The SD-8516 Program ABI

Version 1. What a program is, how it starts, how it ends, and how it finds its arguments.

Why this exists

Until INT $20 the machine had no word for **program**. BASIC, hexmon, ed, forth and tinyc were each a REPL that owned the whole machine, and none of them could call any of the others. Everything ran by hand: assemble some `.sda` files into RAM and type `SYS $2C000`.

That absence is why the shell, the compiler and the game each looked like they were blocked on the other two. The compiler could not produce a *program*, only bytes at a fixed address; a shell could not *run* one, because there was no such thing to run; and writing the shell in C needed a compiler whose output had nowhere to go. Three things apparently waiting on each other, all actually waiting on the same missing noun.

This is not the bootstrap chicken-and-egg – that one was already solved, the usual way, by writing the compiler twice. This was a missing abstraction, and the historical fix is well known and small: it is the step from **monitor** to **operating system**. CP/M's contribution was not its shell but its convention (a flat binary at `$0100`, entered by a jump, terminated by jumping to `$0000`, command tail at `$0080`); the CCP was about 2KB and nearly trivial once that existed. Unix is the same shape – `exec()`, `a.out`, `argv`, an exit status – and the shell arrived right afterwards.

What a program is

A raw binary file. **No header, no relocation, no symbol table.** The file is the image.

This is deliberate. It is the least that works, and none of it has to change when it later becomes cleverer – a header can be added without changing what a program *means*.

address	name	what it is
$020000	PROC_BASE	process area, 256 bytes
$020080	PROC_CMDTAIL	command tail, NUL-terminated, 128 bytes
$020100	USER_ORIGIN	load address, and the entry point

The 256 bytes below the load address belong to the *process* rather than to the program, and that is where its arguments are left. CP/M put the command tail at `$0080` below a TPA at `$0100` for exactly this reason, and the shape is worth keeping.

Starting and finishing

The image is loaded at USER_ORIGIN and entered with CALL, so **the first byte of the file is the first instruction.** No entry-point field is needed, because there is only one place it could be.

A program finishes in one of two ways:

```
RET                 ; exit code in AL
INT $20  AH=$01     ; exit code in AL, from any call depth
```

The second exists because the first requires a balanced stack, and an error path deep inside a program rarely has one. EXEC records SP before entering the program and EXIT restores it — a longjmp, in the same spirit as CP/M's `JMP $0000`.

INT $20 — Process services

Like INT $15, this is a JZ-style calltable: reach it with `INT $20`, not by CALLing the labels in `kernal/int0x20h.sda`.

AH=$00 — EXEC

In	ELM = pointer to NUL-terminated filename
	FLD = pointer to NUL-terminated command tail, or 0 for none
Out	AL = the program's exit code
	CF = 0 if the program ran, 1 if it could not be loaded

One program at a time. EXEC does not return until the program does. The command tail is copied into `PROC_CMDTAIL` before the program starts, so the caller's buffer does not have to survive the call.

A caller of EXEC must not live in the program area it is about to overwrite. This is why CP/M's CCP sits at the top of memory rather than in the TPA, and it is the one thing to get right when the shell is written.

AH=$01 — EXIT

In	AL = exit code
Out	does not return; control resumes inside EXEC

Restoring SP discards everything the program pushed, including the interrupt frame — which is the point. Calling this with no program running returns to a stack that no longer means anything, so don't.

Kernal state

EXECs bookkeeping lives in KERNAL_WORKSPACE (`$00E800`, declared long ago and never used) rather than in the process area, so that a program which runs off the rails cannot take the kernal's way home with it.

```
PROC_SAVED_SP   $00E800   3 bytes - SP at entry, so EXIT can unwind
PROC_EXIT_CODE  $00E803   1 byte
PROC_HANDLE     $00E804   1 byte  - file handle while loading
```

```
PROC_LOADPTR    $00E805   3 bytes - where the next chunk lands
PROC_NAMEPTR    $00E808   3 bytes
PROC_TAILPTR    $00E80B   3 bytes
```

A note on widths

The load pointer is advanced by GLC — the 24-bit register whose low half *is* C — after zeroing GL:

```
LDGL #0             ; GLC = C, zero-extended to 24 bits
ADD ELM, GLC
```

`ADD ELM, C` would mix a 24-bit destination with a 16-bit source. The assembler checks widths for MOV but **not** for arithmetic, and the CPU takes its width from the destination and indexes the register file with the raw encoding. That is the same trap that put Rogueima's monsters off the map for months. Until the assembler grows a width check on the arithmetic path, this is on the programmer.

What this unlocks

- A **shell**: read a line, split it, EXEC the first word. The CCP is the

model, and it is a few hundred lines.

- `cc c.c` producing a file called `c`, instead of `.load` and `.run`

inside tinyc's own REPL.

- The tinyc fixpoint test becomes expressible as a command: compile the

compiler twice and compare the two outputs byte for byte.

- Rogueima becomes `rogueima` typed at a prompt.

Conformance tests

`programs/abi/` holds four programs that exercise all of this end to end against the real filesystem. See its README for the exact commands.

- `hello` — marker written, command tail arrives as `one two three`,

returns exit code 7, carry clear

- `hello2` — EXIT from three frames deep with three unbalanced pushes

still returns exit code 9 with carry clear

- a missing file returns carry set rather than crashing