to the initialization string, what is meant is that:

1. At the top of your xcomp unit, add constant XXX next to other declarations.
2. Between "XCOMPL" and "XCOMPH" loading, insert the loading of BYYY. Order may matter.
3. In INIT word definition, call ZZZ. Order may matter.

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 86), then you almost have all
it takes to make an ATmega328P blink.

First, read doc/avr.txt Page 40. 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 71 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 40 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 Accessing SD cards (hw/sdcard.txt)

SD cards support the SPI protocol. If you have a SPI relay (doc/hw/spi.txt Page 86) and a driver for it that implement the SPI protocol (doc/protocol.txt Page 54), 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 "423 436 LOADR".

Initialization of the SDC system is done in multiple steps. First, the BLK system needs to be initialized with "BLK$". Then you can plug SDC@ and SDC! into BLK with "' SDC@ ' BLK@* **!" and "' SDC! ' BLK! **!". That only needs to be done once per boot.

Then, 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.

4.6 Communicating through SPI (hw/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 86 in arch-specific folders for more information.

4.7 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.

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 88), 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 89) hooks everything together. CE comes from the z80 bus (img/ps2-z80.png Page 90).

The 595 (img/ps2-595.png Page 89) 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

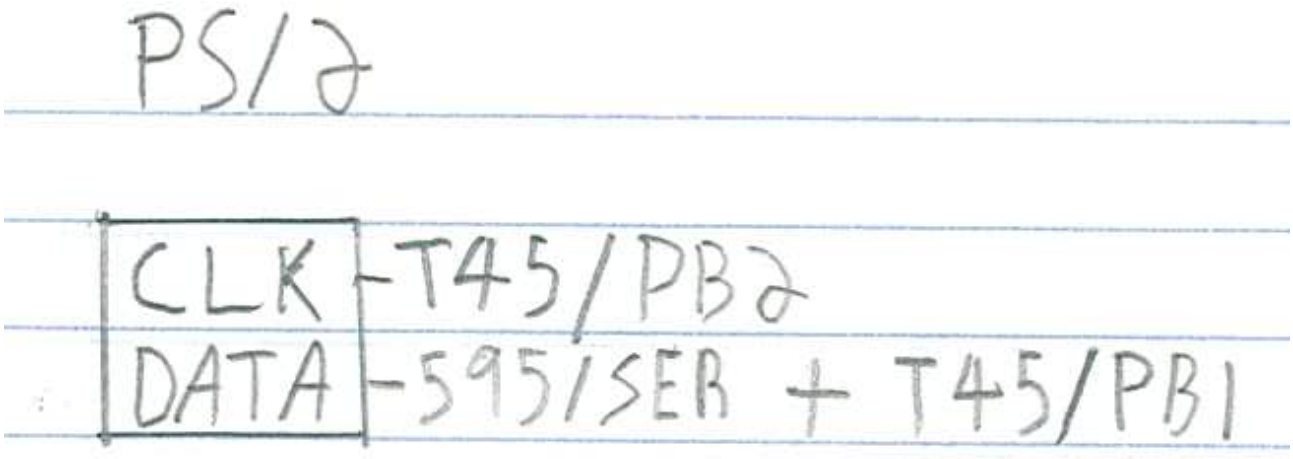
To use this interface, you have to build a new Collapse OS binary. This binary needs two things.

First, we need a "(ps2kc)" routine (see doc/protocol.txt Page 54). 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".

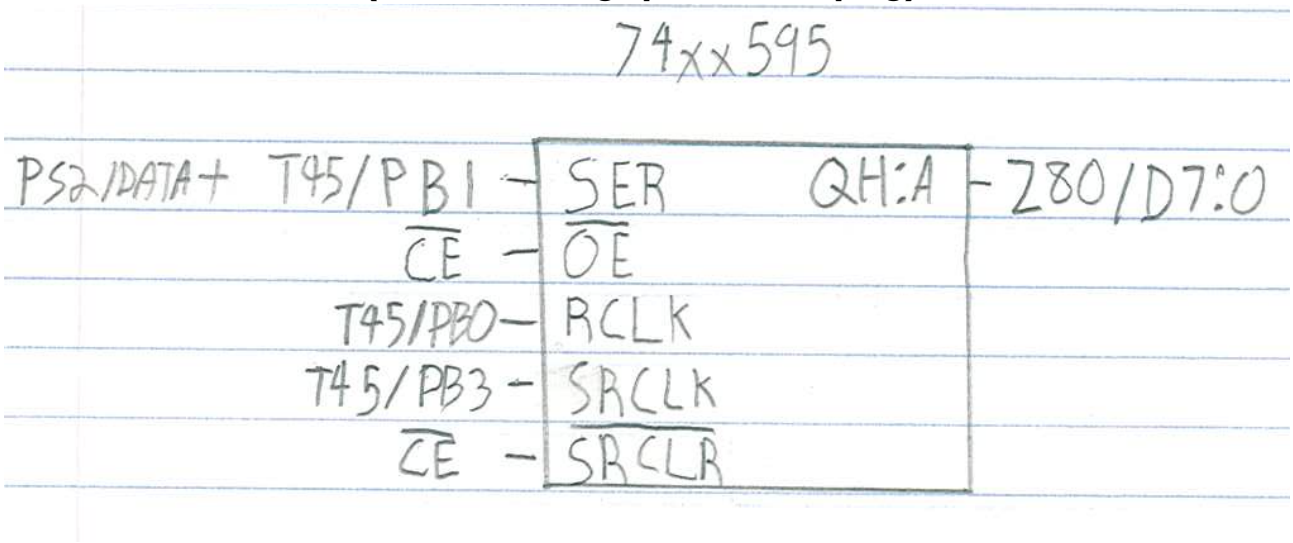
The PS/2 subsystem provides "(key)" from "(ps2kc)".

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.

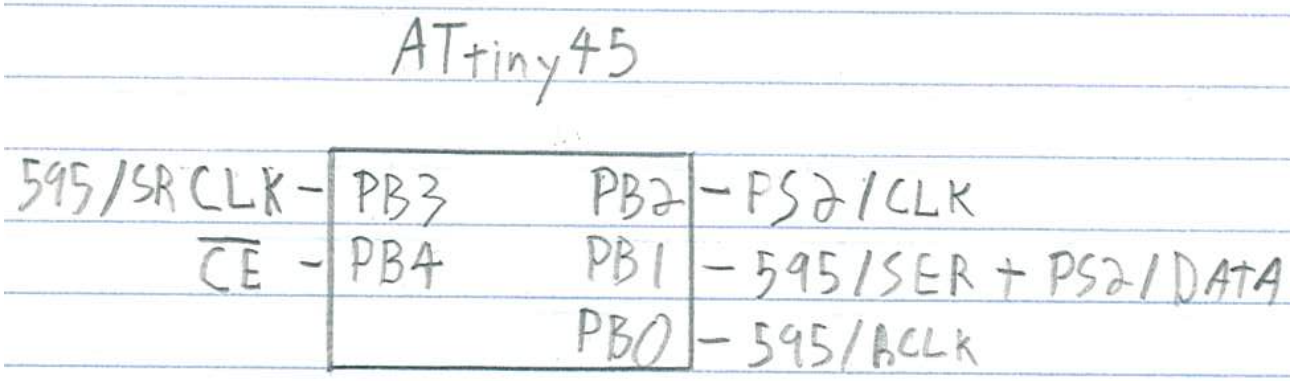
5.2 PS/2 Connector (hw/z80/img/ps2-conn.png)



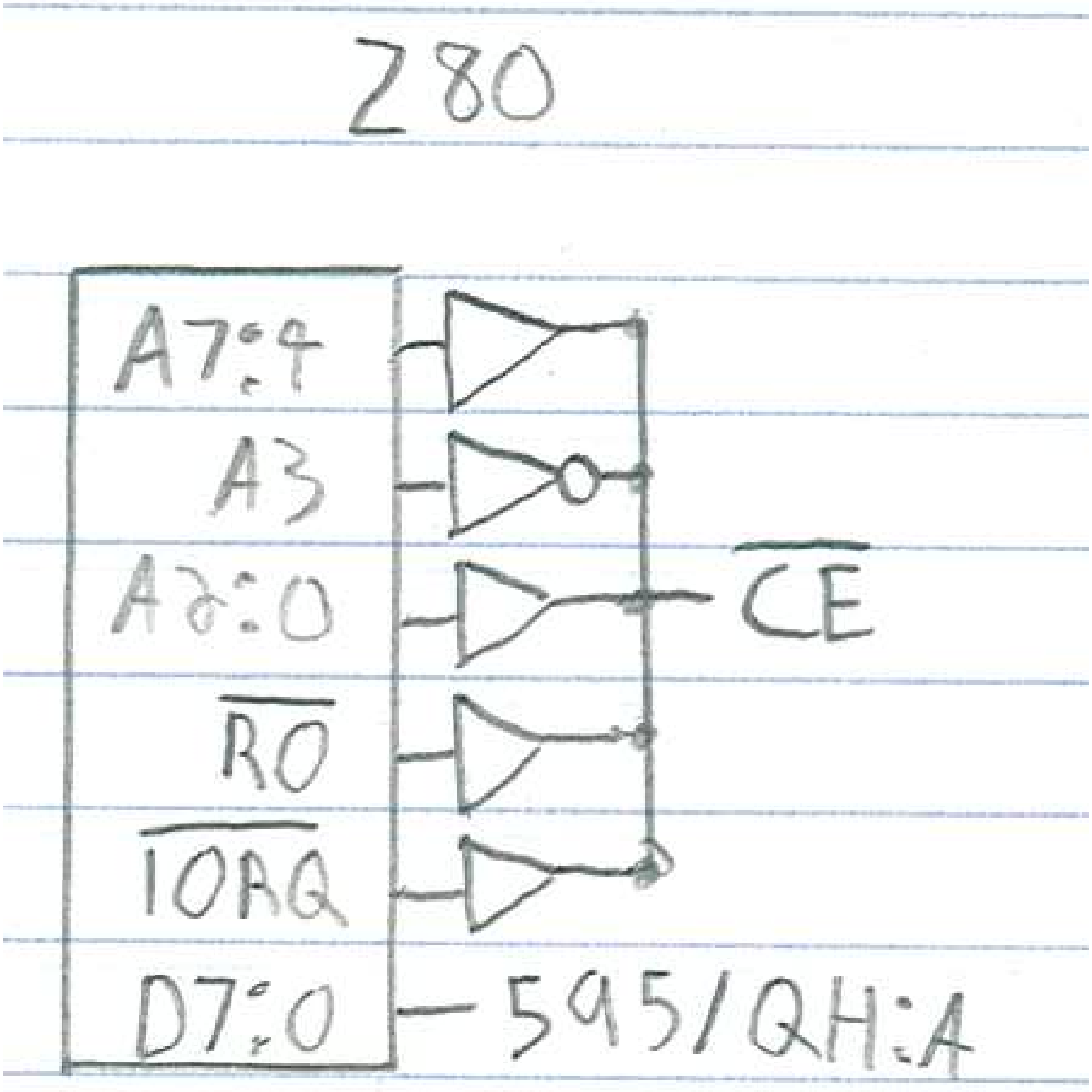
5.3 PS/2 74xx595 (`hw/z80/img/ps2-595.png`)



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 86) 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 92]) 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.

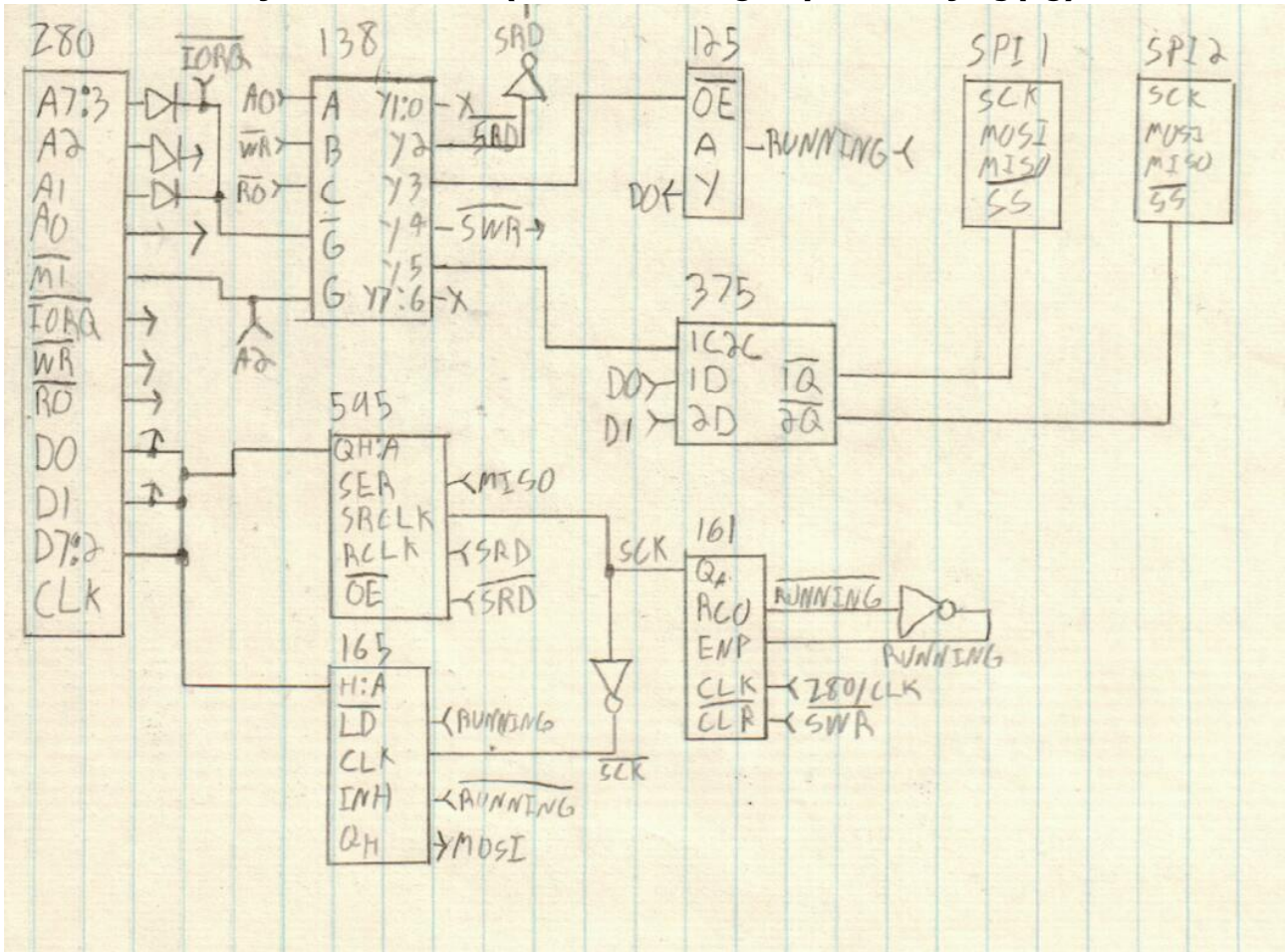
Advice 2: Make input CLK override-able. SD cards are plenty fast enough for us to use the system clock, but you might want to interact with devices that require a slower clock.

Driving the relay

There is a provider for the SPI protocol (doc/protocol.txt Page 54) that
work 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 96

[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.

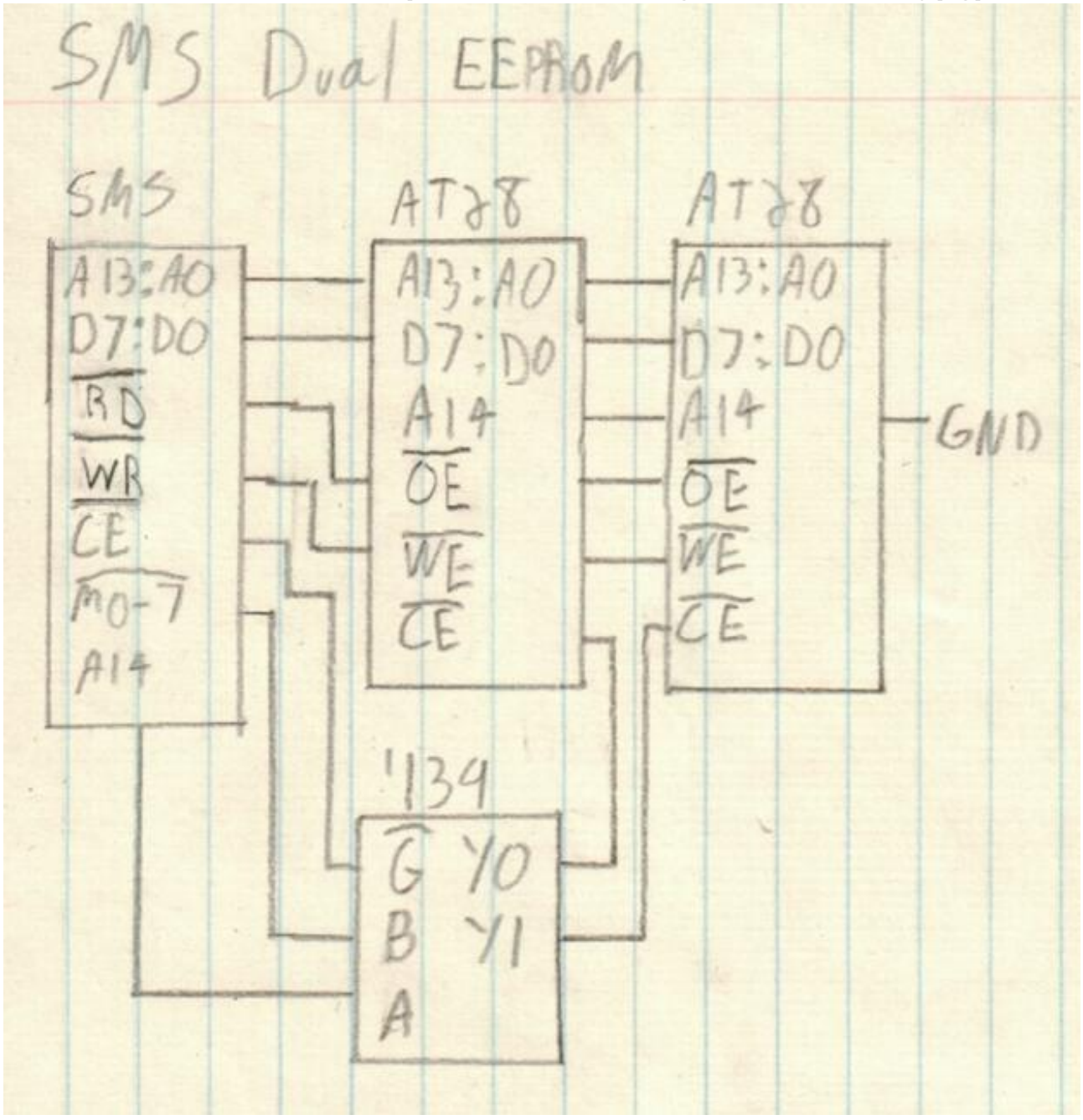
You can see the schematic in dual-at28.jpg Page 96.

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 87]) 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 99. The PS/2-to-AVR part is identical to doc/hw/ps2.txt Page 87.

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

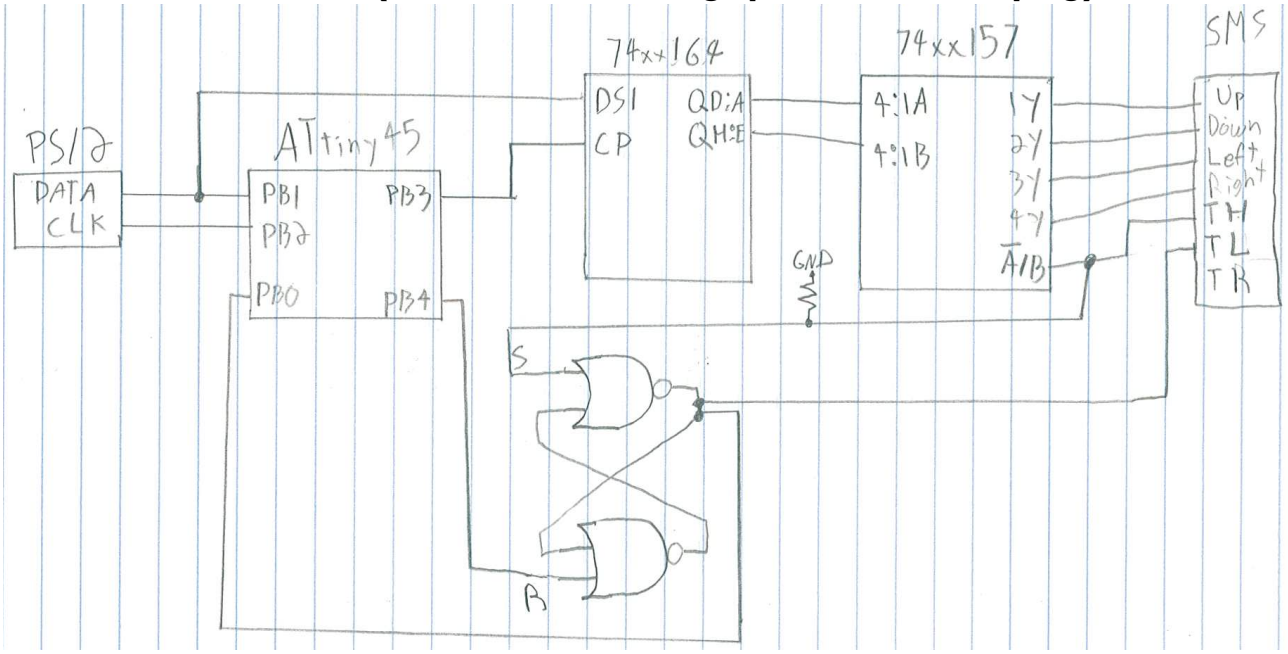
The code for the ATtiny is in B501. It is built with the AVR assembler (doc/asm/avr.txt Page 71). Once built, the binary begins at ORG and can be sent to the ATtiny using the AVR programmer (doc/avr.txt Page 40).

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 55) 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    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 [*TO] 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 55).

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. You can select the active drive with FD0 and FD1. Then, use regular BLK works to interact with blocks.

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> *TO EMIT
' RX<? *TO KEY?
```

"Aliases" in usage.txt[Page 12] 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) *TO EMIT
' (key?) *TO 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).

Floppy organisation

Making blkfs span multiple disk is a bit problematic with regards to absolute block references in the code. You'll need to work a bit to design your very own Collapse OS floppy set. See /doc/usage.txt[Page 12] for details.

Self-hosting

As it is, your installment of Collapse OS is self-hosting using instructions from /doc/selfhost.txt[Page 58]. 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 B121 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:

```
ORG $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.

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.

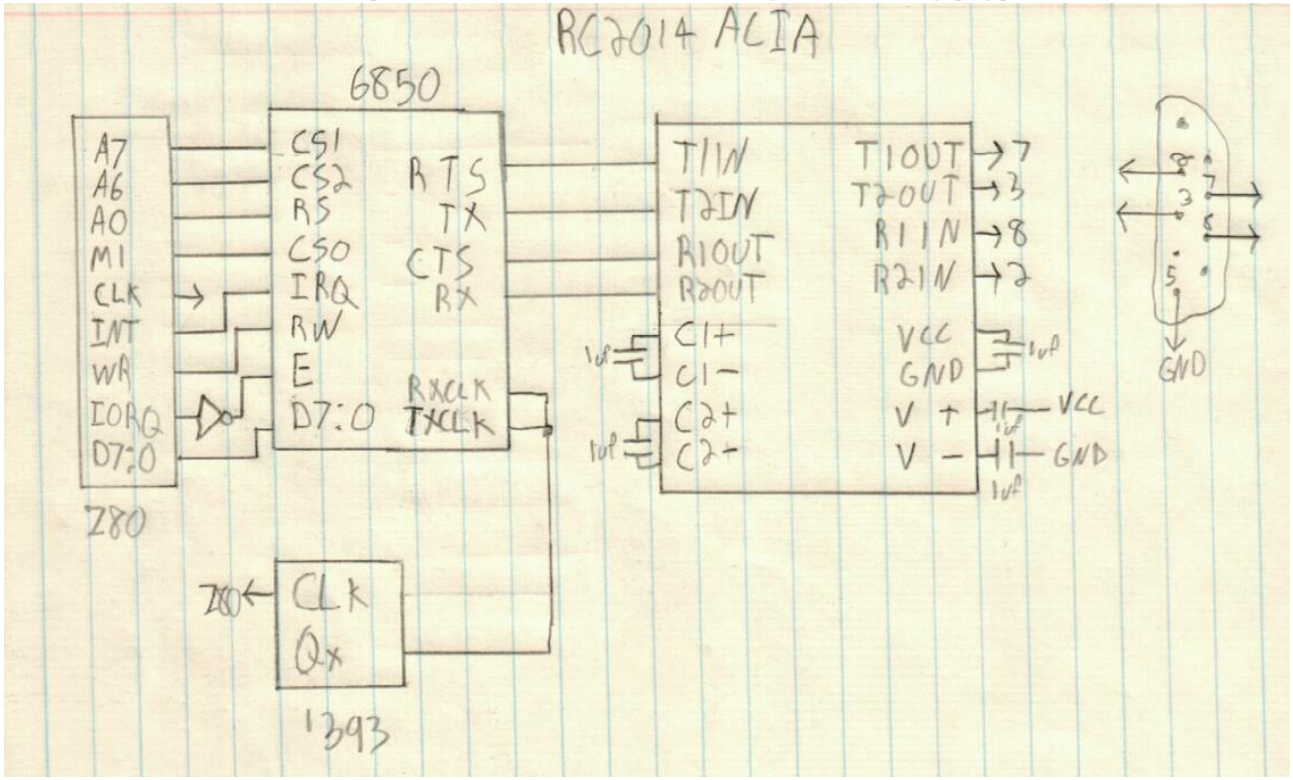
For this reason, I recommend that you build your own ACIA module. A schematic for it is in img/acia.jpg Page 118. This module is exactly the same as the "official" Serial I/O module, with two differences:

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 /arc/z80/ti84 will result in "os.rom" being
created.
```

Emulate

```
Collapse OS has a builtin TI-84+ emulator using XCB for display
in emul/hw/ti. You can invoke it with "make emul".

You will start with a blank screen, it's normal, you haven't
pressed the "ON" key yet. This key is mapped to tilde (~) in the
emulator. Once you press it, the Collapse OS prompt will appear.
See emul/hw/ti/README.md for details.
```

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. But 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.

Usage

The shell works like a normal BASIC 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.

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 125.

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=24576
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, $6000. First, run "HIMEM 24575" to avoid conflict with BASIC and then, in the monitor, type the binary contents.

Regularly, you'll want to come back to BASIC and save your work with something like "BSAVE /VOLUME/COS,A$6000,L$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 "6000G" in the Monitor.

Work In Progress

For now, all this binary does is print "Collapse OS" and loop to infinity. It does so by using as much of the implemented mecanisms as possible, that is:

* core init
* lblnext
* lblxt
* EXIT
* BOOT stable ABI

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)
```

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/arduinouno/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 soon rewrite to Collapse OS' ASM)
* avrdude to send program to Arduino

Schema

Schema is at img/at28wr.jpgPage 131.

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).

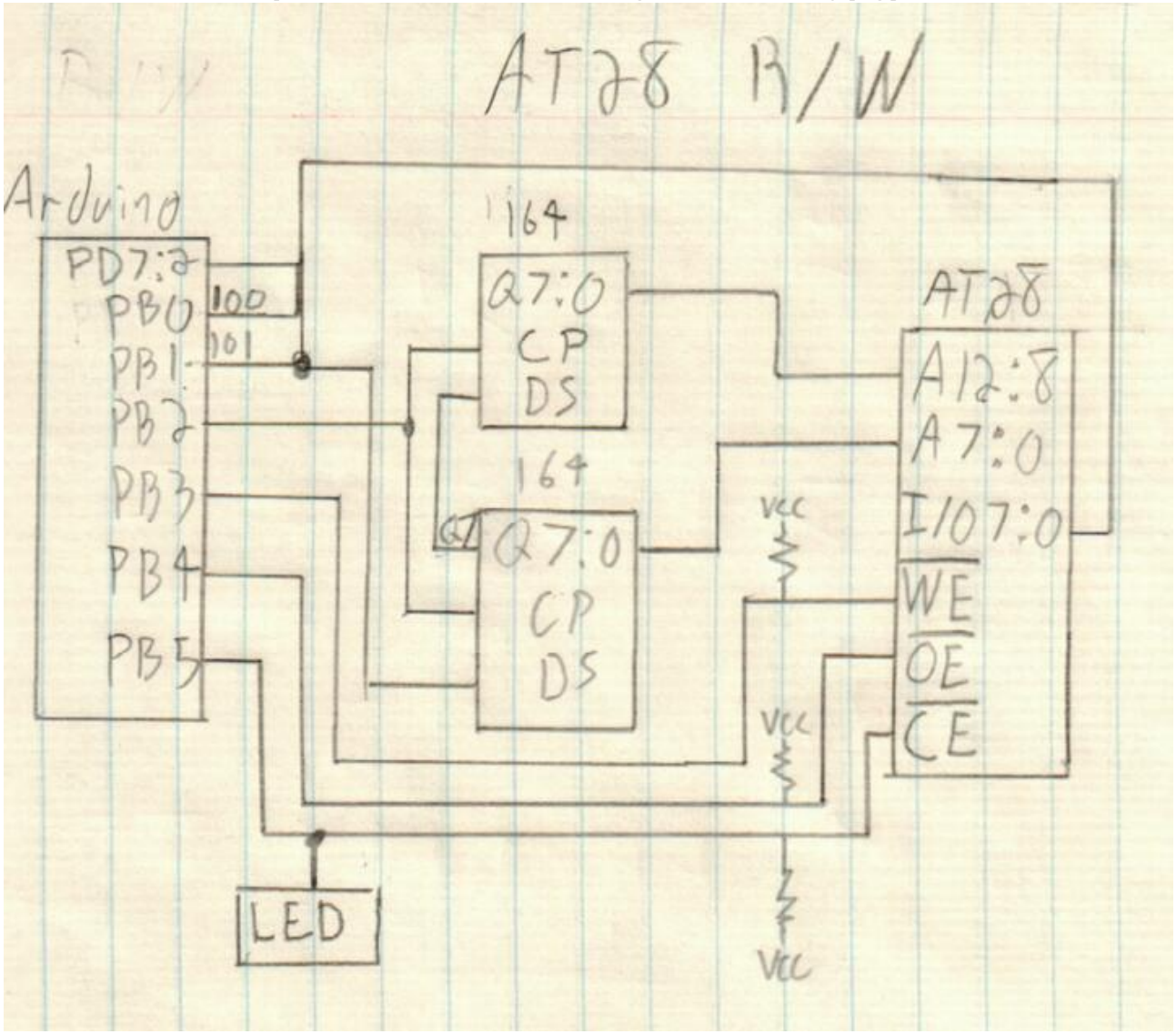
After having written the byte, it re-reads it from the EEPROM and spits it back to the UART, tty-escaped.

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/arduinouno/img/at28wr.jpg`)



Block filesystem

1 Architecture independent

1.1 Master Index: 0

B0

```
MASTER INDEX

002 Common assembler words    05-99 unused
100 Block editor              115 Memory Editor
120 Useful little words
130-149 unused                150 Remote Shell
160 AVR SPI programmer        165 Sega ROM signer
170-199 unused                200 Cross compilation
206 Core words                230 BLK subsystem
240 Grid subsystem            245 PS/2 keyboard subsystem
250 SD Card subsystem         260 Fonts
300 Arch-specific content
```

1.2 Common assembler words: 2-3

B2

```
\ Common assembler words, low
3 VALUES ORG 0 BIN( 0 BIGEND? 0
: PC HERE ORG - BIN( + ;
: <<3 << << << ; : <<4 <<3 << ;
8 VALUES L1 0 L2 0 L3 0 lblnext 0 lblcell 0 lbldoes 0
  lblpush 0 lblxt 0
: |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 ;
: LSET PC TO ;
```

B3

```
\ Common assembler words, high
\ NOTE: this code needs to be cross-compilable
: BEGIN, PC ;
: BR PC - [ JROFF JROPLEN - LITN ] + _bchk ;
: FJR BEGIN, [ JROPLEN LITN ] + 0 ;
: IFZ, Z? ^? FJR ?JRi, ; : IFNZ, Z? FJR ?JRi, ;
: IFC, C? ^? FJR ?JRi, ; : IFNC, C? FJR ?JRi, ;
: FMARK DUP PC ( l l pc ) -^ [ JROFF LITN ] + ( l off )
  \ warning: l is a PC offset, not a mem addr!
  SWAP ORG + BIN( - ( off addr ) C! ;
: THEN, FMARK ; : ELSE, FJR JRi, SWAP FMARK ;
```

1.3 Block editor: 100-111

B100

```
\ Block editor. see doc/ed.txt. B100-B111
\ 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 ;
```

B101

```
\ Block editor, private helpers
: _lpos ( ln -- a ) LNSZ * 'pos ;
: _pln ( ln -- ) \ print line no ln with pos caret
  DUP _lpos DUP LNLEN RANGE DO ( lno )
    I 'EDPOS = IF '^' EMIT THEN
    I C@ DUP SPC < IF DROP SPC THEN EMIT
  LOOP ( lno ) SPC> 1+ . ;
: _zline ( a -- ) LNSZ SPC FILL ; \ zero-out a line
: _zbuf ( buf -- ) 0 SWAP C!+ _zline ; \ zero-out a buf
: _type ( buf -- ) \ type into buf until CR
  IN< DUP CR = IF 2DROP EXIT THEN OVER _zbuf ( buf c )
  OVER 1+ LNSZ RANGE DO ( buf c )
    I C! ( buf ) DUP C@ 1+ OVER C! ( inc len )
    IN< DUP CR = IF LEAVE THEN LOOP 2DROP ;
```

B102

```
\ 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 ;
: _mvln- ( ln -- move ln 1 line up )
    DUP 14 > IF DROP 15 _lpos _zline
    ELSE 1+ _lpos DUP LNSZ - LNSZ MOVE THEN ;
: _U ( U without P, used in VE )
  15 EDPOS LNSZ / - ?DUP IF 0 DO 14 I - _mvln+ LOOP THEN ;
: U _U P ;
```

B103

```
\ Block editor, F i
: _F ( F without _type and _pln. used in VE )
  BLK) 'EDPOS 1+ DO
    I FBUF+ FLEN []= IF I BLK( - EDPOS! LEAVE THEN LOOP ;
: 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 ;
```

B104

```
\ 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 ;
: _X ( n -- )
  ?DUP NOT IF EXIT THEN
  _rbufsz MIN DUP icpy '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 _X EDPOS LNSZ / _pln ;
: _E FLEN _X ;
: E FLEN X ;
: Y FBUF IBUF LNSZ 1+ MOVE ;
```

B105

```
\ Visual text editor. VALUEs, large? width pos@ mode! ...
CREATE CMD '%' C, 0 C,
4 VALUES PREVPOS 0 PREVBLK 0 xoff 0 ACC 0
LNSZ 3 + VALUE MAXW
: large? COLS MAXW > ; : col- MAXW COLS MIN -^ ;
: width large? 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! ;
```

B106

```
\ VE, rfshln contents selblk pos! xoff? setpos
: _ ( ln -- ) \ refresh line ln
  DUP _lpos xoff + SWAP 3 + COLS * large? IF 3 + THEN
  width CELLS! ;
: rfshln pos@ NIP _ ; \ refresh active line
: contents 16 0 DO I _ LOOP ;
: selblk BLK> [TO] PREVBLK 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 -
  large? IF 3 + ( gutter ) THEN SWAP AT-XY ;
```

B107

```
\ VE, cmv buftype bufprint bufs
: cmv ( n -- , char movement ) acc@ * EDPOS + pos! ;
: buftype ( buf 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$ ;
: bufprint ( buf pos -- )
  DUP LNSZ nspcs OVER C@ ROT 1+ ROT> CELLS! ;
: bufs ( -- )
  COLS ( pos ) 'I' OVER CELL! ':' OVER 1+ CELL! ( pos )
  IBUF OVER 3 + bufprint ( pos )
  << 'F' OVER CELL! ':' OVER 1+ CELL! ( pos )
  FBUF SWAP 3 + bufprint ;
```

B108

```
\ VE cmds: G [ ] t I F Y X h H L g @ !
: %G ACC selblk ;
: %[ BLK> acc@ - selblk ; : %] BLK> acc@ + selblk ;
: %t PREVBLK selblk ;
: %I 'I' mode! IBUF 1 buftype _i bufs rfshln ;
: %F 'F' mode! FBUF 2 buftype _F bufs setpos ;
: %Y Y bufs ; : %E _E bufs rfshln ;
: %X acc@ _X bufs rfshln ;
: %h -1 cmv ; : %l 1 cmv ; : %k -64 cmv ; : %j 64 cmv ;
: %H EDPOS $3c0 AND pos! ;
: %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! ;
: %g ACC 1 MAX 1- 64 * pos! ;
: %@ BLK> BLK( (blk@) 0 [*TO] BLKDTY contents ;
: %! BLK> FLUSH [*TO] BLK> ;
```

B109

```
\ VE cmds: w W b B
' NOOP ALIAS C@+-
: C@- ( a -- a-1 c ) DUP C@ SWAP 1- SWAP ;
: go>> ['] C@+ [TO] C@+- ;
: go<< ['] C@- [TO] C@+- ;
: word>> BEGIN C@+- WS? UNTIL ;
: ws>> BEGIN C@+- WS? NOT UNTIL ;
: bpos! BLK( - pos! ;
: %w go>> 'EDPOS acc@ 0 DO word>> ws>> LOOP 1- bpos! ;
: %W go>> 'EDPOS acc@ 0 DO 1+ ws>> word>> LOOP 2 - bpos! ;
: %b go<< 'EDPOS acc@ 0 DO 1- ws>> word>> LOOP 2 + bpos! ;
: %B go<< 'EDPOS acc@ 0 DO word>> ws>> LOOP 1+ bpos! ;
```

B110

```
\ VE cmds: f R O o D
: %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 ;
: %R ( replace mode )
  '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 ;
: %O _U EDPOS $3c0 AND DUP pos! 'pos _zline BLK!! contents ;
: %o EDPOS $3c0 < IF EDPOS 64 + EDPOS! %O THEN ;
: %D %H LNSZ icpy
    acc@ 0 DO 16 EDPOS LNSZ / DO I _mvln- LOOP LOOP
    BLK!! bufs contents ;
```

B111

```
\ VE final: status nums gutter handle VE
: status 0 $20 nspcs 0 0 AT-XY ." BLK" SPC> BLK> . SPC> ACC .
  SPC> pos@ . ',' EMIT . xoff IF '>' EMIT THEN SPC>
  BLKDTY IF '*' EMIT THEN SPC mode! ;
: nums 17 1 DO 0 2 I + AT-XY I . LOOP ;
: gutter large? IF 19 0 DO '|' I COLS * MAXW + CELL! LOOP THEN ;
: handle ( c -- f )
  DUP '0' '9' =><= IF num 0 EXIT THEN
  DUP CMD 1+ C! CMD 2 FIND IF EXECUTE 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 ;
```

1.4 Memory editor: 115-119

B115

```
\ Memory Editor. See doc/me.txt. B115-119
CREATE CMD '#' C, 0 C, \ not same prefix as VE
CREATE BUF '$' C, 4 ALLOT \ always hex
\ POS is relative to ADDR
5 VALUES ADDR 0 POS 0 AWIDTH 16 HALT? 0 ASCII? 0
LINES 2 - VALUE AHEIGHT
AHEIGHT AWIDTH * VALUE PAGE
COLS 33 < [IF] 8 TO AWIDTH [THEN]
: _ ( n -- c ) DUP 9 > IF [ 'a' 10 - LITN ] ELSE '0' THEN + ;
: _p ( c -- n ) '0' - DUP 9 > IF $df AND 'A' '0' - - DUP 6 < IF
    10 + ELSE DROP $100 THEN THEN ;
: addr ADDR POS + ;
: hex! ( c pos -- )
  OVER 16 / _ OVER CELL! ( c pos ) 1+ SWAP $f AND _ SWAP CELL! ;
: bottom 0 LINES 1- AT-XY ;
```

B116

```
\ Memory Editor, line rfshln contents showpos
: line ( ln -- )
  DUP AWIDTH * ADDR + SWAP 1+ COLS * ( a pos )
  ':' OVER CELL! OVER <<8 >>8 OVER 1+ hex! 4 + ( a pos+4 )
  AWIDTH >> 0 DO ( a pos )
    OVER I << + C@+ ( a pos a+1 c ) ROT TUCK hex! ( a a+1 pos )
    2 + SWAP C@ OVER hex! 3 + ( a pos+5 ) LOOP
  SWAP AWIDTH RANGE DO ( pos )
    I C@ DUP SPC < IF DROP '.' THEN OVER CELL! 1+ LOOP DROP ;
: rfshln POS AWIDTH / line ;
: contents LINES 2 - 0 DO I line LOOP ;
: 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 ;
```

B117

```
\ 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 nspcs
  0 0 AT-XY ." A: " ADDR .X SPC> ." C: " POS .X SPC> ." S: "
  PSDUMP POS pos! ;
: type ( cnt -- sa sl ) DUP ( cnt ) 0 DO ( cnt )
  KEY DUP SPC < IF DROP I LEAVE ELSE DUP EMIT BUF 1+ I + C! THEN
  LOOP BUF SWAP 1+ ;
: typep ( cnt -- n? f )
  type ( sa sl ) DUP IF PARSE ELSE NIP THEN ;
```

B118

```
\ 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 ;
```

B119

```
\ Memory Editor, #q handle ME
: #q 1 [TO] HALT? ;
: handle ( c -- f )
  CMD 1+ C! CMD 2 FIND IF EXECUTE THEN ;
: ME clrscr contents 0 pos! BEGIN
    status KEY handle HALT? UNTIL bottom ;
```

1.5 Useful little words: 120-123

B120

```
\ Useful little words. nC, MIN MAX MOVE-
\ parse the next n words and write them as chars
: nC, ( n -- ) 0 DO RUN1 C, LOOP ;
: MIN ( n n - n ) 2DUP > IF SWAP THEN DROP ;
: MAX ( n n - n ) 2DUP < IF SWAP THEN DROP ;
\ Compute CRC16 over a memory range
: CRC16[] ( a u -- c ) 0 ROT> RANGE DO I C@ CRC16 LOOP ;
: MOVE- ( a1 a2 u -- ) \ MOVE starting from the end
  ?DUP IF 1- TUCK + ( a1 u-1 a2+ ) ROT> TUCK + ( a2+ u-1 a1+ )
    SWAP 1+ ( u ) 0 DO ( a2 a1 )
      DUP C@ ROT> 1- ( c a2 a1- ) ROT> TUCK C! 1- SWAP
  LOOP THEN 2DROP ;
```

B121

```
\ Useful little words. MEM>BLK BLK>MEM
\ Copy an area of memory into blocks.
: MEM>BLK ( addr blkno blkcnt )
  ( bcnt ) 0 DO ( a bno )
    DUP I + BLK@ OVER I $400 * + ( a bno a' )
    BLK( $400 MOVE BLK!! LOOP ( a bno ) 2DROP FLUSH ;
\ Copy subsequent blocks in an area of memory
: BLK>MEM ( blkno blkcnt addr )
  ROT> RANGE DO ( a )
    I BLK@ BLK( OVER $400 MOVE $400 + LOOP DROP ;
```

B122

```
\ 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 )
  @ [*TO] CURRENT ;
```

B123

```
\ Grid applications helper words. nspcs clrscr
: nspcs ( pos n ) RANGE DO SPC I CELL! LOOP ;
: clrscr 0 COLS LINES * nspcs ;
```

1.6 Remote Shell: 150-154

B150

```
( Remote Shell. load range B150-B154 )
: _<< ( print everything available from RX<? )
  BEGIN RX<? IF EMIT ELSE EXIT THEN AGAIN ;
: _<<r ( _<< with retries )
  BEGIN _<< 100 TICKS RX<? IF EMIT ELSE EXIT THEN AGAIN ;
: RX< BEGIN RX<? UNTIL ;
: _<<1r RX< EMIT _<<r ;
: rsh BEGIN
  KEY? IF DUP EOT = IF DROP EXIT ELSE TX> THEN THEN _<< AGAIN ;
: rstype ( sa sl --, like STYPE, but remote )
  ( sl ) 0 DO C@+ TX> _<<r LOOP DROP _<<r CR TX> RX< DROP _<<r ;
: rstypep ( like rstype, but read ok prompt )
    rstype BEGIN RX< WS? NOT UNTIL _<<1r ;
```

B151

```
: unpack DUP $f0 OR SWAP $0f OR ;
: out unpack TX> TX> ; : out2 L|M out out ;
: rupload ( loca rema u -- )
  LIT" : in KEY $f0 AND KEY $0f AND OR ;" rstypep
  LIT" : in2 in <<8 in OR ;" rstypep
  \ sig: chk -- chk, a and then u are KEYed in
  LIT" : _ in2 in2 RANGE DO in TUCK + SWAP I C! LOOP ;" rstypep
  DUP ROT ( loca u u rema ) LIT" 0 _" rstype out2 out2
  OVER + SWAP 0 ROT> ( 0 loca+u loca )
  DO '.' EMIT I C@ DUP ROT + SWAP out LOOP
  _<<1r LIT" .X FORGET in" rstypep .X ;
```

B152

```
( XMODEM routines )
: _<<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< DROP RX< DROP ( packet num )
  0 ( addr crc ) SWAP 128 RANGE DO ( crc )
    RX< DUP ( crc n n ) I C! ( crc n ) CRC16 LOOP
  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+ [*TO] BLK> THEN
  DUP _rx>mem1 SWAP 128 + SWAP UNTIL 2DROP ;
```

B153

```
: _snd128 ( a -- a )
    0 ( a crc ) 128 0 DO ( a crc )
      OVER I + C@ DUP TX> ( a crc n ) CRC16 ( a crc ) LOOP
    L|M TX> TX> ( a ) ;
: _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 -- ) SWAP DO ( a )
    'P' EMIT I . SPC> $01 ( SOH ) TX>
    I 1+ ( pkt start at 1 ) DUP TX> $ff -^ TX>
    _snd128 _ack? IF 128 + ( a+128 ) ELSE R> 1- >R THEN
  LOOP DROP ;
```

B154

```
: 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 ) 0 DO ( b1 )
    'B' EMIT DUP I + DUP . SPC> BLK@ BLK(
    I 8 * DUP 8 + ( a pktstart pktend ) _mem>tx
  LOOP DROP
  $4 ( EOT ) TX> RX< DROP ;
```

1.7 AVR SPI programmer: 160-163

B160

```
\ 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 ;
```

B161

```
( 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 ;
```

B162

```
: 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 ;
```

B163

```
: 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 ;
```

1.8 Sega ROM signer: 165

B165

```
( Sega ROM signer. See doc/sega.txt )
: C!+^ ( a c -- a+1 ) OVER C! 1+ ;
: segasig ( addr size -- )
    $2000 OVER LSHIFT ( a sz bytesz )
    ROT TUCK + $10 - ( sz a end )
    TUCK SWAP 0 ROT> ( sz end sum end a ) DO ( sz end sum )
        I C@ + LOOP ( sz end sum ) SWAP ( sz sum end )
    'T' C!+^ 'M' C!+^ 'R' C!+^ SPC C!+^ 'S' C!+^
    'E' C!+^ 'G' C!+^ 'A' C!+^ 0 C!+^ 0 C!+^
    ( sum's LSB ) OVER C!+^ ( MSB ) SWAP >>8 OVER C! 1+
    ( sz end ) 0 C!+^ 0 C!+^ 0 C!+^ SWAP $4a + SWAP C! ;
```

1.9 Cross compilation: 200-205

B200

```
\ Cross compilation program. See doc/cross.txt. B200-B205
: XCOMPH 201 205 LOADR ; : FONTC 262 263 LOADR ;
: COREL 207 224 LOADR ; : COREH 225 229 LOADR ;
: BLKSUB 230 234 LOADR ; : GRIDSUB 240 241 LOADR ;
: PS2SUB 246 248 LOADR ;
'? HERESTART NOT [IF] 0 VALUE HERESTART [THEN]
0 VALUE XCURRENT \ CURRENT in target system, in target's addr
7 VALUES
  (n)* 0 (b)* 0 2>R* 0 (loop)* 0 (br)* 0 (?br)* 0 EXIT* 0
```

B201

```
: _xoff ORG BIN( - ;
: ENTRY
  WORD TUCK MOVE, XCURRENT T, C, HERE _xoff - [TO] XCURRENT ;
: ALIAS ENTRY JMPi, ; : *ALIAS ENTRY (i)>w, JMPw, ;
: VALUE ENTRY i>w, lblpush JMPi, ;
: *VALUE ENTRY (i)>w, lblpush JMPi, ;
: VALUES 0 DO ENTRY RUN1 i>w, lblpush JMPi, LOOP ;
: CREATE ENTRY lblcell CALLi, ;
: W= ( sa sl w -- f ) 2DUP 1- C@ $7f AND = IF ( same len )
    ( sa sl w ) OVER - 3 - ( s+1 len w-3-len ) ROT> []=
    ELSE 2DROP DROP 0 THEN ;
: _xfind ( sa sl -- w? f ) PAD C!+ ! XCURRENT BEGIN ( w )
  _xoff + DUP PAD C@+ SWAP @ SWAP ROT W= IF ( w ) 1 EXIT THEN
  3 - ( prev field ) T@ ?DUP NOT UNTIL 0 ( not found ) ;
: XFIND ( sa sl -- w ) _xfind NOT IF (wnf) THEN _xoff - ;
```

B202

```
: '? WORD _xfind DUP IF SWAP _xoff - SWAP THEN ;
: X' '? NOT IF (wnf) THEN ;
: _codecheck ( lbl str -- )
  XCURRENT _xoff + W= IF XCURRENT SWAP VAL! ELSE DROP THEN ;
: CODE ENTRY
  ['] EXIT* LIT" EXIT" _codecheck ['] (b)* LIT" (b)" _codecheck
  ['] (n)* LIT" (n)" _codecheck ['] 2>R* LIT" 2>R" _codecheck
  ['] (loop)* LIT" (loop)" _codecheck
  ['] (br)* LIT" (br)" _codecheck
  ['] (?br)* LIT" (?br)" _codecheck ;
: ;CODE lblnext JMPi, ;
```

B203

```
: XWRAP
  COREH HERESTART ?DUP NOT IF PC THEN ORG 8 ( LATEST ) + T!
  XCURRENT ORG 6 ( CURRENT ) + T! ;
: LITN DUP $ff > IF (n)* T, T, ELSE (b)* T, C, THEN ;
: imm? ( w -- f ) 1- C@ $80 AND ;
: X: CODE lblxt CALLi, BEGIN
  WORD LIT" ;" 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 ;
```

B204

```
: ['] WORD XFIND LITN ; IMMEDIATE
: COMPILE [COMPILE] ['] LIT" ," XFIND T, ; IMMEDIATE
: DO 2>R* T, HERE ; IMMEDIATE
: LOOP (loop)* T, HERE - C, ; 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
```

B205

```
: [*TO] X' LITN LIT" *VAL!" XFIND T, ; IMMEDIATE
: LIT" (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! ;
: : [ ' X: , ] ;
```

1.10 Core words: 207-229

B207

```
\ Core Forth words. See doc/cross.txt.
\ Load range low: B206-B224 high: B226-B229
CODE EXIT POPr, w>IP, ;CODE
CODE EXECUTE POPp, JMPw,
CODE (br) LSET L1 ( used in ?br and loop ) IP+off, ;CODE
CODE (?br) POPp, w>Z, Z? L1 BR ?JRi, IP+, ;CODE
CODE (loop)
  POPr, INCw, PUSHp, POPr, CMPwp,
  IFNZ, PUSHr, POPp, PUSHr, L1 BR JRi, THEN,
  IP+, DROPp, ;CODE
CODE (b) IP>w, C@w, IP+, PUSHp, ;CODE
CODE (n) IP>w, @w, IP+, IP+, PUSHp, ;CODE
CODE (c) IP>w, IP+off, INCw, JMPw,
```

B208

```
\ Core words, >R R> 2>R 2R> I DUP ?DUP DROP SWAP OVER ..
CODE >R POPp, PUSHr, ;CODE
CODE R> POPr, PUSHp, ;CODE
CODE 2>R POPf, PUSHr, POPp, PUSHr, ;CODE
CODE 2R> DUPp, POPr, PUSHf, POPr, w>p, ;CODE
CODE I POPr, PUSHp, PUSHr, ;CODE
CODE DUP DUPp, ;CODE
CODE ?DUP p>Z, IFNZ, DUPp, THEN, ;CODE
CODE DROP DROPp, ;CODE
CODE SWAP POPf, PUSHp, ;CODE
CODE OVER POPf, PUSHf, PUSHp, ;CODE
CODE ROT POPp, SWAPwp, SWAPwf, PUSHp, ;CODE
CODE ROT> POPp, SWAPwf, SWAPwp, PUSHp, ;CODE
```

B209

```
\ Core words, AND OR XOR NOT + - ! @ C! C@
CODE AND POPp, ANDwp, w>p, ;CODE
CODE OR POPp, ORwp, w>p, ;CODE
CODE XOR POPp, XORwp, w>p, ;CODE
CODE NOT p>Z, Z>w, w>p, ;CODE
CODE + POPp, +wp, w>p, ;CODE
CODE - POPf, -wp, w>p, ;CODE
CODE ! POPp, !wp, DROPp, ;CODE
CODE @ p>w, @w, w>p, ;CODE
CODE C! POPp, C!wp, DROPp, ;CODE
CODE C@ p>w, C@w, w>p, ;CODE
```

B210

```
\ Core words, = > < 1+ 1- << >> <<8 >>8
CODE = POPp, CMPwp, Z>w, w>p, ;CODE
CODE > POPp, CMPwp, C>w, w>p, ;CODE
CODE < POPf, CMPwp, C>w, w>p, ;CODE
CODE 1+ INCp, ;CODE
CODE 1- DECp, ;CODE
CODE >> p>w, >>w, w>p, ;CODE
CODE << p>w, <<w, w>p, ;CODE
CODE >>8 p>w, >>8w, w>p, ;CODE
CODE <<8 p>w, <<8w, w>p, ;CODE
```

B211

```
\ Core words, NOOP CURRENT HERE PC ORG BIN( SYSVARS IOERR ..
CODE NOOP ;CODE
SYSVARS $02 + *VALUE CURRENT
SYSVARS $04 + *VALUE HERE
SYSVARS $04 + *VALUE PC
SYSVARS $18 + VALUE PAD
0 VALUE ORG
BIN( VALUE BIN( SYSVARS VALUE SYSVARS
SYSVARS *VALUE IOERR
\ size of a line. used for INBUF and BLK. Keep this to $40 or
\ you're gonna have a bad time.
$40 VALUE LNSZ
```

B212

```
\ Core words, 0< >= <= =><= 2DUP 2DROP NIP TUCK L|M C@+ ..
: 0< $7fff > ; : >= < NOT ; : <= > NOT ;
: =><= ( n l h -- f ) OVER - ROT> ( h n l ) - >= ;
CODE 2DUP POPf, PUSHf, DUPp, PUSHf, ;CODE
CODE 2DROP DROPp, DROPp, ;CODE
CODE NIP POPf, ;CODE CODE TUCK POPf, DUPp, PUSHf, ;CODE
: L|M DUP <<8 >>8 SWAP >>8 ;
: RSHIFT ?DUP IF 0 DO >> LOOP THEN ;
: LSHIFT ?DUP IF 0 DO << LOOP THEN ; : -^ SWAP - ;
CODE C@+ p>w, INCp, C@w, PUSHp, ;CODE
CODE C!+ POPp, C!wp, INCw, w>p, ;CODE
: LEAVE R> R> DROP I 1- >R >R ;
CODE UNLOOP POPr, POPr, ;CODE
CODE VAL! POPp, INCw, !wp, DROPp, ;CODE
CODE *VAL! POPp, INCw, @w, !wp, DROPp, ;CODE
: / /MOD NIP ; : MOD /MOD DROP ;
```

B213

```
\ Core words, ALLOT FILL IMMEDIATE , L, M, MOVE MOVE- MOVE, ..
: RANGE ( a u -- ah al ) OVER + SWAP ;
: ALLOT HERE + [*TO] HERE ;
: FILL ( a u b -- ) ROT> RANGE DO ( b ) DUP I C! LOOP DROP ;
: ALLOT0 ( u -- ) HERE OVER 0 FILL ALLOT ;
: IMMEDIATE CURRENT 1- DUP C@ 128 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 RANGE DO ( src ) C@+ ( src+1 b ) I C! LOOP
  ELSE DROP THEN DROP ;
: MOVE, ( a u -- ) HERE OVER ALLOT SWAP MOVE ;
```

B214

```
\ Core words, we begin EMITting
SYSVARS $0e + *ALIAS EMIT
: STYPE RANGE DO I C@ EMIT LOOP ;
5 VALUES EOT $04 BS $08 LF $0a CR $0d SPC $20
SYSVARS $0a + *VALUE NL
: SPC> SPC EMIT ;
: NL> NL L|M ?DUP IF EMIT THEN EMIT ;
: STACK? SCNT 0< IF LIT" stack underflow" STYPE ABORT THEN ;
```

B215

```
\ Core words, number formatting
: . ( n -- )
  ?DUP NOT IF '0' EMIT EXIT THEN \ 0 is a special case
  DUP 0< IF '-' EMIT -1 * THEN
  $ff SWAP ( stop ) BEGIN 10 /MOD ( d q ) ?DUP NOT UNTIL
  BEGIN '0' + EMIT DUP 9 > UNTIL DROP ;
: _ DUP 9 > IF [ 'a' 10 - LITN ] ELSE '0' THEN + ;
: .x <<8 >>8 16 /MOD ( l h ) _ EMIT _ EMIT ;
: .X L|M .x .x ;
```

B216

```
\ Core words, literal parsing
: _ud ( sa sl -- n? f ) \ parse unsigned decimal
  0 ROT> RANGE DO ( r )
    10 * I C@ ( r c ) '0' - DUP 9 > IF
      2DROP 0 UNLOOP EXIT THEN + LOOP ( r ) 1 ;
: _d ( sa sl -- n? f ) \ parse possibly signed decimal
  OVER C@ '-' = IF
    SWAP 1+ SWAP 1- _ud DUP IF SWAP 0 -^ SWAP THEN
    ELSE _ud THEN ;
: _h ( sa sl -- n 1 OR sa sl 0 ) \ parse hex
  OVER C@ '$' = NOT IF 0 EXIT THEN
  2DUP 0 ROT> RANGE 1+ DO ( sa sl r )
    16 * I C@ ( r c ) '0' - DUP 9 > IF
      $df AND [ 'A' '0' - LITN ] - DUP 6 < IF
        10 + ELSE 2DROP 0 UNLOOP EXIT THEN THEN
    ( r n ) + LOOP ( sa sl r ) NIP NIP 1 ;
```

B217

```
\ Core words, literal parsing, CRC16
: _c ( sa sl -- n 1 OR sa sl 0 ) \ parse character
  DUP 3 = IF OVER C@ ''' = IF OVER 2 + C@ ''' = IF
    DROP 1+ C@ 1 EXIT THEN THEN THEN 0 ;
: PARSE ( sa sl -- n? f )
  _c ?DUP IF EXIT THEN _h ?DUP NOT IF _d THEN ;
CODE CRC16 ( c n -- c )
  POPp, <<8w, XORwp, w>p, 8 i>w, SWAPwp, BEGIN, ( w=c p=u )
    <<w, IFC, $1021 XORwi, THEN, DECp, p>Z, Z? ^? BR ?JRi,
    w>p, ;CODE
```

B218

```
\ Core words, input buffer
SYSVARS $10 + *ALIAS KEY?
: KEY BEGIN KEY? UNTIL ;
SYSVARS $2e + *VALUE IN(
SYSVARS $30 + *VALUE IN> \ current position in IN(
SYSVARS $08 + *ALIAS LN<
: IN) IN( LNSZ + ;
: BS? DUP $7f ( DEL ) = SWAP BS = OR ;
```

B219

```
\ Core words, input buffer
\ 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 - SPC FILL 1 ELSE
      TUCK EMIT C!+ DUP IN) = THEN THEN ;
: RDLN ( -- ) \ Read 1 line in IN(
  LIT"  ok" STYPE NL> IN( [*TO] IN>
  IN( BEGIN KEY LNTYPE UNTIL DROP NL> ;
: IN< ( -- c ) \ Read one character from INBUF
  IN> IN) = IF LN< THEN IN> C@ IN> 1+ [*TO] IN> ;
: IN$ ['] RDLN [*TO] LN<
  [ SYSVARS $40 ( INBUF ) + LITN ] [*TO] IN( IN) [*TO] IN> ;
```

B220

```
\ Core words, WORD parsing
: ," BEGIN IN< DUP '"' = IF DROP EXIT THEN C, AGAIN ;
: WS? SPC <= ;
: TOWORD ( -- c ) \ Advance IN> to first non-WS and yield it.
  0 ( dummy ) BEGIN DROP IN< DUP WS? NOT UNTIL ;
: CURWORD ( -- sa sl ) [ SYSVARS $12 + LITN ] C@+ SWAP @ SWAP ;
: WORD ( -- sa sl )
  TOWORD DROP IN> 1- DUP BEGIN ( old a )
    C@+ WS? OVER IN) = OR UNTIL ( old new )
  DUP [*TO] IN> ( old new ) OVER - ( old len )
  IN> 1- C@ WS? IF 1- THEN ( adjust len when not EOL )
  2DUP [ SYSVARS $12 ( CURWORD ) + LITN ] C!+ ! ;
```

B221

```
\ Core words, INTERPRET loop
: (wnf) CURWORD STYPE LIT"  word not found" STYPE ABORT ;
: RUN1 \ read next word in stream and interpret it
  WORD PARSE NOT IF
    CURWORD FIND IF EXECUTE STACK? ELSE (wnf) THEN THEN ;
: INTERPRET BEGIN RUN1 AGAIN ;
\ We want to pop the RS until it points to a xt *right after*
\ a reference to INTERPET (yeah, that's pretty hackish!)
: ESCAPE! 0 ( dummy ) BEGIN
  DROP R> DUP 2 - ( xt xt-2 ) @ ['] INTERPRET = UNTIL >R ;
```

B222

```
\ Core words, Dictionary
: ENTRY WORD TUCK MOVE, ( len )
  CURRENT , C, \ write prev value and size
  HERE [*TO] CURRENT ;
X' ENTRY ALIAS CODE
: '? WORD FIND DUP IF NIP THEN ;
: ' WORD FIND NOT IF (wnf) THEN ;
: TO ' VAL! ; : *TO ' *VAL! ;
: 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 ) << >> ( rm IMMEDIATE )
  3 + ( fixed header len ) - [*TO] HERE ( w )
  ( get prev addr ) 3 - DUP @ [*TO] CURRENT ;
```

B223

```
\ Core words, S=, [IF], _bchk
: S= ( sa1 sl1 sa2 sl2 -- f )
  ROT OVER = IF ( same len, s2 s1 l ) []=
  ELSE DROP 2DROP 0 THEN ;
: [IF]
  IF EXIT THEN LIT" [THEN]" BEGIN 2DUP WORD S= UNTIL 2DROP ;
: [THEN] ;
: _bchk DUP $80 + $ff > IF LIT" br ovfl" STYPE ABORT THEN ;
```

B224

```
\ Core words, DUMP, .S
: DUMP ( n a -- )
  SWAP 8 /MOD SWAP IF 1+ THEN 0 DO
    ':' EMIT DUP .x SPC> DUP ( a a )
    4 0 DO C@+ .x C@+ .x SPC> LOOP DROP ( a )
    8 0 DO
      C@+ DUP SPC < IF DROP '.' THEN EMIT LOOP NL>
  LOOP DROP ;
: PSDUMP SCNT NOT IF EXIT THEN
  SCNT PAD ! BEGIN DUP .X SPC> >R SCNT NOT UNTIL
  BEGIN R> SCNT PAD @ = UNTIL ;
: .S ( -- )
  LIT" SP " STYPE SCNT .x SPC> LIT" RS " STYPE RCNT .x SPC>
  LIT" -- " STYPE STACK? PSDUMP ;
```

B225

```
\ Core high, CREATE DOER DOES>
: CREATE CODE [ lblcell LITN ] CALLi, ;
: DOER CODE 0 i>w, [ lbldoes LITN ] CALLi, ;
\ Because we pop RS below, we'll exit parent definition
: DOES> R> CURRENT 1+ ! ;
: ALIAS ( addr -- ) ENTRY JMPi, ;
: *ALIAS ( addr -- ) ENTRY (i)>w, JMPw, ;
: VALUE ENTRY i>w, [ lblpush LITN ] JMPi, ;
: *VALUE ENTRY (i)>w, [ lblpush LITN ] JMPi, ;
: VALUES ( n -- )
  0 DO ENTRY RUN1 i>w, [ lblpush LITN ] JMPi, LOOP ;
: ;CODE [ lblnext LITN ] JMPi, ;
```

B226

```
\ Core high, BOOT
: (main) IN$ INTERPRET BYE ;
XCURRENT ORG $0a ( stable ABI (main) ) + T!
: BOOT
  [ BIN( $06 ( CURRENT ) + LITN ] @ [*TO] CURRENT
  [ BIN( $08 ( LATEST ) + LITN ] @ [*TO] HERE
  ['] (emit) ['] EMIT *VAL! ['] (key?) [*TO] KEY?
  0 [*TO] IOERR $0d0a ( CR/LF ) [*TO] NL
  INIT LIT" Collapse OS" STYPE ABORT ;
XCURRENT ORG $04 ( stable ABI BOOT ) + T!
```

B227

```
\ Core high, See bootstrap doc. DO..LOOP, LITN, :
: DO COMPILE 2>R HERE ; IMMEDIATE
: LOOP COMPILE (loop) HERE - _bchk C, ; IMMEDIATE
: LITN DUP >>8 IF COMPILE (n) , ELSE COMPILE (b) C, THEN ;
: : CODE [ lblxt LITN ] CALLi, BEGIN
    WORD LIT" ;" 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) THEN THEN
  AGAIN ;
```

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
: CODE[ COMPILE (c) HERE 1 ALLOT INTERPRET ; IMMEDIATE
: ]CODE ;CODE [COMPILE] THEN 2R> 2DROP ;
: ( LIT" )" BEGIN 2DUP WORD S= UNTIL 2DROP ; IMMEDIATE
: \ IN) [*TO] IN> ; IMMEDIATE
: LIT"
  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] LIT" 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
: [TO] ' LITN COMPILE VAL! ; IMMEDIATE
: [*TO] ' LITN COMPILE *VAL! ; IMMEDIATE
: [ INTERPRET ; IMMEDIATE
: ] 2R> 2DROP ; \ INTERPRET+RUN1
: COMPILE ' LITN ['] , , ; IMMEDIATE
: [COMPILE] ' , ; IMMEDIATE
: ['] ' LITN ; IMMEDIATE
```

1.11 BLK subsystem: 230-234

B230

```
\ BLK subsystem. See doc/blk.txt. Load range: B230-234
\ Current blk pointer -1 means "invalid"
SYSVARS $38 + *VALUE BLK>
\ Whether buffer is dirty
SYSVARS $3a + *VALUE BLKDTY
SYSVARS 1024 - VALUE BLK(
SYSVARS VALUE BLK)
: BLK$ 0 [*TO] BLKDTY -1 [*TO] BLK> ;
```

B231

```
: BLK! ( -- ) BLK> BLK( (blk!) 0 [*TO] BLKDTY ;
: FLUSH BLKDTY IF BLK! THEN -1 [*TO] BLK> ;
: BLK@ ( n -- )
  DUP BLK> = IF DROP EXIT THEN
  FLUSH DUP [*TO] BLK> BLK( (blk@) ;
: BLK!! 1 [*TO] BLKDTY ;
: WIPE BLK( 1024 SPC FILL BLK!! ;
: COPY ( src dst -- ) FLUSH SWAP BLK@ [*TO] BLK> BLK! ;
```

B232

```
: LNLEN ( a -- len ) \ len based on last visible char in line
  -1 ( res ) LNSZ 0 DO ( a res )
    OVER I + C@ SPC > IF DROP I THEN LOOP 1+ NIP ;
: EMITLN ( a -- ) \ emit LNSZ chars from a or stop at CR
  DUP LNLEN ?DUP IF RANGE DO I C@ EMIT LOOP ELSE DROP THEN NL> ;
: LIST ( n -- ) \ print contents of BLK n
  BLK@ 16 0 DO
    I 1+ DUP 10 < IF SPC> THEN . SPC>
    LNSZ I * BLK( + EMITLN LOOP ;
: INDEX ( b1 b2 -- ) \ print first line of blocks b1 through b2
  1+ SWAP DO I DUP . SPC> BLK@ BLK( EMITLN LOOP ;
```

B233

```
: _ ( -- ) \ set IN( to next line in block
  IN) BLK) = IF ESCAPE! THEN
  IN) [*TO] IN( IN( [*TO] IN> ;
: LOAD
  IN> >R ['] _ [*TO] LN< BLK@ BLK( [*TO] IN( IN( [*TO] IN>
  INTERPRET IN$ R> [*TO] IN> ;
: LOADR 1+ SWAP DO I DUP . SPC> LOAD LOOP ;
```

B234

```
\ Application loader, to include in boot binary
: ED 120 LOAD 100 104 LOADR ;
: VE ED 123 LOAD 105 111 LOADR ;
: ME 123 LOAD 115 119 LOADR ;
: ARCHM 301 LOAD ; : ASML 2 LOAD ; : ASMH 3 LOAD ;
: RSH 150 154 LOADR ;
: AVRP 160 163 LOADR ;
: XCOMPL 200 LOAD ;
```

1.12 Grid subsystem: 240-241

B240

```
\ Grid subsystem. See doc/grid.txt. Load range: B240-B241
GRID_MEM *VALUE XYPOS
'? CURSOR! NOT [IF] : CURSOR! 2DROP ; [THEN]
: XYPOS! COLS LINES * MOD DUP XYPOS CURSOR! [*TO] XYPOS ;
: AT-XY ( x y -- ) COLS * + XYPOS! ;
'? NEWLN NOT [IF]
: NEWLN ( oldln -- newln )
  1+ LINES MOD DUP COLS * COLS RANGE DO SPC I CELL! LOOP ;
[THEN]
'? CELLS! NOT [IF]
: CELLS! ( a pos u -- )
  ?DUP IF RANGE DO ( a ) C@+ I CELL! LOOP
    ELSE DROP THEN DROP ; [THEN]
```

B241

```
: _lf XYPOS COLS / NEWLN COLS * XYPOS! ;
: _bs SPC XYPOS TUCK CELL! ( pos ) 1- XYPOS! ;
: (emit)
    DUP BS? IF DROP _bs EXIT THEN
    DUP CR = IF DROP SPC XYPOS CELL! _lf EXIT THEN
    DUP SPC < IF DROP EXIT THEN
    XYPOS CELL!
    XYPOS 1+ DUP COLS MOD IF XYPOS! ELSE DROP _lf THEN ;
: GRID$ 0 [*TO] XYPOS ;
```

1.13 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 ) ;
```

1.14 SD Card subsystem: 250-258

B250

```
\ SD Card subsystem Load range: B250-B258
SDC_MEM *VALUE 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 0 DO
        DROP _idle DUP $ff = NOT IF LEAVE THEN LOOP ;

( 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 0 DO ( c )
    << ( c<<1 ) DUP >>8 IF
      ( MSB was set, apply polynomial )
      <<8 >>8
      $12 XOR ( $09 << 1, we apply CRC on high bits )
    THEN
  LOOP ;
( 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 [*TO] SDC_SDHC DROP ;
```

B255

```
: _err 0 (spie) LIT" 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 0 DO _idle DROP LOOP
    0 ( dummy ) 10 0 DO  ( r )
        DROP $40 0 0 SDCMDR1  ( CMD0 )
        1 = DUP IF LEAVE THEN
    LOOP 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
  0 SWAP ( crc1 a ) 512 RANGE DO ( crc1 )
    _idle ( crc1 b ) DUP I C! ( crc1 b ) CRC16 LOOP ( 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 0 SWAP ( crc a )
  512 RANGE DO ( crc )
    I C@ ( crc b ) DUP (spix) DROP CRC16 LOOP ( crc )
    DUP >>8 ( crc msb ) (spix) DROP (spix) DROP
    _wait DROP _ready 0 (spie) ;
: SDC! ( blkno blk( -- )
  SWAP << ( 2x ) 2DUP ( a b a b ) _
  ( a b ) 1+ SWAP 512 + SWAP _ ;
```

1.15 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 0 _h 0
: _g ( given a top-left of dot-X in BLK(, spit H bin lines )
  _h 0 DO 0 _w 0 DO ( a r )
    << OVER I + C@ 'X' = IF 1+ THEN
  LOOP 8 _w - LSHIFT C, 64 + LOOP DROP ;
: _l ( a u -- a, spit a line of u glyphs )
  ( u ) 0 DO ( a ) DUP I _w * + _g LOOP ;
```

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 )
    270 267 DO I BLK@ BLK( 12 _l 448 + 12 _l DROP LOOP ( 72 )
    270 BLK@ BLK( 12 _l 448 + 10 _l DROP ( 94! ) ;
: CPFNT7x7 7 [TO] _w 7 [TO] _h
    _h ALLOT0 ( space char )
    276 271 DO I BLK@ BLK( 9 _l 448 + 9 _l DROP LOOP ( 90 )
    276 BLK@ BLK( 4 _l DROP ( 94! ) ;
```

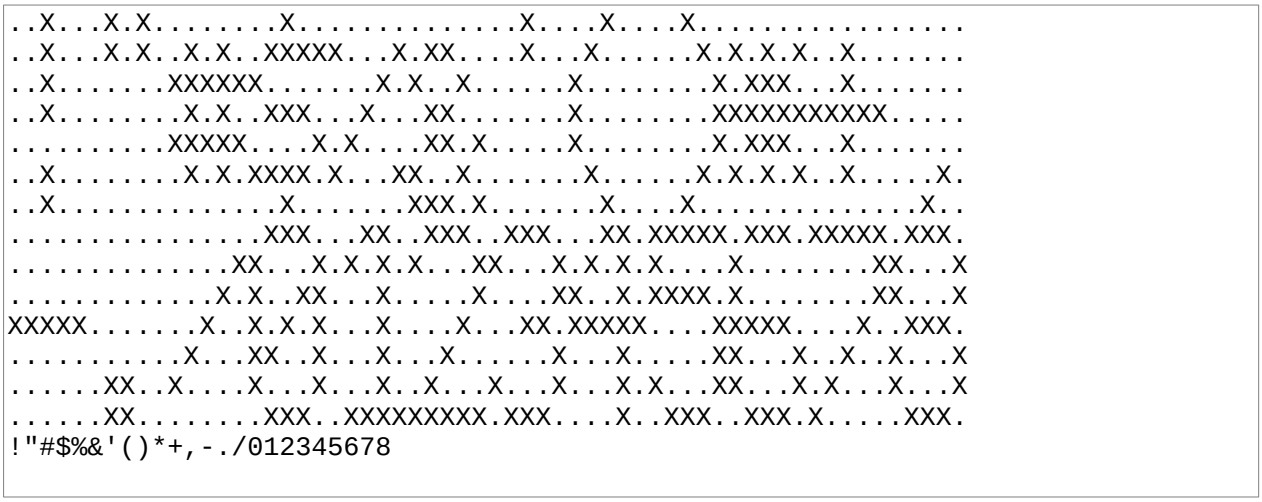
B265

```
.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.XXXXXXXX...XXX.
.XXXXXXXXXXX........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[\]^_
```

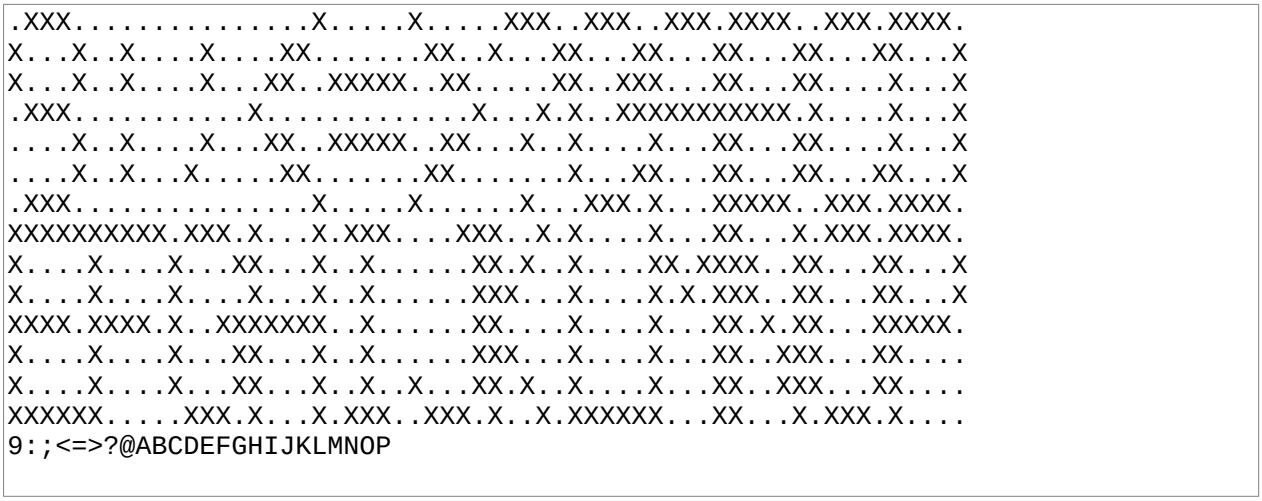
B266

```
X.....X.......X....XX...X...X...XX..XX.....................X.
.X.XX.X...XX..X.X.X...X.X........X.X.X.X.XXX..X.XX..XX.XX.XXXXX
.....XXX.X...XXX.XXX.X.XXX..X...XXX..X.XXXX.XX.XX.XX.XX..XX..X.
...XXXX.XX..X.XXX.X...XXX.X.X...XX.X.X.X.XX.XX.XXX..XXX....X.X.
...XXXXX..XX.XX.XXX..XX.X.X.X.XX.X.X.XXX.XX.X.X.X....XX..XX..XX
.....................XX.X.XX.....................................
X.XX.XX.XX.XX.XXXX.X..X..X..XX
X.XX.XX.X.X..X..XXX...X...XXX.
X.XX.XXXX.X..X.XX..X..X..X....
XXX.X.X.XX.X.X.XXX.XX.X.XX....
`abcdefghijklmnopqrstuvwxyz{|}~
```

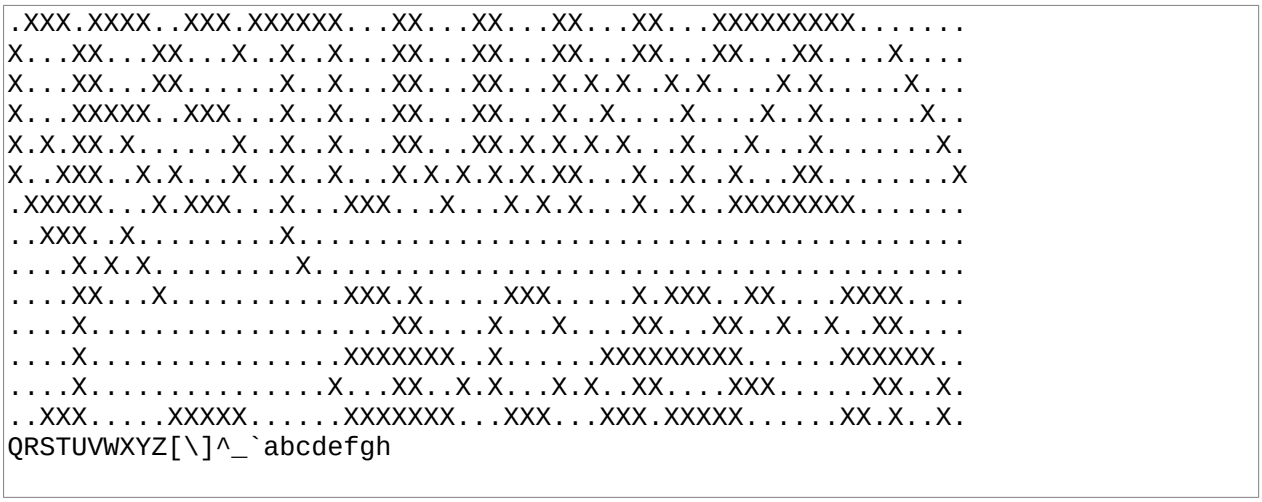
B267



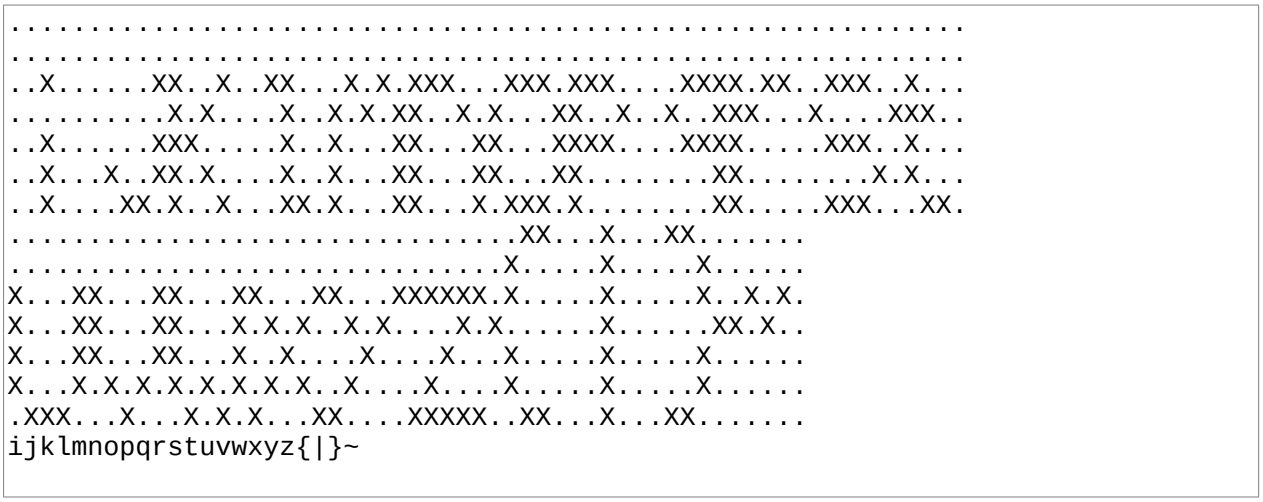
B268



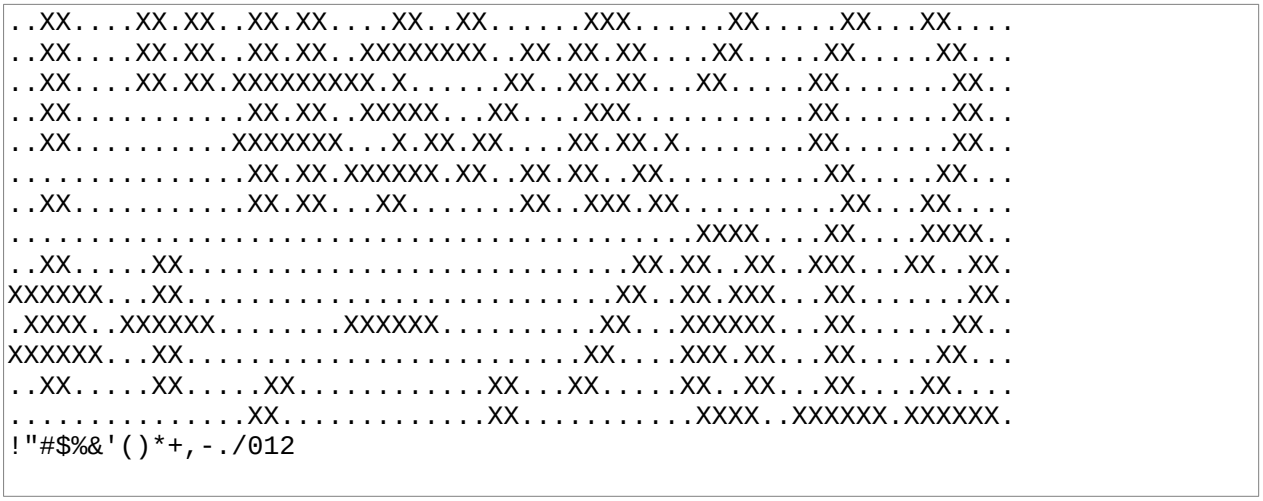
B269



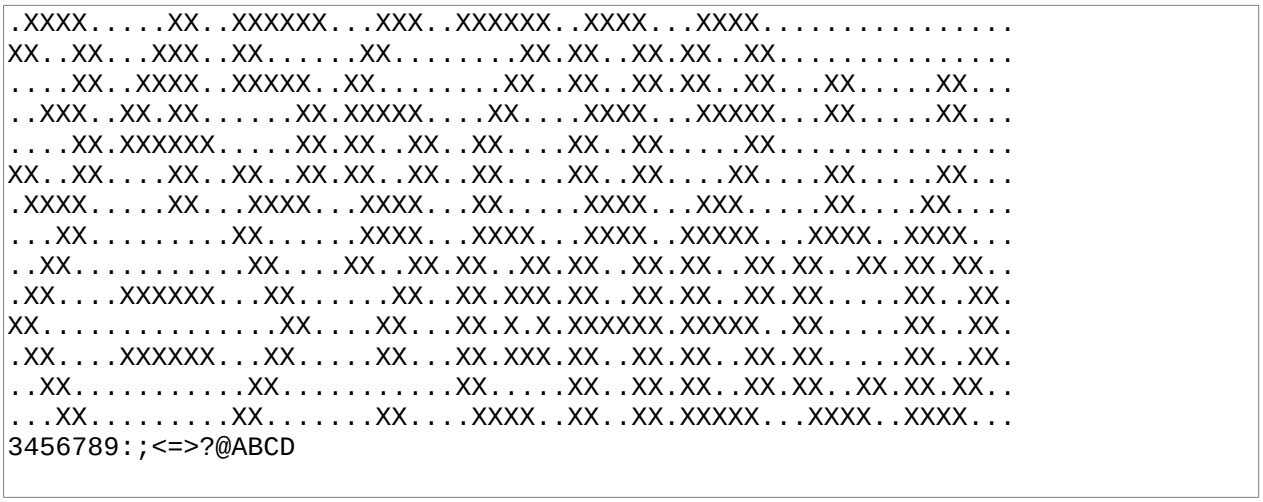
B270



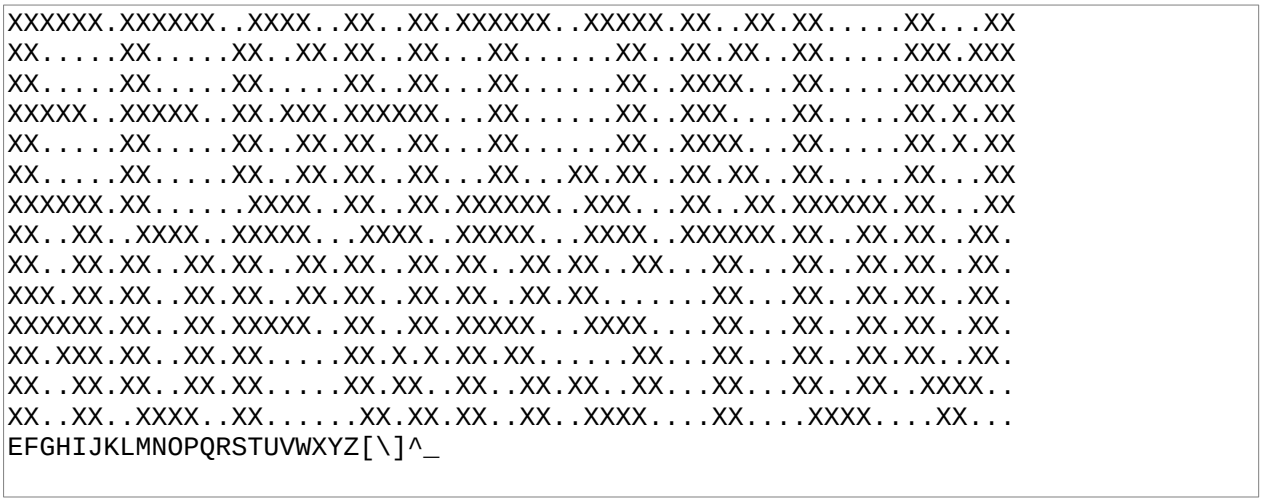
B271



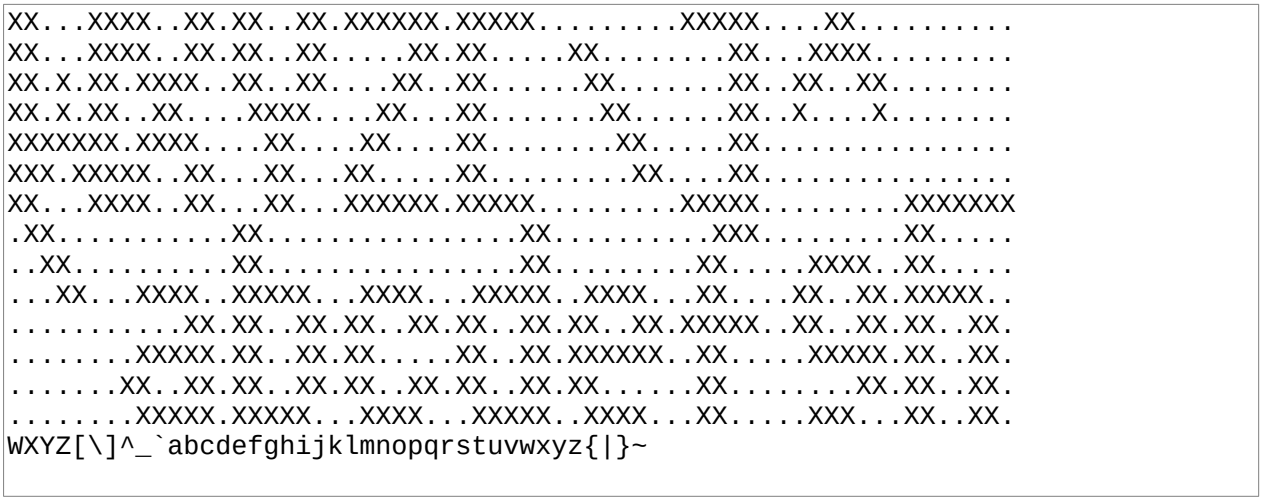
B272



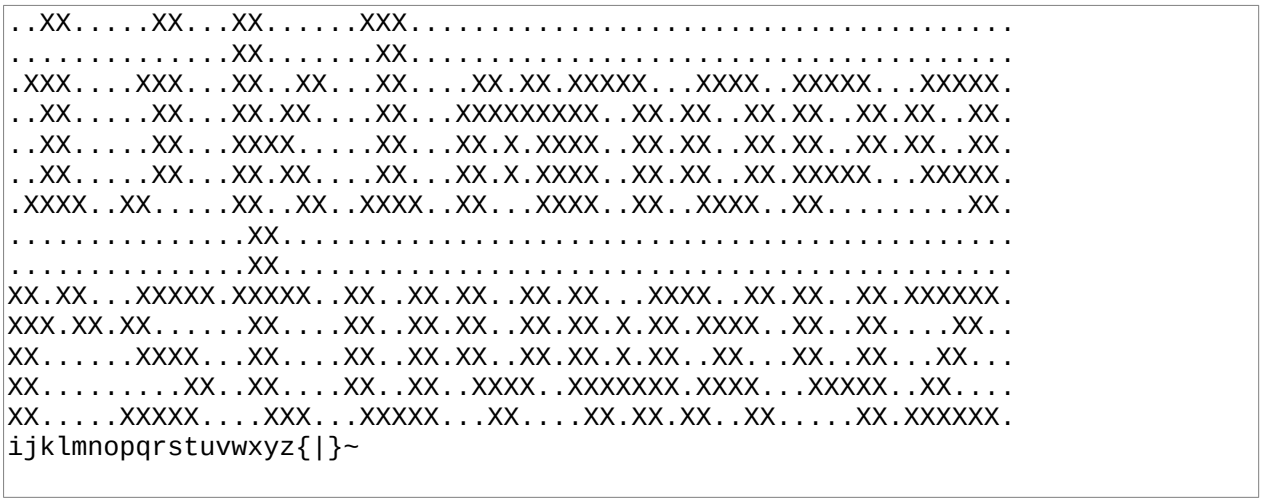
B273



B274



B275



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............
{|}~
```

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 TRS-80 4P drivers
395 Dan SBC drivers
```

2.2 Z80 boot code: 301-308

B301

```
\ Z80 port's Macros and constants. See doc/code/z80.txt
: Z80A ASML 320 327 LOADR 310 LOAD ( HAL flow ) ASMH ;
: Z80C 302 308 LOADR ; : Z80H 310 314 LOADR ;
: TRS804PM 380 LOAD ;
\ see comment at TICKS' definition
\ 7.373MHz target: 737t. outer: 37t inner: 16t
\ tickfactor = (737 - 37) / 16
44 VALUE tickfactor
1 VALUE JROPLEN -1 VALUE JROFF
```

B302

```
\ Z80 port's Code. Load range B281-B299
HERE TO ORG ( STABLE ABI )
FJR JRi, TO L1 ( B282 ) NOP, NOP, ( unused )
NOP, NOP, ( 04, BOOT ) NOP, NOP, ( 06 CURRENT )
NOP, NOP, ( 08, LATEST ) NOP, NOP, ( 0a (main) )
4 ALLOT0 LSET lblxt ( RST 10 )
  IX INCd, IX INCd, 0 IX+ E LDIXYr, 1 IX+ D LDIXYr,
  HL POP, LDDE(HL), HL INCd, EXDEHL, JP(HL), \ 17 bytes
7 ALLOT0
0 JP, ( RST 28 ) 5 ALLOT0
0 JP, ( RST 30 ) 5 ALLOT0
0 JP, ( RST 38 )
```

B303

```
L1 FMARK ( B281 )
  SP PS_ADDR LDdi, IX RS_ADDR LDdi,
  BIN( $04 ( BOOT ) + LDHL(i), JP(HL),
LSET lbldoes IX INCd, IX INCd, 0 IX+ E LDIXYr, 1 IX+ D LDIXYr,
  EXDEHL, \ continue to lblcell
LSET lblcell HL POP, \ continue to lblpush
LSET lblpush PUSHp, \ continue to lblnext
LSET lblnext EXDEHL, LDDE(HL), HL INCd, EXDEHL, JP(HL),
```

B304

```
( 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 DECd, ( 6t )
    IFZ, ( 12t ) BC POP, ;CODE THEN,
    A tickfactor LDri, ( 7t )
    BEGIN, A DECr, ( 4t ) Z? ^? BR ?JRi, ( 12t )
  BR JRi, ( 12t ) ( outer: 37t inner: 16t )
```

B305

```
CODE PC! HL POP, L OUT(C)r, BC POP, ;CODE
CODE PC@ C INr(C), B 0 LDri, ;CODE
CODE []= BC PUSH, EXX, ( protect DE ) BC POP, DE POP, HL POP,
  LSET L1 ( loop )
    LDA(DE), DE INCd, CPI,
    IFNZ, EXX, BC 0 LDdi, ;CODE THEN,
    CPE L1 JPc, ( BC not zero? loop )
  EXX, BC 1 LDdi, ;CODE
CODE (im1) IM1, EI, ;CODE
CODE * HL POP, DE PUSH, EXDEHL, ( DE * BC -> HL )
  HL 0 LDdi, A $10 LDri, BEGIN,
    HL ADDHLd, E RL, D RL,
    IFC, BC ADDHLd, THEN,
    A DECr, Z? ^? BR ?JRi,
  HL>BC, DE POP, ;CODE
```

B306

```
\ Divides AC by DE. quotient in AC remainder in HL
CODE /MOD BC>HL, BC POP, DE PUSH, EXDEHL,
  A B LDrr, B 16 LDri, HL 0 LDdi, BEGIN,
    SCF, C RL, RLA,
    HL ADCHLd, DE SBCHLd,
    IFC, DE ADDHLd, C DECr, THEN,
  BR DJNZ,
  DE POP, HL PUSH, B A LDrr, ;CODE
CODE QUIT LSET L1 ( used in ABORT )
IX RS_ADDR LDdi, BIN( $0a ( main ) + LDHL(i), JP(HL),
CODE ABORT SP PS_ADDR LDdi, L1 BR JRi,
CODE BYE HALT,
CODE RCNT RS_ADDR i>w, PUSHp, IX PUSH, HL POP, -wp, w>p, ;CODE
CODE SCNT 0 i>w, SP ADDHLd, PUSHp, PS_ADDR i>w, -wp, w>p, ;CODE
```

B307

```
CODE FIND ( sa sl -- w? f ) BC PUSH, EXX, BC POP, HL POP,
  BC ADDHLd, HL DECd, \ HL points to the last char of s
  DE SYSVARS $02 ( CURRENT ) + LDd(i),
  BEGIN, \ main loop
    DE DECd, LDA(DE), $7f ANDi, ( IMMEDIATE ) C CPr, IFZ,
      HL PUSH, DE PUSH, BC PUSH,
      DE DECd, DE DECd, \ Skip prev field
      LSET L1 ( loop )
        DE DECd, LDA(DE), CPD, IFZ, TO L2 ( break! )
      CPE L1 JPc, ( BC not zero? loop ) L2 FMARK
      BC POP, DE POP, HL POP, THEN,
```

B308

```
\ At this point, Z is set if we have a match.
    IFZ, ( match ) DE INCd, DE PUSH, EXX, BC 1 LDdi, ;CODE THEN,
\ no match, go to prev and continue
\ we read prev field backwards
    DE DECd, LDA(DE), EXAFAF', DE DECd, LDA(DE),
    E A LDrr, EXAFAF', D A LDrr,
    E ORr, IFZ, \ DE=0, end of dict
      EXX, BC 0 LDdi, ;CODE THEN,
  BR JRi, \ main loop
```

2.3 Z80 HAL: 310-314

B310

```
\ Z80 HAL, flow words. also used in Z80A
SYSVARS $16 + *VALUE ?JROP
: JMPi, $c3 C, L, ;
: CALLi, DUP $38 AND OVER = IF
  ( RST ) $c7 OR C, ELSE $cd C, L, THEN ;
: JRi, $18 C, ( JR ) C, ;
: ?JRi, ?JROP C, C, ;
: Z? $28 [*TO] ?JROP ; : C? $38 [*TO] ?JROP ;
: ^? ?JROP 8 XOR [*TO] ?JROP ;
```

B311

```
\ Z80 HAL, Stack
: w>p, $444d M, ; \ ld b,h; ld c,l
: p>w, $6069 M, ; \ ld h,b; ld l,c
: DROPp, $c1 C, ( pop bc ) ; : POPp, p>w, DROPp, ;
: DUPp, $c5 C, ( push bc ) ; : PUSHp, DUPp, w>p, ;
: POPf, $e1 C, ( pop hl ) ; : PUSHf, $e5 C, ( push hl ) ;
: POPr, $dd6e M, $00 C, ( ld l,ix+0 )
        $dd66 M, $01 C, ( ld h,ix+1 )
        $dd2b M, ( dec ix ) $dd2b M, ( dec ix ) ;
: PUSHr, $dd23 M, ( inc ix ) $dd23 M, ( inc ix )
         $dd75 M, $00 C, ( ld l,ix+0 )
         $dd74 M, $01 C, ( ld h,ix+1 ) ;
\ ld a,h; h,b; b,a; a,l; l,c; c,a
: SWAPwp, $7c60 M, $477d M, $694f M, ;
: SWAPwf, $e3 C, ; \ ex (sp),hl
```

B312

```
\ Z80 HAL, Jumps, Transfer
: JMPw, $e9 C, ; \ jp (hl)
: INCw, $23 C, ( inc hl ) ; : DECw, $2b C, ( dec hl ) ;
: INCp, $03 C, ( inc bc ) ; : DECp, $0b C, ( dec bc ) ;
: i>w, $21 C, L, ; \ ld hl,nn
: (i)>w, $2a C, L, ; \ ld hl,(nn)
: C@w, $6e C, $2600 M, ; \ ld l,(hl); ld h,0
: @w, $7e23 M, $666f M, ; \ ld a,(hl); inc hl; h,(hl); l,a
: C!wp, $71 C, ; \ ld (hl),c
: !wp, C!wp, INCw, $70 C, ; \ ld (hl),b
```

B313

```
\ Z80 HAL, Transfer
: w>Z, $7db4 M, ; \ ld a,l; or h
: p>Z, $78b1 M, ; \ ld a,c; or b
: C>w, 0 i>w, $ed6a M, ; \ adc hl,hl
: Z>w, 0 i>w, Z? ^? 1 ?JRi, INCw, ;
: IP>w, $626b M, ; \ ld h,d; ld l,e
: w>IP, $545d M, ; \ ld d,h; ld e,l
: IP+, $13 C, ; \ inc de
: IP+off, $1a C, ( ld a,(de) ) $cb7f M, ( bit 7,a )
  Z? 1 ?JRi, $15 C, ( dec d ) $83 C, ( add e )
  C? ^? 1 ?JRi, $14 C, ( inc d ) $5f C, ( ld e,a ) ;
```

B314

```
\ Z80 HAL, Arithmetic
: +wp, $09 C, ; \ add hl,bc
: -wp, $b7 C, $ed42 M, ; \ or a; sbc hl,bc
: >>w, $cb3c M, $cb1d M, ; \ srl h; rr l
: <<w, $29 C, ; \ add hl,hl
: >>8w, $6c C, $2600 M, ; \ ld l,h; ld h,0
: <<8w, $65 C, $2e00 M, ; \ ld h,l; ld l,0
: CMPwp, $7cb8 M, Z? ^? 2 ?JRi, $7db9 M, ; \ a,h; cpb; a,l; cpc
: ANDwp, $7ca0 M, $677d M, $a16f M, ; \ a,h;and b;h,a;a,l;and c
: ORwp, $7cb0 M, $677d M, $b16f M, ; \ a,h;or b;h,a;a,l;or c;l,a
: XORwp, $7ca8 M, $677d M, $a96f M, ; \ a,h;xor b;h,a;a,l;xor c
: XORwi, $7cee M, DUP >>8 C, $677d M, $ee C, C, $6f C, ;
```

2.4 Z80 assembler: 320-327

B320

```
\ Z80 Assembler. See doc/asm.txt
21 VALUES A 7 B 0 C 1 D 2 E 3 H 4 L 5 (HL) 6
          BC 0 DE 1 HL 2 AF 3 SP 3
          CNZ 0 CZ 1 CNC 2 CC 3 CPO 4 CPE 5 CP 6 CM 7
\ As a general rule, IX and IY are equivalent to spitting an
\ extra $dd / $fd and then spit the equivalent of HL
: IX $dd C, HL ; : IY $fd C, HL ;
: IX+ <<8 >>8 $dd C, (HL) ;
: IY+ <<8 >>8 $fd C, (HL) ;
: OPXY DOER , DOES> @ ( xyoff opref ) EXECUTE C, ;
```

B321

```
: OP1 DOER C, DOES> C@ C, ;
$f3 OP1 DI,                    $fb OP1 EI,
$eb OP1 EXDEHL,                $d9 OP1 EXX,
$08 OP1 EXAFAF',               $e3 OP1 EX(SP)HL,
$76 OP1 HALT,                  $e9 OP1 JP(HL),
$12 OP1 LD(DE)A,               $1a OP1 LDA(DE),
$02 OP1 LD(BC)A,               $0a OP1 LDA(BC),
$00 OP1 NOP,                   $c9 OP1 RET,
$17 OP1 RLA,                   $07 OP1 RLCA,
$1f OP1 RRA,                   $0f OP1 RRCA,
$37 OP1 SCF,
```

B322

```
: OP1r DOER C, DOES> C@ ( r op ) SWAP <<3 OR C, ;
$04 OP1r INCr,                $05 OP1r DECr,
' INCr, OPXY INC(IXY+),        ' DECr, OPXY DEC(IXY+),
\ OP1r also works for conditions
$c0 OP1r RETc,

: OP1r0 DOER C, DOES> C@ ( r op ) OR C, ;
$80 OP1r0 ADDr,               $88 OP1r0 ADCr,
$a0 OP1r0 ANDr,               $b8 OP1r0 CPr,
$b0 OP1r0 ORr,                $90 OP1r0 SUBr,
$98 OP1r0 SBCr,               $a8 OP1r0 XORr,
' CPr, OPXY CP(IXY+),
```

B323

```
: OP1d DOER C, DOES> C@ ( d op ) SWAP <<4 OR C, ;
$c5 OP1d PUSH,                $c1 OP1d POP,
$03 OP1d INCd,                $0b OP1d DECd,
$09 OP1d ADDHLd,
: ADDIXd, IX DROP ADDHLd, ;  : ADDIXIX, HL ADDIXd, ;
: ADDIYd, IY DROP ADDHLd, ;  : ADDIYIY, HL ADDIYd, ;

: LDrr, ( rd rr ) SWAP <<3 OR $40 OR C, ;
' LDrr, OPXY LDIXYr,
: LDrIXY, ( rd ixy+- HL ) ROT SWAP LDIXYr, ;
: LDri, ( r i ) SWAP <<3 $06 OR C, C, ;
: LDdi, ( d n ) SWAP <<4 $01 OR C, L, ;
: LDd(i), ( d i ) $ed C, SWAP <<4 $4b OR C, L, ;
: LD(i)d, ( i d ) $ed C, <<4 $43 OR C, L, ;
```

B324

```
: OPED DOER C, DOES> $ed C, C@ C, ;
$a1 OPED CPI,       $b1 OPED CPIR,     $a9 OPED CPD,
$b9 OPED CPDR,      $46 OPED IM0,      $56 OPED IM1,
$5e OPED IM2,       $a0 OPED LDI,      $b0 OPED LDIR,
$a8 OPED LDD,       $b8 OPED LDDR,     $44 OPED NEG,
$4d OPED RETI,      $45 OPED RETN,     $a2 OPED INI,
$aa OPED IND,       $a3 OPED OUTI,

: OP2i DOER C, DOES> C@ ( i op ) C, C, ;
$d3 OP2i OUTiA,     $db OP2i INAi,
$c6 OP2i ADDi,      $ce OP2i ADCi,
$e6 OP2i ANDi,      $f6 OP2i ORi,      $d6 OP2i SUBi,
$ee OP2i XORi,      $fe OP2i CPi,
$18 OP2i JR,        $10 OP2i DJNZ,     $38 OP2i JRC,
$30 OP2i JRNC,      $28 OP2i JRZ,      $20 OP2i JRNZ,
```

B325

```
: OP2br DOER C, DOES>
    $cb C, C@ ( b r op ) ROT <<3 OR OR C, ;
$c0 OP2br SET,      $80 OP2br RES,     $40 OP2br BIT,
\ bitwise rotation ops have a similar sig
: OProt DOER C, DOES> $cb C, C@ ( r op ) OR C, ;
$10 OProt RL,       $00 OProt RLC,     $18 OProt RR,
$08 OProt RRC,      $20 OProt SLA,     $38 OProt SRL,

\ cell contains both bytes. MSB is spit as-is, LSB is ORed
\ with r.
: OP2r DOER , DOES> @ L|M ( r lsb msb ) C, SWAP <<3 OR C, ;
$ed41 OP2r OUT(C)r, $ed40 OP2r INr(C),

: OP2d DOER C, DOES> $ed C, C@ ( d op ) SWAP <<4 OR C, ;
$4a OP2d ADCHLd,    $42 OP2d SBCHLd,
```

B326

```
: OP3i DOER C, DOES> C@ ( i op ) C, L, ;
$cd OP3i CALL,                $c3 OP3i JP,
$22 OP3i LD(i)HL,             $2a OP3i LDHL(i),
$32 OP3i LD(i)A,              $3a OP3i LDA(i),

: RST, $c7 OR C, ;
: JP(IX), IX DROP JP(HL), ;
: JP(IY), IY DROP JP(HL), ;
: JPc, SWAP <<3 $c2 OR C, L, ;
: CALLc, SWAP <<3 $c4 OR C, L, ;
```

B327

```
\ Macros
: SUBHLd, A ORr, SBCHLd, ; \ clear carry + SBC
: PUSHA, B 0 LDri, C A LDrr, BC PUSH, ;
: HLZ, A H LDrr, L ORr, ;
: DEZ, A D LDrr, E ORr, ;
: BCZ, A B LDrr, C ORr, ;
: LDDE(HL), E (HL) LDrr, HL INCd, D (HL) LDrr, ;
: LDBC(HL), C (HL) LDrr, HL INCd, B (HL) LDrr, ;
: LDHL(HL), A (HL) LDrr, HL INCd, H (HL) LDrr, L A LDrr, ;
: OUTHL, DUP A H LDrr, OUTiA, A L LDrr, OUTiA, ;
: OUTDE, DUP A D LDrr, OUTiA, A E LDrr, OUTiA, ;
: HL>BC, B H LDrr, C L LDrr, ;
: BC>HL, H B LDrr, L C LDrr, ;
: A>BC, C A LDrr, B 0 LDri, ;
: A>HL, L A LDrr, H 0 LDri, ;
```

2.5 AT28 EEPROM: 330

B330

```
CODE AT28C! ( c a -- )
  BC>HL, BC POP,
  (HL) C LDrr, A C LDrr, ( orig ) B C LDrr, ( save )
  C (HL) LDrr, ( poll ) BEGIN,
    A (HL) LDrr, ( poll ) C CPr, ( same as old? )
    C A LDrr, ( save old poll, Z preserved )
  Z? ^? BR ?JRi,
\ equal to written? SUB instead of CP to ensure IOERR is NZ
  B SUBr, IFNZ, SYSVARS ( IOERR ) LD(i)A, THEN, BC POP, ;CODE
: AT28! ( n a -- ) 2DUP AT28C! 1+ SWAP >>8 SWAP AT28C! ;
```

2.6 SPI relay: 332

B332

```
( SPI relay driver. See doc/hw/z80/spi.txt )
CODE (spix) ( n -- n )
  A C LDrr,
  SPI_DATA OUTiA,
  ( wait until xchg is done )
  BEGIN, SPI_CTL INAi, 1 ANDi, Z? ^? BR ?JRi,
  SPI_DATA INAi,
  C A LDrr, ;CODE
CODE (spie) ( n -- )
  A C LDrr, SPI_CTL OUTiA, BC POP, ;CODE
```

2.7 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 LDrr, TMS_CTLPORT OUTiA, A B LDrr, TMS_CTLPORT OUTiA,
  BC POP, ;CODE
CODE _data
  A C LDrr, TMS_DATAPORT OUTiA, BC POP, ;CODE
```

B336

```
: _zero ( x -- send 0 _data x times )
    ( x ) 0 DO 0 _data LOOP ;
( 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 -- Send font to TMS )
    7 0 DO C@+ _data LOOP DROP
    ( blank row ) 0 _data ;
: _sfont^ ( a -- Send inverted font to TMS )
    7 0 DO C@+ $ff XOR _data LOOP DROP
    ( 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 0 DO ~FNT I 7 * + _sfont LOOP
    $4400 _ctl $5f 0 DO ~FNT I 7 * + _sfont^ LOOP
    $820e _ctl ( name table $3800 )
    $8400 _ctl ( pattern table $0000 )
    $87f0 _ctl ( colors 0 and 1 )
    $8000 _ctl $81d0 _ctl ( text mode, display on ) ;
```

2.8 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,
    6850_CTL INAi, $02 ANDi, ( are we transmitting? )
  Z? BR ?JRi, ( yes, loop )
  A C LDrr, 6850_IO OUTiA, BC POP, ;CODE
```

B341

```
CODE 6850<? BC PUSH,
  A XORr, ( 256x ) A $16 ( RTS lo ) LDri, 6850_CTL OUTiA,
  BC 0 LDdi, ( pre-push a failure )
  BEGIN, EXAFAF', ( preserve cnt )
    6850_CTL INAi, $1 ANDi, ( rcv buff full? )
    IFNZ, ( full )
      6850_IO INAi, PUSHA, C 1 LDri, A XORr, ( end loop )
    ELSE, EXAFAF', ( recall cnt ) A DECr, THEN,
  Z? ^? BR ?JRi,
  A $56 ( RTS hi ) LDri, 6850_CTL OUTiA, ;CODE
```

B342

```
X' 6850<? ALIAS RX<? X' 6850<? ALIAS (key?)
X' 6850> ALIAS TX> X' 6850> ALIAS (emit)
: 6850$ $56 ( RTS high ) [ 6850_CTL LITN ] PC! ;
```

2.9 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,
  A XORr, ( 256x ) BC 0 LDdi, ( pre-push a failure )
  A 5 ( PTR5 ) LDri, SIOA_CTL OUTiA,
  A $68 ( RTS low ) LDri, SIOA_CTL OUTiA,
  BEGIN, EXAFAF', ( preserve cnt )
    SIOA_CTL INAi, $1 ANDi, ( rcv buff full? )
    IFNZ, ( full )
      SIOA_DATA INAi, PUSHA, C 1 LDri, A XORr, ( end loop )
    ELSE, EXAFAF', ( recall cnt ) A DECr, THEN,
  Z? ^? BR ?JRi,
  A 5 ( PTR5 ) LDri, SIOA_CTL OUTiA,
  A $6a ( RTS high ) LDri, SIOA_CTL OUTiA, ;CODE
```

B346

```
CODE SIOA>
  BEGIN,
    SIOA_CTL INAi, $04 ANDi, ( are we transmitting? )
  Z? BR ?JRi, ( yes, loop )
  A C LDrr, SIOA_DATA OUTiA, 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 )
    $21 C, ( CMD2/PTR1 ) 0 C, ( WR1/Rx no INT )
: SIOA$ _ 9 RANGE DO I C@ [ SIOA_CTL LITN ] PC! LOOP ;
```

B347

```
CODE SIOB<? BC PUSH, ( copy/paste of SIOA<? )
  A XORr, ( 256x ) BC 0 LDdi, ( pre-push a failure )
  A 5 ( PTR5 ) LDri, SIOB_CTL OUTiA,
  A $68 ( RTS low ) LDri, SIOB_CTL OUTiA,
  BEGIN, EXAFAF', ( preserve cnt )
    SIOB_CTL INAi, $1 ANDi, ( rcv buff full? )
    IFNZ, ( full )
      SIOB_DATA INAi, PUSHA, C 1 LDri, A XORr, ( end loop )
    ELSE, EXAFAF', ( recall cnt ) A DECr, THEN,
  Z? ^? BR ?JRi,
  A 5 ( PTR5 ) LDri, SIOB_CTL OUTiA,
  A $6a ( RTS high ) LDri, SIOB_CTL OUTiA, ;CODE
```

B348

```
CODE SIOB>
  BEGIN,
    SIOB_CTL INAi, $04 ANDi, ( are we transmitting? )
  Z? BR ?JRi, ( yes, loop )
  A C LDrr, SIOB_DATA OUTiA, BC POP, ;CODE
: SIOB$ _ 9 RANGE DO I C@ [ SIOB_CTL LITN ] PC! LOOP ;
```

2.10 Sega Master System VDP: 350-352

B350

```
\ VDP Driver. see doc/hw/sms/vdp.txt. 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

```
: _sfont ( a -- Send font to VDP )
  7 RANGE DO I C@ _data 3 _zero LOOP ( blank row ) 4 _zero ;
: CELL! ( c pos )
  2 * $7800 OR _ctl ( c )
  $20 - ( glyph ) $5f MOD _data ;
```

B352

```
: 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 0 DO _idat I 2 * + @ _ctl LOOP
  ( 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 0 DO ~FNT I 7 * + _sfont LOOP
  ( bit 6, enable display, bit 7, ?? ) $81c0 _ctl ;
: COLS 32 ; : LINES 24 ;
```

2.11 SMS PAD: 355-358

B355

```
( SMS pad driver. See doc/hw/z80/sms/pad.txt.
  Load range: 335-338 )
: _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@+ I > UNTIL ( a R:c ) R> DROP
  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! ;
```

2.12 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 ) ;
```

2.13 SMS SPI relay: 367

B367

```
: (spie) DROP ; ( always enabled )
CODE (spix) ( x -- x, for port B )
  ( TR = DATA TH = CLK )
  CPORT_MEM LDA(i), $f3 ANDi, ( TR/TH output )
  B 8 LDri, BEGIN,
    $bf ANDi, ( TR lo ) C RL,
    IFC, $40 ORi, ( TR hi ) THEN,
    CPORT_CTL OUTiA, ( clic! ) $80 ORi, ( TH hi )
    CPORT_CTL OUTiA, ( clac! )
    EXAFAF', CPORT_D1 INAi, ( Up Btn is B6 ) RLA, RLA,
      L RL, EXAFAF',
    $7f ANDi, ( TH lo ) CPORT_CTL OUTiA, ( cloc! )
  BR DJNZ, CPORT_MEM LD(i)A,
  C L LDrr, ;CODE
```

2.14 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,
  $11 ANDi, C A LDrr, CPORT_MEM LDA(i),
  $ee ANDi, C ORr, CPORT_CTL OUTiA, CPORT_MEM LD(i)A,
  BC POP, ;CODE
CODE _THA! ( B0 -> B5, B1 -> B1 )
  C RR, RLA, RLA, RLA, RLA, C RR, RLA, RLA,
  $22 ANDi, C A LDrr, CPORT_MEM LDA(i),
  $dd ANDi, C ORr, CPORT_CTL OUTiA, CPORT_MEM LD(i)A,
  BC POP, ;CODE
```

B369

```
CODE _TRB! ( B0 -> B6, B1 -> B2 )
  C RR, RLA, RLA, RLA, RLA, C RR, RLA, RLA, RLA,
  $44 ANDi, C A LDrr, CPORT_MEM LDA(i),
  $bb ANDi, C ORr, CPORT_CTL OUTiA, CPORT_MEM LD(i)A,
  BC POP, ;CODE
CODE _THB! ( B0 -> B7, B1 -> B3 )
  C RR, RLA, RLA, RLA, RLA, C RR, RLA, RLA, RLA, RLA,
  $88 ANDi, C A LDrr, CPORT_MEM LDA(i),
  $77 ANDi, C ORr, CPORT_CTL OUTiA, CPORT_MEM LD(i)A,
  BC POP, ;CODE
CODE _D1@ BC PUSH, CPORT_D1 INAi, C A LDrr, B 0 LDri, ;CODE
CODE _D2@ BC PUSH, CPORT_D2 INAi, C A LDrr, B 0 LDri, ;CODE
```

2.15 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 3 ; : FNTH 5 ;
: 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,
    $10 ( CMD ) INAi,
    RLA, ( When 7th bit is clr, we can send a new cmd )
  C? BR ?JRi, ;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 ;
: _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 )
  SWAP _row! ( u ) 0 DO
    _yinc 0 _col!
    11 0 DO 0 _data! LOOP
    _xinc 0 _data! LOOP ;
: 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
  FNTH 0 DO ( pos coff a )
    OVER 8 -^ SWAP C@+ ( pos coff 8-coff a+1 c ) ROT LSHIFT
    _data@ <<8 OR
    LCD_BUF I + 2DUP FNTH + C!
    SWAP >>8 SWAP C!
  LOOP 2DROP
  DUP _atrow!
  FNTH 0 DO LCD_BUF I + C@ _data! LOOP
  DUP _atrow! _tocol NIP 1+ _col!
  FNTH 0 DO LCD_BUF FNTH + I + C@ _data! LOOP ;
```

2.16 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 LDri,
    KBD_PORT OUTiA,
    A C LDrr,
    KBD_PORT OUTiA,
    KBD_PORT INAi,
  EI,
  C A LDrr,
;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 0 DO
        1 I LSHIFT $ff -^ ( group dmask ) _get
        DUP $ff = IF DROP ELSE I ( dmask gid ) LEAVE THEN
    LOOP _wait
    SWAP ( gid dmask )
    $ff XOR ( dpos ) 0 ( dindex )
    BEGIN 1+ 2DUP RSHIFT NOT UNTIL 1-
    ( gid dpos dindex ) NIP
    ( gid dindex ) SWAP 8 * + ;
```

B379

```
: (key?) ( -- c? f )
    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! ;
```

2.17 TRS-80 4P drivers: 380-390

B380

```
\ TRS-80 drivers declarations and macros
: TRS804PL 381 388 LOADR ; : TRS804PH 389 LOAD ;
$f800 VALUE VIDMEM $bf VALUE CURCHAR
: fdstat $f0 INAi, ;
: fdcmd A SWAP LDri, B $18 LDri,
  $f0 OUTiA, BEGIN, BR DJNZ, ;
: fdwait BEGIN, fdstat RRCA, C? BR ?JRi, RLCA, ;
: vid+, ( reg -- ) HL VIDMEM LDdi, ADDHLd, ;
```

B381

```
\ TRS-80 4P video driver
24 VALUE LINES 80 VALUE COLS
CODE CELL! ( c pos -- ) HL POP,
  A L LDrr, BC vid+, (HL) A LDrr, BC POP, ;CODE
CODE CELLS! ( a pos u -- ) BC PUSH, EXX, BC POP, DE POP,
  DE vid+, EXDEHL, HL POP, BCZ, IFNZ, LDIR, THEN, EXX, BC POP,
;CODE
CODE CURSOR! ( new old -- ) BC vid+, A (HL) LDrr, CURCHAR CPi,
  IFZ, UNDERCUR LDA(i), (HL) A LDrr, THEN,
  BC POP, BC vid+, A (HL) LDrr, UNDERCUR LD(i)A, A CURCHAR LDri,
  (HL) A LDrr, BC POP, ;CODE
CODE SCROLL ( -- )
  EXX, HL VIDMEM 80 + LDdi, DE VIDMEM LDdi, BC 1840 LDdi, LDIR,
  H D LDrr, L E LDrr, DE INCd, A SPC LDri, (HL) A LDrr,
  BC 79 LDdi, LDIR, EXX, ;CODE
: NEWLN ( old -- new ) 1+ DUP LINES = IF 1- SCROLL THEN ;
```

B382

```
LSET L2 ( seek, B=trk )
  A 21 LDri, B CPr, FDMEM LDA(i), IFC, $20 ORi, ( WP ) THEN,
  $80 ORi, $f4 OUTiA, \ FD sel
  A B LDrr, ( trk ) $f3 OUTiA, $1c fdcmd RET,
CODE FDRD ( trksec addr -- st ) BC>HL, BC POP,
  L2 CALL, fdwait $98 ANDi, IFZ, DI,
    A C LDrr, $f2 OUTiA, ( sec ) C $f3 LDri, $84 fdcmd ( read )
    BEGIN, BEGIN, fdstat $b6 ANDi, Z? BR ?JRi, \ DRQ
      $b4 ANDi, IFZ, TO L3 ( error ) INI, Z? ^? BR ?JRi, THEN,
  fdwait $3c ANDi, L3 FMARK A>BC, EI, ;CODE
CODE FDWR ( trksec addr -- st ) BC>HL, BC POP,
  L2 CALL, fdwait $98 ANDi, IFZ, DI,
    A C LDrr, $f2 OUTiA, ( sec ) C $f3 LDri, $a4 fdcmd ( read )
    BEGIN, BEGIN, fdstat $f6 ANDi, Z? BR ?JRi, \ DRQ
      $f4 ANDi, IFZ, TO L3 ( error ) OUTI, Z? ^? BR ?JRi, THEN,
  fdwait $3c ANDi, L3 FMARK A>BC, EI, ;CODE
```

B383

```
FDMEM 1+ *ALIAS FDOP
: _err LIT" 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 + ;
: FD@! ( blk blk( -- )
  SWAP << << ( blk( blk*4=sec ) 4 RANGE DO ( dest )
    I _trksec OVER ( dest trksec dest )
    FDOP ( dest ) ?DUP IF _err THEN $100 +
  LOOP DROP ;
: FD@ ['] FDRD [*TO] FDOP FD@! ;
: FD! ['] FDWR [*TO] FDOP FD@! ;
CODE FDSEL ( fdmask -- )
  A C LDrr, BC POP, FDMEM LD(i)A, $80 ORi, $f4 OUTiA,
  0 fdcmd ( restore ) fdwait ;CODE
: FD$ 2 FDSEL ;
```

B384

```
: CL$ ( baudcode -- )
  $02 $e8 PC! ( UART RST )
  DUP << << << << OR $e9 PC! ( bauds )
  $6d $ea PC! ( word8 no parity no-RTS ) ;
CODE TX> BEGIN,
    $ea INAi, $40 ANDi, IFNZ, ( TX reg empty )
      $e8 INAi, $80 ANDi, IFZ, ( CTS low )
        A C LDrr, $eb OUTiA, ( send byte ) BC POP, ;CODE
  THEN, THEN, BR JRi,
```

B385

```
CODE RX<? BC PUSH,
  A XORr, ( 256x ) BC 0 LDdi, ( pre-push a failure )
  A $6c ( RTS low ) LDri, $ea OUTiA,
  BEGIN, EXAFAF', ( preserve cnt )
    $ea INAi, $80 ANDi, ( rcv buff full? )
    IFNZ, ( full )
      $eb INAi, A>HL, HL PUSH, C INCr, A XORr, ( end loop )
    ELSE, EXAFAF', ( recall cnt ) A DECr, THEN,
  Z? ^? BR ?JRi,
  A $6d ( RTS high ) LDri, $ea OUTiA, ;CODE
```

B386

```
LSET L1 6 nC, '`' 'h' 'p' 'x' '0' '8'
LSET L2 8 nC, $0d 0 $ff 0 0 $08 0 $20
PC ORG $39 + T! ( RST 38 )
AF PUSH, HL PUSH, DE PUSH, BC PUSH,
$ec INAi, ( RTC INT ack )
$f440 LDA(i), A ORr, IFNZ, \ 7th row is special
  HL L2 1- LDdi, BEGIN, HL INCd, RRA, C? ^? BR ?JRi,
  A (HL) LDrr, ELSE, \ not 7th row
  HL L1 LDdi, DE $f401 LDdi, BC $600 LDdi, BEGIN,
    LDA(DE), A ORr, IFNZ,
      C (HL) LDrr, BEGIN, C INCr, RRA, C? ^? BR ?JRi,
      C DECr, THEN,
    E SLA, HL INCd, BR DJNZ,
  A C LDrr, THEN, \ cont.
```

B387

```
\ A=char or zero if no keypress. Now let's debounce
HL KBD_MEM 2 + LDdi, A ORr, IFZ, \ no keypress, debounce
  (HL) A LDrr, ELSE, \ keypress, is it debounced?
  (HL) CPr, IFNZ, \ != debounce buffer
    C A LDrr, (HL) C LDrr, $ff CPi, IFZ, \ BREAK!
      HL POP, HL POP, HL POP, HL POP, HL POP, EI,
      X' QUIT JP, THEN,
    HL DECd, $f480 LDA(i), 3 ANDi, (HL) A LDrr, HL DECd,
    (HL) C LDrr, THEN, THEN,
BC POP, DE POP, HL POP, AF POP, EI, RET,
```

B388

```
KBD_MEM *VALUE KBDBUF \ LSB=char MSB=shift
: KBD$ 0 [*TO] KBDBUF $04 $e0 PC! ( enable RTC INT ) (im1) ;
: (key?) KBDBUF DUP <<8 >>8 NOT IF DROP 0 EXIT THEN
  0 [*TO] 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 ) ;
```

B389

```
: FD0 FLUSH 1 FDSEL ;
: FD1 FLUSH 2 FDSEL ;
```

B390

```
\ TRS-80 4P bootloader. Loads sectors 2-17 to addr 0.
HERE TO ORG
DI, A $86 LDri, $84 OUTiA, \ mode 2, 80 chars, page 1
A $81 LDri, $f4 OUTiA, \ DRVSEL DD, drv0
A $40 LDri, $ec OUTiA, \ MODOUT 4MHZ, no EXTIO
HL 0 LDdi, ( dest addr ) A XORr, $e4 OUTiA, ( no NMI )
A INCr, ( trk1 ) BEGIN,
  $f3 OUTiA, EXAFAF', ( save ) $18 ( seek ) fdcmd fdwait
  A XORr, $f2 OUTiA, C $f3 LDri, BEGIN,
    $80 ( read sector ) fdcmd ( B=0 )
    BEGIN, fdstat RRA, RRA, C? ^? BR ?JRi, ( DRQ )
    INI, A $c1 LDri, BEGIN, $f4 OUTiA, INI, Z? ^? BR ?JRi,
    fdwait $1c ( error mask ) ANDi, IFNZ,
      SPC ADDi, VIDMEM LD(i)A, BEGIN, BR JRi, THEN,
    $f2 INAi, A INCr, $f2 OUTiA, 18 CPi, C? BR ?JRi,
  EXAFAF', ( restore ) A INCr, 3 CPi, C? BR ?JRi, 0 RST,
```

2.18 Dan SBC drivers: 395-409

B395

```
\ Dan SBC drivers. See doc/hw/z80/dan.txt
\ Macros
: OUTii, ( val port -- ) A ROT LDri, OUTiA, ;
: repeat ( n -- ) ' SWAP 0 DO ( w ) DUP EXECUTE LOOP DROP ;
```

B396

```
\ SPI relay driver
CODE (spix) ( n -- n )
  A C LDrr,
  SPI_DATA OUTiA,
  ( wait until xchg is done )
  NOP, NOP, NOP, NOP,
  SPI_DATA INAi,
  C A LDrr, ;CODE
CODE (spie) ( n -- )
  $9A CTL8255 OUTii, $3 CTL8255 OUTii,
  A C LDrr, 1 XORi, 1 ANDi, CTL8255 OUTiA, BC POP, ;CODE
```

B397

```
\ software framebuffer subsystem
VID_MEM *VALUE VD_DECFR
VID_MEM $02 + *VALUE VD_DECTL
VID_MEM $04 + *VALUE VD_CURCL
VID_MEM $06 + *VALUE VD_FRMST
VID_MEM $08 + *VALUE VD_COLS
VID_MEM $0A + *VALUE VD_LINES
VID_MEM $0C + *VALUE VD_FRB
VID_MEM $0E + *VALUE VD_OFS
\ Clear Framebuffer
CODE (vidclr) ( -- ) BC PUSH,
  $9A CTL8255 OUTii, $3 CTL8255 OUTii, $1 CTL8255 OUTii,
  BC VID_MEM $10 + LDdi, HL VID_WDTH VID_SCN * LDdi,
  BEGIN, A XORr, LD(BC)A, BC INCd, HL DECd, HLZ, Z? ^? BR ?JRi,
  BC POP, ;CODE
```

B398

```
: VID_OFS
  [ VID_WDTH 8 * LITN ] * + VD_FRB + [*TO] VD_OFS (vidclr) ;
: VID$ ( -- )
  1 [*TO] VD_DECFR 0 [*TO] VD_DECTL 0 [*TO] VD_CURCL 0
  [*TO] VD_FRMST [ VID_WDTH 1 - LITN ] [*TO] VD_COLS
  [ VID_LN 1 - LITN ] [*TO] VD_LINES
  [ VID_MEM $10 + LITN ] [*TO] VD_FRB 1 4 VID_OFS ;
```

B399

```
: 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 0 DO
  2DUP C@ >> SWAP C! 1+ SWAP
  [ VID_WDTH LITN ] + SWAP LOOP
  DROP 0 SWAP C! ELSE 2DROP THEN ;
```

B400

```
: 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 [*TO] 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 * LDdi, DE VID_MEM $10 + LDdi,
 HL VID_MEM $10 + VID_WDTH 8 * + LDdi,
 LDIR,  HL VID_WDTH 8 * LDdi,
 BEGIN, A XORr, LD(DE)A, DE INCd, HL DECd, HLZ,
 Z? ^? BR ?JRi, EXX, BC POP, ;CODE
: NEWLN DUP 1+ VD_LINES = IF (vidscr) ELSE 1+ THEN ;
```

B401

```
\ Stream video frames, single scan
CODE (vidfr) ( -- ) BC PUSH, EXX,
  C SPI_DATA LDri, DE VID_MEM $04 + LDd(i),
  HL VID_MEM 40 + VID_WDTH - LDdi, DE ADDHLd,
  VID_MEM $06 + LD(i)HL, DE VID_WDTH 24 - LDdi,
  B VID_SCN LDri,
  LSET L1 BEGIN,
    6 CTL8255 OUTii, DE ADDHLd, 7 CTL8255 OUTii,
    A B LDrr, 4 repeat NOP, 24 repeat OUTI,
    B A LDrr, BR DJNZ,
  B 0 LDri, B 0 LDri, B 0 LDri, B VID_VBL 1 - LDri, FJR JRi,
  LSET L2 A VID_VBL 1 - LDri, FJR JRi, FMARK FMARK
    A B LDrr, B 28 LDri, BEGIN, BR DJNZ, HL INCd, B A LDrr,
    7 CTL8255 OUTii, 5 repeat NOP, 6 CTL8255 OUTii,
  L2 BR DJNZ,
```

B402

```
  VID_MEM $02 + LDA(i), B A LDrr, VID_MEM LDA(i),
  B SUBr, IFNZ,
    VID_MEM LD(i)A, B 23 LDri, HL INCd, B 23 LDri,
    BEGIN, BR DJNZ,
    VID_MEM $06 + LDHL(i), B VID_SCN LDri, 7 CTL8255 OUTii,
    5 repeat NOP, 6 CTL8255 OUTii, L1 JMPi,
  THEN, EXX, BC POP, ;CODE
```

B403

```
\ Stream video frames, double scan
CODE (vidfr) ( -- ) BC PUSH, EXX,
  C SPI_DATA LDri, DE VID_MEM $04 + LDd(i),
  HL VID_MEM 40 + VID_WDTH - LDdi, DE ADDHLd,
  VID_MEM $06 + LD(i)HL, DE VID_WDTH 24 - LDdi, B VID_SCN LDri,
  LSET L1 BEGIN,
    6 CTL8255 OUTii, DE ADDHLd, 7 CTL8255 OUTii, A B LDrr,
    DE DECd, DE -25 LDdi, 24 repeat OUTI,
    AF PUSH, DE INCd, 6 CTL8255 OUTii, DE ADDHLd,
    7 CTL8255 OUTii, AF POP, DE VID_WDTH 24 - LDdi,
    24 repeat OUTI, B A LDrr, BR DJNZ,
  B 0 LDri, B 0 LDri, B 0 LDri, B VID_VBL 1 - LDri, FJR JRi,
  LSET L2 A VID_VBL 1 - LDri, FJR JRi, FMARK FMARK
    A B LDrr, B 28 LDri, BEGIN, BR DJNZ, HL INCd, B A LDrr,
    7 CTL8255 OUTii, 5 repeat NOP, 6 CTL8255 OUTii,
  L2 BR DJNZ,
```

B404

```
  VID_MEM $02 + LDA(i), B A LDrr, VID_MEM LDA(i), B SUBr, IFNZ,
    VID_MEM LD(i)A, B 23 LDri, HL INCd, B 23 LDri,
    BEGIN, BR DJNZ, VID_MEM $06 + LDHL(i), B VID_SCN LDri,
    7 CTL8255 OUTii, 5 repeat NOP, 6 CTL8255 OUTii, L1 JMPi,
  THEN, EXX, BC POP, ;CODE
```

B405

```
\ PS2 keyboard driver subsystem
PSK_MEM *VALUE PSK_STAT
PSK_MEM $02 + *VALUE PSK_CC
PSK_MEM $04 + *VALUE PSK_BUFI
PSK_MEM $06 + *VALUE PSK_BUFO
PSK_MEM $08 + VALUE PSK_BUF
PC ORG $39 + T! ( RST 38 )
DI, AF PUSH, $10 SIOA_CTL OUTii, SIOA_CTL INAi,
4 A BIT, IFZ, AF POP, EI, RETI, THEN, ( I1 - T1 )
PSK_MEM LDA(i), A ORr,
IFZ, PTC8255 INAi, 7 A BIT,         ( I1 - )
IFZ, A 1 LDri, PSK_MEM LD(i)A, THEN,  ( I2 - T2 )
```

B406

```
AF POP, EI, RETI, THEN,             ( - T1 )
$9 CPi, FJR JRNZ, TO L3
HL PUSH, PSK_MEM $02 + LDHL(i), H 8 LDri, A XORr,
BEGIN, L RRC, 0 ADCi, H DECr, BR JRNZ,
H A LDrr, PTC8255 INAi, A H LDrr, 0 ADCi, $1 ANDi,
FJR JRZ, TO L1 A XORr, VID_MEM LD(i)A, VID_MEM $02 + LD(i)A,
PSK_MEM $04 + LDA(i), L A LDrr, PSK_MEM $06 + LDA(i),
A INCr, PS2_BMSK ANDi, L CPr, FJR JRZ, TO L1
PSK_MEM $06 + LD(i)A, L A LDrr,
A PSK_MEM $08 + <<8 >>8 LDri, L ADDr, L A LDrr,
A PSK_MEM $08 + >>8 LDri, 0 ADCi,
```

B407

```
H A LDrr, PSK_MEM $02 + LDA(i), (HL) A LDrr,
L1 FMARK A XORr, PSK_MEM LD(i)A, HL POP, AF POP, EI, RETI,
L3 FMARK PTC8255 INAi, RLCA, PSK_MEM $02 + LDA(i),
RRA, PSK_MEM $02 + LD(i)A,
PSK_MEM LDA(i), A INCr, PSK_MEM LD(i)A,
AF POP, EI, RETI,
```

B408

```
CODE (pskset)
  DI, $11 SIOA_CTL OUTii, $19 SIOA_CTL OUTii, IM1, EI, ;CODE
: PSK< ( -- n )
  PSK_BUFI PSK_BUFO = IF 0 ELSE PSK_BUFI
  1+ [ PS2_BMSK LITN ] AND DUP PSK_BUF + C@
  SWAP [*TO] PSK_BUFI THEN ;
: PSKV< ( -- n )
  PSK_BUFI PSK_BUFO = IF
  BEGIN 1 [*TO] VD_DECFR (vidfr)
  PSK_BUFI PSK_BUFO = NOT UNTIL THEN
  PSK_BUFI 1+ [ PS2_BMSK LITN ] AND DUP
  PSK_BUF + C@ SWAP [*TO] PSK_BUFI ;
: PSK$ ( -- )
  0 [*TO] PSK_BUFO 0 [*TO] PSK_BUFI
  0 [*TO] PSK_STAT (pskset) ;
```

B409

```
: (ps2kc) 0 BEGIN DROP PSKV<
  DUP 5 = IF 0 [*TO] VD_CURCL DROP 0 THEN
  DUP 6 = IF VD_CURCL 4 < IF 0 ELSE VD_CURCL 4 - THEN
    [*TO] VD_CURCL DROP 0 THEN
  DUP 4 = IF VD_CURCL [ VID_WDTH 28 - LITN ] > IF
    [ VID_WDTH 24 - LITN ] ELSE VD_CURCL 4 + THEN
    [*TO] 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
```

3.2 AVR macros: 301

B301

```
: AVRA ASML 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 ORG - >> ;
: _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,
```

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 VALUES X  $1c Y  $08 Z  0
         X+ $1d Y+ $19 Z+ $11
         -X $1e -Y $1a -Z $12
: _ldst ( Rd XYZ op ) SWAP DUP $10 AND <<8 SWAP $f AND
    OR OR ( Rd op' ) SWAP _Rdp L, ;
: LD, $8000 _ldst ; : ST, SWAP $8200 _ldst ;
```

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!
  << ORG + PC 1- HERE ( opw addr tgt hbkp )
  ROT [*TO] HERE ( opw tgt hbkp )
  SWAP ROT EXECUTE HERE ! ( hbkp ) [*TO] 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 VALUES R0 0 R1 1 R2 2 R3 3 R4 4 R5 5 R6 6 R7 7 R8 8 R9 9
  R10 10 R11 11 R12 12 R13 13 R14 14 R15 15 R16 16 R17 17
  R18 18 R19 19 R20 20 R21 21 R22 22 R23 23 R24 24 R25 25
  R26 26 R27 27 R28 28 R29 29 R30 30 R31 31
  XL 26 XH 27 YL 28 YH 29 ZL 30 ZH 31
```

3.4 ATmega328P definitions: 315

B315

```
( ATmega328P definitions ) 87 VALUES
UDR0 $c6 UBRR0L $c4 UBRR0H $c5 UCSR0C $c2 UCSR0B $c1 UCSR0A $c0
TWAMR $bd TWCR $bc TWDR $bb TWAR $ba TWSR $b9 TWBR $b8 ASSR $b6
OCR2B $b4 OCR2A $b3 TCNT2 $b2 TCCR2B $b1 TCCR2A $b0 OCR1BL $8a
OCR1BH $8b OCR1AL $88 OCR1AH $89 ICR1L $86 ICR1H $87 TCNT1L $84
TCNT1H $85 TCCR1C $82 TCCR1B $81 TCCR1A $80 DIDR1 $7f DIDR0 $7e
ADMUX $7c ADCSRB $7b ADCSRA $7a ADCH $79 ADCL $78 TIMSK2 $70
TIMSK1 $6f TIMSK0 $6e PCMSK1 $6c PCMSK2 $6d PCMSK0 $6b EICRA $69
PCICR $68 OSCCAL $66 PRR $64 CLKPR $61 WDTCSR $60 SREG $3f
SPL $3d SPH $3e SPMCSR $37 MCUCR $35 MCUSR $34 SMCR $33 ACSR $30
SPDR $2e SPSR $2d SPCR $2c GPIOR2 $2b GPIOR1 $2a OCR0B $28
OCR0A $27 TCNT0 $26 TCCR0B $25 TCCR0A $24 GTCCR $23 EEARH $22
EEARL $21 EEDR $20 EECR $1f GPIOR0 $1e EIMSK $1d EIFR $1c
PCIFR $1b TIFR2 $17 TIFR1 $16 TIFR0 $15 PORTD $0b DDRD $0a
PIND $09 PORTC $08 DDRC $07 PINC $06 PORTB $05 DDRB $04 PINB $03
```

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 VALUES SRAM_START $0060 RAMEND $015f SPL $3d SPH $3e
          GPIOR0 $11 MCUCR $35 TCCR0B $33 GIMSK $3b
          TIFR $38 TCNT0 $32 PINB $16 DDRB $17 PORTB $18
          CLK 2 DATA 1 CP 3 LQ 0 LR 4
$100 100 - VALUE TIMER_INITVAL
\ We need a lot of labels in this program...
5 VALUES L4 0 L5 0 L6 0 L7 0 L8 0
```

B325

```
HERE TO ORG
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,
```

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: 301-305

B301

```
\ 8086 macros
: 8086A ASML 311 317 LOADR 306 LOAD ( HAL flow ) ASMH ;
: 8086C 302 305 LOADR ;
: 8086H 306 310 LOADR ;
1 VALUE JROPLEN -1 VALUE JROFF
```

B302

```
\ 8086 boot code. PS=SP, RS=BP, IP=DX
HERE TO ORG
FJR JRi, TO L1 ( main ) 0 C, 0 C, ( 03, boot driveno )
8 ALLOT0
\ End of Stable ABI
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 lbldoes BP INCx, BP INCx, [BP] 0 DX []+x MOV[], ( pushRS )
  DX AX MOVxx, \ continue to lblcell
LSET lblcell AX POPx, \ continue to lblpush
LSET lblpush PUSHp, \ continue to lblnext
LSET lblnext DI DX MOVxx, ( <-- IP ) DX INCx, DX INCx,
  DI [DI] x[] MOV[], DI JMPr,
LSET lblxt BP INCx, BP INCx, [BP] 0 DX []+x MOV[], ( pushRS )
  DX POPx, lblnext BR JRi,
```

B303

```
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 []= ( a1 a2 u -- f ) CX BX MOVxx, SI POPx, DI POPx,
  CLD, REPZ, CMPSB, BX 0 MOVxI, IFZ, BX INCx, THEN, ;CODE
CODE QUIT LSET L1 ( used in ABORT )
  BP RS_ADDR MOVxI, DI $0a ( main ) MOVxm, DI JMPr,
CODE ABORT SP PS_ADDR MOVxI, L1 BR JRi,
CODE BYE HLT, BEGIN, BR JRi,
CODE RCNT RS_ADDR i>w, PUSHp, AX BP MOVxx, -wp, w>p, ;CODE
CODE SCNT AX SP MOVxx, PUSHp, PS_ADDR i>w, -wp, w>p, ;CODE
```

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 3 SUBxi, DI [DI] x[] MOV[], ( prev ) DI DI ORxx,
  Z? ^? BR ?JRi, ( loop ) BX BX XORxx, ;CODE
```

B305

```
( See comment in B294 TODO: test on real hardware. in qemu,
  the resulting delay is more than 10x too long. )
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
```

4.3 8086 HAL: 306-310

B306

```
\ HAL flow words, also used in 8086A
SYSVARS $16 + *VALUE ?JROP
: JMPi, $e9 C, ( jmp near ) PC - 2 - L, ;
: CALLi, $e8 C, ( jmp near ) PC - 2 - L, ;
: JRi, $eb C, ( jmp short ) C, ;
: ?JRi, ?JROP C, C, ;
: Z? $74 [*TO] ?JROP ; : C? $72 [*TO] ?JROP ;
: ^? ?JROP 1 XOR [*TO] ?JROP ;
```

B307

```
: w>p, $89c3 M, ; \ mov bx,ax
: p>w, $89d8 M, ; \ mov ax,bx
: DROPp, $5b C, ( pop bx ) ; : POPp, p>w, DROPp, ;
: DUPp, $53 C, ( push bx ) ; : PUSHp, DUPp, w>p, ;
: POPf, $58 C, ( pop ax ) ; : PUSHf, $50 C, ( push ax ) ;
: POPr, $8b46 M, $00 C, ( mov ax,[bp+0] )
  $4d4d M, ( dec bp;dec bp ) ;
: PUSHr, $4545 M, ( inc bp;inc bp )
  $8946 M, $00 C, ( mov [bp+0],ax ) ;
: SWAPwp, $93 C, ( xchg ax,bx ) ;
: SWAPwf, $89c1 M, ( mov cx,ax ) POPf, $51 C, ( push cx ) ;
```

B308

```
: JMPw, $ffe0 M, ; \ jmp ax
: INCw, $40 C, ( inc ax ) ; : DECw, $48 C, ( dec ax ) ;
: INCp, $43 C, ( inc bx ) ; : DECp, $4b C, ( dec bx ) ;
: i>w, $b8 C, L, ; \ mov ax,nn
: (i)>w, $a1 C, L, ; \ mov ax,(nn)
: C@w, $89c7 M, ( mov di,ax ) $30e4 M, ( xor ah,ah )
  $8a05 M, ( mov al,[di] ) ;
: @w, $89c7 M, ( mov di,ax ) $8b05 M, ( mov ax,[di] ) ;
: C!wp, $89c7 M, ( mov di,ax ) $881d M, ( mov [di],bl ) ;
: !wp, $89c7 M, ( mov di,ax ) $891d M, ( mov [di],bx ) ;
```

B309

```
: w>Z, $09c0 M, ( or ax,ax ) ;
: p>Z, $09db M, ( or bx,bx ) ;
: C>w, $b8 C, 0 L, ( mov ax,0 ) $1400 M, ( adc al,0 ) ;
: Z>w, $b8 C, 0 L, $7501 M, ( jrnz+1 ) INCw, ;
: IP>w, $89d0 M, ; \ mov ax,dx
: w>IP, $89c2 M, ; \ mov dx,ax
: IP+, $42 C, ; \ inc dx
: IP+off, $50 C, ( push ax ) $89d7 M, ( mov di,dx )
  $30e4 M, ( xor ah,ah ) $8a05 M, ( mov al,[di] )
  $98 C, ( cbw ) $01c2 M, ( add dx,ax ) $58 C, ( pop ax ) ;
```

B310

```
: +wp, $01d8 M, ( add ax,bx ) ;
: -wp, $29d8 M, ( sub ax,bx ) ;
: >>w, $d1e8 M, ; \ shr ax
: <<w, $d1e0 M, ; \ shl ax
: >>8w, $88e0 M, $30e4 M, ; \ mov al,ah;xor ah,ah
: <<8w, $88c4 M, $30c0 M, ; \ mov ah,al;xor al,al
: CMPwp, $39d8 M, ( cmp ax,bx ) ;
: ANDwp, $21d8 M, ( and ax,bx ) ;
: ORwp, $09d8 M, ( or ax,bx ) ;
: XORwp, $31d8 M, ( xor ax,bx ) ;
: XORwi, $35 C, L, ( xor ax,nn ) ;
```

4.4 8086 assembler: 311-317

B311

```
\ 8086 assembler. See doc/asm.txt.
28 VALUES AL 0 CL 1 DL 2 BL 3
          AH 4 CH 5 DH 6 BH 7
          AX 0 CX 1 DX 2 BX 3
          SP 4 BP 5 SI 6 DI 7
          ES 0 CS 1 SS 2 DS 3
          [BX+SI] 0 [BX+DI] 1 [BP+SI] 2 [BP+DI] 3
          [SI] 4 [DI] 5 [BP] 6 [BX] 7
```

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,     $20 OPrr ANDrr,    $21 OPrr ANDxx,
```

B315

```
: OPmodrm ( opbase modrmbase ) DOER C, C, DOES>
    @ L|M ( disp? modrm opoff modrmbase opbase ) ROT + C,
    ( disp? modrm modrmbase ) + DUP C, ( disp? modrm )
    $c0 AND DUP IF ( has disp ) $80 AND IF
        ( disp low+high ) L, ELSE ( low only ) C, THEN
    ELSE ( no disp ) DROP THEN ;
( -- disp? modrm opoff )
: [b] ( r/m ) 0 ; : [w] ( r/m ) 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

```
$fe 0 OPmodrm INC[],          $fe $8 OPmodrm DEC[],
$fe $30 OPmodrm PUSH[],       $8e 0 OPmodrm POP[],
$88 0 OPmodrm MOV[],          $38 0 OPmodrm CMP[],

: OPi DOER C, DOES> C@ C, C, ;
$04 OPi ADDALi,     $24 OPi ANDALi,    $2c OPi SUBALi,
$cd OPi INT,        $eb OPi JRi,       $74 OPi JRZi,
$75 OPi JRNZi,      $72 OPi JRCi,      $73 OPi JRNCi,
: OPI DOER C, DOES> C@ C, L, ;
$05 OPI ADDAXI,     $25 OPI ANDAXI,    $2d OPI SUBAXI,
$e9 OPI JMPi,       $e8 OPI CALLi,
```

B317

```
: CMPri, $80 C, SWAP $f8 OR C, C, ;
: CMPxI, $81 C, SWAP $f8 OR C, L, ;
: CMPxi, $83 C, SWAP $f8 OR C, C, ;
: 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, ;
: SUBxi, $83 C, SWAP $e8 OR C, C, ;
: ADDxi, $83 C, SWAP $c0 OR C, C, ;
: JMPr, $ff C, 7 AND $e0 OR C, ;
: JMPf, ( seg off ) $ea C, L, L, ;
```

4.5 8086 drivers: 320-324

B320

```
( PC/AT drivers. Load range: 320-326 )
CODE (key?)
  AH AH XORrr, $16 INT, AH AH XORrr, PUSHp, DUPp, ;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 VALUE FDSPT
DRV_ADDR 1+ VALUE 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 VALUES COLS 80 LINES 25
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 _spit ( c )
  POPp, AH $0e MOVri, ( print char ) $10 INT, ;CODE
: CELL! ( c pos -- ) 0 CURSOR! _spit ;
```

5 6809

5.1 Architecture index: 300

B300

```
6809 MASTER INDEX

301 6809 macros                302 6809 boot code
306 6809 HAL                   311 6809 assembler
320 TRS-80 Color Computer 2
```

5.2 6809 macros: 301

B301

```
( 6809 declarations )
: 6809A ASML 311 318 LOADR 306 LOAD ( HAL flow ) ASMH ;
: 6809C 302 305 LOADR ;
: 6809H 306 310 LOADR ;
: COCO2 320 322 LOADR ;
1 VALUE JROPLEN -1 VALUE JROFF
```

5.3 6809 boot code: 302-305

B302

```
( 6809 Boot code. IP=Y, PS=S, RS=U  ) HERE TO ORG
FJR JRi, TO L1 ( main ) $0a ALLOT0
\ end of stable ABI
L1 FMARK ( main ) PS_ADDR # LDS, RS_ADDR # LDU,
BIN( 4 + ( BOOT ) () LDX, X+0 JMP,
LSET lblpush PUSHp,
LSET lblcell LSET lblnext Y++ LDX, X+0 JMP,
LSET lbldoes U++ STY, ( IP->RS ) D Y TFR, lblnext BR JRi,
LSET lblxt U++ STY, ( IP->RS ) PULS, Y lblnext BR JRi,
```

B303

```
CODE QUIT LSET L1 ( for ABORT ) RS_ADDR # LDU,
  BIN( $0a + ( main ) () LDX, X+0 JMP,
CODE ABORT PS_ADDR # LDS, L1 BR JRi,
CODE BYE BEGIN, BR JRi,
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
```

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 BHSi, ( if < ) S+0 ADDD, 3 S+N DEC, ( a lsb ) THEN,
  0 <> DEC, Z? ^? BR ?JRi,
  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, Z? ^? BR ?JRi, 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, Z? ^? BR ?JRi, \ not zero, loop
  ( end of dict ) 0 <> LDY, S+0 STX, ( X=0 ) ;CODE
```

5.4 6809 HAL: 306-310

B306

```
\ 6809 HAL, flow words. Also used in 6809A
SYSVARS $16 + *VALUE ?JROP
: JMPi, $7e C, M, ( jmp nn ) ; : CALLi, $bd C, M, ( jsr nn ) ;
: JRi, $20 C, C, ( bra n ) ; : ?JRi, ?JROP C, C, ;
: Z? $27 [*TO] ?JROP ( beq ) ; : C? $25 [*TO] ?JROP ( bcs ) ;
: ^? ?JROP 1 XOR [*TO] ?JROP ( bne/bcc ) ;
```

B307

```
\ 6809 HAL, Stack
: w>p, $ed60 M, ( std 0,S ) ; : p>w, $ec60 M, ( ldd 0,S ) ;
: DUPp, $ae60 M, ( ldx 0,S ) $3410 M, ( pshs x ) ;
: DROPp, $3262 M, ( leas 2,S ) ;
: PUSHp, $3406 M, ( pshs d ) ; : POPp, $3506 M, ( puls d ) ;
: POPf, $3510 M, ( puls x ) POPp, $3410 M, ( pshs x ) ;
: PUSHf, $3510 M, ( puls x ) PUSHp, $3410 M, ( pshs x ) ;
: PUSHr, $edc1 M, ( std u++ ) ; : POPr, $ecc3 M, ( ldd --u ) ;
: SWAPwp, $ae60 M, ( ldx 0,S ) w>p, $1f10 M, ( tfr x,d ) ;
: SWAPwf, $ae62 M, ( ldx 2,S ) $ed62 M, ( std 2,S )
  $1f10 M, ( tfr x,d ) ;
```

B308

```
\ 6809 HAL, Jump, flags
: JMPw, $1f01 M, ( tfr d,x ) $6e00 M, ( jmp 0,x ) ;
\ TODO: add some compile time flag to determine when this code
\ generation is spurious and when its not so that we can avoid
\ emitting it when unnecessary.
: w>Z, $1083 M, 0 M, ( cmpd 0 ) ;
: p>Z, $ae60 M, ( ldx 0,S ) ;
: i>w, $cc C, M, ( ldd nn ) ;
: (i)>w, $fc C, M, ( ldd (nn) ) ;
: C>w, 0 i>w, $c900 M, ( adcb 0 ) ;
: Z>w, Z? 5 ?JRi, 0 i>w, 3 JRi, 1 i>w, ;
```

B309

```
\ 6809 HAL, transfer
: C@w, $1f01 M, ( tfr d,x ) $e600 M, ( ldb 0,X )
  $4f C, ( clra ) ;
: @w, $1f01 M, ( tfr d,x ) $ec00 M, ( ldd 0,X ) ;
: C!wp, $1f01 M, ( tfr d,x ) $e661 M, ( ldb 1,S )
  $e700 M, ( stb 0,X ) $1f10 M, ( tfr x,d ) ;
: !wp, $1f01 M, ( tfr d,x ) $ec60 M, ( ldd 0,S )
  $ed00 M, ( std 0,X ) ;
: IP>w, $1f20 M, ( tfr y,d ) ;
: w>IP, $1f02 M, ( tfr d,y ) ;
: IP+, $6da0 M, ( tst ,y+ ) ;
: IP+off, $a620 M, ( lda 0,y ) $31a6 M, ( leay a,y ) ;
```

B310

```
\ 6809 HAL, arithmetic
: +wp, $e360 M, ( addd 0,s ) ; : -wp, $a360 M, ( subd 0,s ) ;
: >>w, $4456 M, ( lsra rorb ) ; : <<w, $5849 M, ( lslb rola ) ;
: >>8w, $1f89 M, ( tfr a,b ) $4f C, ( clra ) ;
: <<8w, $1f98 M, ( tfr b,a ) $5f C, ( clrb ) ;
: INCw, $c3 C, 1 M, ( addd 1 ) ;
: DECw, $c3 C, -1 M, ( addd -1 ) ;
: INCp, $6c61 M, ( inc 1,s ) Z? ^? 2 ?JRi, $6c60 M, ( inc0,s ) ;
: DECp, $6d61 M, ( tst 1,s ) Z? ^? 2 ?JRi, $6a60 M, ( dec 0,s )
  $6a61 M, ( dec 1,s ) ;
: CMPwp, $10a3 M, $60 C, ( cmpd 0,s ) ;
: ANDwp, $a460 M, ( anda 0,s ) $e461 M, ( andb 1,s ) ;
: ORwp, $aa60 M, ( ora 0,s ) $ea61 M, ( orb 1,s ) ;
: XORwp, $a860 M, ( eora 0,s ) $e861 M, ( eorb 1,s ) ;
: XORwi, $88 C, DUP >>8 C, ( eora nn ) $c8 C, C, ( eorb nn ) ;
```

5.5 6809 assembler: 311-318

B311

```
\ 6809 assembler. See doc/asm.txt.
1 TO BIGEND?
\ For TFR/EXG
10 VALUES D 0 X 1 Y 2 U 3 S 4 PCR 5 A 8 B 9 CCR 10 DPR 11
\ 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 ;
```

B312

```
: 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] ;
```

B313

```
$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]
```

B314

```
: ,? DUP $ff > IF M, ELSE C, THEN ;
: ,N ( cnt ) 0 DO C, LOOP ;
: 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, ;
```

B315

```
$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,
```

B316

```
$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,
```

B317

```
$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 BCCi,        $1024 OPLBR LBCCi,  $25 OPBR BCSi,
$1025 OPLBR LBCSi,    $27 OPBR BEQi,      $1027 OPLBR LBEQi,
$2c OPBR BGEi,        $102c OPLBR LBGEi,  $2e OPBR BGTi,
$102e OPLBR LBGTi,    $22 OPBR BHIi,      $1022 OPLBR LBHIi,
$24 OPBR BHSi,        $1024 OPLBR LBHSi,  $2f OPBR BLEi,
$102f OPLBR LBLEi,    $25 OPBR BLOi,      $1025 OPLBR LBLOi,
$23 OPBR BLSi,        $1023 OPLBR LBLSi,  $2d OPBR BLTi,
$102d OPLBR LBLTi,    $2b OPBR BMIi,      $102b OPLBR LBMIi,
```

B318

```
$26 OPBR BNEi,        $1026 OPLBR LBNEi,  $2a OPBR BPLi,
$102a OPLBR LBPLi,    $20 OPBR BRAi,      $16 OPLBR LBRAi,
$21 OPBR BRNi,        $1021 OPLBR BRNi,   $8d OPBR BSRi,
$17 OPLBR LBSRi,      $28 OPBR BVCi,      $1028 OPLBR LBVCi,
$29 OPBR BVSi,        $1029 OPLBR LBVSi,

: _ ( 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 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,
```

5.6 TRS-80 Color Computer 2: 320-322

B320

```
\ CoCo2 drivers
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,
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, C? BR ?JRi,
  ( no key ) CLRB, RTS,
```

B321

```
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, C? BR ?JRi,
    ( A = our char ) 1 S+N STA, TSTA, IFNZ, ( valid key )
      1 # LDD, ( f ) PSHS, D ( wait for keyup )
      BEGIN, L1 () JSR, Z? ^? BR ?JRi, THEN,
  THEN, ;CODE
```

B322

```
32 VALUE COLS 16 VALUE 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! ;
```

6 6502

6.1 Architecture index: 300

B300

```
6502 MASTER INDEX

301 6502 macros and consts     302 6502 assembler
310 6502 boot code
```

6.2 6502 macros and consts: 301

B301

```
\ 6502 macros and constants. See doc/code/6502.txt
: 6502A ASML 302 305 LOADR ;
: 6502C 310 311 LOADR ;
1 VALUE JROPLEN -1 VALUE JROFF
```

6.3 6502 assembler: 302-305

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 asm, HAL underpinnings
0 VALUE ?JROP
' JMP, ALIAS JMPi,
: JRi, PC + [ JROFF JROPLEN - LITN ] - JMP, ; \ no BRA!
: ?JRi, ?JROP C, C, ;
: Z? $f0 [TO] ?JROP ; : C? $b0 [TO] ?JROP ;
: ^? ?JROP $20 XOR [TO] ?JROP ;
' JSR, ALIAS CALLi,
```

6.4 6502 boot code: 310-311

B310

```
\ 6502 boot code PS=SP RS=Y IP=1<> 0<>=6C (JMP[])
HERE TO ORG 0 JMP, 9 ALLOT0 \ STABLE ABI
PC ORG $01 ( main jmp ) + T!
$ff # LDX, TXS, ( PS ) 0 # LDY, ( RS ) $6c # LDA, 0 <> STA,
BIN( $04 ( BOOT ) + JMP[],
LSET lblxt INY, INY, 1 <> LDA, $100 (Y+) STA, 2 <> LDA,
  $101 (Y+) STA, PLA, 1 <> STA, PLA, 2 <> STA,
  1 <> INC, IFZ, 2 <> INC, THEN, $00 JMP,
LSET lblnext 1 <> LDA, CLC, 2 # ADC, 1 <> STA,
  IFC, 2 <> INC, THEN, $00 JMP,
```

B311

```
CODE EXIT DEY, DEY, $100 (Y+) LDA, 1 <> STA, $101 (Y+) LDA,
  2 <> STA, ;CODE
LSET L2 11 C, ," Collapse OS"
CODE foo L2 () LDA, 3 <> STA, 1 # LDX, BEGIN,
    L2 (X+) LDA, $80 # ORA, $fded JSR, INX, 3 <> DEC,
  BR Z? ^? ?JRi, BEGIN, BR JRi,
: BOOT foo ;
XCURRENT ORG $04 ( stable ABI BOOT ) + T!
```