Tidepod 1.0.0-rc5 Generated 2026-10-21

Tidepod Documentation & Developer Guide

Guides, tool references and the complete library reference for building TI-99/4A software with Tidepod.

Contents

Getting Started

1. What is Tidepod?
2. Installation
3. Quick start: build and run a project
4. Project types and the .project file
5. The Tidepod IDE

Tutorials

1. Tutorial 1: Minimal 8K C Cartridge
2. Tutorial 2: Sprites, Joystick & 3-Voice Sound
3. Tutorial 3: The Binary Asset Pipeline (`.dat`)
4. Tutorial 4: RAM Placement & Stack Safety
5. Tutorial 5: F18A FPGA Graphics & GPU in C
6. Tutorial 6: Disk I/O & Standalone EA5 Programs

Build System

1. The build pipeline
2. Banking and the packer
3. Code placement: banked, persistent, scratchpad
4. Binary resources (.dat files)
5. F18A GPU code in C
6. Extending the build with Makefile.custom
7. Reading bank_report.json

Tools & Workflows

1. Tidepod command line
2. The stand-alone emulator (ti994a-emu)
3. Disks and peripherals
4. Debugging
5. Frame profiler
6. Headless testing and automation
7. Troubleshooting

API Reference

1. API Reference
2. cgetc, kbhit, getchar
3. conio
4. cputc, cputs, cprintf, clrscr
5. gotoxy, gotox, gotoy, wherex, wherey
6. fclose
7. fcopy_path
8. fgetc
9. fgets
10. fopen, fclose, fread, fwrite, fflush, fcopy_path
11. fprintf
12. fputc
13. fputs
14. fread, fwrite
15. fseek, ftell, rewind
16. ftell
17. fwrite
18. printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf
19. putchar
20. puts
21. snprintf
22. sprintf
23. sscanf, scanf
24. stdio_enable_chip, stdio_enable_fat, stdio_enable_wds, stdio_enable_prog, stdio_enable_all_disk
25. stdio_enable_all_disk
26. stdio_enable_chip

27. stdio_enable_fat
28. stdio_enable_prog
29. stdio_enable_wds
30. stdio_last_error
31. stdio_set_wds_vdp_buf, stdio_get_wds_vdp_buf
32. ti_text_mode, ti_text_load_builtin_font
33. ti_vdp_detect, ti_vdp_text80_ok
34. remove, rename
35. ti_dsr_enum
36. ti_opendir, ti_readdir, ti_closedir, ti_dir_volume
37. memchr
38. memcpy
39. memmove
40. memset
41. strcat, strncat
42. strcmp, strncmp
43. stpcpy, stpncpy, strcpy, strncpy
44. strlen, strnlen
45. strcat, strncat
46. strcmp, strncmp
47. stpcpy, stpncpy, strcpy, strncpy
48. strnlen
49. strstr
50. strtok
51. isalnum
52. isalpha
53. iscntrl
54. isdigit
55. islower
56. isprint
57. ispunct
58. isspace
59. isupper
60. isxdigit
61. tolower
62. toupper
63. abort
64. abs
65. atoi, atol
66. atol
67. calloc
68. exit, _Exit
69. free
70. malloc
71. qsort
72. rand, rand_r, srand, sranddev
73. realloc
74. rand, rand_r, srand, sranddev
75. strtol, strtoul, atoi, atol
76. strtoul
77. atanf
78. floorf, ceilf
79. sinf, cosf, tanf
80. expf
81. fabsf
82. floorf, ceilf
83. fmodf
84. logf
85. powf
86. sinf, cosf, tanf
87. sqrtf
88. sinf, cosf, tanf
89. libc
90. ASC
91. CHAR
92. CHARSET
93. CHR$
94. CLEAR
95. COINC
96. COLOR
97. DELSPRITE
98. DISPLAY AT
99. DISTANCE
100. GCHAR
101. HCHAR
102. JOYST
103. KEY
104. LEN
105. LOAD
106. LOCATE
107. MAGNIFY
108. MOTION
109. PATTERN
110. PEEK
111. PEEKV
112. POKE
113. POKEV
114. POSITION
115. RND
116. SCREEN
117. SEG$
118. SOUND
119. SPRITE
120. STR$
121. VAL
122. VCHAR

123. VSYNC
124. xb-control
125. xb-data
126. xb-input
127. xb-io
128. xb-math
129. xb-misc
130. xb-screen
131. xb-sound
132. xb-sprites
133. xb-string
134. xb-sub
135. xb-tiles
136. xb-variables
137. xb-vdp

AI Agent Skill

1. The ti99-tidepod-gamedev agent skill
2. Rules the skill enforces

What is Tidepod?

Overview

Tidepod is a development suite for the TI-99/4A. You write C (or compiled Extended BASIC, or assembly), and Tidepod turns it into a cartridge image or disk program, runs it in an emulator and lets you debug it at source level. The same `tidepod` binary is both the graphical IDE and a command-line project manager.

This page describes the pieces and how they fit together. You do not need to build any of them: every release ships with the complete toolchain.

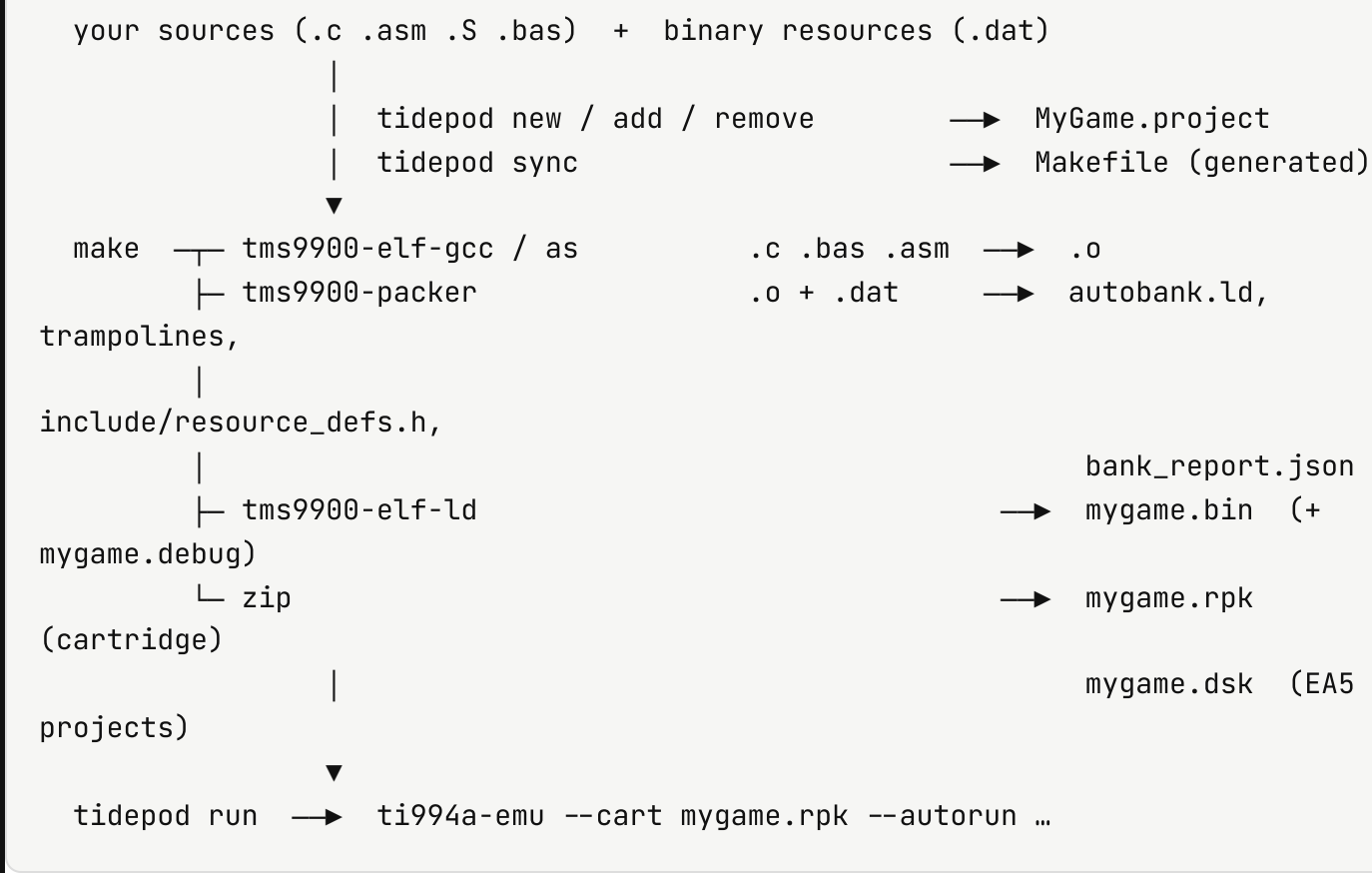
Components

Component	What it is	You interact with it via
`tidepod`	IDE (editor, build, embedded emulator, debugger, graphics viewers, profiler) and CLI project manager.	GUI, or `tidepod new/add/remove/list/set-type/sync/run`
`.project`	JSON project description: type, source list, per-file placement, build and emulator settings.	CLI and IDE dialogs (reference)
`Makefile`	Generated from `.project` by `tidepod sync`. Never edited by hand.	`make`, `make debug`, `make clean`
`tms9900-elf-gcc`	GCC 14.2 cross compiler for the TMS9900, plus binutils (`as`, `ld`, `objdump`, `objcopy`, ...) and `tms9900-elf-gdb`. Also compiles Extended BASIC (`-x xb`).	Invoked by the Makefile
`tms9900-packer`	Auto-banker: distributes code and data over 8 KB cartridge banks, writes the linker script, cross-bank trampolines and resource headers, and produces EA5 disk images.	Invoked by the Makefile (details)
`gpu-image-map`	Generates the map header for F18A GPU images.	Invoked by the Makefile (details)
`libc.a`	C library: strings, ctype, stdlib (no heap), console I/O and conio, keyboard, soft-float math, disk I/O through `fopen` and friends, VDP detection, F18A GPU host helpers, ROM resource copy.	`#include` (API reference)
`libxb.a`	Runtime for compiled Extended BASIC programs (`CALL SPRITE`, `CALL SOUND`, ...).	XB projects
`crt0_*.o`	Start-up code per project type (8K, banked, EA5, XB, GPU).	Selected by project type

Component	What it is	You interact with it via
`ti994a-emu`	Stand-alone emulator with GDB server, headless mode, key injection and screenshots. The IDE embeds the same core.	`tidepod run` or directly (reference)
`man pages`	Reference pages for libc, the XB runtime and the XB language.	IDE help, `man`, and the API section of this site
`Agent skill`	`ti99-tidepod-gamedev`: instructions for AI coding agents, copied into every new project.	Your AI agent (details)
`Examples`	Complete projects: `pong`, `pong_xb`, `f18a_bump`, `ghostbusters`, `ccleste_ti`, `fileman`, …	`resources/examples/` in the install

From source to cartridge

Build flow



The details of each step, and every generated file, are covered in Build pipeline.

Project types

`--type`	`"type" in .project`	Language	Output	Needs 32K expansion
`8k`	0	C	Single 8 KB cartridge, no banking. `.data` / `.bss` live in scratchpad.	No
`banked` (default)	1	C	Bank-switched cartridge (`paged378` PCB).	Yes
`xb`	2	Extended BASIC, compiled	Bank-switched cartridge.	Yes
`ea5`	3	C	Editor/Assembler option 5 program files on a `.dsk` image.	Yes
`custom`	4	any	You provide the build; no Makefile generation.	—
—	5	C	8 KB cartridge that uses the 32K expansion (IDE only).	Yes

Memory map as Tidepod uses it

Range	Region	Used by Tidepod for
`0x2000-0x3FFF`	Low 8K expansion RAM	`.persistent`: runtime stubs, cross-bank trampolines, code you mark persistent.
`0x4000-0x5FFF`	Peripheral DSR space	Disk controller ROMs (not yours).
`0x6000-0x7FFF`	Cartridge window	One 8 KB bank of your code/data/resources at a time.
`0x8300-0x83FF`	Scratchpad RAM	Workspace registers; `.scratchpad` code from `0x8342` by default. `0x83C0-0x83FF` belongs to the console ISR/GPL.
`0x8400`, `0x8800-0x8C02`	Sound / VDP ports	Memory-mapped I/O, not RAM.
`0xA000-0xFFFF`	High 24K expansion RAM	`.data`, `.bss` and the C stack (grows down from the top).

Video RAM (16 KB) is on a separate bus and is only reachable through the VDP ports. Where your code and data end up is controlled per file; see Memory placement.

What happens at power-on (banked carts)

1. **Bank 0:** the console finds the cartridge header and jumps to `_start`. Interrupts are off. A small copier is copied into scratchpad (`0x8342`) and started.
2. **Scratchpad copier:** copies `.persistent` to low RAM (`0x2000`), `.data` to high RAM (`0xA000`), then your `.scratchpad` code over itself.
3. `_persistent_init`: clears `.bss` and calls `main()`.

`main()` should never return on a cartridge; end it with an endless loop.

Type model

`int` and `size_t` are 16-bit, `long` is 32-bit, and `float` is IEEE binary32 implemented in software. There is no heap: `malloc`, `free`, `calloc` and `realloc` are not provided. Use static or global buffers.

Installation

macOS · Linux · Windows

Each release contains the IDE/CLI, the full `tms9900-elf-*` toolchain, libraries, man pages, examples, the agent skill and the emulator. Download from the Downloads page.

macOS (Apple Silicon)

1. Open `tidepod-<version>-macos-arm64.dmg` and drag **Tidepod.app** to **Applications**.
2. To use the CLI and toolchain from a terminal, add both directories to your `PATH`:

~/.zshrc

```
export PATH="/Applications/Tidepod.app/Contents/MacOS:$PATH"
export PATH="/Applications/Tidepod.app/Contents/Resources/toolchain/bin:$PATH"
```

Linux (Debian / Ubuntu, amd64 or arm64)

Terminal

```
sudo apt install ./tidepod-<version>-linux-amd64.deb
```

The package installs to `/opt/tidepod` and links `/usr/bin/tidepod`. The toolchain lives in `/opt/tidepod/resources/toolchain/bin`; add it to your `PATH` if you want to call `tms9900-elf-gcc`, `tms9900-elf-gdb` or `ti994a-emu` directly:

~/.bashrc

```
export PATH="/opt/tidepod/resources/toolchain/bin:$PATH"
```

Windows (x86-64)

- **Installer** (`…-windows-x86_64-setup.exe`): installs to `Program Files\Tidepod`, adds Start Menu and desktop shortcuts, and by default puts Tidepod and the toolchain on the system `PATH`.

Uninstall from Settings → Apps.

- **Portable** (`…-windows-x86_64.zip`): unpack anywhere and run `tidepod.exe`. `PATH` is not changed; the IDE still finds its bundled toolchain. The bundle includes the small set of Unix tools the generated Makefiles need.

Check the installation

Terminal

```
tidepod new --help            # CLI usage for "new"
tms9900-elf-gcc --version     # GCC 14.2.0 for the TMS9900
ti994a-emu --version
```

> **Note**
>
> `tidepod` with no command, or with an argument it doesn't recognise as a command, opens the IDE. Use a subcommand with `--help` to stay in the terminal.

What's in the install

Path (inside the install)	Contents
`toolchain/bin`	`tms9900-elf-*`, `tms9900-packer`, `gpu-image-map`, `ti994a-emu`
`toolchain/lib`	`libc.a`, `libxb.a`, `crt0_*.o`, cartridge headers
`toolchain/include`	libc headers (`stdio.h`, `conio.h`, `resource_copy.h`, `f18a_gpu_host.h`, `debug.h`, `profiler.h`, …)
`toolchain/ldscripts`	Linker script templates used by the packer
`resources/man`	Man pages (also in the API reference)
`resources/examples`	Example projects
`resources/agent-skills`	The `ti99-tidepod-gamedev` agent skill

On macOS these directories are under `Tidepod.app/Contents/Resources/`; on Linux under `/opt/tidepod/resources/`.

Quick start: build and run a project

5 minutes

This creates a banked C cartridge from the built-in template, builds it and runs it. It is the shortest path to a working setup; the tutorials go further.

1. Create the project

Terminal

```
tidepod new hello --type banked --menu "HELLO" --path ./hello
cd hello
```

You get:

File	Purpose	Edit?
`hello.project`	Project description (JSON)	Through the CLI/IDE; hand edits need `tidepod sync`
`main.c`	Your program (template prints two lines)	Yes
`cart_header.asm`	Cartridge header with the menu name	Rarely
`include/bankswitch.h`	`switchtobank()` macro	No
`Makefile`	Generated build	**Never**
`Makefile.custom`	Your extra make rules; included by the Makefile and never regenerated	Yes
`layout.xml`	RPK cartridge layout	No (regenerated by sync)
`.agents/skills/`, `.cursor/skills/`	The agent skill	No

2. Build

```
Terminal
```

```
make
```

The result is `hello.bin` and the cartridge package `hello.rpk`. Use `make debug` instead for an unoptimised build with DWARF debug info (needed for the debugger).

> **First build of a new project**
>
> The first `make` may stop with *"resource_defs.h changed during packing ... Re-run make."* That is the packer creating `include/resource_defs.h` for the first time. Run `make` again.

3. Run

```
Terminal
```

```
tidepod run
```

This starts the emulator with `hello.rpk` and skips the title screen. `tidepod run` does **not** build: run `make` first, or you will be testing the previous ROM.

4. Add files as the project grows

```
Terminal
```

```
tidepod add src/player.c src/level.c   # new C sources
tidepod add resources/tiles.dat         # a binary resource
tidepod list                            # what's in the build
make
```

`add` and `remove` update the `.project` file and regenerate the Makefile in one go. See the CLI reference.

Same thing in the IDE

Open the project with `tidepod hello/hello.project` (or File → Open Project…). Build with `F7`, build and run in the embedded emulator with `Cmd + R` (`Ctrl + R` on Linux/Windows), or build a debug image and start the debugger with `F5`. See The IDE.

Project types and the .project file

Reference

Every Tidepod project is a directory with one `<name>.project` file in it. The file is JSON and describes everything the generated `Makefile` needs: the project type, the source list, per-file placement and compiler options. The IDE and the CLI both read and write it.

Project types

Pick the type with `tidepod new --type …`, change it later with `tidepod set-type` or in Project Settings.

`--type`	`"type"`	Output	Use it for
`8k`	0	8 KB cartridge (`.bin` + `.rpk`), no 32K RAM expansion required	Small programs that fit one ROM bank and the console's 256 bytes of scratchpad RAM
—	5	8 KB cartridge that uses the 32K expansion	Small code, more RAM. Selected in Project Settings (“8K Cart + 32K RAM”)
`banked` (default)	1	Bank-switched cartridge, any number of 8 KB banks; requires 32K	Almost everything: the packer spreads your code and data over banks automatically
`xb`	2	Banked cartridge built from Extended BASIC source (`main.bas`) compiled to machine code	BASIC programs that need to run fast
`ea5`	3	Editor/Assembler option 5 program files (`NAME1`, `NAME2`, …) and a `.dsk` image containing them	Disk-loaded programs
`custom`	4	Whatever your own Makefile builds	Projects with their own build; Tidepod never writes the Makefile

A typical file

mygame.project

```
{
  "name": "mygame",
  "cartMenuName": "MY GAME",
  "type": 1,
  "workspacePath": "/Users/me/projects/mygame",
  "toolchainPath": "/Applications/Tidepod.app/Contents/Resources/toolchain",
  "optLevelDebug": 0,
  "optLevelRelease": 3,
  "target_fps": 60,
  "includePaths": "",
  "libPaths": "",
  "customCompilerFlags": "",
  "customLinkerFlags": "",
  "scratchpadBase": "0x8380",
  "scratchpadSize": 128,
  "packing_algo": 0,
  "pcbType": "paged378",
  "dsk1": "disks/data.dsk",
  "srcFiles": [
    {"path": "cart_header.asm"},
    {"path": "src/main.c"},
    {"path": "src/isr.c", "forcePersistent": true},
    {"path": "src/gpu.c", "forceGpu": true, "gpuBase": "0x3000", "gpuStack":
"0x3F00"},
    {"path": "resources/tiles.dat"}
  ],
  "libFiles": [
  ]
}
```

Field reference

Key	Meaning
`name`	Project name. The output files are named after it in lower case (`mygame.bin`, `mygame.rpk`).
`cartMenuName`	Text shown in the console's cartridge menu (upper case, max. 31 characters).
`type`	Project type, see the table above.
`workspacePath`	Project directory.
`toolchainPath`	Toolchain used by the Makefile. Tidepod corrects it automatically when it points at a toolchain that no longer exists (for example after moving the install).
`optLevelDebug`, `optLevelRelease`	0, 1, 2 → `-O0` … `-O2`; 3 → `-Os`. Defaults: debug 0, release 3.
`target_fps`	Frame-rate target used by the profiler to compute the frame budget: 60, 30, 20, 15 or 10. Budget = (60 / target) × 50,065 CPU cycles.
`includePaths`	Extra include directories, separated by spaces or `;`. Each becomes `-I…`.
`libPaths`	Extra library directories.
`customCompilerFlags`, `customLinkerFlags`	Appended verbatim to the compile / link commands.
`scratchpadBase`, `scratchpadSize`	Where the packer may place scratchpad code. Omitted = packer default (base `0x8342`, up to `0x8400`). Project Settings offers presets: full (`0x8320`), music libraries (`0x8340`), disk I/O (`0x8380`).
`packing_algo`	Bank packing strategy: 0 = best-fit decreasing (`bfd`), 1 = first-fit decreasing (`ffd`), 2 = call-graph clustering (`cluster`).

Key	Meaning
`pcbType`	Cartridge board written to `layout.xml`: `standard` (8K), `paged378` (banked default) or `paged379i` (inverted bank order; old projects that carry it by mistake are migrated automatically).
`dsk1` – `dsk3`	Disk images mounted when the project is opened in the IDE or run with `tidepod run`. Workspace-relative or absolute paths. See Disks & peripherals.
`hard1` – `hard3`	Raw hard-disk images (`.hd`, `.hdf`, `.raw`) for an HFDC controller.
`emu`	Optional per-project emulator overrides (PEB cards, VDP). See The emulator.
`srcFiles`	Build members, see below.
`libFiles`	Extra static libraries (`.a`) to link. `"forcePersistent": true` puts the whole library in persistent RAM (`--persistent-lib`).

`srcFiles` entries

The file extension decides how a member is built:

- `.c` — compiled with `tms9900-elf-gcc`.
- `.asm` — assembled with `tms9900-elf-as`.
- `.bas` — Extended BASIC, compiled with `tms9900-elf-gcc -x xb` (XB projects).
- `.dat` — any binary file; becomes a ROM resource.

Option	CLI equivalent	Effect
`"forcePersistent": true`	`add --persistent`	Code and data go to RAM that is always mapped, instead of a ROM bank.
`"forceScratchpad": true`	`add --scratchpad`	Code goes to the fast 16-bit scratchpad RAM.
`"forceGpu": true`, `"gpuBase"`, `"gpuStack"`	`add --gpu --gpu-base … --gpu-stack …`	Builds the file as an F18A GPU image instead of host code. See F18A GPU code.

The same placement can be requested from inside a C file with `#pragma bank persistent`, `#pragma bank scratchpad` or `#pragma bank gpu` (plus `#pragma gpu base` / `#pragma gpu stack`). The pragmas are read when the project is synced.

> **Editing by hand**
>
> You can edit the `.project` file in any editor. Run `tidepod sync` afterwards so the Makefile and `layout.xml` are regenerated from it.

The Tidepod IDE

Reference

The IDE combines an editor, the build, the emulator and a source-level debugger in one window. Start it with `tidepod` (empty) or `tidepod path/to/name.project`. It runs the same `make` and the same emulator core as the command line, so a project builds identically in both.

Window layout

Area	What it does
Explorer (left)	File tree of the project directory. Toggle with View → Toggle Explorer.
Editor (centre)	Tabbed editor with syntax highlighting for C, assembly and Extended BASIC; multi-cursor; click in the gutter to set a breakpoint. Man pages open in read-only tabs.
Emulator	The running TI-99/4A. Click it to give it keyboard focus; click elsewhere to return focus to the editor. Toggle with View → Toggle Emulator.
Debug panel	Tabs **CPU** (registers, memory), **VDP** (VDP registers, VRAM), **ASM** (disassembly), **BRK** (breakpoints), **WCH** (watches, with struct expansion), **GFX** (pattern, colour and sprite tables), **DSK** (disk mounts and contents) and **PRF** (profiler).
Output	Build output and logpoint messages. Toggle with View → Toggle Output.

Menus

- **File** — New / Open / Save / Save As / Close Tab; Save Snapshot... / Load Snapshot... (emulator state); New / Open / Save / Close Project; Project Settings...; Preferences....
- **Edit** — Undo, Redo, Cut, Copy, Paste, Select All, Find..., Find in Project..., Replace..., Replace in Project....
- **View** — Toggle Explorer, Toggle Output, Toggle Emulator.
- **Build** — Build, Run, Debug, Clean Workspace.

Default keyboard shortcuts

`Mod` is `Cmd` on macOS and `Ctrl` on Linux and Windows. All shortcuts can be rebound in Preferences.

Action	Shortcut
Build (release)	`F7`
Build & Run	`Mod` + `R`
Build & Debug (debug build, debugger attached)	`F5`
Pause / continue	`F6`
Clean	`Mod` + `Shift` + `K`
Step over / into / out (source)	`F10` / `F11` / `Shift` + `F11`
Step one instruction	`Shift` + `F10`
Save file	`Mod` + `S`
New / open project	`Mod` + `N` / `Mod` + `O`
Find / find in project	`Mod` + `F` / `Mod` + `Shift` + `F`
Next / previous result	`Mod` + `G` / `Mod` + `Shift` + `G`
Replace in project	`Mod` + `Shift` + `R`
Undo / redo	`Mod` + `Z` / `Mod` + `Shift` + `Z`
Add cursor above / below	macOS `Ctrl` + `Shift` + `↑` / `↓` ; others `Ctrl` + `Alt` + `↑` / `↓`
Select next occurrence	`Mod` + `D`

Project Settings

File → Project Settings… edits the `.project` file: project type, cartridge menu name, optimisation levels, target frame rate, scratchpad preset, packing algorithm, include/library paths, extra compiler and linker flags. Saving regenerates the Makefile.

Disks and emulator settings

The Disk dialog and the **DSK** tab mount disk and hard-disk images and choose the PEB configuration (Inherit / None / TI FDC / HFDC / WHT SCSI). Mounts are stored in the project; machine-wide defaults (PEB cards, VDP) live in Preferences. See Disks & peripherals.

Session restore

The IDE remembers the open project, tabs and layout and restores them on the next launch.

Tutorial 1: Minimal 8K C Cartridge

Beginner Walkthrough

> **Draft — being rewritten**
>
> This tutorial's code has not yet been verified against the current Tidepod toolchain and contains API calls that do not exist in libc. Use the API reference and the examples shipped in `resources/examples/` until this page is updated.

In this tutorial, we will scaffold, write, compile, and run a minimal standard 8K ROM cartridge in C. We will walk through **every single line of code** to explain how the TMS9900 16-bit microprocessor and the TMS9918A Video Display Processor (VDP) interact at the register level.

Hardware Architecture & Memory Map

An 8K non-banked cartridge occupies the CPU address window `0x6000-0x7FFF` (8,192 bytes). The TI-99/4A console ROM checks this memory range at power-on for a standard cartridge header (containing the string "HELLO CART", entrypoint vectors, and menu item index).

Memory Address	Bus Access	Function & Description
`0x6000 - 0x7FFF`	16-bit Read-Only	Cartridge ROM space. Contains cartridge header, startup bootstrap, and compiled C code.
`0x8300 - 0x83FF`	16-bit Read/Write	256 bytes of internal CPU zero-wait-state Scratchpad RAM. Holds active CPU register workspaces (R0–R15).
`0x8800 / 0x8C00`	8-bit Read / Write	VDP Data Port. Reads and writes VRAM byte streams into the TMS9918A.
`0x8802 / 0x8C02`	8-bit Read / Write	VDP Control Port. Read queries the Status Register (VDPST); write sets the 14-bit VRAM address pointer or configures VDP Registers 0–7.

Step 1: Create the Project via CLI

Scaffold a clean workspace targeting standard 8K cartridge geometry. This automatically writes the project manifest (`HelloCart.project`), source boilerplate, and builds the Makefile:

Terminal

```
tidepod new HelloCart --type 8k --menu "HELLO CART" --path ./HelloCart
cd HelloCart
```

Step 2: Complete Code Listing (`src/main.c`)

Replace the contents of `src/main.c` with this complete, self-contained demonstration:

src/main.c

```c
#include <stdio.h>
#include <conio.h>

int main(void) {
    // Initialize 40-column hardware text mode
    ti_text_mode(40);

    // Set screen border color to Dark Blue (palette index 4)
    bordercolor(4);

    // Clear screen and position cursor at column 4, row 3
    clrscr();
    gotoxy(4, 3);
    puts("HELLO FROM C ON TI-99/4A!");

    // Display toolchain banner
    gotoxy(4, 5);
    puts("BUILT WITH TIDEPOD C99 SUITE");

    // Display interactive prompt
    gotoxy(4, 8);
    puts("PRESS ANY KEY TO CONTINUE...");

    // Scan keyboard matrix and wait for user keypress
    int key = cgetc();

    // Output confirmation with ASCII character value
    gotoxy(4, 11);
    printf("KEY DETECTED: ASCII %d ('%c')
", key, (char)key);

    gotoxy(4, 13);
    puts("SYSTEM READY. HALTING IN IDLE LOOP.");

    // Bare-metal halt loop (prevents crash on OS return)
    while (1) {
    }

```

```
38:     return 0;
39: }
```

Line-by-Line Code Breakdown

Here is the exhaustive line-by-line explanation detailing every operation executed by the TMS9900 CPU and 9918A VDP:

Lines 1-2 — **`#include <stdio.h>` & `#include <conio.h>`:** Ingests standard C formatted output declarations (`printf`) and Tidepod's direct hardware console I/O library (`conio.h`). On the TI-99/4A, `conio.h` functions bypass heavy file-stream abstractions and talk directly to the VDP memory-mapped I/O ports at `0x8C00` and `0x8C02`.

Line 4 — **`int main(void)`:** The application entrypoint. When the console boots, the TI ROM checks `0x6000`, detects the cartridge header, and displays "HELLO CART" on the master menu. Selecting it jumps to the Tidepod C runtime bootstrap (`_start`), which initializes the TMS9900 Workspace Pointer (WP), sets up the R10 stack pointer in high RAM, clears `.bss`, copies initialized globals from ROM to RAM, and calls `main()`.

Line 6 — **`ti_text_mode(40);`:** Programs the TMS9918A Video Display Processor into Text Mode (40 columns by 24 rows, 960 total character tiles). This writes a 2-byte command sequence to VDP control port `0x8C02`: clearing Bit 1 in Register 0 (mode select M2=0) and setting Bit 4 in Register 1 (mode select M1=1, screen unblanked). It also sets up the Pattern Name Table at VRAM `0x0000` and Character Pattern Generator at VRAM `0x0800`.

Line 9 — **`bordercolor(4);`:** Sets the screen backdrop and overscan border color. This writes palette index 4 (Dark Blue) to VDP Register 7 via control port `0x8C02`.

Line 12 — **`clrscr();`:** Clears the entire 40×24 display by streaming 960 ASCII space characters (byte `0x20`) into VRAM address `0x0000` through VDP data port `0x8C00`. It then resets the internal hardware cursor to (0, 0).

Lines 13-14

`gotoxy(4, 3); puts("HELLO FROM C ON TI-99/4A!");`: `gotoxy(col, row)` computes the VRAM offset using the formula `address = 0x0000 + (row * 40) + col` and transmits the 14-bit VRAM address to port `0x8C02`. `puts(...)` then loops through each character of the string and writes it to port `0x8C00`. Because the TMS9918A hardware auto-increments its internal address latch on each write, subsequent characters draw horizontally across the screen at maximum bus speed.

Lines 17-18

`gotoxy(4, 5); puts("BUILT WITH TIDEPOD C99 SUITE");`: Repurposes the VRAM address pointer to row 5, column 4 and writes the second header line.

Lines 21-22

`gotoxy(4, 8); puts("PRESS ANY KEY TO CONTINUE...");`: Renders the user prompt at row 8.

Line 25

`int key = cgetc();`: Executes a hardware keyboard scan routine. The TI-99/4A keyboard is not memory-mapped; it is connected to the TMS9901 Communications Register Unit (CRU) serial I/O chip. `cgetc()` pulses column select wires, reads back row pins, performs software switch debouncing, translates matrix scan codes into standard ASCII, and blocks CPU execution until a key is pressed.

Lines 28-29

`gotoxy(4, 11); printf("KEY DETECTED: ASCII %d ('%c') ", key, (char)key);`: Uses the lightweight Tidepod C99 `printf` formatter to format the integer ASCII code and corresponding glyph character into a scratch buffer and streams the characters into VRAM.

Lines 31-32

`gotoxy(4, 13); puts("SYSTEM READY. HALTING IN IDLE LOOP.");`: Prints confirmation that the program flow has concluded successfully.

Lines 35-36

`while (1) {}`: A fundamental bare-metal embedded rule: a vintage microcomputer running a cartridge has no operating system command prompt or shell to return to. If `main()` returned, the program counter (PC) would branch into unmapped memory, executing random garbage opcodes and

crashing the console. The infinite loop safely keeps the CPU in a stable idle halt.

Lines 38-39 **`return 0; }`:** Closes the function definition for ISO C99 conformance.

Step 3: Compile and Test

Terminal

```
tidepod sync && make
tidepod run
```

`tidepod run` launches the native standalone emulator (`ti994a-emu`), mounts your generated cartridge image (`HelloCart.rpk`), and runs the program at real-time 3.0 MHz TMS9900 clock speed.

Tutorial 2: Sprites, Joystick & 3-Voice Sound

`Intermediate Walkthrough`

> **Draft — being rewritten**
>
> This tutorial's code has not yet been verified against the current Tidepod toolchain and contains API calls that do not exist in libc. Use the API reference and the examples shipped in `resources/examples/` until this page is updated.

In this tutorial, we will build a complete interactive arcade loop targeting a **Banked 512KB Cartridge**. We will design custom hardware sprites, poll analog joysticks via CRU, synthesize audio waveforms on the SN76489 sound chip, and lock the animation to the 60Hz vertical blank interrupt.

Hardware Subsystems Explained

- **Hardware Sprites:** The TMS9918A supports up to 32 simultaneous hardware sprites (indices 0 to 31). Each sprite's position, pattern glyph, and color are stored in a 4-byte record inside the Sprite Attribute Table (SAT at VRAM `0x0380`).
- **PSG Sound Generation:** The TI-99/4A features a Texas Instruments SN76489 Complex Sound Generator mapped to CPU write port `0x8400`. It contains 3 independent square-wave tone channels and 1 periodic/white noise generator. Volume levels range from `0` (maximum loudness / 0 dB) to `30` (complete attenuation / silence).
- **VSYNC Synchronization:** Reading the VDP Status Register (VDPST at `0x8802`) checks the vertical blanking flag (Bit 7). Crucial rule: *reading this port automatically resets the flag to zero*. Tidepod's `vsync()` polls this register in a tight assembly loop, locking the game update rate to exactly 60 FPS (NTSC) or 50 FPS (PAL) and eliminating sprite flicker and tearing.

Step 1: Scaffold Banked Game

Terminal

```
tidepod new SpaceBlaster --type banked --size 512 --menu "SPACE BLASTER" --path
./SpaceBlaster
cd SpaceBlaster
```

Step 2: Complete Code Listing (`src/main.c`)

src/main.c

```
 1: #include <stdio.h>
 2: #include <conio.h>
 3:
 4: // Tidepod Extended BASIC C runtime library bindings
 5: extern void pattern(int char_code, const char *hex_pattern);
 6: extern void sprite(int num, int char_code, int color, int r, int c, int rv,
int cv);
 7: extern void locate(int num, int r, int c);
 8: extern void color(int char_set, int f_color, int b_color);
 9: extern void joyst(int unit, int *x, int *y);
10: extern void sound(int duration_ms, int freq, int vol);
11: extern void vsync(void);
12:
13: int main(void) {
14:     // Player spaceship coordinates (pixels)
15:     int ship_x = 128;
16:     int ship_y = 96;
17:     int joy_x = 0;
18:     int joy_y = 0;
19:
20:     // Enemy star coordinates and automatic velocity
21:     int enemy_x = 30;
22:     int enemy_y = 40;
23:     int enemy_vx = 2;
24:     int enemy_vy = 1;
25:
26:     // Switch to Graphics I Mode (32x24 tiles with 32 hardware sprites)
27:     ti_text_mode(32);
28:     clrscr();
29:
30:     // Define custom 8x8 sprite pattern 65 ('A') for Player Spaceship
31:     // Hex: 18 (..XX....), 3C (.XXXX...), 7E (XXXXXX..), DB (XX.XX.XX), FF
(XXXXXXXX), 24 (..X..X..), 5A (.X.XX.X.), 42 (.X....X.)
32:     pattern(65, "183C7EDBFF245A42");
33:
34:     // Define custom 8x8 sprite pattern 66 ('B') for Enemy Hazard Star
35:     pattern(66, "002418FF18240000");
```

```
36: 
37:     // Spawn Hardware Sprite #1: Player Ship (Cyan, 7) at (96, 128)
38:     sprite(1, 65, 7, ship_y, ship_x, 0, 0);
39: 
40:     // Spawn Hardware Sprite #2: Enemy Star (Dark Red, 6) at (40, 30)
41:     sprite(2, 66, 6, enemy_y, enemy_x, 0, 0);
42: 
43:     // Play startup fanfare on PSG: 440 Hz (A4) for 200 ms at full volume
(0)
44:     sound(200, 440, 0);
45: 
46:     // 60 FPS Interactive Game Loop
47:     while (1) {
48:         // 1. Wait for VDP Vertical Blank interrupt (prevents visual
tearing)
49:         vsync();
50: 
51:         // 2. Poll Joystick #1 (Wired to 9901 CRU)
52:         joyst(1, &joy_x, &joy_y);
53: 
54:         // 3. Update player position based on joystick deflection
55:         if (joy_x ≠ 0 || joy_y ≠ 0) {
56:             ship_x += (joy_x * 2);
57:             ship_y -= (joy_y * 2); // Invert Y because screen rows increase
downwards
58: 
59:             // Clamp player to visible screen boundaries
60:             if (ship_x < 8) ship_x = 8;
61:             if (ship_x > 248) ship_x = 248;
62:             if (ship_y < 8) ship_y = 8;
63:             if (ship_y > 184) ship_y = 184;
64: 
65:             // Update Sprite #1 position in VRAM Attribute Table
66:             locate(1, ship_y, ship_x);
67:         }
68: 
69:         // 4. Update and bounce enemy hazard
70:         enemy_x += enemy_vx;
71:         enemy_y += enemy_vy;
72: 
73:         // Bounce off screen walls and play a bounce blip
74:         if (enemy_x ≤ 8 || enemy_x ≥ 248) {
```

```
75:             enemy_vx = -enemy_vx;
76:             sound(30, 880, 5); // Short high-pitch chirp (880 Hz, volume 5)
77:         }
78:         if (enemy_y ⩽ 8 || enemy_y ⩾ 184) {
79:             enemy_vy = -enemy_vy;
80:             sound(30, 660, 5); // Short mid-pitch chirp (660 Hz, volume 5)
81:         }
82:
83:         // Update Sprite #2 position in VRAM
84:         locate(2, enemy_y, enemy_x);
85:     }
86:
87:     return 0;
88: }
```

Line-by-Line Code Breakdown

Lines 1-2 — **`#include <stdio.h>` & `#include <conio.h>`:** Includes basic console and runtime headers.

Lines 5-11 — **Extended BASIC Runtime Bindings:** Declares the standard Tidepod C runtime functions matching TI Extended BASIC semantics. `pattern` sets sprite shape, `sprite` registers a hardware sprite in VRAM, `locate` repositions it, `joyst` scans hardware controllers, `sound` triggers PSG frequency generation, and `vsync` halts execution until the electron beam enters vertical blank.

Lines 14-18 — **Player State Variables:** Allocates integers on the local stack frame for player coordinates (`ship_x`, `ship_y`) and joystick offsets (`joy_x`, `joy_y`). Scalar integer locals are safe and register-promoted by `tms9900-gcc`.

Lines 21-24 — **Enemy Hazard State Variables:** Initializes position (30, 40) and per-frame delta velocity (vx=2, vy=1) for an automated bouncing hazard star.

Lines 27-28 — **`ti_text_mode(32); clrscr();`:** Switches the VDP to Graphics Mode I (32 columns × 24 rows, 256×192 pixels). Unlike 40-column text mode, 32-column

graphics mode enables hardware sprite rendering and individual tile color attributes.

Lines 32 & 35	**`pattern(65, "183C7EDBFF245A42"); pattern(66, ...);:`** Uploads 8 bytes (16 hex digits) of bitmap data to the Sprite Pattern Generator table in VRAM. Each hex pair corresponds to 8 horizontal pixels. 18 in binary is `00011000`, creating the tip of the spaceship. Pattern 65 is assigned to character 'A' and Pattern 66 to 'B'.
Lines 38 & 41	**`sprite(1, 65, 7, ...); sprite(2, 66, 6, ...);:`** Writes entries into the VRAM Sprite Attribute Table (SAT at `0x0380`). Sprite #1 is configured with pattern 65, color Cyan (index 7), and placed at pixel coordinates (96, 128). Sprite #2 is configured with pattern 66, Dark Red (index 6), at (40, 30). Velocities `(0, 0)` keep automatic hardware motion disabled so we control coordinates manually in C.
Line 44	**`sound(200, 440, 0);:`** Programs Channel 1 of the SN76489 sound chip mapped to `0x8400`. It emits a 440 Hz square wave (musical pitch A4) for 200 milliseconds at full volume (0 dB attenuation).
Line 47	**`while (1) {:`** The main game loop. Runs indefinitely at a rock-solid 60 frames per second.
Line 49	**`vsync();:`** Polls the VDP Status Register (`0x8802`) until Bit 7 (VBLANK) becomes high. This synchronizes the loop with the TV refresh cycle, preventing video tearing and standardizing game speed across all hardware.
Line 52	**`joyst(1, &joy_x, &joy_y);:`** Queries Joystick #1 via the 9901 CRU bus. Fills `joy_x` and `joy_y` with deflection vectors: `-4` (left / down), `0` (centered), or +4 (right / up).
Lines 55-67	**Player Movement & Clamping:** Adjusts `ship_x` and `ship_y` by the joystick vectors, clamps values within visible CRT display boundaries (8 to 248 for X, 8

to 184 for Y), and calls `locate(1, ship_y, ship_x)` to update Sprite #1's coordinate bytes directly in VRAM SAT.

Lines 70-82	**Hazard Physics & Sound Effects:** Advances enemy position by its velocity. When the star strikes any screen edge, its velocity is negated (`enemy_vx = -enemy_vx`) and `sound(30, freq, 5)` fires a 30ms high-pitch audio blip (880 Hz or 660 Hz at moderate volume) on the PSG chip.
Line 84	**`locate(2, enemy_y, enemy_x);`:** Updates Sprite #2's position record in VRAM.
Lines 87-88	**Function Exit:** Closes the main game loop and completes the C function frame.

Step 3: Compile and Test

Terminal

```
tidepod sync && make
tidepod run
```

Tutorial 3: The Binary Asset Pipeline (`.dat`)

`Resource Management`

> **Draft — being rewritten**
>
> This tutorial's code has not yet been verified against the current Tidepod toolchain and contains API calls that do not exist in libc. Use the API reference and the examples shipped in `resources/examples/` until this page is updated.

Complex homebrew games require hundreds of kilobytes of level maps, tile pattern tables, PCM digitized voice samples, and music tracks. Tidepod provides an automated asset pipeline that ingests any raw binary file, assigns it to a cartridge ROM bank, and generates type-safe cross-bank access macros.

How Banking Works Under the Hood

On the TI-99/4A, cartridge ROM space is fixed at 8KB (`0x6000-0x7FFF`). To support cartridges up to 512KB (or larger), hardware banking boards swap 8KB chunks into this address window whenever the CPU writes to a specific trigger address.

Tidepod's `tms9900-packer` tool automatically groups your compiled C objects and binary assets (`.dat`) into separate 8KB banks. It creates header symbol files:

- `resource_defs.h`: Defines symbol names, bank indices, and byte lengths for all `.dat` files.
- `resource_copy.h`: Provides high-speed streaming routines (`rom_to_vram` , `rom_to_ram`) that latch the bank, transfer data, and restore the previous bank safely.

Step 1: Adding Assets to the Build

Place your raw binary files (e.g. `resources/tileset.dat` and `resources/level_map.dat`) into the project folder, then register them with the CLI:

Terminal

```
tidepod add resources/tileset.dat
tidepod add resources/level_map.dat
tidepod sync
```

Step 2: Complete Code Listing (`src/main.c`)

src/main.c

```
 1: #include <stdio.h>
 2: #include <conio.h>
 3: #include "resource_defs.h"
 4: #include "resource_copy.h"
 5:
 6: // Allocate persistent level buffer in CPU BSS (off the C stack!)
 7: static unsigned char s_active_level[1024];
 8:
 9: int main(void) {
10:     // Initialize 32-column graphics mode
11:     ti_text_mode(32);
12:     clrscr();
13:
14:     // 1. Stream 2048 bytes of background tileset from banked ROM into VRAM
at 0x0000.
15:     //    rom_to_vram handles bank switching, VDP address register
programming,
16:     //    burst byte streaming, and original bank restoration!
17:     rom_to_vram(unwrap(tileset), 0x0000);
18:
19:     // 2. Stream 1024 bytes of level map data from banked ROM into CPU RAM
buffer
20:     rom_to_ram(unwrap(level_map), s_active_level, sizeof(s_active_level));
21:
22:     // 3. Draw first 32 tiles of the level onto the top row of the screen
23:     gotoxy(0, 0);
24:     for (int i = 0; i < 32; i++) {
25:         cputc(s_active_level[i]);
26:     }
27:
28:     gotoxy(2, 5);
29:     puts("ASSET PIPELINE STREAMING COMPLETE!");
30:     gotoxy(2, 7);
31:     printf("TILES LOADED: %u BYTES TO VRAM
", (unsigned int)tileset_size);
32:     gotoxy(2, 8);
33:     printf("MAP LOADED:   %u BYTES TO RAM
```

```
", (unsigned int)level_map_size);
34:
35:     // Idle loop
36:     while (1) {
37:     }
38:
39:     return 0;
40: }
```

Line-by-Line Code Breakdown

Lines 1-4 — **Headers:** Includes standard I/O, console routines, and the auto-generated headers `resource_defs.h` and `resource_copy.h` produced by `tms9900-packer`.

Lines 6-7 — **Static RAM Buffer (`s_active_level`):** Declares a 1024-byte buffer for level map tiles. **Notice the `static` keyword:** this places the array in the `.bss` segment located in upper 24K expansion RAM (`0xA000`). Never declare arrays of this size as local variables inside a function, as it would cause an instant stack overflow on the TMS9900!

Lines 10-12 — **Screen Setup:** Activates 32-column graphics mode and erases the screen.

Lines 14-17 — **`rom_to_vram(unwrap(tileset), 0x0000);`:** `unwrap(tileset)` is a macro defined in `resource_defs.h` that expands to: `tileset_bank, tileset_rom_addr, tileset_size`. The `rom_to_vram` routine reads the target ROM bank, switches the hardware cartridge latch to map that bank into `0x6000`, sets the VDP address register to `0x0000`, copies all pattern bytes directly from ROM into VRAM, and then switches back to the caller's bank.

Lines 19-20 — **`rom_to_ram(unwrap(level_map), s_active_level, ...);`:** Transfers the raw level layout from its assigned ROM bank directly into CPU expansion RAM for rapid tile lookup during gameplay.

Lines 23-26

`cputc(s_active_level[i]);`: Renders the loaded level tiles onto the top row of the screen, proving that the RAM buffer was successfully populated from banked ROM.

Lines 28-33

Diagnostics: Prints confirmation messages and displays the exact byte sizes of the assets using the generated constants `tileset_size` and `level_map_size`.

Lines 36-40

Halt Loop & Clean Exit: Keeps the CPU safely running in an idle loop.

Tutorial 4: RAM Placement & Stack Safety

`Advanced Optimization`

> **Draft — being rewritten**
>
> This tutorial's code has not yet been verified against the current Tidepod toolchain and contains API calls that do not exist in libc. Use the API reference and the examples shipped in `resources/examples/` until this page is updated.

The TMS9900 CPU has unique memory layout requirements and hazards. Understanding memory segmentation and the strict rules governing the C workspace stack is vital for building robust homebrew applications that never crash.

TI-99/4A Memory Segmentation Map

Address Window	Size	Bus Speed	Placement Role & Strategy
`0x2000 - 0x3FFF`	8 KB	8-bit multiplexed (Wait states)	**Lower Expansion RAM (Persistent):** Always resident, never bank-swapped. Use for interrupt service routines, audio playback drivers, and shared state across ROM banks (`--persistent`).
`0x6000 - 0x7FFF`	8 KB	8-bit Cartridge ROM	**Cartridge Window:** Bank-switched ROM space for code, read-only tables, and packed `.dat` assets.
`0x8300 - 0x83FF`	256 Bytes	16-bit Zero-Wait-State	**Internal CPU Scratchpad RAM:** Extremely fast on-die RAM. Stores active register workspaces R0–R15. Critical hot-loops can be placed here using `--scratchpad` .
`0xA000 - 0xFFFF`	24 KB	8-bit multiplexed (Wait states)	**Upper Expansion RAM:** Standard C `.text` , `.data` , `.bss` , and C call stack (pointed to by R10).

The Cardinal Stack Rule: Never Allocate Large Locals

> ⚠ **Stack Overflow Signature: Runaway Crash at 0x839C**
>
> The C workspace stack on the TMS9900 is managed via Register 10 (R10). The stack starts near the top of memory and grows downwards. **Never** allocate sector buffers (`char buf[256]`), directory slot tables, or file copy windows as local function variables. Doing so smashes the return address and corrupts system scratchpad vectors around `0x839C`, causing an immediate console crash! Always use file-scope `static` variables.

Complete Code Listing (`src/safe_memory.c`)

src/safe_memory.c

```
 1: #include <stdio.h>
 2: #include <string.h>
 3: #include <conio.h>
 4:
 5: // Fast raster math function placed in ultra-fast 16-bit Scratchpad RAM
(0x8300)
 6: __attribute__((section(".scratchpad")))
 7: int fast_sin(int angle) {
 8:     static const int s_sin_lut[8] = { 0, 70, 100, 70, 0, -70, -100, -70 };
 9:     return s_sin_lut[angle & 7];
10: }
11:
12: // Audio playback routine placed in Lower 8K Persistent RAM (0x2000)
13: // Guaranteed to remain accessible regardless of which ROM bank is currently
active!
14: __attribute__((section(".persistent")))
15: void persistent_audio_tick(void) {
16:     // Audio driver code safe from bank swaps
17: }
18:
19: // SAFE: 256-byte sector buffer placed in .bss Expansion RAM (NOT on the
stack!)
20: static unsigned char s_disk_sector_buffer[256];
21:
22: int safe_process_sector(unsigned char fill_pattern) {
23:     // Fills the 256-byte static buffer safely
24:     memset(s_disk_sector_buffer, fill_pattern,
sizeof(s_disk_sector_buffer));
25:
26:     int checksum = 0;
27:     for (int i = 0; i < 256; i++) {
28:         checksum += s_disk_sector_buffer[i];
29:     }
30:     return checksum;
31: }
32:
33: int main(void) {
```

```
34:     ti_text_mode(40);
35:     clrscr();
36:
37:     int val = fast_sin(2);
38:     int sum = safe_process_sector(0x42);
39:
40:     printf("SCRATCHPAD SIN(2): %d
", val);
41:     printf("STATIC BSS BUFFER CHECKSUM: %d
", sum);
42:     puts("MEMORY TEST COMPLETED WITH ZERO STACK VIOLATIONS.");
43:
44:     while (1) {
45:     }
46:
47:     return 0;
48: }
```

Line-by-Line Code Breakdown

Lines 1-3

Headers: Includes standard declarations for `stdio.h`, `string.h` (for `memset`), and `conio.h`.

Lines 5-10

`fast_sin(...)` in Scratchpad RAM: The `__attribute__((section(".scratchpad")))` attribute directs the linker to place this function inside the 256-byte internal 16-bit CPU RAM at `0x8300`. Because this memory has zero wait states and does not go through the console 8-bit multiplexer, instruction fetch and lookup operations execute up to 300% faster than ROM.

Lines 12-17

`persistent_audio_tick(...)` in Persistent RAM: The `__attribute__((section(".persistent")))` attribute compiles this routine into Lower 8K RAM (`0x2000-0x3FFF`). This memory is permanently connected and never affected by cartridge banking latches, making it the ideal home for interrupt-driven sound players that must execute while other ROM banks are swapped in. (You can also achieve this via CLI: `tidepod add src/audio.c --persistent`).

Lines 19-20

`static unsigned char s_disk_sector_buffer[256];`: Crucial best practice: By declaring the 256-byte buffer at file-scope with `static`, the compiler allocates storage in the uninitialized data segment (`.bss`) in high RAM. Zero bytes are taken from the R10 stack pointer.

Lines 22-31

`safe_process_sector(...)`: Operates on `s_disk_sector_buffer` without any danger of stack corruption. The local variables `fill_pattern` and `checksum` fit into CPU registers R1 and R2.

Lines 33-48

Main Execution Flow: Initializes text mode, exercises the Scratchpad routine, computes the checksum, prints diagnostic messages to screen, and halts in an idle loop.

Tutorial 5: F18A FPGA Graphics & GPU in C

`FPGA Coprocessor`

> **Draft — being rewritten**
>
> This tutorial's code has not yet been verified against the current Tidepod toolchain and contains API calls that do not exist in libc. Use the API reference and the examples shipped in `resources/examples/` until this page is updated.

The **F18A** is a modern FPGA drop-in replacement for the TMS9918A VDP. In addition to 80-column text mode, 64-color palettes, and bitmap scrolling, it includes an embedded **100MHz 16-bit GPU coprocessor** with direct access to 16KB VRAM. Tidepod is the only toolchain in the world that allows you to write F18A GPU routines directly in ISO C99!

Dual-Processor Architecture

Building an F18A GPU project involves two separate programs executing in tandem:

1. **GPU Coprocessor Shader (`src/gpu_fx.c`):** Compiled by `tms9900-gcc` with GPU target flags. Runs entirely inside the F18A FPGA at 100MHz (over 30× faster than the 3MHz host CPU!).
2. **Host TMS9900 Controller (`src/main.c`):** Runs on the console CPU. Unlocks F18A enhanced registers, uploads the GPU binary to VRAM, and triggers execution.

Step 1: Write the GPU Coprocessor Shader (`src/gpu_fx.c`)

src/gpu_fx.c

```
 1: // Code executed natively on the 100MHz 16-bit F18A GPU coprocessor
 2: void gpu_main(void) {
 3:     // Direct pointer to F18A VRAM (mapped at address 0x0000 in GPU memory
space)
 4:     volatile unsigned char *vram = (volatile unsigned char *)0x0000;
 5:
 6:     // Rapidly fill screen tiles with a dynamic XOR geometric pattern
 7:     for (int y = 0; y < 24; y++) {
 8:         for (int x = 0; x < 40; x++) {
 9:             int offset = (y * 40) + x;
10:             vram[offset] = (unsigned char)((x ^ y) + 65);
11:         }
12:     }
13:
14:     // Halt GPU execution cleanly and wait for next host trigger
15:     __asm__ volatile ("idle");
16: }
```

Line-by-Line Breakdown: GPU Shader

Lines 1-4

`volatile unsigned char *vram = 0x0000;`: On the host TMS9900, VRAM must be accessed through slow 8-bit ports (`0x8C00`). On the F18A GPU, the entire 16KB VRAM is mapped directly into the GPU's memory address space starting at `0x0000`. Memory reads and writes execute in a single 100MHz clock cycle!

Lines 6-12

Procedural Screen Generation: Loops through all 960 character tiles (40×24) and writes an XOR calculation directly into VRAM. At 100MHz, this entire 960-byte screen refresh completes in a fraction of a millisecond.

Lines 14-16

`__asm__ volatile ("idle");`: Executes the TMS9900 `IDLE` opcode. On the F18A GPU, `IDLE` stops the coprocessor pipeline and clears the GPU busy

bit in VDP status, signaling to the host CPU that the GPU task is complete.

Step 2: Register GPU File with the Build System

Terminal

```
tidepod add src/gpu_fx.c --gpu --gpu-base 0x3000 --gpu-stack 0x3F00
tidepod sync
```

This tells the toolchain to compile `gpu_fx.c` to link at VRAM address `0x3000` with its stack at `0x3F00`, and to generate the header `gpu_fx_map.h`.

Step 3: Host Execution from TMS9900 CPU (`src/main.c`)

src/main.c

```
 1: #include <stdio.h>
 2: #include <conio.h>
 3: #include "resource_defs.h"
 4: #include "resource_copy.h"
 5: #include "gpu_fx_map.h"
 6:
 7: // F18A hardware register control routines
 8: extern void f18a_unlock(void);
 9: extern void f18a_gpu_start(unsigned int vram_addr);
10:
11: int main(void) {
12:     ti_text_mode(40);
13:     clrscr();
14:     puts("INITIALIZING F18A HARDWARE...");
15:
16:     // 1. Unlock enhanced F18A registers by writing magic sequence >55, >55
to VR57
17:     f18a_unlock();
18:
19:     // 2. Stream compiled GPU machine code from cartridge ROM into VRAM at
0x3000
20:     rom_to_vram(unwrap(gpu_fx), 0x3000);
21:
22:     gotoxy(0, 5);
23:     puts("LAUNCHING 100 MHZ F18A GPU SHADER...");
24:
25:     // 3. Trigger GPU execution at VRAM address 0x3000 via VDP Register 54
26:     f18a_gpu_start(0x3000);
27:
28:     gotoxy(0, 12);
29:     puts("GPU EXECUTED SUCCESSFULLY!");
30:
31:     while (1) {
32:     }
33:
34:     return 0;
35: }
```

Line-by-Line Breakdown: Host Controller

Lines 1-5	**Headers:** Includes standard I/O and the auto-generated `gpu_fx_map.h` header, which defines symbols for the compiled GPU coprocessor image.
Lines 7-9	**F18A Library Prototypes:** Declares `f18a_unlock()` and `f18a_gpu_start()` from Tidepod's hardware abstraction library.
Lines 16-17	**`f18a_unlock();`:** The F18A starts in strict TMS9918A compatibility mode. To enable the GPU and enhanced registers 16–63, the software must write the 2-byte magic unlock sequence (`0x55`, `0x55`) to VDP Register 57.
Lines 19-20	**`rom_to_vram(unwrap(gpu_fx), 0x3000);`:** Transfers the compiled GPU machine code from the cartridge ROM bank into VRAM starting at address `0x3000`.
Lines 25-26	**`f18a_gpu_start(0x3000);`:** Writes the target execution address (`0x3000`) to VDP Registers 54 and 55. This activates the 100MHz GPU coprocessor clock, which instantly begins executing `gpu_main()`!
Lines 31-35	**Host Idle Loop:** Keeps the TMS9900 CPU active while the GPU runs in hardware.

Tutorial 6: Disk I/O & Standalone EA5 Programs

File Operations

> **Draft — being rewritten**
>
> This tutorial's code has not yet been verified against the current Tidepod toolchain and contains API calls that do not exist in libc. Use the API reference and the examples shipped in `resources/examples/` until this page is updated.

Tidepod includes a fully compliant C `stdio` subsystem supporting standard file operations on TI floppy sector disk images (`.dsk`) and hard disks (`.hd`). It also supports building standalone binary programs executable via the classic Editor/Assembler Option 5 menu without requiring a cartridge ROM.

TI Disk Architecture & Path Notation

TI-99/4A file access uses disk controller DSR (Device Service Routine) protocols. File paths are specified using standard TI notation:

- `"DSK1.SCORES"`: Refers to file `SCORES` on Floppy Drive 1.
- `"DSK2.LEVELDATA"`: Refers to file `LEVELDATA` on Floppy Drive 2.
- Tidepod's C runtime automatically converts standard POSIX stream calls (`fopen`, `fprintf`, `fgets`, `fclose`) into TI sector PAB (Peripheral Access Block) requests.

Complete Code Listing (`src/main.c`)

src/main.c

```
 1: #include <stdio.h>
 2: #include <string.h>
 3: #include <conio.h>
 4:
 5: // Static I/O buffer in BSS to prevent stack overflow!
 6: static char s_io_buffer[128];
 7:
 8: int main(void) {
 9:     ti_text_mode(40);
10:     clrscr();
11:     puts("TI-99/4A DISK I/O DEMONSTRATION");
12:
13:     // 1. Open file for writing on Floppy Drive 1
14:     FILE *fp = fopen("DSK1.SCORES", "w");
15:     if (!fp) {
16:         puts("ERROR: CANNOT CREATE DSK1.SCORES");
17:         while (1) {}
18:     }
19:
20:     // 2. Write high score records to disk
21:     fprintf(fp, "DANNY:98500
");
22:     fprintf(fp, "RETRO:82100
");
23:     fclose(fp);
24:     puts("HIGH SCORES SAVED SUCCESSFULLY.");
25:
26:     // 3. Re-open file for sequential reading
27:     fp = fopen("DSK1.SCORES", "r");
28:     if (!fp) {
29:         puts("ERROR: CANNOT READ DSK1.SCORES");
30:         while (1) {}
31:     }
32:
33:     // 4. Stream and print records line by line
34:     int line_num = 1;
35:     while (fgets(s_io_buffer, sizeof(s_io_buffer), fp)) {
```

```
36:         gotoxy(2, 6 + line_num);
37:         printf("RECORD %d: %s", line_num, s_io_buffer);
38:         line_num++;
39:     }
40:     fclose(fp);
41:
42:     gotoxy(2, 11);
43:     puts("DISK I/O OPERATION COMPLETED OK.");
44:
45:     while (1) {
46:     }
47:
48:     return 0;
49: }
```

Line-by-Line Code Breakdown

Lines 1-3 — **Headers:** Includes standard C declarations for `stdio.h` (file streams), `string.h`, and `conio.h`.

Lines 5-6 — **Static Line Buffer:** Allocates a 128-byte line buffer `s_io_buffer` in the `.bss` segment. Maintaining file I/O buffers in static memory is essential to prevent C workspace stack overflows.

Lines 8-11 — **Screen Initialization:** Configures 40-column text mode, clears the screen, and outputs the title header.

Lines 14-18 — **`fopen("DSK1.SCORES", "w");`:** Instructs the TI disk controller DSR to allocate a file descriptor record (FDR) on disk drive 1 and open a sequential write stream. If the drive is unmounted or write-protected, `fopen` returns `NULL`, which is caught and reported.

Lines 21-24 — **`fprintf` & `fclose`:** Writes two formatted string records into the disk sector buffer. Calling `fclose(fp)` flushes remaining dirty sectors to floppy media,

updates the catalog directory track, and releases the DSR Peripheral Access Block.

Lines 27-31 — **`fopen("DSK1.SCORES", "r");`:** Re-opens the newly written file in read-only sequential mode.

Lines 34-40 — **`fgets(s_io_buffer, sizeof(s_io_buffer), fp);`:** Reads records from disk until encountering EOF (End of File). Each line is printed to the screen using `printf` at incrementing row coordinates. Calling `fclose(fp)` cleanly closes the stream.

Lines 42-49 — **Completion & Idle Loop:** Displays completion banner and enters safe bare-metal halt loop.

Building Standalone EA5 Disks

If you wish to distribute your application as a disk-based binary that users can launch from the TI Editor/Assembler cartridge (Option 5: RUN PROGRAM FILE):

Terminal

```
tidepod new FileTool --type ea5 --path ./FileTool
cd FileTool && make
```

This links your code with an origin at `0xA000` in Expansion RAM and packages the resulting binary onto an autorun floppy image (`FileTool.dsk`).

The build pipeline

`make · make debug`

Tidepod generates a plain GNU `Makefile` from the `.project` file. The IDE's Build command runs exactly this Makefile, so `make` in a terminal gives the same result. This page follows a banked cartridge build step by step; the other cartridge types use the same steps with a different start-up object and linker template.

Targets

Command	Result
`make`	Release build: `<name>.bin` and `<name>.rpk`.
`make debug`	Debug build: the same plus `<name>.debug` (an ELF file with DWARF debug info), which is also packed into the `.rpk` for the debugger.
`make clean`	Removes objects, packer output and build results.

Switching between `make` and `make debug` deletes the project's objects first (the mode is remembered in `.build_mode`), so you never link a mix of release and debug objects.

What happens during `make`

Each step prints a short tag:

make output (banked project with one resource)

```
[RES] 1 resources (2 KB, 1 banks)
[AS] cart_header.asm
[CC] main.c
[BK] Running tms9900-banker ...
[AS] autobank_trampolines.s
[LD] hello.bin
[PAD] hello.bin padded to 16KB
[ZP] hello.rpk
```

Tag	Step
`[RES]`	Each `.dat` resource is wrapped into an object file, and the packer computes where every resource will live. The result is `include/resource_defs.h`, which your C code includes to find resources. See Binary resources.
`[CC]` / `[AS]`	Each C file is compiled with `tms9900-elf-gcc`, each `.asm` file assembled with `tms9900-elf-as`. Every source file (translation unit) becomes one object.
`[BK]`	`tms9900-packer` reads all objects plus `libc.a`, measures each translation unit, builds the call graph and distributes the units over 8 KB banks. It writes the linker script and the cross-bank call stubs. See Banking and the packer.
`[AS]` trampolines	The generated `autobank_trampolines.s` is assembled.
`[LD]`	`tms9900-elf-ld` links everything with the generated `autobank.ld`. The link map is written to `link.map`.
`[PAD]`	The ROM image is padded to the next power of two (8, 16, 32 KB, ...) so cartridge boards and emulators accept it.
`[ZP]`	The `.rpk` is a zip of `layout.xml`, the `.bin`, the `.debug` file (debug builds) and any disk images in the project directory.

Compiler flags

Build	Flags
Release	`-std=c99 -Werror -Wall -Os -s -fno-function-sections -fno-data-sections -Iinclude -I<toolchain>/include`
Debug	`-std=c99 -Werror -Wall -O0 -g -gdwarf-2 -fno-function-sections -fno-data-sections -Iinclude -I<toolchain>/include`

The optimisation level comes from `optLevelRelease` / `optLevelDebug`, include paths and `customCompilerFlags` are appended. Note `-Werror`: every warning stops the build. Files

placed in persistent RAM or scratchpad additionally get `-mpersistent` or `-mscratchpad` (see Code placement).

Generated files

Do not edit these; they are rewritten by `tidepod sync` or by every build.

File	Written by	Contents
`Makefile`	sync	The build. Starts with `# [GENERATED BY TI-99 IDE - DO NOT EDIT MANUALLY]`.
`layout.xml`	sync	Cartridge board description for the `.rpk`.
`.cart_header.asm`	sync	Cartridge header with the menu name.
`include/resource_defs.h`	build	Bank, offset and length of every resource.
`autobank.ld`	build	Linker script with the bank assignment.
`autobank_trampolines.s`, `autobank_wrap.txt`	build	Cross-bank call stubs and the matching `--wrap` linker options.
`autobank_mapping.h`	build	Which function ended up in which bank.
`bank_report.json`	build	Machine-readable layout report, see Reading bank_report.json.
`link.map`	build	Linker map.
`include/<prefix>_map.h`	build	Addresses of an F18A GPU image, see F18A GPU code.

Your own build steps go in `Makefile.custom`, which is included by the Makefile and never overwritten. See Makefile.custom.

Things to know

“resource_defs.h changed during packing … Re-run make.”

The packer found that resource positions differ from what the objects were compiled against (always the case on the first build of a project with resources, and whenever resources change size). It updates `include/resource_defs.h` and stops. Run `make` again.

Header changes are not tracked

The `%.o: %.c` rule has no header dependencies, and `make clean` does not delete the start-up objects. After changing a header, rebuild everything: `make clean && rm -f crt0_*.o include/resource_defs.h .cart_header.o && make`

EA5 projects

EA5 projects link a single ELF and run `tms9900-packer --ea5`, which slices it into Editor/Assembler option 5 program files (`NAME1`, `NAME2`, …) and writes them to `<name>.dsk`. There is no bank packing.

Banking and the packer

`tms9900-packer`

A cartridge sees ROM through one 8 KB window at `0x6000–0x7FFF`. Larger cartridges are split into 8 KB *banks*; writing to an address in the window selects which bank is visible. Tidepod hides this from you: you write normal C, and `tms9900-packer` decides which bank every source file goes into and generates the glue for calls between banks.

The unit of packing is the source file

The packer places whole **translation units** (one `.c` or `.asm` file = one object). All functions of one file end up in the same bank. Consequences:

- No single source file may be larger than one bank (8 KB of code and read-only data). Split big files.
- Calls between functions in the same file are always direct and cost nothing extra.
- Keep functions that call each other a lot in the same file, or put the hot one in persistent RAM.

Memory layout of a banked cartridge

Region	Address	What the toolchain puts there
Cartridge ROM window	`0x6000–0x7FFF`	The current bank: header and start-up code (bank 0 only), your code (“auto” units), resources.
Low expansion RAM	`0x2000–0x3FFF`	“Persistent” code: C runtime, bank-switch stubs, files you mark persistent.
High expansion RAM	`0xA000–0xFFFF`	`.data`, `.bss`, persistent overflow, and the stack (top of RAM, growing down).
Scratchpad RAM	`0x8300–0x83FF`	Registers/workspace, and code you mark scratchpad (from `0x8342` by default).

Start-up

When the console starts a banked cartridge:

1. `_start` (bank 0) disables interrupts and copies a small copier routine into scratchpad.
2. The copier copies the persistent section to `0x2000`, `.data` to high RAM and the scratchpad section over itself. It can read from any bank, so these sections can sit at the end of the ROM.
3. `_persistent_init` clears `.bss`, enables interrupts and calls `main()`.

Cross-bank calls (trampolines)

When a function in bank 2 calls a function in bank 5, the call is redirected (with the linker's `--wrap`) to a small stub in persistent RAM. The stub saves the current bank, switches to bank 5, calls the function, switches back and returns. You do not write anything for this, but it costs a few dozen cycles per call. The number of stubs is reported as `summary.total_trampolines` in `bank_report.json`.

> **Function pointers are not wrapped**
>
> Only direct calls get a trampoline. If you take the address of a function and call it through a pointer from another bank, the call jumps into whatever bank is currently mapped. Put functions that are called through pointers (callbacks, state-machine tables) in persistent RAM, or call them only from their own file.

Strings and constants across banks

A pointer to read-only data in bank 2 is useless once bank 5 is mapped. For every file that calls into another bank, the packer therefore moves string literals and small constants (`.rodata.str*`, `.rodata.cst*` and plain `.rodata` up to 128 bytes) into always-mapped RAM. So `printf("score %d", s)` from banked code just works. Large `const` tables stay in their bank: don't pass pointers to them into functions in other files — copy the data first, or put the table in a persistent file.

Switching banks yourself

Normally you never do this. Banked projects get `include/bankswitch.h`, which defines `switchtobank(n)` (a write to `0x6000 + (n << 1)`) and declares `currentbank`, the variable the trampolines keep up to date. The resource copy functions use it to restore the bank after reading from another one. If you switch manually, only do it from persistent or scratchpad code and switch back before returning.

Packing algorithms

packing_algo	Packer option	Strategy
0 (default)	`bfd`	Best-fit decreasing: largest units first, each into the fullest bank it fits. Fewest banks.
1	`ffd`	First-fit decreasing: each unit into the first bank with room.
2	`cluster`	Groups units that call each other into the same bank to reduce trampolines.

Change it in Project Settings or in the `.project` file and compare `total_banks` / `total_trampolines`.

Packer options

The Makefile passes the right options; you can add more through `PACKER_EXTRA_FLAGS` in `Makefile.custom`. For reference (`tms9900-packer --help`):

Option	Meaning
`--bank-size N`	Bank capacity in bytes (default 8192).
`--bank-base N`	Physical cartridge bank of logical bank 0 (default 0); for images merged behind other ROM banks.
`--heap-size N`	Reserve RAM for dynamic use.
`--stack-size N`	Override the stack size computed from the call graph.
`--no-32k`	No 32K expansion: data and bss in scratchpad (used by 8K projects).
`--flat-rom`	Unbanked ROM, everything in bank 0.
`--persistent-lib P`	Put a whole static library in persistent RAM (what `libFiles[].forcePersistent` does).
`--scratchpad-base H`, `--scratchpad-size N`	Scratchpad area for scratchpad code (default `0x8342` to `0x8400`).
`--packing-algo`	`bfd`, `ffd` or `cluster`.
`--ea5 ELF BASE`	EA5 mode: slice a linked ELF into program files and a disk image.
`-v`	Verbose output.

Code placement: banked, persistent, scratchpad

Performance

By default every C file is *banked*: it lives in cartridge ROM and is reached through the bank window. You can move a file to RAM instead. That costs RAM but removes cross-bank call overhead, and scratchpad RAM is also the fastest memory in the machine (16-bit, no wait states).

Placement	Where	Use for	Cost
Banked (default)	Cartridge ROM bank	Everything else	Trampoline on calls from other banks
Persistent	Expansion RAM (`0x2000` region first, then high RAM)	Code called from many banks, callbacks, interrupt handlers, code called through function pointers	32K expansion RAM; copied from ROM at start-up
Scratchpad	Console scratchpad (`0x8342` – `0x83FF` by default)	A few small, very hot routines (inner loops, blitters)	Fewer than 200 bytes in total

Marking a file

From the command line:

Terminal

```
tidepod add src/isr.c --persistent
tidepod add src/blit.c --scratchpad
```

Or inside the file, before the first line of code:

src/blit.c

```
/* Inner loop for the tile renderer */
#pragma bank scratchpad
#include "blit.h"

void blit_row(...)
{
    ...
}
```

Tidepod looks for `#pragma bank persistent` or `#pragma bank scratchpad` only in the leading block of comments and preprocessor lines; a pragma after the first line of code is ignored. Run `tidepod sync` (or save Project Settings) after adding one. Either way, the Makefile compiles the file with `-mpersistent` or `-mscratchpad`, which tells the compiler to emit its code into the matching section.

> **8K projects**
>
> Persistent and scratchpad placement need the 32K expansion and are ignored for the 8K type (type 0). Use “8K Cart + 32K RAM” or banked.

Libraries

A static library listed in `libFiles` with `"forcePersistent": true` is placed in persistent RAM as a whole (`--persistent-lib`). Useful for a sound or music player that is called every frame from everywhere.

Scratchpad area

Scratchpad code is placed from `scratchpadBase` up to `0x8400` (or `scratchpadSize` bytes). The default base `0x8342` follows the C workspace. Project Settings has presets:

Preset	Base	When
Full	`0x8320`	Maximum space for your code.
Music libraries	`0x8340`	Leaves room for music players that use low scratchpad.
Disk I/O	`0x8380`	Programs that use disk files: the disk system uses the area below `0x8380`.
Custom	any ≥ `0x8320`	Your own value.

> **Top of scratchpad**
>
> `0x83C0-0x83FF` is used by the console's interrupt routine. Code placed there is only safe if your program keeps interrupts disabled and never returns to the console.

Checking the result

`bank_report.json` lists every placed unit with its classification (`persistent`, `persistent_high`, `scratchpad`, `auto`, ...) and the free space per region. Check `regions.*.free` after moving files: a region that is nearly full leaves no room for the stack and `.bss`.

Binary resources (.dat files)

`resource_copy.h`

Graphics, fonts, level data, music: anything that is just bytes can be added to a cartridge as a *resource*. Resources are stored in ROM banks after your code and copied to VRAM or RAM when you need them. They cost no RAM until you load them.

1. Add the file

Give the file a `.dat` extension and a base name that is a valid C identifier, then add it:

Terminal

```
tidepod add resources/font_5x8.dat
make          # run it twice if it asks you to (first build after adding a
resource)
```

The Makefile turns the file into an object (`ld -r -b binary`, renamed to a `.resource` section), and the packer assigns it a place in ROM.

2. The generated header

The packer writes three constants per resource, named after the file, into `include/resource_defs.h`:

include/resource_defs.h (generated)

```
#define height_bank   0
#define height_offset 0
#define height_length 6144
#define gpu_bank   0
#define gpu_offset 6144
#define gpu_length 564
```

Treat bank and offset as opaque values for the copy functions. `_length` is the file size in bytes.

3. Copy it where you need it

`resource_copy.h` (in the toolchain include directory) provides:

resource_copy.h

```
#define unwrap(x) x##_bank, x##_offset, x##_length

void rom_to_vram(unsigned int srcbank, unsigned int srcoffset,
                 unsigned int length, unsigned int dst);
void rom_to_ram(unsigned int srcbank, unsigned int srcoffset,
                unsigned int length, unsigned char *dst);

typedef void (*resource_copy_hook_t)(unsigned int bytes_copied, unsigned int
total_bytes);

void rom_to_vram_ex(unsigned int srcbank, unsigned int srcoffset, unsigned int
length,
                    unsigned int dst, unsigned int chunk_size,
resource_copy_hook_t hook);
void rom_to_ram_ex(unsigned int srcbank, unsigned int srcoffset, unsigned int
length,
                   unsigned char *dst, unsigned int chunk_size,
resource_copy_hook_t hook);
```

- `unwrap(name)` expands to the three constants, so the first three arguments are always written as `unwrap(name)`.
- `rom_to_vram` copies to a VRAM address; `rom_to_ram` to a CPU RAM buffer.
- The `_ex` variants copy in chunks of `chunk_size` bytes and call `hook` with progress, e.g. to draw a loading bar or keep music playing during a long copy.
- The functions switch banks to read the data and restore `currentbank` afterwards, so they can be called from any bank.

Example, from the `f18a_bump` sample:

src/main.c

```
#include "resource_copy.h"
#include "resource_defs.h"

    rom_to_vram(unwrap(height), HEIGHT_BASE);   /* resources/height.dat → VRAM
0x0C00 */
    rom_to_vram(unwrap(gpu), gpu_base);          /* the GPU program image → VRAM
0x3000 */
```

Producing .dat files

Tidepod does not convert images or audio. Produce raw bytes in the format your code expects (pattern tables, colour tables, sound lists, ...) with your own tools, and add the result. A `Makefile.custom` rule can regenerate a `.dat` from its source automatically; see Makefile.custom.

> **Never edit RESOURCE_OBJS**
>
> The resource list in the Makefile comes from the project. Use `tidepod add` / `tidepod remove`; hand edits are lost on the next sync.

F18A GPU code in C

`f18a_gpu.h · f18a_gpu_host.h`

The F18A (and the Pico9918) video replacements contain their own TMS9900-compatible processor, the GPU, which runs from VRAM in parallel with the console CPU. Tidepod can compile a C file for the GPU, pack the resulting program as a resource, and give the host code the addresses it needs to load and start it.

The canonical, complete example is `resources/examples/f18a_bump` in the Tidepod install. The snippets on this page come from it.

1. Add the GPU source

Terminal

```
tidepod add src/gpu.c --gpu --gpu-base 0x3000 --gpu-stack 0x3F00
```

- `--gpu-base` is the VRAM address the program is linked for and loaded at; it is also the entry point.
- `--gpu-stack` is the initial GPU stack pointer (VRAM address; the stack grows down).
- Both are required. Alternatively put `#pragma bank gpu`, `#pragma gpu base 0x3000` and `#pragma gpu stack 0x3F00` in the file.
- Choose addresses that don't overlap the VRAM tables your host program uses.

2. What the build does with it

A GPU file is not part of the cartridge code. The Makefile:

1. compiles it with `-ffreestanding -nostdlib -fno-builtin -fomit-frame-pointer -ffixed-r15 -Os` (`[GPU-CC]`),
2. links it with the GPU start-up code `crt0_gpu.o` at the base address (`[GPU-LD]`),
3. turns it into a binary `src/gpu.dat` plus a header `include/gpu_map.h` with `gpu-image-map` (`[GPU-BIN]`),
4. packs `gpu.dat` as a normal resource named `gpu`.

include/gpu_map.h (generated)

```
#define gpu_base  0x3000u
#define gpu_entry 0x3000u  /* _gpu_start */
#define gpu_stack 0x3F00u
#define gpu_size  564u
```

The prefix is the file's base name: `src/fx.c` gives `include/fx_map.h`, `fx_base`, `fx_entry`, and the resource `fx`.

3. The GPU side

GPU code includes `<f18a_gpu.h>` and nothing from libc. A minimal frame loop:

src/gpu.c (shortened)

```
#include <f18a_gpu.h>
#include "layout.h"           /* VRAM addresses shared with the host */

void main(void) __attribute__((noreturn));

void main(void)
{
    for (;;) {
        F18A_GPU_IDLE();      /* sleep until the host-configured trigger (VSYNC)
*/

        int lx = (signed char)F18A_GPU_MMIO8(LIGHT_LX);   /* read a value the
host wrote */
        ...
        /* write pixels straight into VRAM through pointers */
    }
}
```

f18a_gpu.h	Meaning
`F18A_GPU_IDLE()`	Executes `IDLE`: the GPU sleeps until the trigger set with VR50 by the host (e.g. VSYNC). Use this to pace the GPU instead of polling the blanking register.
`F18A_GPU_MMIO8(addr)`, `F18A_GPU_MMIO16(addr)`	Volatile byte / word access to a VRAM or register address.
`F18A_GPU_VRAM_BASE` `0x0000`	VRAM as seen by the GPU.
`F18A_GPU_GRAM_BASE` `0x4000`	2 KB GPU-only RAM on the classic F18A.
`F18A_GPU_WS_BASE` `0x4800`	Hardware workspace.
`F18A_GPU_SCANLINE` `0x7000`, `F18A_GPU_BLANKING` `0x7001`	Status registers. Prefer `F18A_GPU_IDLE()` over polling `0x7001`.
`F18A_ECM3_P0/P1/P2`	ECM3 pattern-plane offsets from the pattern table base.

GPU rules

The GPU's workspace pointer is fixed by hardware: never use `LWPI` in GPU code (including inline assembly). R15 is reserved (the build passes `-ffixed-r15`); don't use it in assembly either. The GPU program has no libc: no `printf`, no `memcpy`.

4. The host side

src/main.c (excerpt)

```
#include "vdp.h"               /* project-local VDP helpers (copied from the
examples) */
#include "f18a_gpu_host.h"
#include "resource_copy.h"
#include "resource_defs.h"
#include "gpu_map.h"

void f18a_unlock(void)
{
    VDP_SET_REGISTER(57, 0x1C);   /* writing 0x1C to VR57 twice unlocks the F18A
registers */
    VDP_SET_REGISTER(57, 0x1C);
}

int main(void)
{
    f18a_unlock();
    ...
    rom_to_vram(unwrap(gpu), gpu_base);   /* copy the GPU program into VRAM */
    VDP_SET_REGISTER(50, 0x30);           /* trigger setting used by f18a_bump
for its F18A_GPU_IDLE() loop */
    f18a_gpu_start(gpu_entry);            /* set the GPU PC and start it */
    for (;;) { ... }
}
```

f18a_gpu_host.h	Meaning
`f18a_gpu_start(entry_pc)`	Sets the GPU program counter (VR54/VR55) and starts it.
`f18a_gpu_stop()`	Stops the GPU (clears the run bit in VR56).
`f18a_gpu_load_at(vram_addr, src, len)`	Copies a program from a CPU-RAM buffer to VRAM.
`f18a_gpu_load_and_start(vram_addr, src, len, entry_pc)`	Both in one call.

With a packed GPU image you normally use `rom_to_vram(unwrap(gpu), gpu_base)` followed by `f18a_gpu_start(gpu_entry)`; the `load` functions are for programs you already have in RAM.

> **Host checklist**
>
> Unlock the F18A before writing registers above VR7. After unlocking, also set VR30 (sprites per scanline): its reset value is 4, not 32. Load GPU programs into normal VRAM (below `0x4000`), not into GPU RAM. Detect the chip first (`ti_vdp_detect`) if your program must also run on a stock 9918A.

Running it

Run with an F18A-capable VDP: `ti994a-emu --cart game.rpk --f18a`, or select `tidepod-f18a` / `pico-f18a` as VDP in Preferences or in the project's `emu` settings.

Extending the build with Makefile.custom

`Extension point`

The generated `Makefile` is rewritten whenever the project changes, so you never edit it. Everything you want to add goes into `Makefile.custom` in the project directory. Tidepod creates it once (with a comment header) and never touches it again.

How it is included

The Makefile includes `Makefile.custom` *after* it has defined its variables (`OBJECT_LIST`, `RESOURCE_OBJS`, `CCFLAGS`, ...) and `.DEFAULT_GOAL := all`, but *before* the `all` and `debug` recipes. So you can:

- read and change variables — the recipes use your values;
- add your own rules and targets without changing what plain `make` builds;
- add prerequisites to existing objects.

Available variables include `$(NAME)`, `$(CC)`, `$(AS)`, `$(LD)`, `$(OBJCOPY)`, `$(CCFLAGS)`, `$(CCFLAGS_DBG)`, `$(OBJECT_LIST)`, `$(RESOURCE_OBJS)`, `$(STARTUP_OBJS)`, `$(GCC_TOOLS)` (toolchain `bin`), `$(TOOLCHAIN_INC)` and `$(TOOLCHAIN_LIB)`.

Hooks

Variable	Effect
`POST_BUILD`	A shell command run at the very end of `all` and `debug`, after the `.rpk` is zipped. Use it for post-processing instead of overriding the targets.
`PACKER_EXTRA_FLAGS`	Extra packer options, e.g. `--stack-size 1280` or `--bank-base 2`.
`MAKEFILE_CUSTOM_POST := 1`	Includes `Makefile.custom` a second time at the end of the Makefile, after the stock recipes, for the rare case where you need to see or wrap them.

Recipes

Track header dependencies. The generated `%.o: %.c` rule doesn't know about headers. List them yourself so objects rebuild when a header changes:

Makefile.custom

```
main.o player.o level.o: game.h
player.o enemy.o: sprites.h
```

Generate a resource from its source. If `resources/tiles.dat` is in the project, make it depend on the file it is converted from:

Makefile.custom

```
resources/tiles.dat: art/tiles.png tools/png2pat.py
	python3 tools/png2pat.py $< $@
```

The Makefile's own `%.o: %.dat` rule then picks up the new `.dat`.

Copy the result somewhere after each build.

Makefile.custom

```
POST_BUILD = cp $(NAME).rpk ~/Dropbox/ti99/
```

Raise the stack estimate. The packer computes the stack size from the call graph; recursion and calls through pointers are invisible to it.

Makefile.custom

```
PACKER_EXTRA_FLAGS := --stack-size 1024
```

A large real-world example

`resources/examples/fileman/Makefile.custom` adds header dependencies, reorders resources, uses its own start-up objects, checks GPU image sizes and merges the cartridge with another ROM in `POST_BUILD`.

Don't re-implement the build

Avoid redefining `all` or `debug` (make will warn about overriding recipes, and the IDE relies on them). Add sources with `tidepod add` rather than appending to `OBJECT_LIST`, so the packer, the IDE and the debugger all know about them.

Reading bank_report.json

Packer report

Every build writes `bank_report.json`: how full each bank and RAM region is, where each object went, and which calls cross banks. Comparing it with the report of the last good build catches most layout problems before you test.

Top-level keys

Key	Contents
`summary`	`total_banks`, `total_trampolines`.
`ram_overhead`	`data_bytes`, `bss_bytes`, `max_stack_depth` (static estimate in bytes), `stack_vma` (top of stack, normally `65534` = `0xFFFE`).
`regions`	`persistent_lower` (the `0x2000` RAM) and `persistent_high` (the `0xA000` RAM), each with `capacity`, `used`, `free`.
`ram_items`	Items placed in RAM: `filename`, `classification`, `size`, `lma` (ROM address), `vma` (run address).
`banks`	One entry per bank: `bank_id`, `capacity`, `used`, and `items` (same fields as above plus `hotness`).
`call_edges`	The call graph: `caller`, `callee`, `weight`, `caller_tu`, `callee_tu`, `caller_bank`, `callee_bank`, `is_trampoline`.
`trampolines`	One entry per cross-bank stub: `symbol`, `target_bank`.

Classifications

Value	Meaning
`pinned`	Must be at the start of bank 0 (cartridge header, `_start`).
`boot_copier`	The start-up copier (runs once from scratchpad).
`auto`	Normal banked code, placed by the packer.
`persistent`, `persistent_high`	Code/data in low or high expansion RAM.
`scratchpad`	Code in scratchpad RAM.
`resource`	A `.dat` resource.
`system_data`, `system_bss`, `stack`, `wp_reserved`	RAM reservations for `.data`, `.bss`, the stack and the register workspace.

The four numbers to watch

1. **`regions.persistent_high.free`** / **`persistent_lower.free`** — RAM headroom. When high RAM gets tight, the stack (which grows down from the top) runs into `.bss`.
2. **`ram_overhead.max_stack_depth`** — computed from the call graph. Recursion and calls through function pointers are not included, so the real depth can be larger. Raise it with `--stack-size` if needed (Makefile.custom).
3. **`summary.total_trampolines`** — a sudden jump means new cross-bank calls, often in a hot path.
4. **Bank of each object** — code that silently moved to another bank can break function pointers (see Banking).

Quick queries

With `jq`:

Terminal

```
# Overview
jq '{summary, ram_overhead, regions}' bank_report.json

# Fill level per bank
jq -r '.banks[] | "\(.bank_id)\t\(.used)/\(.capacity)"' bank_report.json

# Which bank each of my objects is in
jq -r '.banks[] | .bank_id as $b | .items[] | select(.classification=="auto") |
"\($b)\t\(.filename)\t\(.size)"' bank_report.json

# All cross-bank calls
jq -r '.call_edges[] | select(.is_trampoline) | "\(.caller) (\(.caller_bank)) →
\(.callee) (\(.callee_bank))"' bank_report.json
```

Keep a baseline

Commit `bank_report.json` (or copy it next to a release) so you can `diff` it against the next build.

Tidepod command line

`tidepod <command>`

The `tidepod` executable is both the IDE and a command-line tool. With a command it works on a project and exits; without one (or with a `.project` path) it opens the IDE.

Usage

```
tidepod <command> [options]
tidepod [project.project]          # open the IDE
tidepod <command> --help           # options of one command
```

Commands that work on an existing project find it through `--project <file>`, a `*.project` argument, or the single `.project` file in the current directory.

Command	What it does
`new`	Create a project directory with boilerplate.
`add`	Add files to the build and regenerate the Makefile.
`remove`	Remove files from the build (the files stay on disk).
`list`	Show project info and the file list with their flags.
`set-type`	Change the project type and regenerate the Makefile.
`sync`	Regenerate the Makefile (and `layout.xml`, cartridge header) from the `.project` file.
`run`	Start the stand-alone emulator with the project's cartridge.

There is no `build` command: run `make` or `make debug` in the project directory.

tidepod new

Synopsis

```
tidepod new <name> [--type 8k|banked|xb|ea5|custom] [--menu <name>] [--path
<dir>]
```

Option	Default	Meaning
`--type`	`banked`	Project type.
`--menu`	name in upper case	Text in the console's cartridge menu.
`--path`	`./<name>`	Project directory (created if needed).

Creates `<name>.project`, a starter `main.c` (`main.bas` for XB), the cartridge header, the Makefile, `Makefile.custom`, `layout.xml`, and installs the agent skill under `.agents/skills/` and `.cursor/skills/`.

tidepod add

Synopsis

```
tidepod add <file...> [--persistent | --scratchpad | --gpu --gpu-base HEX --gpu-
stack HEX] [--project <file>]
```

Accepts `.c`, `.asm`, `.S` and `.dat` resources. Paths are relative to the project directory.

Option	Meaning
`--persistent`	Place the file in persistent RAM. See Code placement.
`--scratchpad`	Place the file in scratchpad RAM.
`--gpu`	Build the file as an F18A GPU image. Requires `--gpu-base` (VRAM link/entry address) and `--gpu-stack` (stack top, grows down). See F18A GPU code.
`--project`	Explicit project file.

Examples

```
tidepod add src/player.c src/enemy.c
tidepod add src/isr.c --persistent
tidepod add resources/tiles.dat
tidepod add src/gpu.c --gpu --gpu-base 0x3000 --gpu-stack 0x3F00
```

tidepod remove

Synopsis

```
tidepod remove <file...> [--project <file>]
```

Removes the files from the build list and regenerates the Makefile. Nothing is deleted from disk.

tidepod list

Prints the project name, type, and every build member with its placement flags.

tidepod set-type

Synopsis

```
tidepod set-type 8k|banked|xb|ea5|custom [--project <file>]
```

tidepod sync

Regenerates the Makefile, `layout.xml` and the cartridge header from the `.project` file, and re-reads `#pragma bank …` / `#pragma gpu …` from the sources. Run it after editing the `.project` file by hand or adding a placement pragma.

tidepod run

Synopsis

```
tidepod run [--debug] [--project <file>]
```

Starts `ti994a-emu` with `<name>.rpk` from the project directory and `--autorun`. The emulator configuration is layered:

1. the machine defaults from Tidepod's `preferences.json` (PEB cards, VDP engine),
2. the project's optional `"emu"` overrides,
3. the project's disk mounts `dsk1`–`dsk3` and `hard1`–`hard3`.

`--debug` also starts the GDB server on port 3333 (see Debugging). `run` does **not** build — run `make` first — and it opens the stand-alone emulator, not the IDE. For headless runs and other options, call `ti994a-emu` directly (The emulator).

The stand-alone emulator (ti994a-emu)

`ti994a-emu`

`ti994a-emu` is the emulator used by the IDE, packaged as a separate program. It ships in the toolchain's `bin` directory next to the compiler. `tidepod run` starts it for you; call it directly for headless runs, screenshots, snapshots and anything else on this page.

Common invocations

Terminal

```
ti994a-emu --cart game.rpk                          # boot to the title screen,
pick the cart yourself
ti994a-emu --cart game.rpk --autorun                # start the cartridge directly
ti994a-emu --cart game.rpk --f18a                   # with an F18A
ti994a-emu --cart game.rpk --dsk1 data.dsk          # with a disk in DSK1
ti994a-emu --cart game.rpk --gdb                    # with the GDB server on :3333
```

Cartridge and machine state

Option	Meaning
`--cart <file>`	Load a cartridge: `.rpk` (recommended, includes the board type and debug info) or raw `.bin`.
`--carts <list>`	Comma-separated multi-file cartridge (C/D/G images).
`--autorun`	Skip the title screen and menu and start the cartridge immediately.
`--save-state <path>`	Save a snapshot on exit (use with `--frames`).
`--load-state <path>`	Restore a snapshot after loading the cartridge.
`--pause-at <N>`	Pause when frame N is reached.
`--step-n <K>`	When paused, execute K instructions.
`--print-config`	Print the resolved VDP, cartridge and disk configuration and exit.
`--version`, `-h` / `--help`	Version / option list.

> **`--autorun` is a shortcut, not a test**
>
> `--autorun` jumps into the cartridge without going through the console's title screen and menu, a path real users never take. A program can behave differently that way. To check that a build really boots, start normally and select it with key injection; see Headless testing.

Video

`--vdp engine`	Emulates
`tidepod-tms9918a`	The stock TMS9918A (default).
`tidepod-f18a`	F18A (also selected by `--f18a`).
`pico-f18a`	F18A, Pico9918-based core.
`pico-v9938`	V9938 (Pico9918-based core).
`pico-pico9918`	Pico9918 PRO.

Disks and peripherals

`--dsk1` … `--dsk4`, `--hard1` … `--hard3`, `--peb-card`, `--peb-none`, `--dsr` and `--fast-disk` are described in Disks & peripherals.

Automation and debugging

Option	Meaning
`--headless`	No window.
`--frames <N>`	Run N frames (60 per second), then exit.
`--keys <script>`, `--key-fifo <path>`	Inject keystrokes. See Headless testing.
`--screenshot-at <list>`, `--screenshot-path <p>`	Save PNG screenshots at the given frames.
`--gdb [port]`	Start the GDB server (default port 3333).
`--debug-dwarf <file>`	Load a `.debug` file for source-level information (not needed for `.rpk` files built with `make debug`, which contain it).
`--dump-regs <N>`	Headless: run N frames, then print CPU registers, interrupt state, scratchpad and VRAM tables.
`-v`, `--verbose`	Log every interrupt with its source (VDP / timer / external).

See Debugging for GDB and `--dump-regs` in practice.

Configuration used by `tidepod run` and the IDE

The emulator itself only knows its command-line options. Tidepod builds them from three layers: the machine defaults in Preferences (stored in `preferences.json`), the optional `"emu"` block in the project, and the project's disk mounts. The `"emu"` block:

mygame.project (excerpt)

```
"emu": {
  "peb": "custom",
  "pebCards": [ { "type": "hfdc", "cru": 4352 } ],
  "vdp": "tidepod-f18a"
}
```

Key	Values
`peb`	`inherit` (default; use Preferences), `none` (empty PEB, disk mounts are skipped), `custom` (use `pebCards` only).
`pebCards`	List of `{"type": "tifdc"|"hfdc"|"whtscsi", "cru": N}`. `cru` is decimal (4352 = `0x1100`, 4608 = `0x1200`).
`vdp`	`inherit` or an engine name from the table above.

The block is omitted from the file when everything is `inherit`.

Disks and peripherals

DSK · HARD · PEB

Programs that read or write files need a disk controller card in the emulated Peripheral Expansion Box (PEB) and a disk image to work on. This page covers how to set that up for the IDE, `tidepod run` and `ti994a-emu`. For the C side (`fopen` and friends) see the API reference.

Disk images in a project

Mount images in the IDE's Disk dialog or **DSK** tab, or put them in the project file. Paths can be relative to the project directory:

mygame.project (excerpt)

```
"dsk1": "disks/game.dsk",
"dsk2": "disks/save.dsk",
"hard1": "disks/work.hd",
```

The IDE mounts them when the project opens and `tidepod run` passes them to the emulator. Disk images (`*.dsk`, `*.hd`, …) lying in the project directory are also zipped into the `.rpk` by the build.

Emulator options

Option	Meaning
`--dsk1` … `--dsk3 <file>`	Mount a sector disk image (`.dsk`).
`--dsk4 <file>`	Fourth floppy drive; needs the HFDC card.
`--hard1` … `--hard3 <file>`	Mount a hard-disk image (HFDC).
`--peb-card <type@cru>`	Add a card: `tifdc@1100`, `hfdc@1100` or `whtscsi@1200`. Repeatable; the first use replaces the default TI FDC.
`--peb-none`	Empty PEB: no disk controller at all.
`--dsr <base>:<file>`	Load a DSR ROM at a CRU base, e.g. `1100:disk.bin`.
`--fast-disk`	Disable realistic disk timing.

Controller cards

Type	Default CRU	Notes
`tifdc`	`1100`	TI floppy disk controller. The default when nothing is configured.
`hfdc`	`1100`	Myarc HFDC: floppies (including DSK4) and hard disks.
`whtscsi`	`1200`	WHT SCSI card; work in progress.

Combinations are explicit: `--peb-card tifdc@1100 --peb-card whtscsi@1200`. Two cards on the same CRU base are emulated as a bus clash, not refused.

Hard-disk images

Hard-disk images must be raw sector dumps (`.hd`, `.hdf`, `.raw`). MAME CHD files are refused at mount time; convert them first:

Terminal

```
chdman extracthd -i disk.chd -o disk.hd
```

`chdman` is part of MAME's tools (Homebrew package `rom-tools`).

Testing without a disk controller

Set `"peb": "none"` in the project's `"emu"` block (or pass `--peb-none`) to check how your program behaves on a console without disks. The `dsk` / `hard` entries stay in the project but are not mounted.

Programs that use disk files

- Use the **Disk I/O** scratchpad preset (`0x8380`) in Project Settings; the disk system uses lower scratchpad. See Code placement.
- By default `fopen` goes through the disk controller's own file routines. Other back-ends are opt-in: see stdio_enable(3).
- After a failed call, stdio_last_error(3) tells you why.
- Don't put sector-sized buffers (256 bytes and up) on the stack; make them `static`. The stack is small.

Debugging

IDE · GDB · logpoints

Debug builds (`make debug`) are compiled with `-O0 -g` and produce `<name>.debug`, an ELF file with DWARF information that is also packed into the `.rpk`. With it you can debug at the C source level in the IDE or with GDB.

In the IDE

1. Click in the editor's gutter to set breakpoints.
2. Press `F5` (Build & Debug).
3. Step with `F10` (over), `F11` (into), `Shift` + `F11` (out), `Shift` + `F10` (one instruction); continue/pause with `F6`.
4. Inspect state in the debug panel: **CPU** (registers, memory), **VDP** (registers, VRAM), **ASM** (disassembly), **BRK** (breakpoint list), **WCH** (watch expressions, structs expand), **GFX** (patterns, colours, sprites).

With GDB

Use the toolchain's `tms9900-elf-gdb`; a stock GDB doesn't know the TMS9900.

Terminal 1

```
make debug
tidepod run --debug                       # emulator window + GDB server on :3333
# or headless:
ti994a-emu --cart mygame.rpk --autorun --headless --gdb 3333
```

Terminal 2

```
tms9900-elf-gdb mygame.debug -ex 'target remote :3333'
(gdb) break main
(gdb) continue
(gdb) next
(gdb) print player
(gdb) info registers
(gdb) bt
```

- After connecting, the emulator is usually stopped at the console's reset vector (`PC=0024`). Type `continue`.
- `x/` examines CPU memory only. VRAM is a separate address space; read it with the IDE's VDP tab or `--dump-regs`.

Logpoints: printf without cost

`debug.h` provides `__dbg_printf`. In a debug build each call compiles to a single 2-byte `NOP`; the format string and the *names* of the arguments go into a separate ELF section in the `.debug` file. When the CPU reaches the NOP, the IDE reads the variables through the debug information, formats the message on the host and prints it in the Output panel. The TI spends no time formatting.

C

```
#include "debug.h"

__dbg_printf("x=%d y=%d", x, y);                        /* up to 8 arguments,
normal printf formats */
__dbg_printf(DBG_RED "hit" DBG_RESET " at %x", addr);  /* ANSI colour helpers */
BRK();                                                  /* soft breakpoint:
pauses the IDE debugger */
```

- Arguments must be variables the debugger can find through the debug information (locals, globals, parameters), not arbitrary expressions.
- Colour and cursor helpers: `DBG_RESET`, `DBG_CLEAR`, `DBG_HOME`, `DBG_GOTO(r,c)`, `DBG_RED` … `DBG_WHITE`, `DBG_BRIGHT_*`, `DBG_BG_*`, `DBG_BOLD`.

- When `NDEBUG` is defined, `__dbg_printf` and `BRK()` expand to nothing. The generated Makefile does not define `NDEBUG` for release builds, so remove or `#ifdef` logpoints you don't want in a release.

Hangs and crashes: `--dump-regs` and `-v`

When a program locks up, run it headless and dump the machine state at a given frame:

Terminal

```
ti994a-emu --headless --cart mygame.rpk -v \
  --keys "WAIT:240 SPACE WAIT:600" \
  --dump-regs 900 --frames 920 2>int.log | tee dump.log
```

- The dump (stdout) contains the CPU registers, interrupt state, interrupt-routine statistics, key scratchpad words, the registers saved at the last interrupt, and the screen and sprite tables from VRAM.
- With `-v`, every interrupt is logged on stderr with its source (VDP, timer, external).
- If PC is inside the console ROM's interrupt routine at dump time, look at the saved registers to see where your program was interrupted.

> **Typical symptoms**
>
> A flood of illegal-opcode messages usually means a jump into the wrong bank (for example a function pointer called across banks, see Banking), a wild pointer, or a stack overflow into `.bss` (check `max_stack_depth` in bank_report.json).

Frame profiler

`profiler.h · PRF tab`

The IDE's **PRF** tab shows how many CPU cycles each part of your frame takes, against the frame budget. You mark the parts in your code with macros from `profiler.h`. The macros only create symbol labels: they add zero bytes and zero cycles to your program.

Instrumenting a main loop

main.c

```
#include "profiler.h"

for (;;) {
    PROF_FRAME_END();                /* once per frame, top of the loop, before
the vblank wait */

    PROF_BEGIN(808080, vblank_wait);  /* grey: idle time */
    wait_for_vblank();                /* your own vsync wait */
    PROF_END(808080, vblank_wait);

    PROF_BEGIN(4FC3F7, input);
    read_input();
    PROF_END(4FC3F7, input);

    PROF_BEGIN(81C784, update);
    update_world();
      PROF_BEGIN(FFB74D, physics);    /* blocks may nest */
      physics();
      PROF_END(FFB74D, physics);
    PROF_END(81C784, update);

    PROF_BEGIN(E57373, render);
    render();
    PROF_END(E57373, render);
}
```

Rules

1. The colour is six hex digits `RRGGBB` without `0x` or `#`.
2. The name is a bare C identifier, not a string: `PROF_BEGIN(FF8800, render)`.
3. Every `PROF_BEGIN(colour, name)` needs a `PROF_END` with the same colour and name.
4. Blocks may nest but must not overlap.
5. `PROF_FRAME_END()` exactly once per game frame. The profiler counts VDP interrupts between two markers to detect the real frame rate.
6. Measure idle time with an ordinary block around your vblank wait.

The profiler reads the labels from the `.debug` file, so profile a `make debug` build (`F5`). Remember that debug builds are unoptimised (`-O0`): absolute numbers are higher than in a release build, but the proportions are still useful. Raise `optLevelDebug` in Project Settings for numbers closer to release.

Frame budget

The budget line comes from the project's `target_fps` (selectable in the PRF tab: 60, 30, 20, 15, 10). One 60 Hz frame is about 50,065 CPU cycles; the budget is (60 / target) × 50,065. At 30 fps you get two frames, about 100k cycles. The tab flags frames where the detected rate falls below the target.

Headless testing and automation

CI · scripts

`ti994a-emu` can run without a window, type on the TI keyboard from a script and save screenshots. That is enough to check in a script or CI job that a build boots and reaches a given screen.

A boot test

Terminal

```
ti994a-emu --headless --cart mygame.rpk \
  --keys "WAIT:240 SPACE WAIT:90 2 WAIT:300" \
  --screenshot-at 600 --screenshot-path /tmp/boot.png --frames 660
```

1. Wait 240 frames (4 s) for the title screen.
2. Press `Space` to get to the cartridge menu, wait 1.5 s.
3. Press `2` to start the second menu entry (your cartridge; entry 1 is TI BASIC), wait 5 s.
4. Save a screenshot at frame 600 and exit at frame 660.

This goes through the same title screen and menu as a real user, which `--autorun` skips.

Key script syntax (`--keys`)

Tokens are separated by spaces or commas.

Token	Meaning
`WAIT:n`	Do nothing for n frames (60 frames = 1 second).
`HOLD:n`	Hold the following keys down for n frames (default 6).
`GAP:n`	Release for n frames between the following keys (default 10).
`ENTER SPACE SHIFT CTRL FCTN EQUAL PERIOD COMMA SLASH SEMI`	Named keys.
`FCTN+E`, `CTRL+SHIFT+X`	Combinations with up to three modifiers.
`2`, `R`, `q`	Any single key on the TI keyboard.

Keys go through the emulated keyboard matrix, exactly as if typed.

Live input (`--key-fifo`)

Terminal

```
mkfifo /tmp/k
ti994a-emu --cart mygame.rpk --key-fifo /tmp/k &
echo "SPACE WAIT:60 2" > /tmp/k
```

Screenshots

- `--screenshot-at 300,600,900` captures several frames; with more than one, the frame number is appended to the file name.
- The default path is `screenshot.png`.
- Images are cropped to the active display. In 80-column mode every scanline is doubled (512×384) so the pixels are square.

Making it a test

- Look for illegal-opcode messages in the output: they mean the program jumped somewhere it shouldn't.
- Compare the screenshot with a known-good reference image.
- `--dump-regs N` gives you register and VRAM state at frame N as text, easy to grep. See Debugging.

> **Merged cartridges**
>
> If your cartridge image is placed behind other ROM banks (packer `--bank-base`), the bank numbers the emulator reports are offset from the ones in `bank_report.json`.

Troubleshooting

FAQ

Symptoms you are likely to run into, and what to do about them.

Build

Symptom	Cause and fix
`[!!] resource_defs.h changed during packing … Re-run make.`	Normal on the first build and after resources change size. Run `make` again. See Build pipeline.
A warning stops the build	Builds use `-Werror`. Fix the warning.
A header change has no effect	Objects don't depend on headers. Clean rebuild: `make clean && rm -f crt0_*.o include/resource_defs.h .cart_header.o && make`, or list the dependencies in `Makefile.custom`.
The packer reports a unit that doesn't fit in a bank	One source file has more than 8 KB of code and constant data. Split it. See Banking.
Changes to the `.project` file are ignored	Run `tidepod sync`.
`undefined reference to malloc`	There is no heap allocator; see malloc(3). Use static buffers.
Make warns about overriding recipes	`Makefile.custom` redefines `all` or `debug`. Use `POST_BUILD` instead.

Running

Symptom	Cause and fix
`tidepod run` shows the old version	`run` doesn't build. Run `make` first.
Works with `--autorun` but not from the menu (or the other way round)	Test the real boot path with key injection; see Headless testing.
Crash or illegal-opcode flood after calling through a function pointer	Calls through pointers are not bank-switched. Move the target function to persistent RAM. See Banking.
A string or table passed to another file reads as garbage	It points into a bank that is no longer mapped. Copy it first or move the table to a persistent file.
Random corruption that gets worse as the program grows	Stack overflow into `.bss`. Move large locals to `static`, check `max_stack_depth` and RAM headroom in `bank_report.json`.
Disk access corrupts the program or hangs	Scratchpad code overlaps the disk system's area. Use the Disk I/O preset (`0x8380`). See Code placement.
Hard-disk image is refused	CHD images are not supported; extract a raw image with `chdman extracthd`. See Disks.
Text mode shows nothing	`ti_text_mode` sets up the mode but loads no font. Copy a font into the pattern table, e.g. with `rom_to_vram`.
Only 4 sprites per line on the F18A	Set VR30 after unlocking; its reset value is 4. See F18A GPU code.

Debugging

Symptom	Cause and fix
No source lines in the debugger	Build with `make debug` (F5 in the IDE).
GDB stops at `PC=0024`	That's the reset vector; `continue`.
GDB shows nonsense for VRAM addresses	`x/` reads CPU memory. Use the VDP tab or `--dump-regs`.
Logpoints print nothing	They only fire in a debug build running under the IDE debugger.

Known issues in this release

- `tidepod new --type 8k`: the first build fails because `cart_header.asm` is listed in the project but not created. Copy `.cart_header.asm` to `cart_header.asm` and build again.
- `ti_text_mode()` can blank the screen on a stock TMS9918A. Use an F18A engine (`--f18a`) while this is being fixed.
- Release builds don't define `NDEBUG`, so `__dbg_printf` calls remain in release binaries (as a 2-byte `NOP` each).

API Reference

136 man pages

Every page in this section is generated from the man pages shipped with Tidepod (`resources/man`), so it always matches the library you have installed. The same pages are available offline in the IDE and with `man`. 136 pages are listed below; use the filter or the tree.

Function(s)	Page	Description
`cgetc, kbhit, getchar`	`cgetc(3)`	keyboard and joystick 1 input
`conio`	`conio(3)`	console cursor I/O (cc65/libti99-shaped)
`cputc, cputs, cprintf, clrscr`	`cputc(3)`	console output at cursor
`gotoxy, gotox, gotoy, wherex, wherey`	`gotoxy(3)`	console cursor position
`fclose`	`fclose(3)`	close a disk stream
`fcopy_path`	`fcopy_path(3)`	bit-preserving copy via fopen family
`fgetc`	`fgetc(3)`	character/string stream I/O
`fgets`	`fgets(3)`	character/string stream I/O
`fopen, fclose, fread, fwrite, fflush, fcopy_path`	`fopen(3)`	disk file streams (DSR / AU)
`fprintf`	`fprintf(3)`	formatted output (see printf)
`fputc`	`fputc(3)`	character/string stream I/O
`fputs`	`fputs(3)`	character/string stream I/O
`fread, fwrite`	`fread(3)`	stream I/O (record or PROGRAM AU cache)

Function(s)	Page	Description
`fseek, ftell, rewind`	`fseek(3)`	reposition a disk stream
`ftell`	`ftell(3)`	report stream position
`fwrite`	`fwrite(3)`	write to a stream
`printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf`	`printf(3)`	formatted output
`putchar`	`putchar(3)`	output a character to standard output
`puts`	`puts(3)`	output a line to standard output
`snprintf`	`snprintf(3)`	formatted output (see printf)
`sprintf`	`sprintf(3)`	formatted output (see printf)
`sscanf, scanf`	`sscanf(3)`	formatted input (minimal)
`stdio_enable_chip, stdio_enable_fat, stdio_enable_wds, stdio_enable_prog, stdio_enable_all_disk`	`stdio_enable(3)`	opt-in native disk backends for fopen
`stdio_enable_all_disk`	`stdio_enable_all_disk(3)`	enable every optional fopen disk backend
`stdio_enable_chip`	`stdio_enable_chip(3)`	enable TI chip FS stdio backend for DSK*

Function(s)	Page	Description
`stdio_enable_fat`	`stdio_enable_fat(3)`	enable FAT12 HFDC stdio backend for DSK*
`stdio_enable_prog`	`stdio_enable_prog(3)`	enable DSR PROGRAM AU byte-stream backend
`stdio_enable_wds`	`stdio_enable_wds(3)`	enable HFDC WDS* chip stdio backend
`stdio_last_error`	`stdio_last_error(3)`	last disk/DSR error code from Tidepod libc
`stdio_set_wds_vdp_buf`, `stdio_get_wds_vdp_buf`	`stdio_set_wds_vdp_buf(3)`	choose HFDC/WDS Level-2 sector buffer in VRAM
`ti_text_mode`, `ti_text_load_builtin_font`	`ti_text_mode(3)`	select console text geometry and load a default font
`ti_vdp_detect, ti_vdp_text80_ok`	`ti_vdp_detect(3)`	detect F18A / Pico9918 and 80-column capability
`remove, rename`	`remove(3)`	delete or rename a disk file
`ti_dsr_enum`	`ti_dsr_enum(3)`	list catalogable DSR unit names via CRU scan
`ti_opendir, ti_readdir, ti_closedir, ti_dir_volume`	`ti_opendir(3)`	TI disk catalog

Function(s)	Page	Description
`memchr`	`memchr(3)`	locate byte in byte string
`memcpy`	`memcpy(3)`	copy memory area
`memmove`	`memmove(3)`	copy byte string
`memset`	`memset(3)`	fill a byte string with a byte value
`strcat, strncat`	`strcat(3)`	concatenate strings
`strcmp, strncmp`	`strcmp(3)`	compare strings
`stpcpy, stpncpy, strcpy, strncpy`	`strcpy(3)`	copy strings
`strlen, strnlen`	`strlen(3)`	find length of string
`strcat, strncat`	`strncat(3)`	concatenate strings
`strcmp, strncmp`	`strncmp(3)`	compare strings
`stpcpy, stpncpy, strcpy, strncpy`	`strncpy(3)`	copy strings
`strnlen`	`strnlen(3)`	bounded string length
`strstr`	`strstr(3)`	locate a substring in a string
`strtok`	`strtok(3)`	string tokens
`isalnum`	`isalnum(3)`	alphanumeric character test
`isalpha`	`isalpha(3)`	alphabetic character test
`iscntrl`	`iscntrl(3)`	control character test

Function(s)	Page	Description
`isdigit`	`isdigit(3)`	decimal-digit character test
`islower`	`islower(3)`	lower-case character test
`isprint`	`isprint(3)`	printing character test (including space)
`ispunct`	`ispunct(3)`	punctuation character test
`isspace`	`isspace(3)`	white-space character test
`isupper`	`isupper(3)`	upper-case character test
`isxdigit`	`isxdigit(3)`	hexadecimal-digit character test
`tolower`	`tolower(3)`	upper case to lower case letter conversion
`toupper`	`toupper(3)`	lower case to upper case letter conversion
`abort`	`abort(3)`	cause abnormal program termination
`abs`	`abs(3)`	integer absolute value function
`atoi, atol`	`atoi(3)`	convert string to integer
`atol`	`atol(3)`	convert string to long

Function(s)	Page	Description
`calloc`	`calloc(3)`	allocate zeroed memory (deferred)
`exit, _Exit`	`exit(3)`	perform normal program termination
`free`	`free(3)`	free memory (deferred)
`malloc`	`malloc(3)`	allocate memory (deferred)
`qsort`	`qsort(3)`	sort an array
`rand, rand_r, srand, sranddev`	`rand(3)`	bad random number generator
`realloc`	`realloc(3)`	resize allocation (deferred)
`rand, rand_r, srand, sranddev`	`srand(3)`	bad random number generator
`strtol, strtoul, atoi, atol`	`strtol(3)`	convert string to integer
`strtoul`	`strtoul(3)`	convert string to unsigned long
`atanf`	`atanf(3)`	inverse tangent (float)
`floorf, ceilf`	`ceilf(3)`	floor and ceiling functions (float)
`sinf, cosf, tanf`	`cosf(3)`	trigonometric functions (float)
`expf`	`expf(3)`	exponential function (float)

Function(s)	Page	Description
`fabsf`	`fabsf(3)`	absolute value (float)
`floorf, ceilf`	`floorf(3)`	floor and ceiling functions (float)
`fmodf`	`fmodf(3)`	floating-point remainder
`logf`	`logf(3)`	natural logarithm (float)
`powf`	`powf(3)`	power function (float)
`sinf, cosf, tanf`	`sinf(3)`	trigonometric functions (float)
`sqrtf`	`sqrtf(3)`	square root (float)
`sinf, cosf, tanf`	`tanf(3)`	trigonometric functions (float)
`libc`	`libc(3)`	TI-99/4A C library overview
`ASC`	`asc(3xb)`	return ASCII code of a string's first character
`CHAR`	`char(3xb)`	redefine a character's graphical pattern
`CHARSET`	`charset(3xb)`	restore standard character patterns
`CHR$`	`chr(3xb)`	return character corresponding to ASCII code
`CLEAR`	`clear(3xb)`	clear the screen and reset the VDP cursor

Function(s)	Page	Description
`COINC`	`coinc(3xb)`	detect collisions between sprites or dots
`COLOR`	`color(3xb)`	define foreground and background colors for a character set
`DELSPRITE`	`delsprite(3xb)`	delete a specific sprite or all active sprites
`DISPLAY AT`	`display_at(3xb)`	print text at a specific screen location
`DISTANCE`	`distance(3xb)`	compute the squared distance between two sprites
`GCHAR`	`gchar(3xb)`	read the character code at a screen position
`HCHAR`	`hchar(3xb)`	draw a horizontal line of characters
`JOYST`	`joyst(3xb)`	read joystick directional input
`KEY`	`key(3xb)`	scan the keyboard or joysticks for input
`LEN`	`len(3xb)`	return the length of a string
`LOAD`	`load(3xb)`	write bytes directly into CPU RAM

Function(s)	Page	Description
`LOCATE`	`locate(3xb)`	move an existing sprite to a new row and column
`MAGNIFY`	`magnify(3xb)`	set the global sprite magnification factor
`MOTION`	`motion(3xb)`	update the row and column velocities of a sprite
`PATTERN`	`pattern(3xb)`	change the character pattern used by a sprite
`PEEK`	`peek(3xb)`	read bytes from CPU RAM/ROM
`PEEKV`	`peekv(3xb)`	read bytes from VDP (Video) RAM
`POKE`	`poke(3xb)`	write bytes directly into CPU RAM
`POKEV`	`pokev(3xb)`	write bytes directly into VDP (Video) RAM
`POSITION`	`position(3xb)`	read the current screen coordinates of a sprite
`RND`	`rnd(3xb)`	generate a pseudo-random number
`SCREEN`	`screen(3xb)`	set the background screen color

Function(s)	Page	Description
`SEG$`	`seg(3xb)`	extract a substring from a string
`SOUND`	`sound(3xb)`	play sounds/noise through the SN76489 sound chip
`SPRITE`	`sprite(3xb)`	define and display a sprite
`STR$`	`str(3xb)`	convert numeric value to string
`VAL`	`val(3xb)`	convert string to numeric value
`VCHAR`	`vchar(3xb)`	draw a vertical line of characters
`VSYNC`	`vsync(3xb)`	halt execution until the next VDP vertical blanking interrupt
`xb-control`	`xb-control(7)`	XB BASIC control flow statements
`xb-data`	`xb-data(7)`	XB BASIC DATA, READ, RESTORE, and BYTES directives
`xb-input`	`xb-input(7)`	XB BASIC joystick, keyboard, and timing commands
`xb-io`	`xb-io(7)`	XB BASIC print, display, and input statements

Function(s)	Page	Description
`xb-math`	`xb-math(7)`	XB BASIC built-in numeric functions
`xb-misc`	`xb-misc(7)`	XB BASIC miscellaneous statements and directives
`xb-screen`	`xb-screen(7)`	XB BASIC screen, color, and character definition commands
`xb-sound`	`xb-sound(7)`	XB BASIC sound generation commands
`xb-sprites`	`xb-sprites(7)`	XB BASIC sprite creation, movement, and collision commands
`xb-string`	`xb-string(7)`	XB BASIC built-in string functions
`xb-sub`	`xb-sub(7)`	XB BASIC subroutine definition and invocation
`xb-tiles`	`xb-tiles(7)`	XB BASIC tile placement and reading commands
`xb-variables`	`xb-variables(7)`	XB BASIC variable types, arrays, and declarations

Function(s)	Page	Description
`xb-vdp`	xb-vdp(7)	XB BASIC VDP memory access and bulk transfer commands

cgetc, kbhit, getchar

cgetc(3)

keyboard and joystick 1 input

NAME

`cgetc`, `kbhit`, `getchar` — keyboard and joystick 1 input

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <conio.h>

unsigned char
cgetc(void);

unsigned char
kbhit(void);

#include <stdio.h>

int
getchar(void);
```

DESCRIPTION

CRU matrix scan with edge detection. `kbhit`() is nonzero when a new key (or joystick) edge is available. `cgetc`() blocks until input arrives and returns `KEY_*` codes or ASCII. `getchar`() returns `EOF` if nothing is waiting, otherwise the same codes.

Direction

- Fctn-E / Fctn-S / Fctn-D / Fctn-X map to `KEY_UP`, `KEY_LEFT`, `KEY_RIGHT`, `KEY_DOWN`.
- Joystick 1 (and host arrow keys in the Tidepod emu) map to the same `KEY_*` values. Joystick fire maps to `KEY_TAB` (pane-switch convention for apps like fileman).

Common chords

`Fctn-=`	`KEY_ESC` (TI QUIT)
`Fctn-6`	`KEY_TAB` (TI PROC'D)
`Ctrl-I`	`KEY_TAB`
`Ctrl-C`	`KEY_ESC`

Plain digit and letter keys return ASCII (e.g. `3`, `5`, `Q`). Fctn+digit is *not* translated to `KEY_F1` … `KEY_F5`; prefer plain numerics for app commands.

SEE ALSO

`conio(3)`

Source: `resources/man/man3/cgetc.3`

conio

conio(3)

console cursor I/O (cc65/libti99-shaped)

NAME

`conio` — console cursor I/O (cc65/libti99-shaped)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <conio.h>
```

DESCRIPTION

Addressable console functions sharing state with `putchar(3)` / `printf(3)`. Coordinates are 0-based. Use conio for panel UIs; avoid mixing with streaming `printf`() on the same screen.

Absolute draws (`cputcxy`, `cputsxy`, `cclearxy`, `chlinexy`, `cvlinexy`) clip at the screen edge and do *not* wrap or scroll (safe for dual-pane UIs). Streaming `cputc`() / `putchar`() still wrap and scroll.

`chline`() / `cvline`() emit CP437 box-drawing codes (`CH_HLINE`, `CH_VLINE`, `CH_DHLINE`, `CH_DVLINE`, `corners/tees, etc.`) so the pattern descriptor table must include those glyphs (e.g. fileman's 6×8 box set in the 5×8 font resource).

API inspired by Tursi's public-domain libti99 conio; reimplemented on Tidepod libc.

SEE ALSO

`gotoxy(3)`, `cputc(3)`, `cgetc(3)`, `ti_text_mode(3)`

Source: `resources/man/man3/conio.3`

cputc, cputs, cprintf, clrscr

cputc(3)

console output at cursor

NAME

`cputc`, `cputs`, `cprintf`, `clrscr` — console output at cursor

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <conio.h>

void
cputc(int ch);

void
cputs(const char *s);

int
cprintf(const char *fmt, ...);

void
clrscr(void);
```

DESCRIPTION

Write at the current cursor with wrap/scroll like putchar(3).

SEE ALSO

conio(3), gotoxy(3)

Source: `resources/man/man3/cputc.3`

gotoxy, gotox, gotoy, wherex, wherey

gotoxy(3)

console cursor position

NAME

`gotoxy`, `gotox`, `gotoy`, `wherex`, `wherey` — console cursor position

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <conio.h>

void
gotoxy(int x, int y);

int
wherex(void);

int
wherey(void);
```

DESCRIPTION

0-based cursor. Clamped to the current `screensize`().

SEE ALSO

conio(3)

Source: `resources/man/man3/gotoxy.3`

fclose

`fclose(3)`

close a disk stream

NAME

`fclose` — close a disk stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

int
fclose(FILE *stream);
```

DESCRIPTION

Flushes and closes a disk `FILE` opened with `fopen(3)`.

SEE ALSO

`fopen(3)`

Source: `resources/man/man3/fclose.3`

fcopy_path

fcopy_path(3)

bit-preserving copy via fopen family

NAME

`fcopy_path` — bit-preserving copy via fopen family

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

int
fcopy_path(const char *src, const char *dst);
```

DESCRIPTION

Probes *src* (same FDR path as) `fopen`() on an existing file , opens *dst* with a matching create mode (e.g. `wb:pg`, `wb:if:64`, `w:dv:80`), and copies with `fread`() / `fwrite`() only. No VDP LOAD/SAVE.

RETURN VALUES

Returns 0 on success, -1 on failure. Check `stdio_last_error(3)` first (DSR status); `errno` may also be set by some backends (see) `docs/stdio_errors.md`.

SEE ALSO

`fopen(3)`, `stdio_last_error(3)`, `stdio_enable(3)`, `remove(3)`

Source: `resources/man/man3/fcopy_path.3`

fgetc

fgetc(3)

character/string stream I/O

NAME

`fgetc` — character/string stream I/O

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

int
fgetc(FILE *stream);
```

DESCRIPTION

Reads one byte from a disk stream or console stdin. Disk streams use FIXED-128 records; see fopen(3).

SEE ALSO

fopen(3), putchar(3)

Source: `resources/man/man3/fgetc.3`

fgets

fgets(3)

character/string stream I/O

NAME

`fgets` — character/string stream I/O

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

char *
fgets(char *s, int n, FILE *stream);
```

DESCRIPTION

Reads at most n-1 bytes into s, stopping at newline or EOF. Disk streams use FIXED-128 records; see fopen(3).

SEE ALSO

fopen(3), putchar(3)

Source: `resources/man/man3/fgets.3`

fopen, fclose, fread, fwrite, fflush, fcopy_path

`fopen(3)`

disk file streams (DSR / AU)

NAME

`fopen`, `fclose`, `fread`, `fwrite`, `fflush`, `fcopy_path` — disk file streams (DSR / AU)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

FILE *
fopen(const char *path, const char *mode);

int
fclose(FILE *stream);

size_t
fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

size_t
fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

int
fflush(FILE *stream);

int
fcopy_path(const char *src, const char *dst);
```

DESCRIPTION

Paths are TI device names such as "`DSK1.MYFILE`" or "`WDS1.DIR.FILE`".

`fopen`() mode grammar:

```
mode := access [ "b" ] [ "+" ] [ ":" type [ ":" reclen ] ]
access := "r" | "w" | "a"
type := "df" | "dv" | "if" | "iv" | "pg"
```

FDR probe always wins for type and record length. A `:type[:reclen]` suffix on open is ignored so helpers can reuse one mode builder.

- no `b` → DISPLAY/VARIABLE 80 (`w`) creates DV80
- `b` present → INTERNAL/FIXED 128 (`wb`) creates IF128
- explicit `:df|:dv|:if|:iv|:pg[:reclen]` overrides the `b` defaults (e.g. `wb:df:80`, `wb:pg`, `wb:if:64`)

`b` on create without a type selects INTERNAL/FIXED 128. On an existing file (`df` / `dv`), plain `r` / `rb` applies record→newline text munging. Opaque DISPLAY I/O requires an explicit type with `b` (e.g. `rb:df:80`, `wb:dv:80`). `fcopy_path` always uses typed opaque modes for bit-identical copies.

(`pg`) uses an AU/sector byte stream inside libc (256-byte) `FILE.sector_buf` cache with `buf_pos / buf_len` Apps never issue LOAD/SAVE for copy or normal read/write. Record files (df/dv/if/iv) use Level-3 READ/WRITE; libc still caches one record in `sector_buf` so a one-byte `fread`() does not re-hit the DSR every time. See `docs/fopen_stream_buffering.md`.

Paths and *mode* are copied into stack scratch at entry (up to `TI_PATH_MAX`) / `STDIO_MODE_MAX` so a later cart-bank trampoline cannot invalidate a caller pointer. Banked callers should keep mode strings in RAM (not banked .rodata).

At most `STDIO_MAX_FILES` simultaneous disk `FILE` objects are supported.

by default `fopen()` uses Level-3 DSR PAB only (classic floppy `DSK*`) Native chip FS, FAT12, WDS*, and DSR PROGRAM-AU streams are opt-in so game carts do not pull their `.bss` from `libc.a`. Call `stdio_enable_chip(3)`, `stdio_enable_fat(3)`, `stdio_enable_wds(3)`, and/or `stdio_enable_prog(3)` (or) `stdio_enable_all_disk(3)` before disk I/O when those paths are required. See `stdio_enable(3)` and `docs/stdio_backends.md`.

HFDC/WDS Level-2 AU I/O uses a 256-byte VRAM sector buffer (default `0x3DA8`), high under FILES . Call `stdio_set_wds_vdp_buf(3)` if that window overlaps your screen, attributes, or PAB arena (fileman parks it at `0x3600`); Text80 attrs live at `>0800`.

`fcopy_path()` copies *src* to *dst* using only `fopen(/)` `fread(/)` `fwrite(/)` `fclose()` (probes source type; bit-preserving for probed kinds).

`stdin`, `stdout`, and `stderr` are console sentinels, not disk files.

RETURN VALUES

`fopen()` returns a `FILE *` or `NULL` on failure. `fread()` / `fwrite()` return the number of complete elements transferred. `fcopy_path()` returns 0 on success, -1 on failure.

ERRORS

On `fopen()` / `fcopy_path()` failure, call `stdio_last_error(3)` for the DSR-style status (primary signal). `errno` may also be set by native backends (chip / FAT / WDS) but is often left unchanged after classic PAB/DSR failures — do not rely on `errno` alone.

SEE ALSO

`stdio_last_error(3)`, `fgetc(3)`, `fseek(3)`, `remove(3)`, `stdio_enable(3)`, `stdio_set_wds_vdp_buf(3)`, `ti_opendir(3)`, `printf(3)`

Source: `resources/man/man3/fopen.3`

fprintf

`fprintf(3)`

formatted output (see printf)

NAME

`fprintf` — formatted output (see printf)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>
```

See `printf(3)`.

DESCRIPTION

TI-99/4A `fprintf` is documented with the `printf(3)` family. `asprintf`() is not available (no malloc).

SEE ALSO

`printf(3)`

Source: `resources/man/man3/fprintf.3`

fputc

fputc(3)

character/string stream I/O

NAME

`fputc` — character/string stream I/O

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

int
fputc(int c, FILE *stream);
```

DESCRIPTION

Writes one byte to a disk stream or console stdout/stderr. Disk streams use FIXED-128 records; see `fopen(3)`.

SEE ALSO

`fopen(3)`, `putchar(3)`

Source: `resources/man/man3/fputc.3`

fputs

fputs(3)

character/string stream I/O

NAME

`fputs` — character/string stream I/O

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

int
fputs(const char *s, FILE *stream);
```

DESCRIPTION

Writes the string s without a trailing newline. Disk streams use FIXED-128 records; see `fopen(3)`.

SEE ALSO

`fopen(3)`, `putchar(3)`

Source: `resources/man/man3/fputs.3`

fread, fwrite

fread(3)

stream I/O (record or PROGRAM AU cache)

NAME

`fread`, `fwrite` — stream I/O (record or PROGRAM AU cache)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

size_t
fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

size_t
fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

DESCRIPTION

Transfer *size* × *nmemb* bytes to/from *stream*. Bytes come from `FILE.sector_buf` when the cache still has unread data; otherwise libc fills the next Level-3 record or PROGRAM AU (256 bytes) and continues. A one-byte `fread` () does *not* re-read the disk sector every call.

DISPLAY streams without `b` may map newlines to VARIABLE record boundaries. INTERNAL / `b` / PROGRAM streams are opaque byte streams.

See `fopen(3)` and `docs/fopen_stream_buffering.md`.

SEE ALSO

`fopen(3)`, `fgetc(3)`, `fseek(3)`

Source: `resources/man/man3/fread.3`

fseek, ftell, rewind

fseek(3)

reposition a disk stream

NAME

`fseek`, `ftell`, `rewind` — reposition a disk stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

int
fseek(FILE *stream, long offset, int whence);

long
ftell(FILE *stream);

void
rewind(FILE *stream);
```

DESCRIPTION

Byte offsets are mapped onto FIXED-128 disk records. Seeking is limited: prefer `SEEK_SET` with offsets that are multiples of 128 when possible. `SEEK_END` support is minimal.

`rewind`() is equivalent to `fseek`(*stream*, *0*, *SEEK_SET*) followed by clearing the error indicator.

RETURN VALUES

`fseek`() returns 0 on success, nonzero on failure. `ftell`() returns the current byte offset, or -1 on error.

SEE ALSO

fopen(3), fread(3)

Source: `resources/man/man3/fseek.3`

ftell

ftell(3)

report stream position

NAME

`ftell` — report stream position

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

long
ftell(FILE *stream);
```

DESCRIPTION

See `fseek(3)`.

SEE ALSO

`fseek(3)`

Source: `resources/man/man3/ftell.3`

fwrite

`fwrite(3)`

write to a stream

NAME

`fwrite` — write to a stream

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

size_t
fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

DESCRIPTION

See `fread(3)`.

SEE ALSO

`fread(3)`, `fopen(3)`

Source: `resources/man/man3/fwrite.3`

printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf

`printf(3)`

formatted output

NAME

`printf`, `fprintf`, `sprintf`, `snprintf`, `vprintf`, `vfprintf`, `vsprintf`, `vsnprintf` — formatted output

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

int
printf(const char *fmt, ...);

int
fprintf(FILE *stream, const char *fmt, ...);

int
sprintf(char *str, const char *fmt, ...);

int
snprintf(char *str, size_t size, const char *fmt, ...);

int
vprintf(const char *fmt, va_list ap);

int
vfprintf(FILE *stream, const char *fmt, va_list ap);
```

```
int
vsprintf(char *str, const char *fmt, va_list ap);

int
vsnprintf(char *str, size_t size, const char *fmt, va_list ap);
```

DESCRIPTION

Subset of ISO C `printf`() for the TI-99/4A.

`printf`() and `vprintf`() write to the VDP nametable via the console cursor (see `putchar(3)`). Call `ti_text_mode`() first to select 32, 40, or 80 columns. `fprintf`() / `vfprintf`() to `stdout` / `stderr` use the same path; disk `FILE *` streams use `fputc(3)`.

Supported conversions: `%s %c %d %i %u %x %X %p %%` with optional width and precision digits. Flags `- + 0 #` are parsed but only minimally honored.

Not supported: `%f %e %g`, `asprintf`(), positional `%n$`, and length modifiers other than a single `l` (skipped).

TYPE MODEL

`int` and `size_t` are 16-bit; `long` is 32-bit.

RETURN VALUES

Number of characters emitted (or that would be emitted for `snprintf`()), not counting a trailing NUL written into string buffers.

SEE ALSO

`putchar(3)`, `puts(3)`, `sprintf(3)`

Source: `resources/man/man3/printf.3`

putchar

putchar(3)

output a character to standard output

NAME

`putchar` — output a character to standard output

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

int
putchar(int c);
```

DESCRIPTION

The `putchar`() function writes the character *c* (converted to an `unsigned char`) to the standard output stream.

On the TI-99/4A, standard output is the VDP display. Characters are written to the current cursor position in VRAM and the cursor advances.

RETURN VALUES

The `putchar`() function returns the character written. If a write error occurs, it returns `EOF`.

SEE ALSO

`puts(3)`, `printf(3)`, `fputc(3)`

STANDARDS

The `putchar`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/putchar.3`

puts

puts(3)

output a line to standard output

NAME

`puts` — output a line to standard output

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

int
puts(const char *s);
```

DESCRIPTION

The `puts`() function writes the string *s*, and a terminating newline character, to the standard output stream.

On the TI-99/4A, standard output is the VDP display.

RETURN VALUES

The `puts`() function returns a non-negative integer on success and `EOF` on failure.

SEE ALSO

`putchar(3)`, `printf(3)`, `fputs(3)`

STANDARDS

The `puts`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/puts.3`

snprintf

snprintf(3)

formatted output (see printf)

NAME

`snprintf` — formatted output (see printf)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>
```

See printf(3).

DESCRIPTION

TI-99/4A `snprintf` is documented with the printf(3) family. `asprintf`() is not available (no malloc).

SEE ALSO

printf(3)

Source: `resources/man/man3/snprintf.3`

sprintf

sprintf(3)

formatted output (see printf)

NAME

`sprintf` — formatted output (see printf)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>
```

See printf(3).

DESCRIPTION

TI-99/4A `sprintf` is documented with the printf(3) family. `asprintf`() is not available (no malloc).

SEE ALSO

printf(3)

Source: `resources/man/man3/sprintf.3`

sscanf, scanf

sscanf(3)

formatted input (minimal)

NAME

`sscanf`, `scanf` — formatted input (minimal)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

int
sscanf(const char *str, const char *fmt, ...);

int
scanf(const char *fmt, ...);
```

DESCRIPTION

Minimal conversions matching the printf(3) subset: `%s %c %d %i %u %x` `scanf`() reads from the keyboard via `getchar`(). Positional `%n$` and floating conversions are not implemented.

RETURN VALUES

Number of assigned input items, or `EOF` on input failure before any conversion.

SEE ALSO

printf(3), getchar(3)

Source: `resources/man/man3/sscanf.3`

stdio_enable_chip, stdio_enable_fat, stdio_enable_wds, stdio_enable_prog, stdio_enable_all_disk

`stdio_enable(3)`

opt-in native disk backends for fopen

NAME

`stdio_enable_chip`, `stdio_enable_fat`, `stdio_enable_wds`, `stdio_enable_prog`, `stdio_enable_all_disk` — opt-in native disk backends for fopen

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

void
stdio_enable_chip(void);

void
stdio_enable_fat(void);

void
stdio_enable_wds(void);

void
stdio_enable_prog(void);

void
stdio_enable_all_disk(void);
```

DESCRIPTION

By default, `fopen(3)` uses Level-3 DSR PAB I/O only (classic `DSK1.NAME`) paths . That keeps game carts slim: unused chip / FAT / WDS / PROGRAM-AU translation units are not pulled from `libc.a` into upper RAM.

Apps that need the native backends must call one or more of these enable routines *before* disk I/O (typically before) `stdio_disk_init`() or the first `fopen`():

`stdio_enable_chip()`	TIFDC / HFDC floppy `DSK*` via `ti_fs_chip` (no Level-3 PAB).
`stdio_enable_fat()`	FAT12 volumes on HFDC `DSK*` (opaque byte stream).
`stdio_enable_wds()`	HFDC hard `WDS*` via chip AU streams (also installs the optional HFDC face poke used before PAB opens).
`stdio_enable_prog()`	DSR PROGRAM files as AU byte streams (`:pg`) and untyped fall-through to PROGRAM .
`stdio_enable_all_disk()`	Convenience for file managers and disk tests: enables chip, FAT, WDS, and PROGRAM. Lives in its own archive member so games that never call it do not pull those backends.

Each `stdio_enable_*`() registers function pointers used by `fopen`() / `fread`() / `fwrite`() / `fseek`() Calling an enable pulls that backend's `.text` and `.bss` from the archive; call only what you need.

that only save/load on a classic floppy DSR should call *none* of these (e.g. Commander Keen). and the disk suite examples call `stdio_enable_all_disk`().

RETURN VALUES

None.

SEE ALSO

`fopen(3)`, `stdio_set_wds_vdp_buf(3)`, `fcopy_path(3)`, `ti_opendir(3)`, `docs/stdio_backends.md`

Source: `resources/man/man3/stdio_enable.3`

stdio_enable_all_disk

stdio_enable_all_disk(3)

enable every optional fopen disk backend

NAME

`stdio_enable_all_disk` — enable every optional fopen disk backend

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

void
stdio_enable_all_disk(void);
```

DESCRIPTION

Calls `stdio_enable_chip`(), `stdio_enable_fat`(), `stdio_enable_wds`(), and `stdio_enable_prog`(). Use in file managers and disk test carts. Do *not* call from RAM-tight games that only need Level-3 PAB `fopen`(); this symbol lives in its own archive member so omitting the call keeps chip / WDS `.bss` out of the link.

Full backend overview: stdio_enable(3).

SEE ALSO

stdio_enable(3), fopen(3), docs/stdio_backends.md

Source: `resources/man/man3/stdio_enable_all_disk.3`

stdio_enable_chip

enable TI chip FS stdio backend for DSK*

NAME

`stdio_enable_chip` — enable TI chip FS stdio backend for DSK*

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

void
stdio_enable_chip(void);
```

DESCRIPTION

Registers the `ti_fs_chip` path matcher and stream ops so `fopen(3)` can open TIFDC / HFDC floppy `DSK*` paths without Level-3 PAB. Call once at startup before disk I/O. Pulls chip FS `.bss` into upper RAM; omit on carts that only need classic DSR PAB.

Full backend overview: `stdio_enable(3)`.

SEE ALSO

`stdio_enable(3)`, `stdio_enable_all_disk(3)`, `fopen(3)`

Source: `resources/man/man3/stdio_enable_chip.3`

stdio_enable_fat

enable FAT12 HFDC stdio backend for DSK*

NAME

`stdio_enable_fat` — enable FAT12 HFDC stdio backend for DSK*

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

void
stdio_enable_fat(void);
```

DESCRIPTION

Registers the FAT12 path matcher and stream ops so `fopen(3)` can open HFDC `DSK*` volumes formatted as FAT12 (opaque byte stream). Call once at startup before disk I/O.

Full backend overview: `stdio_enable(3)`.

SEE ALSO

`stdio_enable(3)`, `stdio_enable_all_disk(3)`, `fopen(3)`

Source: `resources/man/man3/stdio_enable_fat.3`

stdio_enable_prog

stdio_enable_prog(3)

enable DSR PROGRAM AU byte-stream backend

NAME

`stdio_enable_prog` — enable DSR PROGRAM AU byte-stream backend

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

void
stdio_enable_prog(void);
```

DESCRIPTION

Registers PROGRAM (`:pg`) open and stream ops so `fopen(3)` can treat DSR PROGRAM images as sequential AU byte streams (no app-visible LOAD/SAVE). Without this enable, untyped opens that resolve to PROGRAM fall back toward record OPEN (which) `fopen`() rejects for true PROGRAM FDRs .

Full backend overview: `stdio_enable(3)`.

SEE ALSO

`stdio_enable(3)`, `fopen(3)`, `docs/fopen_stream_buffering.md`

Source: `resources/man/man3/stdio_enable_prog.3`

stdio_enable_wds

stdio_enable_wds(3)

enable HFDC WDS* chip stdio backend

NAME

`stdio_enable_wds` — enable HFDC WDS* chip stdio backend

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

void
stdio_enable_wds(void);
```

DESCRIPTION

Registers WDS* chip AU stream ops and the optional HFDC face poke used before PAB opens on HFDC-owned units. Required for `WDS1....` paths through `fopen(3)`. Also set `stdio_set_wds_vdp_buf(3)` if the default Level-2 VRAM window overlaps your screen tables.

Full backend overview: `stdio_enable(3)`.

SEE ALSO

`stdio_enable(3)`, `stdio_set_wds_vdp_buf(3)`, `fopen(3)`

Source: `resources/man/man3/stdio_enable_wds.3`

stdio_last_error

stdio_last_error(3)

last disk/DSR error code from Tidepod libc

NAME

`stdio_last_error` — last disk/DSR error code from Tidepod libc

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

unsigned int
stdio_last_error(void);
```

DESCRIPTION

Disk path APIs (`fopen`, `remove`, `ti_opendir`, `fcopy_path`, `…`) record a DSR-style status in an internal `stdio_last_err` and expose it via `stdio_last_error`(). This is the failure signal for classic Level-3 PAB I/O.

Defined codes (see `stdio.h`):

`STDIO_ERR_NONE`	0 — success / cleared
`STDIO_ERR_BADDEV`	1 — bad device
`STDIO_ERR_PROTECT`	2 — write protect
`STDIO_ERR_BADOPEN`	3 — bad open
`STDIO_ERR_NOSPACE`	4 — out of space
`STDIO_ERR_EOF`	5 — end of file / missing
`STDIO_ERR_DEVICE`	6 — device error
`STDIO_ERR_FILE`	7 — file error

Some helpers also use extended codes 9–12 (verify / layout / bad path). Successful opens and many helpers clear the code to 0.

is *not* set on every DSR failure. Native backends (chip / FAT / WDS / HFDC Level-2) may set `errno` (e.g. `EINVAL`, `ENOSPC`, `ENOTSUP`) in addition to `stdio_last_error`(). Prefer `stdio_last_error`() for portable disk error handling; check `errno` only when a backend documents it.

RETURN VALUES

The last recorded unsigned status code.

SEE ALSO

fopen(3), fcopy_path(3), errno(3), libc(3)

docs/stdio_errors.md

Source: `resources/man/man3/stdio_last_error.3`

stdio_set_wds_vdp_buf, stdio_get_wds_vdp_buf

`stdio_set_wds_vdp_buf(3)`

choose HFDC/WDS Level-2 sector buffer in VRAM

NAME

`stdio_set_wds_vdp_buf`, `stdio_get_wds_vdp_buf` — choose HFDC/WDS Level-2 sector buffer in VRAM

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

void
stdio_set_wds_vdp_buf(uint16_t vdp_addr);

uint16_t
stdio_get_wds_vdp_buf(void);
```

DESCRIPTION

Myarc HFDC Level-2 sector I/O (AU read/write via subprogram `>10`) uses a 256-byte VRAM window. The DSR programs HDC DMA to the same numeric address; on this toolchain the emulator mirrors that transfer into VDP RAM, so the window must not overlap screen nametable, F18A attribute table, bitmap color table, catalog PABs, or the `stdio_set_vdp_base`() PAB arena.

The libc default is `0x3DA8` (high under FILES(2)), the traditional Myarc location. Console Text80 attributes live at `>0800` so HFDC mirrors around `>221A` do not paint the UI; apps that place their own tables high should still call `stdio_set_wds_vdp_buf`() (e.g. fileman uses `0x3600`), below its PAB arena at `>3800`.

Addresses that cannot fit 256 bytes below `>4000` are rejected and the default is restored. `stdio_get_wds_vdp_buf`() returns the address currently in use.

SEE ALSO

fopen(3), stdio_enable(3), stdio_enable_wds(3), stdio_set_vdp_base(3), ti_opendir(3)

Source: `resources/man/man3/stdio_set_wds_vdp_buf.3`

ti_text_mode, ti_text_load_builtin_font

`ti_text_mode(3)`

select console text geometry and load a default font

NAME

`ti_text_mode`, `ti_text_load_builtin_font` — select console text geometry and load a default font

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdio.h>

void
ti_text_mode(int cols);

void
ti_text_load_builtin_font(void);
```

DESCRIPTION

`ti_text_mode`() selects 32, 40, or 80 column text, programs VDP registers for the console nametable at (>0000) and pattern table at (>1000), and resets the console cursor. If *cols* is 80 but `ti_vdp_text80_ok`() is false, the call falls back to 40 columns. On a stock TMS9918A only VR0–7 are written; F18A/Pico enhanced registers are gated behind detect so they cannot alias into VR1 and blank the display.

`ti_text_mode`() does *not* load glyph patterns. Call `ti_text_load_builtin_font`() afterward to upload a bundled 5×8 font into the PDT, or `rom_to_vram`() a project resource (as fileman does with `font_5x8.dat`). `ti_text_load_builtin_font`() lives in its own libc object and is linked only if referenced.

SEE ALSO

ti_vdp_detect(3), rom_to_vram(3), conio(3)

Source: `resources/man/man3/ti_text_mode.3`

ti_vdp_detect, ti_vdp_text80_ok

detect F18A / Pico9918 and 80-column capability

NAME

`ti_vdp_detect`, `ti_vdp_text80_ok` — detect F18A / Pico9918 and 80-column capability

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <vdpinfo.h>

int
ti_vdp_detect(void);

int
ti_vdp_text80_ok(void);
```

DESCRIPTION

`ti_vdp_detect`() unlocks an F18A-compatible VDP (VR57 twice) and reads status register 14. It returns `TI_VDP_STOCK`, `TI_VDP_F18A`, or `TI_VDP_PICO9918` (currently reported as F18A-compatible). Results are cached.

`ti_vdp_text80_ok`() is nonzero when 80-column text mode is safe (enhanced VDP with firmware >= 1.6).

SEE ALSO

`ti_text_mode(3)`

Source: `resources/man/man3/ti_vdp_detect.3`

remove, rename

remove(3)

delete or rename a disk file

NAME

`remove`, `rename` — delete or rename a disk file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <dirent.h>

int
remove(const char *path);

int
rename(const char *oldpath, const char *newpath);
```

DESCRIPTION

`remove`() deletes a TI path such as `DSK1.MYFILE` (Level-3 DELETE or FS mutate, binding-dispatched).

`rename`() renames within the same volume when the bound FS supports it (TIFDC / HFDC Level-2 meta; may fall back to copy+delete). Cross-volume rename is not supported — use `fcopy_path`() then `remove`().

SEE ALSO

ti_opendir(3), fopen(3)

Source: `resources/man/man3/remove.3`

ti_dsr_enum

ti_dsr_enum(3)

list catalogable DSR unit names via CRU scan

NAME

`ti_dsr_enum` — list catalogable DSR unit names via CRU scan

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <dirent.h>

int
ti_dsr_enum(char out[][TI_DSR_NAME_MAX + 1], int max);
```

DESCRIPTION

Walk PEB CRU bases (`>1000`) through `>1F00`, map each card ROM, and collect type-8 device names from the list at `>4008`. Only catalogable unit names are returned: `DSK1` through `DSK8` and `WDS1` through `WDS3` (bare `DSK`, `WDS`, `TIME`, and other names are skipped).

Names are written without a trailing period. Pass the result to `ti_opendir`() as `DSK1.` or `WDS1.` to catalog.

At most `TI_DSR_MAX` (16) names are returned; *max* is clamped to that limit.

RETURN VALUES

Number of names written to *out* (zero if none found or arguments are invalid).

SEE ALSO

ti_opendir(3), libc(3)

Source: `resources/man/man3/ti_dsr_enum.3`

ti_opendir, ti_readdir, ti_closedir, ti_dir_volume

`ti_opendir(3)`

TI disk catalog

NAME

`ti_opendir`, `ti_readdir`, `ti_closedir`, `ti_dir_volume` — TI disk catalog

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <dirent.h>

ti_DIR *
ti_opendir(const char *device);

int
ti_readdir(ti_DIR *dir, ti_dirent *ent);

void
ti_closedir(ti_DIR *dir);

int
ti_dir_volume(const ti_DIR *dir, ti_volume_t *out);
```

DESCRIPTION

Open *device* (e.g. `DSK1.` or `WDS1.`) as an INTERNAL/FIXED 38 catalog (OPEN passes record length 0 so the DSR fills 38; required for Myarc HFDC). The device name is copied into always-mapped storage at entry before any further catalog work (bank-safe; see) `docs/rom_architecture.md`.

`ti_readdir`() fills *ent* and returns nonzero while file entries remain (the volume record is consumed internally and is not returned as an entry).

`ti_dir_volume`() copies volume metadata gathered from catalog record 0 into *out*. Call it before `ti_closedir`(). It returns nonzero when volume data is available.

ti_dirent

`name`	NUL-terminated file name, max `TI_NAME_MAX` (10) characters.
`type`	1=DISPLAY/FIXED, 2=DISPLAY/VARIABLE, 3=INTERNAL/FIXED, 4=INTERNAL/VARIABLE, 5=PROGRAM; 0=empty/unknown.
`sectors`	Sectors allocated to the file.
`reclen`	Record length for DF/DV/IF/IV files, or EOF offset in the last sector for PROGRAM files.
`flags`	Bit 0 set if the file is write-protected.

ti_volume_t

`volume`	Disk volume name.
`total_sectors, free_sectors`	From catalog record 0.

Numeric catalog fields use BASIC INTERNAL encoding: a length-prefixed name followed by three length-prefixed RADIX-100 floats (each numeric is `08 then 8 float bytes`). Negative type means write-protected (first two float bytes are ones-complemented). Libc also accepts layouts without per-float length bytes and flat emulator catalogs (10-char name at offset 0 with floats at 10/18/26, or big-endian words at 10/12/14). The standard FIXED-38 catalog does *not* include timestamps.

Supported on classic floppy DSRs, Myarc HFDC hard disks (`WDSn.`), and emulator DSK/HD images. Discover unit names with `ti_dsr_enum(3)` rather than hardcoding `DSKn.` / `WDSn.`.

RETURN VALUES

`ti_opendir`() returns a directory handle, or `NULL` on failure. `ti_readdir`() returns nonzero for a valid entry, zero at end-of-catalog or on error. `ti_dir_volume`() returns nonzero when *out* was filled.

SEE ALSO

ti_dsr_enum(3), remove(3), fopen(3), libc(3)

Source: `resources/man/man3/ti_opendir.3`

memchr

memchr(3)

locate byte in byte string

NAME

`memchr` — locate byte in byte string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <string.h>

void *
memchr(const void *s, int c, size_t n);
```

DESCRIPTION

The `memchr`() function locates the first occurrence of *c* (converted to an `unsigned char`) in the initial *n* bytes of memory area *s*.

RETURN VALUES

The `memchr`() function returns a pointer to the byte located, or `NULL` if no such byte exists within *n* bytes.

SEE ALSO

`strchr(3)`, `memcmp(3)`, `memcpy(3)`, `memset(3)`

STANDARDS

The `memchr`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/memchr.3`

memcpy

`memcpy(3)`

copy memory area

NAME

`memcpy` — copy memory area

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <string.h>

void *
memcpy(void *restrict dst, const void *restrict src, size_t n);
```

DESCRIPTION

The `memcpy`() function copies *n* bytes from memory area *src* to memory area *dst*. If *dst* and *src* overlap, behavior is undefined. Applications in which *dst* and *src* might overlap should use `memmove(3)` instead.

RETURN VALUES

The `memcpy`() function returns the original value of *dst*.

SEE ALSO

`bcopy(3)`, `memccpy(3)`, `memmove(3)`, `strcpy(3)`, `wmemcpy(3)`

STANDARDS

The `memcpy`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/memcpy.3`

memmove

memmove(3)

copy byte string

NAME

`memmove` — copy byte string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <string.h>

void *
memmove(void *dst, const void *src, size_t len);
```

DESCRIPTION

The `memmove`() function copies *len* bytes from string *src* to string *dst*. The two strings may overlap; the copy is always done in a non-destructive manner.

RETURN VALUES

The `memmove`() function returns the original value of *dst*.

SEE ALSO

`bcopy(3)`, `memccpy(3)`, `memcpy(3)`, `strcpy(3)`, `wmemmove(3)`

STANDARDS

The `memmove`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/memmove.3`

memset

memset(3)

fill a byte string with a byte value

NAME

`memset` — fill a byte string with a byte value

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <string.h>

void *
memset(void *b, int c, size_t len);
```

DESCRIPTION

The `memset`() function writes *len* bytes of value *c* (converted to an `unsigned char`) to the string *b*.

RETURN VALUES

The `memset`() function returns its first argument.

SEE ALSO

`bzero(3)`, `memset_pattern(3)`, `memset_s.3`, `swab(3)`, `wmemset(3)`

STANDARDS

The `memset`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/memset.3`

strcat, strncat

strcat(3)

concatenate strings

NAME

`strcat`, `strncat` — concatenate strings

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <string.h>

char *
strcat(char *restrict s1, const char *restrict s2);

char *
strncat(char *restrict s1, const char *restrict s2, size_t n);
```

ADMONISHMENT

YOU SHOULD ALMOST CERTAINLY USE `strlcat`() INSTEAD.

See "Security Considerations" below.

DESCRIPTION

The `strcat`() and `strncat`() functions append a copy of the null-terminated string *s2* to the end of the null-terminated string *s1*, then add a terminating '`\0`'. The string *s1* must have sufficient space to hold the result.

The `strncat`() function appends not more than *n* characters from *s2*, and then adds a terminating '`\0`'.

The source and destination strings should not overlap, as the behavior is undefined.

RETURN VALUES

The `strcat`() and `strncat`() functions return the pointer *s1*.

SECURITY CONSIDERATIONS

The `strcat`() function is easily misused in a manner which enables malicious users to arbitrarily change a running program's functionality through a buffer overflow attack. (See the FSA.)

Avoid using `strcat`(). Instead, use `strncat`() or `strlcat`() and ensure that no more characters are copied to the destination buffer than it can hold.

Note that `strncat`() can also be problematic. It may be a security concern for a string to be truncated at all. Since the truncated string will not be as long as the original, it may refer to a completely different resource and usage of the truncated resource could result in very incorrect behavior. Example:

```
void
foo(const char *arbitrary_string)
{
        char onstack[8] = "";

#if defined(BAD)
        /*
         * This first strcat is bad behavior.  Do not use strcat!
         */
        (void)strcat(onstack, arbitrary_string);        /* BAD! */
#elif defined(BETTER)
        /*
         * The following two lines demonstrate better use of
         * strncat().
         */
        (void)strncat(onstack, arbitrary_string,
            sizeof(onstack) - strlen(onstack) - 1);
#elif defined(BEST)
        /*
         * These lines are even more robust due to testing for
         * truncation.
         */
        if (strlen(arbitrary_string) + 1 >
```

```
            sizeof(onstack) - strlen(onstack))
                err(1, "onstack would be truncated");
        (void)strncat(onstack, arbitrary_string,
            sizeof(onstack) - strlen(onstack) - 1);
#endif
}
```

SEE ALSO

bcopy(3), memccpy(3), memcpy(3), memmove(3), strcpy(3), strlcat(3), strlcpy(3), wcscat(3)

STANDARDS

The `strcat`() and `strncat`() functions conform to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/strcat.3`

strcmp, strncmp

strcmp(3)

compare strings

NAME

`strcmp`, `strncmp` — compare strings

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <string.h>

int
strcmp(const char *s1, const char *s2);

int
strncmp(const char *s1, const char *s2, size_t n);
```

DESCRIPTION

The `strcmp`() and `strncmp`() functions lexicographically compare the null-terminated strings *s1* and *s2*.

The `strncmp`() function compares not more than *n* characters. Because `strncmp`() is designed for comparing strings rather than binary data, characters that appear after a '`\0`' character are not compared.

RETURN VALUES

The `strcmp`() and `strncmp`() functions return an integer greater than, equal to, or less than 0, according as the string *s1* is greater than, equal to, or less than the string *s2*. The comparison is done using unsigned characters, so that '`\200`' is greater than '`\0`'.

SEE ALSO

bcmp(3), memcmp(3), strcasecmp(3), strcoll(3), strxfrm(3), wcscmp(3)

STANDARDS

The `strcmp`() and `strncmp`() functions conform to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/strcmp.3`

stpcpy, stpncpy, strcpy, strncpy

strcpy(3)

copy strings

NAME

`stpcpy, stpncpy, strcpy`, `strncpy` — copy strings

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <string.h>

char *
stpcpy(char * dst, const char * src);

char *
stpncpy(char * dst, const char * src, size_t len);

char *
strcpy(char * dst, const char * src);

char *
strncpy(char * dst, const char * src, size_t len);
```

ADMONISHMENT

YOU SHOULD ALMOST CERTAINLY USE `strlcpy`() INSTEAD.

See "Examples" below.

DESCRIPTION

The `stpcpy`() and `strcpy`() functions copy the string *src* to *dst* (including the terminating '`\0`' character.)

The `stpncpy`() and `strncpy`() functions copy at most *len* characters from *src* into *dst*. If *src* is less than *len* characters long, the remainder of *dst* is filled with '`\0`' characters. Otherwise, *dst* is *not* terminated.

The source and destination strings should not overlap, as the behavior is undefined.

RETURN VALUES

The `strcpy`() and `strncpy`() functions return *dst*. The `stpcpy`() and `stpncpy`() functions return a pointer to the terminating '`\0`' character of *dst*. If `stpncpy`() does not terminate *dst* with a `NUL` character, it instead returns a pointer to `dst[n]` (which does not necessarily refer to a valid memory location.)

EXAMPLES

The following sets `chararray` to "`abc\0\0\0`":

```
char chararray[6];

(void)strncpy(chararray, "abc", sizeof(chararray));
```

The following sets `chararray` to "`abcdef`":

```
char chararray[6];

(void)strncpy(chararray, "abcdefgh", sizeof(chararray));
```

Note that it does *not* NUL terminate `chararray` because the length of the source string is greater than or equal to the length argument.

The following copies as many characters from `input` to `buf` as will fit and NUL terminates the result. Because `strncpy`() does *not* guarantee to NUL terminate the string itself, this must be done explicitly.

```
char buf[1024];

(void)strncpy(buf, input, sizeof(buf) - 1);
buf[sizeof(buf) - 1] = '\0';
```

This could be better achieved using `strlcpy(3)`, as shown in the following example:

```
(void)strlcpy(buf, input, sizeof(buf));
```

Note that because `strlcpy(3)` is not defined in any standards, it should only be used when portability is not a concern.

SEE ALSO

`bcopy(3)`, `memccpy(3)`, `memcpy(3)`, `memmove(3)`, `strlcpy(3)`, `wcscpy(3)`

STANDARDS

The `strcpy`() and `strncpy`() functions conform to ISO/IEC 9899:1990 (“ISO C90”). The `stpcpy`() and `stpncpy`() functions conform to IEEE Std 1003.1-2008 (“POSIX.1”).

HISTORY

The `stpcpy`() function first appeared in FreeBSD 4.4, and `stpncpy`() was added in FreeBSD 8.0.

SECURITY CONSIDERATIONS

The `strcpy`() function is easily misused in a manner which enables malicious users to arbitrarily change a running program's functionality through a buffer overflow attack.

Source: `resources/man/man3/strcpy.3`

strlen, strnlen

strlen(3)

find length of string

NAME

`strlen, strnlen` — find length of string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <string.h>

size_t
strlen(const char *s);

size_t
strnlen(const char *s, size_t maxlen);
```

DESCRIPTION

The `strlen`() function computes the length of the string *s*. The `strnlen`() function attempts to compute the length of *s*, but never scans beyond the first *maxlen* bytes of *s*.

RETURN VALUES

The `strlen`() function returns the number of characters that precede the terminating `NUL` character. The `strnlen`() function returns either the same result as `strlen`() or *maxlen*, whichever is smaller.

SEE ALSO

`string(3)`, `wcslen(3)`, `wcswidth(3)`

STANDARDS

The `strlen`() function conforms to ISO/IEC 9899:1990 (“ISO C90”). The `strnlen`() function conforms to IEEE Std 1003.1-2008 (“POSIX.1”).

Source: `resources/man/man3/strlen.3`

strcat, strncat

strncat(3)

concatenate strings

NAME

`strcat`, `strncat` — concatenate strings

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <string.h>

char *
strcat(char *restrict s1, const char *restrict s2);

char *
strncat(char *restrict s1, const char *restrict s2, size_t n);
```

ADMONISHMENT

YOU SHOULD ALMOST CERTAINLY USE `strlcat`() INSTEAD.

See "Security Considerations" below.

DESCRIPTION

The `strcat`() and `strncat`() functions append a copy of the null-terminated string *s2* to the end of the null-terminated string *s1*, then add a terminating '`\0`'. The string *s1* must have sufficient space to hold the result.

The `strncat`() function appends not more than *n* characters from *s2*, and then adds a terminating '`\0`'.

The source and destination strings should not overlap, as the behavior is undefined.

RETURN VALUES

The `strcat`() and `strncat`() functions return the pointer *s1*.

SECURITY CONSIDERATIONS

The `strcat`() function is easily misused in a manner which enables malicious users to arbitrarily change a running program's functionality through a buffer overflow attack. (See the FSA.)

Avoid using `strcat`(). Instead, use `strncat`() or `strlcat`() and ensure that no more characters are copied to the destination buffer than it can hold.

Note that `strncat`() can also be problematic. It may be a security concern for a string to be truncated at all. Since the truncated string will not be as long as the original, it may refer to a completely different resource and usage of the truncated resource could result in very incorrect behavior. Example:

```
void
foo(const char *arbitrary_string)
{
        char onstack[8] = "";

#if defined(BAD)
        /*
         * This first strcat is bad behavior.  Do not use strcat!
         */
        (void)strcat(onstack, arbitrary_string);        /* BAD! */
#elif defined(BETTER)
        /*
         * The following two lines demonstrate better use of
         * strncat().
         */
        (void)strncat(onstack, arbitrary_string,
            sizeof(onstack) - strlen(onstack) - 1);
#elif defined(BEST)
        /*
         * These lines are even more robust due to testing for
         * truncation.
         */
        if (strlen(arbitrary_string) + 1 >
```

```
            sizeof(onstack) - strlen(onstack))
                err(1, "onstack would be truncated");
        (void)strncat(onstack, arbitrary_string,
            sizeof(onstack) - strlen(onstack) - 1);
#endif
}
```

SEE ALSO

bcopy(3), memccpy(3), memcpy(3), memmove(3), strcpy(3), strlcat(3), strlcpy(3), wcscat(3)

STANDARDS

The `strcat`() and `strncat`() functions conform to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/strncat.3`

strcmp, strncmp

strncmp(3)

compare strings

NAME

`strcmp`, `strncmp` — compare strings

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <string.h>

int
strcmp(const char *s1, const char *s2);

int
strncmp(const char *s1, const char *s2, size_t n);
```

DESCRIPTION

The `strcmp`() and `strncmp`() functions lexicographically compare the null-terminated strings *s1* and *s2*.

The `strncmp`() function compares not more than *n* characters. Because `strncmp`() is designed for comparing strings rather than binary data, characters that appear after a '`\0`' character are not compared.

RETURN VALUES

The `strcmp`() and `strncmp`() functions return an integer greater than, equal to, or less than 0, according as the string *s1* is greater than, equal to, or less than the string *s2*. The comparison is done using unsigned characters, so that '`\200`' is greater than '`\0`'.

SEE ALSO

bcmp(3), memcmp(3), strcasecmp(3), strcoll(3), strxfrm(3), wcscmp(3)

STANDARDS

The `strcmp`() and `strncmp`() functions conform to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/strncmp.3`

stpcpy, stpncpy, strcpy, strncpy

strncpy(3)

copy strings

NAME

`stpcpy, stpncpy, strcpy`, `strncpy` — copy strings

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <string.h>

char *
stpcpy(char * dst, const char * src);

char *
stpncpy(char * dst, const char * src, size_t len);

char *
strcpy(char * dst, const char * src);

char *
strncpy(char * dst, const char * src, size_t len);
```

ADMONISHMENT

YOU SHOULD ALMOST CERTAINLY USE `strlcpy`() INSTEAD.

See "Examples" below.

DESCRIPTION

The `stpcpy`() and `strcpy`() functions copy the string *src* to *dst* (including the terminating '`\0`' character.)

The `stpncpy`() and `strncpy`() functions copy at most *len* characters from *src* into *dst*. If *src* is less than *len* characters long, the remainder of *dst* is filled with '`\0`' characters. Otherwise, *dst* is *not* terminated.

The source and destination strings should not overlap, as the behavior is undefined.

RETURN VALUES

The `strcpy`() and `strncpy`() functions return *dst*. The `stpcpy`() and `stpncpy`() functions return a pointer to the terminating '`\0`' character of *dst*. If `stpncpy`() does not terminate *dst* with a `NUL` character, it instead returns a pointer to `dst[n]` (which does not necessarily refer to a valid memory location.)

EXAMPLES

The following sets `chararray` to "`abc\0\0\0`":

```
char chararray[6];

(void)strncpy(chararray, "abc", sizeof(chararray));
```

The following sets `chararray` to "`abcdef`":

```
char chararray[6];

(void)strncpy(chararray, "abcdefgh", sizeof(chararray));
```

Note that it does *not* NUL terminate `chararray` because the length of the source string is greater than or equal to the length argument.

The following copies as many characters from `input` to `buf` as will fit and NUL terminates the result. Because `strncpy`() does *not* guarantee to NUL terminate the string itself, this must be done explicitly.

```
char buf[1024];

(void)strncpy(buf, input, sizeof(buf) - 1);
buf[sizeof(buf) - 1] = '\0';
```

This could be better achieved using `strlcpy(3)`, as shown in the following example:

```
(void)strlcpy(buf, input, sizeof(buf));
```

Note that because `strlcpy(3)` is not defined in any standards, it should only be used when portability is not a concern.

SEE ALSO

`bcopy(3)`, `memccpy(3)`, `memcpy(3)`, `memmove(3)`, `strlcpy(3)`, `wcscpy(3)`

STANDARDS

The `strcpy`() and `strncpy`() functions conform to ISO/IEC 9899:1990 (“ISO C90”). The `stpcpy`() and `stpncpy`() functions conform to IEEE Std 1003.1-2008 (“POSIX.1”).

HISTORY

The `stpcpy`() function first appeared in FreeBSD 4.4, and `stpncpy`() was added in FreeBSD 8.0.

SECURITY CONSIDERATIONS

The `strcpy`() function is easily misused in a manner which enables malicious users to arbitrarily change a running program's functionality through a buffer overflow attack.

Source: `resources/man/man3/strncpy.3`

strnlen

strnlen(3)

bounded string length

NAME

`strnlen` — bounded string length

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <string.h>

size_t
strnlen(const char *s, size_t maxlen);
```

DESCRIPTION

Like strlen(3) but never scans past *maxlen* bytes.

SEE ALSO

strlen(3)

Source: `resources/man/man3/strnlen.3`

strstr

strstr(3)

locate a substring in a string

NAME

`strstr` — locate a substring in a string

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <string.h>

char *
strstr(const char *haystack, const char *needle);
```

DESCRIPTION

The `strstr`() function locates the first occurrence of the null-terminated string *needle* in the null-terminated string *haystack*.

RETURN VALUES

If *needle* is an empty string, *haystack* is returned; if *needle* occurs nowhere in *haystack*, `NULL` is returned; otherwise a pointer to the first character of the first occurrence of *needle* is returned.

SEE ALSO

`strchr(3)`, `strrchr(3)`, `strcmp(3)`, `strncmp(3)`

STANDARDS

The `strstr`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/strstr.3`

strtok

strtok(3)

string tokens

NAME

`strtok` — string tokens

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <string.h>

char *
strtok(char *str, const char *sep);
```

DESCRIPTION

The `strtok`() function is used to isolate sequential tokens in a null-terminated string, *str*. These tokens are separated in the string by at least one of the characters in *sep*. The first time that `strtok`() is called, *str* should be specified; subsequent calls, wishing to obtain further tokens from the same string, should pass a null pointer instead. The separator string, *sep*, must be supplied each time, and may change between calls.

The `strtok`() function modifies its first argument by replacing delimiter characters with null bytes.

RETURN VALUES

The `strtok`() function returns a pointer to the beginning of each subsequent token in the string, after replacing the separator character following it with a null byte. When no more tokens remain, a null pointer is returned.

SEE ALSO

strchr(3), strstr(3), strcmp(3)

STANDARDS

The `strtok`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

BUGS

The `strtok`() function uses a static variable to maintain state between calls, so it is not reentrant.

Source: `resources/man/man3/strtok.3`

isalnum

isalnum(3)

alphanumeric character test

NAME

`isalnum` — alphanumeric character test

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <ctype.h>

int
isalnum(int c);
```

DESCRIPTION

The `isalnum`() function tests for any character for which `isalpha(3)` or `isdigit(3)` is true.

RETURN VALUES

The `isalnum`() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

`isalpha(3)`, `isdigit(3)`, `ctype(3)`

STANDARDS

The `isalnum`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/isalnum.3`

isalpha

isalpha(3)

alphabetic character test

NAME

`isalpha` — alphabetic character test

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <ctype.h>

int
isalpha(int c);
```

DESCRIPTION

The `isalpha`() function tests for any character for which `isupper(3)` or `islower(3)` is true. In the ASCII character set, this includes the following characters (listed in their numeric order):

`A-Z` a-z

RETURN VALUES

The `isalpha`() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

`isalnum(3)`, `isdigit(3)`, `islower(3)`, `isupper(3)`, ctype(3)

STANDARDS

The `isalpha`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/isalpha.3`

iscntrl

iscntrl(3)

control character test

NAME

`iscntrl` — control character test

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <ctype.h>

int
iscntrl(int c);
```

DESCRIPTION

The `iscntrl`() function tests for any control character. Control characters are those with ASCII values 0x00 through 0x1F and 0x7F (DEL).

RETURN VALUES

The `iscntrl`() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

isprint(3), isspace(3), ctype(3)

STANDARDS

The `iscntrl`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/iscntrl.3`

isdigit

isdigit(3)

decimal-digit character test

NAME

`isdigit` — decimal-digit character test

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <ctype.h>

int
isdigit(int c);
```

DESCRIPTION

The `isdigit`() function tests for a decimal digit character (0-9).

RETURN VALUES

The `isdigit`() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

isalnum(3), isalpha(3), isxdigit(3), ctype(3)

STANDARDS

The `isdigit`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/isdigit.3`

islower

lower-case character test

NAME

`islower` — lower-case character test

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <ctype.h>

int
islower(int c);
```

DESCRIPTION

The `islower`() function tests for any lower-case letter (a-z).

RETURN VALUES

The `islower`() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

isupper(3), isalpha(3), tolower(3), ctype(3)

STANDARDS

The `islower`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/islower.3`

isprint

isprint(3)

printing character test (including space)

NAME

`isprint` — printing character test (including space)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <ctype.h>

int
isprint(int c);
```

DESCRIPTION

The `isprint`() function tests for any printing character, including space. This is any character for which `iscntrl(3)` is false. The printable characters are those with ASCII values 0x20 (space) through 0x7E (~).

RETURN VALUES

The `isprint`() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

`iscntrl(3)`, `isalnum(3)`, `ispunct(3)`, `ctype(3)`

STANDARDS

The `isprint`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/isprint.3`

ispunct

ispunct(3)

punctuation character test

NAME

`ispunct` — punctuation character test

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <ctype.h>

int
ispunct(int c);
```

DESCRIPTION

The `ispunct`() function tests for any printing character that is not a space or a character for which `isalnum(3)` is true.

RETURN VALUES

The `ispunct`() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

`isalnum(3)`, `isprint(3)`, `isspace(3)`, `ctype(3)`

STANDARDS

The `ispunct`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/ispunct.3`

isspace

isspace(3)

white-space character test

NAME

`isspace` — white-space character test

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <ctype.h>

int
isspace(int c);
```

DESCRIPTION

The `isspace`() function tests for the standard white-space characters. These are:

	horizontal tab
	newline
	vertical tab
	form feed
	carriage return
" "	space

RETURN VALUES

The `isspace`() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

isalnum(3), isalpha(3), isprint(3), ctype(3)

STANDARDS

The `isspace`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/isspace.3`

isupper

isupper(3)

upper-case character test

NAME

`isupper` — upper-case character test

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <ctype.h>

int
isupper(int c);
```

DESCRIPTION

The `isupper`() function tests for any upper-case letter (A-Z).

RETURN VALUES

The `isupper`() function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

islower(3), isalpha(3), toupper(3), ctype(3)

STANDARDS

The `isupper`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/isupper.3`

isxdigit

isxdigit(3)

hexadecimal-digit character test

NAME

`isxdigit` — hexadecimal-digit character test

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <ctype.h>

int
isxdigit(int c);
```

DESCRIPTION

The `isxdigit` () function tests for any hexadecimal-digit character. These are: 0-9, A-F, and a-f.

RETURN VALUES

The `isxdigit` () function returns zero if the character tests false and returns non-zero if the character tests true.

SEE ALSO

isdigit(3), isalnum(3), ctype(3)

STANDARDS

The `isxdigit` () function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/isxdigit.3`

tolower

tolower(3)

upper case to lower case letter conversion

NAME

`tolower` — upper case to lower case letter conversion

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <ctype.h>

int
tolower(int c);
```

DESCRIPTION

The `tolower`() function converts an upper-case letter to the corresponding lower-case letter. The argument must be representable as an `unsigned char` or the value of `EOF`.

RETURN VALUES

If the argument is an upper-case letter, the `tolower`() function returns the corresponding lower-case letter if there is one; otherwise, the argument is returned unchanged.

SEE ALSO

`toupper(3)`, `isupper(3)`, `islower(3)`, `ctype(3)`

STANDARDS

The `tolower`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/tolower.3`

toupper

toupper(3)

lower case to upper case letter conversion

NAME

`toupper` — lower case to upper case letter conversion

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <ctype.h>

int
toupper(int c);
```

DESCRIPTION

The `toupper`() function converts a lower-case letter to the corresponding upper-case letter. The argument must be representable as an `unsigned char` or the value of `EOF`.

RETURN VALUES

If the argument is a lower-case letter, the `toupper`() function returns the corresponding upper-case letter if there is one; otherwise, the argument is returned unchanged.

SEE ALSO

`tolower(3)`, `isupper(3)`, `islower(3)`, `ctype(3)`

STANDARDS

The `toupper`() function conforms to ISO/IEC 9899:1990 (“ISO C90”).

Source: `resources/man/man3/toupper.3`

abort

abort(3)

cause abnormal program termination

NAME

`abort` — cause abnormal program termination

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdlib.h>

void
abort(void);
```

DESCRIPTION

The `abort`() function causes abnormal program termination to occur, unless the signal `SIGABRT` is being caught and the signal handler does not return.

Any open streams are flushed and closed.

IMPLEMENTATION NOTES

The `abort`() function is thread-safe. It is unknown if it is async-cancel-safe.

The `abort`() function causes a report to be generated by Crash Reporter. If you wish to terminate without generating a crash report, use `exit(3)` instead.

RETURN VALUES

The `abort`() function never returns.

SEE ALSO

sigaction(2), exit(3)

STANDARDS

The `abort`() function conforms to IEEE Std 1003.1-1990 (“POSIX.1”). The `abort`() function also conforms to ISO/IEC 9899:1999 (“ISO C99”) with the implementation specific details as noted above.

Source: `resources/man/man3/abort.3`

abs

abs(3)

integer absolute value function

NAME

`abs` — integer absolute value function

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdlib.h>

int
abs(int i);
```

DESCRIPTION

The `abs`() function computes the absolute value of the integer *i*.

RETURN VALUES

The `abs`() function returns the absolute value.

SEE ALSO

`cabs(3)`, `fabs(3)`, `floor(3)`, `hypot(3)`, `imaxabs(3)`, `labs(3)`, `llabs(3)`, `math(3)`

STANDARDS

The `abs`() function conforms to ISO/IEC 9899:1999 (“ISO C99”).

BUGS

The absolute value of the most negative integer remains negative.

Source: `resources/man/man3/abs.3`

atoi, atol

atoi(3)

convert string to integer

NAME

`atoi`, `atol` — convert string to integer

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdlib.h>

int
atoi(const char *str);

long
atol(const char *str);
```

DESCRIPTION

See strtol(3).

SEE ALSO

strtol(3)

Source: `resources/man/man3/atoi.3`

atol

atol(3)

convert string to long

NAME

`atol` — convert string to long

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdlib.h>

long
atol(const char *str);
```

DESCRIPTION

See strtol(3).

SEE ALSO

strtol(3)

Source: `resources/man/man3/atol.3`

calloc

calloc(3)

allocate zeroed memory (deferred)

NAME

`calloc` — allocate zeroed memory (deferred)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdlib.h>

void *
calloc(size_t count, size_t size);
```

DESCRIPTION

Heap allocation (`malloc`, `free`, `calloc`, `realloc`) is deferred. Programs should use static/global buffers, scratchpad, or cart RAM instead. `asprintf`() is also unavailable for the same reason.

RETURN VALUES

These functions are not provided; linking a call will fail.

SEE ALSO

abs(3), qsort(3)

Source: `resources/man/man3/calloc.3`

exit, _Exit

exit(3)

perform normal program termination

NAME

`exit`, `_Exit` — perform normal program termination

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdlib.h>

void
exit(int status);

void
_Exit(int status);
```

DESCRIPTION

The `exit`() and `_Exit`() functions terminate a process.

Before termination, `exit`() performs the following functions in the order listed:

1. Call the functions registered with the `atexit(3)` function, in the reverse order of their registration.
2. Flush all open output streams.
3. Close all open streams.
4. Unlink all files created with the `tmpfile(3)` function.

The `_Exit`() function terminates without calling the functions registered with the `atexit(3)` function, and may or may not perform the other actions listed. Both functions make the low-order eight bits of the *status* argument available to a parent process which has called a `wait(2)`-family function.

The C Standard (ISO/IEC 9899:1999 (“ISO C99”)) defines the values `0`, `EXIT_SUCCESS`, and `EXIT_FAILURE` as possible values of *status*. Cooperating processes may use other values; in a program which might be called by a mail transfer agent, the values described in `sysexits(3)` may be used to provide more information to the parent process.

Note that `exit`() does nothing to prevent bottomless recursion should a function registered using `atexit(3)` itself call `exit`(). Such functions must call `_Exit`() instead (although this has other effects as well which may not be desired).

RETURN VALUES

The `exit`() and `_Exit`() functions never return.

SEE ALSO

`_exit(2)`, `wait(2)`, `atexit(3)`, `intro(3)`, `sysexits(3)`, `tmpfile(3)`

STANDARDS

The `exit`() and `_Exit`() functions conform to ISO/IEC 9899:1999 (“ISO C99”).

Source: `resources/man/man3/exit.3`

free

free(3)

free memory (deferred)

NAME

`free` — free memory (deferred)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdlib.h>

void
free(void *ptr);
```

DESCRIPTION

Heap allocation (`malloc`, `free`, `calloc`, `realloc`) is deferred. Programs should use static/global buffers, scratchpad, or cart RAM instead. `asprintf`() is also unavailable for the same reason.

RETURN VALUES

These functions are not provided; linking a call will fail.

SEE ALSO

abs(3), qsort(3)

Source: `resources/man/man3/free.3`

malloc

malloc(3)

allocate memory (deferred)

NAME

`malloc` — allocate memory (deferred)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdlib.h>

void *
malloc(size_t size);
```

DESCRIPTION

Heap allocation (`malloc`, `free`, `calloc`, `realloc`) is deferred. Programs should use static/global buffers, scratchpad, or cart RAM instead. `asprintf`() is also unavailable for the same reason.

RETURN VALUES

These functions are not provided; linking a call will fail.

SEE ALSO

abs(3), qsort(3)

Source: `resources/man/man3/malloc.3`

qsort

qsort(3)

sort an array

NAME

`qsort` — sort an array

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdlib.h>

void
qsort(void *base, size_t nel, size_t width, int (*compar)(const void *, const void *));
```

DESCRIPTION

Sorts *nel* elements of *width* bytes each. Stability is not guaranteed. Uses a small in-place algorithm suitable for 16-bit `size_t`.

SEE ALSO

abs(3)

Source: `resources/man/man3/qsort.3`

rand, rand_r, srand, sranddev

rand(3)

bad random number generator

NAME

`rand`, `rand_r`, `srand`, `sranddev` — bad random number generator

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdlib.h>

int
rand(void);

int
rand_r(unsigned *seed);

void
srand(unsigned seed);

void
sranddev(void);
```

DESCRIPTION

arc4random(3)

The `rand`() function computes a sequence of pseudo-random integers in the range of 0 to `RAND_MAX` (as defined by the header file `<stdlib.h>`).

The `srand`() function sets its argument *seed* as the seed for a new sequence of pseudo-random numbers to be returned by `rand`(). These sequences are repeatable by calling `srand`() with the same seed value.

If no *seed* value is provided, the functions are automatically seeded with a value of 1.

The `sranddev`() function initializes a seed, using the `random(4)` random number device which returns good random numbers. However, the `rand`() function still remains unsuitable for cryptographic use.

The `rand_r`() function provides the same functionality as `rand`(). A pointer to the context value *seed* must be supplied by the caller.

SEE ALSO

`arc4random(3)`, `random(3)`, `random(4)`

STANDARDS

The `rand`() and `srand`() functions conform to ISO/IEC 9899:1990 (“ISO C90”).

The `rand_r`() function is as proposed in the POSIX.4a Draft #6 document.

Source: `resources/man/man3/rand.3`

realloc

realloc(3)

resize allocation (deferred)

NAME

`realloc` — resize allocation (deferred)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdlib.h>

void *
realloc(void *ptr, size_t size);
```

DESCRIPTION

Heap allocation (`malloc`, `free`, `calloc`, `realloc`) is deferred. Programs should use static/global buffers, scratchpad, or cart RAM instead. `asprintf`() is also unavailable for the same reason.

RETURN VALUES

These functions are not provided; linking a call will fail.

SEE ALSO

abs(3), qsort(3)

Source: `resources/man/man3/realloc.3`

rand, rand_r, srand, sranddev

srand(3)

bad random number generator

NAME

`rand`, `rand_r`, `srand`, `sranddev` — bad random number generator

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdlib.h>

int
rand(void);

int
rand_r(unsigned *seed);

void
srand(unsigned seed);

void
sranddev(void);
```

DESCRIPTION

arc4random(3)

The `rand`() function computes a sequence of pseudo-random integers in the range of 0 to `RAND_MAX` (as defined by the header file `<stdlib.h>`).

The `srand`() function sets its argument *seed* as the seed for a new sequence of pseudo-random numbers to be returned by `rand`(). These sequences are repeatable by calling `srand`() with the same seed value.

If no *seed* value is provided, the functions are automatically seeded with a value of 1.

The `sranddev`() function initializes a seed, using the `random(4)` random number device which returns good random numbers. However, the `rand`() function still remains unsuitable for cryptographic use.

The `rand_r`() function provides the same functionality as `rand`(). A pointer to the context value *seed* must be supplied by the caller.

SEE ALSO

`arc4random(3)`, `random(3)`, `random(4)`

STANDARDS

The `rand`() and `srand`() functions conform to ISO/IEC 9899:1990 (“ISO C90”).

The `rand_r`() function is as proposed in the POSIX.4a Draft #6 document.

Source: `resources/man/man3/srand.3`

strtol, strtoul, atoi, atol

strtol(3)

convert string to integer

NAME

`strtol`, `strtoul`, `atoi`, `atol` — convert string to integer

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdlib.h>

long
strtol(const char *str, char **endptr, int base);

unsigned long
strtoul(const char *str, char **endptr, int base);

int
atoi(const char *str);

long
atol(const char *str);
```

DESCRIPTION

`long` is 32-bit on this target. *base* 0 means auto-detect (`0x`) hex, leading `0` octal, else decimal. No `errno` is set; overflow clamps to the type range. `atoi`() is `(int)strtol`(*str*, *NULL*, *10*).

SEE ALSO

abs(3)

Source: `resources/man/man3/strtol.3`

strtoul

strtoul(3)

convert string to unsigned long

NAME

`strtoul` — convert string to unsigned long

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <stdlib.h>

unsigned long
strtoul(const char *str, char **endptr, int base);
```

DESCRIPTION

See strtol(3).

SEE ALSO

strtol(3)

Source: `resources/man/man3/strtoul.3`

atanf

atanf(3)

inverse tangent (float)

NAME

`atanf` — inverse tangent (float)

LIBRARY

`<math.h>`

SYNOPSIS

```
float
atanf(float x);
```

DESCRIPTION

The `atanf`() function computes the principal value of the arc tangent of *x*.

RETURN VALUES

The arc tangent of *x* in the range [-π/2, +π/2] radians.

SEE ALSO

sinf(3), tanf(3)

Source: `resources/man/man3/atanf.3`

floorf, ceilf

ceilf(3)

floor and ceiling functions (float)

NAME

`floorf`, `ceilf` — floor and ceiling functions (float)

LIBRARY

`<math.h>`

SYNOPSIS

```
float
floorf(float x);

float
ceilf(float x);
```

DESCRIPTION

The `floorf`() function returns the largest integer value not greater than *x*.

The `ceilf`() function returns the smallest integer value not less than *x*.

RETURN VALUES

`floorf`() returns floor(x); `ceilf`() returns ceil(x). Both return float values.

EXAMPLES

```
floorf(3.7f) /* returns 3.0f */
```

```
floorf(-3.7f) /* returns -4.0f */
```

```
ceilf(3.2f) /* returns 4.0f */
```

```
ceilf(-3.2f) /* returns -3.0f */
```

SEE ALSO

fabsf(3), fmodf(3)

Source: `resources/man/man3/ceilf.3`

sinf, cosf, tanf

cosf(3)

trigonometric functions (float)

NAME

`sinf`, `cosf`, `tanf` — trigonometric functions (float)

LIBRARY

`<math.h>`

SYNOPSIS

```
float
sinf(float x);

float
cosf(float x);

float
tanf(float x);
```

DESCRIPTION

The `sinf`(), `cosf`(), and `tanf`() functions compute the sine, cosine, and tangent of *x*, where *x* is given in radians.

RETURN VALUES

The trigonometric value of *x*.

SEE ALSO

atanf(3)

Source: `resources/man/man3/cosf.3`

expf

expf(3)

exponential function (float)

NAME

`expf` — exponential function (float)

LIBRARY

`<math.h>`

SYNOPSIS

```
float
expf(float x);
```

DESCRIPTION

The `expf`() function computes the base-e exponential of *x*.

RETURN VALUES

The value e^x.

SEE ALSO

logf(3), powf(3)

Source: `resources/man/man3/expf.3`

fabsf

fabsf(3)

absolute value (float)

NAME

`fabsf` — absolute value (float)

LIBRARY

`<math.h>`

SYNOPSIS

```
float
fabsf(float x);
```

DESCRIPTION

The `fabsf`() function computes the absolute value of the floating-point number *x*.

RETURN VALUES

The absolute value of *x*.

SEE ALSO

abs(3), fmodf(3)

Source: `resources/man/man3/fabsf.3`

floorf, ceilf

floorf(3)

floor and ceiling functions (float)

NAME

floorf, ceilf — floor and ceiling functions (float)

LIBRARY

<math.h>

SYNOPSIS

```
float
floorf(float x);

float
ceilf(float x);
```

DESCRIPTION

The floorf() function returns the largest integer value not greater than *x*.

The ceilf() function returns the smallest integer value not less than *x*.

RETURN VALUES

floorf() returns floor(x); ceilf() returns ceil(x). Both return float values.

EXAMPLES

```
floorf(3.7f) /* returns 3.0f */
```

```
floorf(-3.7f) /* returns -4.0f */
```

```
ceilf(3.2f) /* returns 4.0f */
```

```
ceilf(-3.2f) /* returns -3.0f */
```

SEE ALSO

fabsf(3), fmodf(3)

Source: `resources/man/man3/floorf.3`

fmodf

fmodf(3)

floating-point remainder

NAME

`fmodf` — floating-point remainder

LIBRARY

`<math.h>`

SYNOPSIS

```
float
fmodf(float x, float y);
```

DESCRIPTION

The `fmodf`() function computes the floating-point remainder of *x* divided by *y*. The result has the same sign as *x*.

RETURN VALUES

The remainder x - n*y, where n is the quotient x/y truncated toward zero.

SEE ALSO

fabsf(3)

Source: `resources/man/man3/fmodf.3`

logf

logf(3)

natural logarithm (float)

NAME

`logf` — natural logarithm (float)

LIBRARY

`<math.h>`

SYNOPSIS

```
float
logf(float x);
```

DESCRIPTION

The `logf`() function computes the natural (base-e) logarithm of *x*.

RETURN VALUES

The natural logarithm of *x*. If *x* is zero or negative, the result is undefined.

SEE ALSO

expf(3), powf(3)

Source: `resources/man/man3/logf.3`

powf

power function (float)

NAME

`powf` — power function (float)

LIBRARY

`<math.h>`

SYNOPSIS

```
float
powf(float x, float y);
```

DESCRIPTION

The `powf`() function computes *x* raised to the power *y*. Implemented as expf(y * logf(x)).

RETURN VALUES

The value x^y.

SEE ALSO

`expf(3)`, `logf(3)`, `sqrtf(3)`

Source: `resources/man/man3/powf.3`

sinf, cosf, tanf

sinf(3)

trigonometric functions (float)

NAME

`sinf`, `cosf`, `tanf` — trigonometric functions (float)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <math.h>

float
sinf(float x);

float
cosf(float x);

float
tanf(float x);
```

DESCRIPTION

IEEE 754 binary32 soft-float implementations. *x* is in radians. Accuracy is modest; prefer fixed-point for hot loops.

SEE ALSO

atanf(3), sqrtf(3)

Source: `resources/man/man3/sinf.3`

sqrtf

sqrtf(3)

square root (float)

NAME

`sqrtf` — square root (float)

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <math.h>

float
sqrtf(float x);
```

DESCRIPTION

IEEE soft-float square root. Negative inputs return 0.0f.

SEE ALSO

powf(3), fabsf(3)

Source: `resources/man/man3/sqrtf.3`

sinf, cosf, tanf

tanf(3)

trigonometric functions (float)

NAME

`sinf`, `cosf`, `tanf` — trigonometric functions (float)

LIBRARY

`<math.h>`

SYNOPSIS

```
float
sinf(float x);

float
cosf(float x);

float
tanf(float x);
```

DESCRIPTION

The `sinf`(), `cosf`(), and `tanf`() functions compute the sine, cosine, and tangent of *x*, where *x* is given in radians.

RETURN VALUES

The trigonometric value of *x*.

SEE ALSO

atanf(3)

Source: `resources/man/man3/tanf.3`

libc

libc(3)

TI-99/4A C library overview

NAME

`libc` — TI-99/4A C library overview

DESCRIPTION

Tidepod/tms9900-gcc `libc` is a freestanding-friendly C library built as tiny translation units so the packer can bank unused symbols away.

Provided: string and ctype; stdlib without a heap (`atoi`, `strtol`, `rand`, `qsort`, `exit`); console `printf` / `putchar` and conio(3) cursor I/O on the VDP; keyboard (`cgetc`, `kbhit`); ti_vdp_detect(3) for F18A/Pico9918 and 80-column text; mathf via soft-float; disk fopen(3) family (mode grammar with `:df|:dv|:if|:iv|:pg`), FDR probe on existing files, PROGRAM as AU byte streams, `fcopy_path`; catalog ti_opendir(3), DSR unit discovery ti_dsr_enum(3), metadata (`ti_stat`, `ti_protect`, `ti_mkdir`, `ti_rmdir`), and remove(3) / `rename`. Default `fopen`() is Level-3 PAB only; native chip / FAT / WDS / PROGRAM-AU backends are opt-in via stdio_enable(3) (see) `docs/stdio_backends.md`. Disk path APIs copy path/mode into stack scratch at entry so banked callers remain safe across trampolines (packer also hoists string literals for trampoline TUs; see) `docs/rom_architecture.md`. HFDC Level-2 sector DMA uses stdio_set_wds_vdp_buf(3) (default `0x3DA8`).

after a failed `fopen` / `fcopy_path` / `remove` / `ti_opendir`, read stdio_last_error(3) (DSR status; see also) `docs/stdio_errors.md`. `errno` is secondary and not always updated for Level-3 PAB failures.

Deferred: `malloc`, `free`, `calloc`, `realloc`, `asprintf`.

Type model: `int` / `size_t` are 16-bit; `long` is 32-bit; `float` is IEEE binary32 soft-float.

SEE ALSO

conio(3), printf(3), fopen(3), stdio_last_error(3), stdio_enable(3), ti_vdp_detect(3), ti_opendir(3), ti_dsr_enum(3), malloc(3)

Source: `resources/man/man3/libc.3`

ASC

asc(3xb)

return ASCII code of a string's first character

NAME

`ASC` — return ASCII code of a string's first character

SYNOPSIS

`ASC`(*string$*);

DESCRIPTION

The `ASC`() function returns the ASCII code (0-255) of the first character in the provided string. If the string is empty, the behavior may be undefined or return 0.

EXAMPLES

```
100 C = ASC("HELLO")
```

SEE ALSO

chr(3xb)

Source: `resources/man/man3/asc.3xb`

CHAR

char(3xb)

redefine a character's graphical pattern

NAME

CHAR — redefine a character's graphical pattern

SYNOPSIS

```
CALL CHAR(char_code, pattern$);
```

DESCRIPTION

The CALL CHAR() statement redefines the visual representation of a character. *char_code* is the ASCII code of the character to redefine (32-143). *pattern$* is a 16-character hexadecimal string representing the 8×8 pixel grid.

EXAMPLES

```
100 CALL CHAR(33, "FFFFFFFFFFFFFFFF")
```

SEE ALSO

charset(3xb)

Source: resources/man/man3/char.3xb

CHARSET

charset(3xb)

restore standard character patterns

NAME

CHARSET — restore standard character patterns

SYNOPSIS

```
CALL CHARSET();
```

DESCRIPTION

The CALL CHARSET() statement resets characters 32 through 95 to their default TI-99/4A patterns, erasing any custom definitions created by CALL CHAR for those characters.

EXAMPLES

```
100 CALL CHARSET
```

SEE ALSO

char(3xb)

Source: resources/man/man3/charset.3xb

CHR$

chr(3xb)

return character corresponding to ASCII code

NAME

`CHR$` — return character corresponding to ASCII code

SYNOPSIS

`CHR$` (*ascii_code*);

DESCRIPTION

The `CHR$` () function returns a string consisting of a single character corresponding to the provided *ascii_code* (0-255).

EXAMPLES

```
100 A$ = CHR$(65)
```

SEE ALSO

asc(3xb)

Source: `resources/man/man3/chr.3xb`

CLEAR

clear(3xb)

clear the screen and reset the VDP cursor

NAME

`CLEAR` — clear the screen and reset the VDP cursor

SYNOPSIS

`CALL CLEAR`();

DESCRIPTION

The `CALL CLEAR`() statement clears the screen by replacing all characters with the space character (ASCII 32). It also resets the VDP cursor to the top-left corner of the screen.

EXAMPLES

```
100 CALL CLEAR
110 PRINT "SCREEN CLEARED"
```

SEE ALSO

screen(3xb)

Source: `resources/man/man3/clear.3xb`

COINC

coinc(3xb)

detect collisions between sprites or dots

NAME

COINC — detect collisions between sprites or dots

SYNOPSIS

CALL COINC(*#n1, #n2, tol, var*);

CALL COINC(*ALL, var*);

DESCRIPTION

The CALL COINC() statement checks if two sprites have collided (distance less than tolerance *tol*), or if any sprites have overlapped pixels. The result is returned in *var* (normally -1 for true, 0 for false).

EXAMPLES

```
100 CALL COINC(#1, #2, 8, HIT)
110 IF HIT THEN 500
```

SEE ALSO

distance(3xb)

Source: resources/man/man3/coinc.3xb

COLOR

color(3xb)

define foreground and background colors for a character set

NAME

`COLOR` — define foreground and background colors for a character set

SYNOPSIS

`CALL COLOR`(*character_set, foreground, background*);

DESCRIPTION

The `CALL COLOR`() statement assigns a foreground and background color to a specific character set. The TI-99/4A divides its 256 characters into 32 sets of 8 characters each.

character_set must be between 1 and 14 (or up to 32, depending on graphics mode). *foreground* and *background* must be colors from 1 to 16.

The available colors are:

1 Transparent

2 Black

3 Medium Green

4 Light Green

5 Dark Blue

6 Light Blue

7 Dark Red

8 Cyan

9 Medium Red

10 Light Red

11 Dark Yellow

12 Light Yellow

13 Dark Green

14 Magenta

15 Gray

16 White

EXAMPLES

```
100 CALL COLOR(1, 16, 2)
```

SEE ALSO

screen(3xb)

Source: `resources/man/man3/color.3xb`

DELSPRITE

delsprite(3xb)

delete a specific sprite or all active sprites

NAME

`DELSPRITE` — delete a specific sprite or all active sprites

SYNOPSIS

`CALL DELSPRITE(`*(#no | ALL)*`);`

DESCRIPTION

The `CALL DELSPRITE()` statement removes sprites from the screen. Passing a sprite number removes only that sprite. Passing *ALL* removes all currently active sprites.

EXAMPLES

```
100 CALL DELSPRITE(#1)
110 CALL DELSPRITE(ALL)
```

SEE ALSO

sprite(3xb)

Source: `resources/man/man3/delsprite.3xb`

DISPLAY AT

display_at(3xb)

print text at a specific screen location

NAME

DISPLAY AT — print text at a specific screen location

SYNOPSIS

DISPLAY AT(row, col)(*text_or_vars*);

DESCRIPTION

The DISPLAY AT() statement positions the text cursor at the specified *row* and *col*, then prints the following strings, numbers, or variables.

EXAMPLES

```
100 DISPLAY AT(12, 10):"HELLO WORLD"
```

Source: resources/man/man3/display_at.3xb

DISTANCE

distance(3xb)

compute the squared distance between two sprites

NAME

DISTANCE — compute the squared distance between two sprites

SYNOPSIS

CALL DISTANCE(*#n1, #n2, var*);

DESCRIPTION

The CALL DISTANCE() statement calculates the squared pixel distance between sprite *#n1* and sprite *#n2*, and stores the result in *var*.

EXAMPLES

```
100 CALL DISTANCE(#1, #2, DIST)
```

SEE ALSO

coinc(3xb)

Source: resources/man/man3/distance.3xb

GCHAR

gchar(3xb)

read the character code at a screen position

NAME

GCHAR — read the character code at a screen position

SYNOPSIS

```
CALL GCHAR(row, col, return_var);
```

DESCRIPTION

The CALL GCHAR() statement reads the ASCII code of the character currently displayed at the specified *row* and *col*, and stores it in *return_var*.

EXAMPLES

```
100 CALL GCHAR(12, 16, CH)
110 IF CH=32 THEN 200
```

SEE ALSO

hchar(3xb), vchar(3xb)

Source: resources/man/man3/gchar.3xb

HCHAR

hchar(3xb)

draw a horizontal line of characters

NAME

`HCHAR` — draw a horizontal line of characters

SYNOPSIS

```
CALL HCHAR(row, col, char_code, [count]);
```

DESCRIPTION

The `CALL HCHAR()` statement places a character at the specified *row* and *col*. If the optional *count* is provided, it repeats the character horizontally that many times.

EXAMPLES

```
100 CALL HCHAR(12, 16, 33, 10)
```

SEE ALSO

vchar(3xb), gchar(3xb)

Source: `resources/man/man3/hchar.3xb`

JOYST

joyst(3xb)

read joystick directional input

NAME

`JOYST` — read joystick directional input

SYNOPSIS

`CALL JOYST`(*unit, return_x, return_y*);

DESCRIPTION

The `CALL JOYST`() statement reads the directional status of joystick *unit* (1 or 2). *return_x* and *return_y* are populated with -4, 0, or 4 depending on the joystick's deflection.

EXAMPLES

```
100 CALL JOYST(1, X, Y)
```

SEE ALSO

key(3xb)

Source: `resources/man/man3/joyst.3xb`

KEY

key(3xb)

scan the keyboard or joysticks for input

NAME

`KEY` — scan the keyboard or joysticks for input

SYNOPSIS

`CALL KEY`(*unit, return_key, return_status*);

DESCRIPTION

The `CALL KEY`() statement reads the keyboard or joystick inputs without halting the program. *unit* defines which hardware to scan:

0 Console keyboard

1 Joystick 1

2 Joystick 2

3 Left side of split keyboard

4 Right side of split keyboard

5 Pascal keyboard mode

return_key is populated with the ASCII keycode of the pressed key (or 255 if none). *return_status* is populated with 1 if a new key was pressed, -1 if the same key was held, or 0 if no key is pressed.

EXAMPLES

```
100 CALL KEY(0, K, S)
110 IF S=0 THEN 100
```

SEE ALSO

joyst(3xb)

Source: `resources/man/man3/key.3xb`

LEN

`len(3xb)`

return the length of a string

NAME

`LEN` — return the length of a string

SYNOPSIS

`LEN`(*string$*);

DESCRIPTION

The `LEN`() function returns the number of characters in the provided string.

EXAMPLES

```
100 L = LEN("HELLO")
```

SEE ALSO

seg(3xb)

Source: `resources/man/man3/len.3xb`

LOAD

load(3xb)

write bytes directly into CPU RAM

NAME

LOAD — write bytes directly into CPU RAM

SYNOPSIS

CALL LOAD(*address, val1, [val2...]*);

DESCRIPTION

The CALL LOAD() statement writes one or more byte values sequentially into the CPU memory address space starting at *address*.

This statement functions exactly identically to CALL POKE().

EXAMPLES

```
100 CALL LOAD(8192, 255, 0)
```

SEE ALSO

poke(3xb)

Source: resources/man/man3/load.3xb

LOCATE

locate(3xb)

move an existing sprite to a new row and column

NAME

LOCATE — move an existing sprite to a new row and column

SYNOPSIS

CALL LOCATE(*#no, r, c*);

DESCRIPTION

The CALL LOCATE() statement immediately changes the screen position of an existing sprite. *r* and *c* are the pixel coordinates on screen (1-256).

EXAMPLES

```
100 CALL LOCATE(#1, 50, 50)
```

SEE ALSO

sprite(3xb), position(3xb)

Source: resources/man/man3/locate.3xb

MAGNIFY

magnify(3xb)

set the global sprite magnification factor

NAME

MAGNIFY — set the global sprite magnification factor

SYNOPSIS

CALL MAGNIFY(*factor*);

DESCRIPTION

The CALL MAGNIFY() statement changes the magnification of all sprites on screen. Valid factors are 1 (1×1 normal), 2 (2×2 unmagnified), 3 (1×1 magnified), and 4 (2×2 magnified).

EXAMPLES

```
100 CALL MAGNIFY(4)
```

SEE ALSO

sprite(3xb)

Source: resources/man/man3/magnify.3xb

MOTION

motion(3xb)

update the row and column velocities of a sprite

NAME

`MOTION` — update the row and column velocities of a sprite

SYNOPSIS

`CALL MOTION(#no, rv, cv);`

DESCRIPTION

The `CALL MOTION()` statement changes the speed and direction of a sprite. *rv* is the row velocity and *cv* is the column velocity (-128 to 127).

EXAMPLES

```
100 CALL MOTION(#1, 5, 5)
```

SEE ALSO

sprite(3xb)

Source: `resources/man/man3/motion.3xb`

PATTERN

pattern(3xb)

change the character pattern used by a sprite

NAME

`PATTERN` — change the character pattern used by a sprite

SYNOPSIS

`CALL PATTERN` *(#no, char)*;

DESCRIPTION

The `CALL PATTERN` () statement changes the visual graphics of an existing sprite by assigning it a new ASCII *char* code.

EXAMPLES

```
100 CALL PATTERN(#1, 43)
```

SEE ALSO

sprite(3xb)

Source: `resources/man/man3/pattern.3xb`

PEEK

peek(3xb)

read bytes from CPU RAM/ROM

NAME

PEEK — read bytes from CPU RAM/ROM

SYNOPSIS

CALL PEEK(*address, var1, [var2...]*);

DESCRIPTION

The CALL PEEK() statement reads one or more bytes sequentially from the CPU memory address space starting at *address*, and stores them in the provided variables.

EXAMPLES

```
100 CALL PEEK(8192, A, B)
```

SEE ALSO

poke(3xb), load(3xb), peekv(3xb)

Source: resources/man/man3/peek.3xb

PEEKV

peekv(3xb)

read bytes from VDP (Video) RAM

NAME

PEEKV — read bytes from VDP (Video) RAM

SYNOPSIS

CALL PEEKV(*address, var1, [var2...]*);

DESCRIPTION

The CALL PEEKV() statement reads one or more bytes sequentially from the VDP memory address space starting at *address*, and stores them in the provided variables.

EXAMPLES

```
100 CALL PEEKV(0, A, B)
```

SEE ALSO

pokev(3xb), peek(3xb)

Source: resources/man/man3/peekv.3xb

POKE

poke(3xb)

write bytes directly into CPU RAM

NAME

POKE — write bytes directly into CPU RAM

SYNOPSIS

CALL POKE(*address, val1, [val2...]*);

DESCRIPTION

The CALL POKE() statement writes one or more byte values sequentially into the CPU memory address space starting at *address*.

Note: CALL LOAD() is functionally identical and supported as a direct alias for backwards compatibility.

EXAMPLES

```
100 CALL POKE(8192, 255, 0)
```

SEE ALSO

peek(3xb), load(3xb), pokev(3xb)

Source: resources/man/man3/poke.3xb

POKEV

pokev(3xb)

write bytes directly into VDP (Video) RAM

NAME

POKEV — write bytes directly into VDP (Video) RAM

SYNOPSIS

CALL POKEV(*address, val1, [val2...]*);

DESCRIPTION

The CALL POKEV() statement writes one or more byte values sequentially into the VDP memory address space starting at *address*.

EXAMPLES

```
100 CALL POKEV(0, 255, 0)
```

SEE ALSO

peekv(3xb), poke(3xb)

Source: resources/man/man3/pokev.3xb

POSITION

position(3xb)

read the current screen coordinates of a sprite

NAME

`POSITION` — read the current screen coordinates of a sprite

SYNOPSIS

`CALL POSITION(#no, r, c)`;

DESCRIPTION

The `CALL POSITION()` statement reads the current row and column pixel coordinates of sprite *#no* into variables *r* and *c*.

EXAMPLES

```
100 CALL POSITION(#1, R, C)
```

SEE ALSO

sprite(3xb), locate(3xb)

Source: `resources/man/man3/position.3xb`

RND

rnd(3xb)

generate a pseudo-random number

NAME

`RND` — generate a pseudo-random number

SYNOPSIS

`RND`();

DESCRIPTION

The `RND`() function returns a pseudo-random floating-point number greater than or equal to 0 and less than 1.

EXAMPLES

```
100 X = INT(RND * 10) + 1
```

Source: `resources/man/man3/rnd.3xb`

SCREEN

screen(3xb)

set the background screen color

NAME

`SCREEN` — set the background screen color

SYNOPSIS

```
CALL SCREEN(color);
```

DESCRIPTION

The `CALL SCREEN()` statement sets the background color of the screen. The *color* parameter must be an integer from 1 to 16.

The available colors are:

1	Transparent
2	Black
3	Medium Green
4	Light Green
5	Dark Blue
6	Light Blue
7	Dark Red
8	Cyan
9	Medium Red
10	Light Red
11	Dark Yellow
12	Light Yellow
13	Dark Green

14 Magenta

15 Gray

16 White

EXAMPLES

```
100 CALL SCREEN(2)
```

SEE ALSO

color(3xb)

Source: `resources/man/man3/screen.3xb`

SEG$

seg(3xb)

extract a substring from a string

NAME

`SEG$` — extract a substring from a string

SYNOPSIS

`SEG$`(*string$, position, length*);

DESCRIPTION

The `SEG$`() function extracts a substring from *string$*, starting at 1-indexed *position*, for up to *length* characters.

EXAMPLES

```
100 S$ = SEG$("HELLO", 2, 3)
```

SEE ALSO

len(3xb)

Source: `resources/man/man3/seg.3xb`

SOUND

sound(3xb)

play sounds/noise through the SN76489 sound chip

NAME

`SOUND` — play sounds/noise through the SN76489 sound chip

SYNOPSIS

`CALL SOUND`(*duration, freq, vol, [freq2, vol2, freq3, vol3]*);

DESCRIPTION

The `CALL SOUND`() statement plays a tone or noise. *duration* is the time in milliseconds. A negative duration indicates that the program should not halt while the sound plays; instead, it plays asynchronously in the background. Note that asynchronous sound playback requires `CALL VSYNC`() to be called in your game loop!

freq is the frequency in Hertz (110-44733). Frequencies between -1 and -8 are special and trigger the noise channel.

vol is the volume from 0 (loudest) to 30 (quietest).

EXAMPLES

```
100 CALL SOUND(100, 440, 2)
110 CALL SOUND(-500, -7, 0)
```

SEE ALSO

vsync(3xb)

Source: `resources/man/man3/sound.3xb`

SPRITE

sprite(3xb)

define and display a sprite

NAME

`SPRITE` — define and display a sprite

SYNOPSIS

`CALL SPRITE(#no, char, color, r, c, [rv, cv])`;

DESCRIPTION

The `CALL SPRITE()` statement creates a hardware sprite. *#no* is the sprite number (1-32). *char* is the ASCII character code to use for its graphics. *color* is the sprite's color (1-16). *r* and *c* are the pixel coordinates on screen (1-256). *rv* and *cv* are optional row and column velocities for automotion (-128 to 127).

EXAMPLES

```
100 CALL SPRITE(#1, 42, 16, 128, 128, 5, 0)
```

SEE ALSO

delsprite(3xb), locate(3xb), motion(3xb)

Source: `resources/man/man3/sprite.3xb`

STR$

str(3xb)

convert numeric value to string

NAME

`STR$` — convert numeric value to string

SYNOPSIS

`STR$`(*number*);

DESCRIPTION

The `STR$`() function converts a numeric value into its string representation, adhering to standard TI BASIC formatting rules (including a leading space for positive numbers and zero).

EXAMPLES

```
100 A$ = STR$(123.45)
```

SEE ALSO

val(3xb)

Source: `resources/man/man3/str.3xb`

VAL

val(3xb)

convert string to numeric value

NAME

`VAL` — convert string to numeric value

SYNOPSIS

`VAL`(*string$*);

DESCRIPTION

The `VAL`() function converts a string representation of a number into a numeric value.

EXAMPLES

```
100 A = VAL("123.45")
```

SEE ALSO

str(3xb)

Source: `resources/man/man3/val.3xb`

VCHAR

vchar(3xb)

draw a vertical line of characters

NAME

VCHAR — draw a vertical line of characters

SYNOPSIS

```
CALL VCHAR(row, col, char_code, [count]);
```

DESCRIPTION

The CALL VCHAR() statement places a character at the specified *row* and *col*. If the optional *count* is provided, it repeats the character vertically that many times.

EXAMPLES

```
100 CALL VCHAR(12, 16, 33, 10)
```

SEE ALSO

hchar(3xb), gchar(3xb)

Source: `resources/man/man3/vchar.3xb`

VSYNC

vsync(3xb)

halt execution until the next VDP vertical blanking interrupt

NAME

`VSYNC` — halt execution until the next VDP vertical blanking interrupt

SYNOPSIS

`CALL VSYNC`();

DESCRIPTION

The `CALL VSYNC`() statement synchronizes program execution with the screen refresh rate (60Hz NTSC / 50Hz PAL).

This statement is *new* to this Extended Basic compiler and libxb. In compiled programs, you *must* call `CALL VSYNC`() inside your main game loops. It is responsible for processing background tasks, including:

- Sprite automotion (CALL MOTION)
- Asynchronous sound updates (CALL SOUND with negative durations)

EXAMPLES

```
100 CALL VSYNC
110 GOTO 100
```

Source: `resources/man/man3/vsync.3xb`

xb-control

xb-control(7)

XB BASIC control flow statements

NAME

`xb-control` — XB BASIC control flow statements

DESCRIPTION

Branching, looping, and conditional execution.

IF / THEN / ELSE / ELSEIF / ENDIF

Conditional execution. Single-line form:

```
IF condition THEN statement :: statement
```

Multi-line (structured) form:

```
IF condition THEN
  statements
ELSEIF condition THEN
  statements
ELSE
  statements
ENDIF
```

FOR / TO / STEP / NEXT

Counted loop. *STEP* defaults to 1 if omitted.

```
FOR I%=1 TO 10 STEP 2
  CALL HCHAR(I%,1,42)
NEXT I%
```

WHILE / WEND

Pre-tested loop. The body executes while *condition* is true.

```
WHILE RUNNING%
  ! game loop
WEND
```

REPEAT / UNTIL

Post-tested loop. The body executes at least once, repeating until *condition* becomes true.

```
REPEAT
  CALL KEY(0,K%,S%)
UNTIL S%<>0
```

GOTO label

Transfer control to a line number or @label.

```
GOTO 100
```

```
GOTO @MAIN_LOOP
```

GOSUB label / RETURN

Call a subroutine at a line number or @label. `RETURN` transfers back to the statement after the GOSUB.

```
GOSUB @DRAW_SCREEN
! ...
@DRAW_SCREEN:
  CALL CLEAR
  RETURN
```

ON expr GOTO label1, label2, ...

ON expr GOSUB label1, label2, ...

Computed branch. If *expr* evaluates to 1, go to the first label; if 2, the second; etc.

```
ON CHOICE% GOTO @OPT1,@OPT2,@OPT3
```

SEE ALSO

xb-sub(7), xb-variables(7), xb-misc(7)

Source: `resources/man/man7/xb-control.7`

xb-data

xb-data(7)

XB BASIC DATA, READ, RESTORE, and BYTES directives

NAME

`xb-data` — XB BASIC DATA, READ, RESTORE, and BYTES directives

DESCRIPTION

Inline data storage and retrieval.

DATA val1, val2, ...

Store constant values in a program-wide data pool. Values can be integers, floats, or quoted strings.

```
DATA 10,20,30,"HELLO",42
```

READ var1, var2, ...

Read the next value(s) from the DATA pool into variables. Types must match: read integers into %, strings into $.

```
READ X%,Y%,Z%,MSG$,N%
```

RESTORE

Reset the DATA read pointer to the beginning of the data pool.

```
RESTORE
```

BYTES val, val, ...

(Extension) Define raw binary data bytes under a label. Values are 8-bit (0–255 or >00⟶FF). Used with CALL VDPMEMCPY for bulk VDP transfers.

```
@PADDLE_PATTERN:
BYTES >18,>18,>18,>18,>18,>18,>18,>18
```

Multiple BYTES lines under the same label are concatenated.

DATA ENCODING

DATA values are compiled into a binary blob in ROM with typed tags:

0x00	Integer: tag + 2 bytes (big-endian)
0x01	String: tag + chars + NUL
0xFF	End sentinel

SEE ALSO

xb-variables(7), xb-vdp(7), xb-misc(7)

Source: `resources/man/man7/xb-data.7`

xb-input

xb-input(7)

XB BASIC joystick, keyboard, and timing commands

NAME

`xb-input` — XB BASIC joystick, keyboard, and timing commands

DESCRIPTION

Hardware input and timing via the TMS9900 CRU and VDP interrupt.

CALL JOYST(unit, x, y)

Read joystick *unit* (1 or 2). *x* receives -4 (left), 0 (center), or 4 (right). *y* receives -4 (down), 0 (center), or 4 (up).

```
CALL JOYST(1,JX%,JY%)
```

CALL KEY(unit, key, status)

Scan the keyboard. *unit* selects the input source:

0	Any key (full keyboard scan)
1	Left side of keyboard / joystick 1 fire
2	Right side of keyboard / joystick 2 fire

key receives the ASCII code of the pressed key (0 if none). *status* receives:

- 0 — no key pressed
- -1 — new key pressed
- 1 — same key still held

```
CALL KEY(0,K%,S%)
```

CALL VBLANK

Wait for the next vertical blank interrupt (frame sync). Use this for consistent ~60Hz game timing.

```
CALL VBLANK
```

EXAMPLES

```
! Simple input loop
@LOOP:
  CALL KEY(0,K%,S%)
  IF K%=13 THEN GOTO @DONE
  CALL JOYST(1,JX%,JY%)
  CALL VBLANK
  GOTO @LOOP
@DONE:
```

SEE ALSO

xb-sprites(7), xb-sound(7), xb-control(7)

Source: `resources/man/man7/xb-input.7`

xb-io

xb-io(7)

XB BASIC print, display, and input statements

NAME

`xb-io` — XB BASIC print, display, and input statements

DESCRIPTION

Screen output and keyboard input for text-mode interaction.

PRINT expr [; expr] [, expr]

Print values to the screen at the current cursor position.

- Semicolon '`;`' suppresses spacing between items.
- Comma '`,`' advances to the next tab stop.
- A trailing semicolon suppresses the newline.

```
PRINT SCORE: ;S1%
```

```
PRINT X%,Y%,Z%
```

DISPLAY AT(row, col): expr

Write text at a specific screen position without moving the cursor.

```
DISPLAY AT(1,10):"PLAYER 1"
```

```
DISPLAY AT(24,1):S1%
```

PRINT USING fmt: expr

Formatted output using a format string or IMAGE reference.

```
PRINT USING ###.##: SCORE
```

IMAGE format

Define a format template on a line number for use with PRINT USING.

```
100 IMAGE ###.## POINTS
PRINT USING 100: SCORE
```

INPUT prompt: var

Read a value from the keyboard.

```
INPUT ENTER NAME: :NAME$
```

```
INPUT HOW MANY? :N%
```

LINPUT prompt: var$

Read an entire line (including commas and semicolons) into a string.

```
LINPUT TEXT: :LINE$
```

ACCEPT AT(row, col) [SIZE(n)] [VALIDATE(...)] [BEEP]: var

Read input at a specific screen position. Optional modifiers:

`'SIZE(n)'`	Limit input field to *n* characters.
`'VALIDATE(...)'`	Restrict allowed input characters.
`'BEEP'`	Sound a beep when the field is ready.

```
ACCEPT AT(12,5) SIZE(10): NAME$
```

```
ACCEPT AT(1,1) VALIDATE("YN") BEEP: CHOICE$
```

SEE ALSO

xb-control(7), xb-string(7), xb-variables(7)

Source: `resources/man/man7/xb-io.7`

xb-math

xb-math(7)

XB BASIC built-in numeric functions

NAME

`xb-math` — XB BASIC built-in numeric functions

DESCRIPTION

Numeric functions available in all XB BASIC programs. Functions operating on or returning floats require float mode (`-mfloat-ieee`) or `-mfloat-fixed`.

ABS(x)

Return the absolute value of *x*.

```
Y%=ABS(-42)! Y% = 42
```

SGN(x)

Return the sign of *x*: -1 if negative, 0 if zero, 1 if positive.

```
S%=SGN(-5)! S% = -1
```

INT(x)

Return the integer part of *x* (truncates toward zero).

```
N%=INT(3.7)! N% = 3
```

SQR(x)

Return the square root of *x*.

```
R=SQR(144)! R = 12.0
```

RND

Return a pseudo-random integer. Use RANDOMIZE to seed.

```
X%=RND
```

PI

The constant π (3.14159265...).

```
AREA=PI*R*R
```

SIN(x), COS(x), TAN(x)

Trigonometric functions. *x* is in radians.

```
Y=SIN(PI/4)
```

ATN(x)

Inverse tangent (arctangent) of *x*. Returns radians.

```
ANGLE=ATN(1)! PI/4
```

EXP(x)

Return e raised to the power *x*.

```
Y=EXP(1)! 2.71828...
```

LOG(x)

Return the natural logarithm of *x*.

```
Y=LOG(2.71828)! approximately 1.0
```

OPERATORS

`+`	Addition
`-`	Subtraction / negation
`*`	Multiplication
`/`	Division
`^`	Power (integer exponent)
`%`	Modulo (integer remainder)

COMPARISON

`=`	Equal
`<>`	Not equal
`<`	Less than
`>`	Greater than
`<=`	Less or equal
`>=`	Greater or equal

LOGICAL

`AND`	Logical AND
`OR`	Logical OR
`NOT`	Logical NOT

SEE ALSO

xb-string(7), xb-variables(7)

Source: `resources/man/man7/xb-math.7`

xb-misc

xb-misc(7)

XB BASIC miscellaneous statements and directives

NAME

`xb-misc` — XB BASIC miscellaneous statements and directives

DESCRIPTION

Utility statements, program termination, and error handling.

LET var = expr

Explicit assignment (the '`LET`' keyword is optional).

```
LET X%=42
```

```
X%=42
```

END

Terminate program execution.

```
END
```

STOP

Halt program execution (equivalent to END in compiled code).

```
STOP
```

REM text

! text

Comment. Everything after REM or ! is ignored until the end of the line.

```
REM This is a comment
```

```
! This is also a comment
```

RANDOMIZE [seed]

Seed the random number generator. Without an argument, uses a time-based seed.

```
RANDOMIZE
```

```
RANDOMIZE 42
```

DEF name(param) = expr

Define a single-expression function.

```
DEF DOUBLE(X)=X*2
```

```
Y%=DOUBLE(21)
```

ON ERROR label

Set an error handler. When a runtime error occurs, execution transfers to *label*.

```
ON ERROR @ERR_HANDLER
```

RESUME

Resume execution after an error handler. Returns to the statement that caused the error.

```
RESUME
```

SEE ALSO

xb-control(7), xb-variables(7), xb-sub(7)

Source: `resources/man/man7/xb-misc.7`

xb-screen

xb-screen(7)

XB BASIC screen, color, and character definition commands

NAME

`xb-screen` — XB BASIC screen, color, and character definition commands

DESCRIPTION

Screen initialization, color table manipulation, and character pattern definition for the TI-99/4A VDP in Graphics I mode.

CALL CLEAR

Clear the screen (fill name table with spaces) and reset the cursor position.

```
CALL CLEAR
```

CALL SCREEN(color)

Set the screen background color. *color* is a VDP color code (1–16):

1 Transparent	5 Dark Blue	9 Medium Red	13 Dark Green
2 Black	6 Light Blue	10 Light Red	14 Magenta
3 Medium Green	7 Dark Red	11 Dark Yellow	15 Gray
4 Light Green	8 Cyan	12 Light Yellow	16 White

```
CALL SCREEN(2)
```

CALL COLOR(set, fg, bg)

Set the foreground and background colors for a character set. Each set controls 8 consecutive character codes:

1	0–7
2	8–15
…	…
32	248–255

fg and *bg* are VDP color codes (1–16).

```
CALL COLOR(2,16,2)
```

CALL CHAR(code, pattern)

Redefine the pixel pattern of character *code* (0–255). *pattern* is a 16-character hex string representing 8 bytes (8×8 pixels).

```
CALL CHAR(42,"007E424242427E00")
```

Each hex pair represents one row, MSB–left. The string must be exactly 16 characters.

TI-99/4A NOTES

- The TI-99/4A VDP operates in Graphics I mode with 32 character sets.
- CALL CLEAR writes 0x20 (space) to the entire 768-byte name table.
- Color table is 32 bytes: one byte per set (high nybble = fg, low = bg).
- Pattern table is 2048 bytes: 8 bytes per character × 256 characters.

SEE ALSO

xb-tiles(7), xb-sprites(7), xb-vdp(7)

Source: `resources/man/man7/xb-screen.7`

xb-sound

xb-sound(7)

XB BASIC sound generation commands

NAME

`xb-sound` — XB BASIC sound generation commands

DESCRIPTION

Sound output via the TI-99/4A SN76489 sound chip.

CALL SOUND(duration, freq, vol)

Play a tone on sound channel 0.

duration is in milliseconds (converted internally to VBlank ticks by dividing by 16; minimum 1 tick).

freq is the frequency in Hz. If *freq* is 0, only the duration timer is set (silence).

vol is the volume using TI Extended BASIC convention: 0 = loudest, 30 = silent. Internally converted to SN76489 attenuation by dividing by 2.

```
CALL SOUND(200,440,2)
```

Note: The current runtime supports a single tone channel (channel 0). Multi-channel sound (passing additional freq/vol pairs) is not yet implemented.

CALL SAY(word)

Speak a word using the TI speech synthesizer. *Note:* Speech synthesis is not supported in the current runtime or emulator.

```
CALL SAY("HELLO")
```

TIMER

Sound duration is managed by the VBlank ISR. The ISR decrements a tick counter each frame (~60Hz); when it reaches zero, all sound channels are silenced. One tick ~ 17ms (1/60s).

EXAMPLES

```
! Play a simple tone
CALL SOUND(500,262,2)   ! Middle C for ~500ms

! Play descending tones
FOR I%=1 TO 5
  CALL SOUND(200,440-I%*50,0)
NEXT I%
```

SEE ALSO

xb-input(7), xb-screen(7)

Source: `resources/man/man7/xb-sound.7`

xb-sprites

xb-sprites(7)

XB BASIC sprite creation, movement, and collision commands

NAME

`xb-sprites` — XB BASIC sprite creation, movement, and collision commands

DESCRIPTION

Hardware sprite management via the TMS9918A VDP. Up to 32 sprites are supported, numbered #1 through #32.

CALL SPRITE(#id, pat, color, y, x)

Create or update sprite *#id* with pattern *pat*, VDP color *color*, at pixel position (*y*, *x*).

```
CALL SPRITE(#1,33,7,96,128)
```

Pattern numbers refer to character codes. In 8×8 magnify mode, each sprite uses one character pattern; in 16×16 mode, four consecutive character patterns are used.

CALL DELSPRITE(#id)

Delete sprite *#id* from the sprite table (moves it off-screen).

```
CALL DELSPRITE(#1)
```

Use '`CALL DELSPRITE(ALL)`' to delete all sprites.

CALL MOTION(#id, vy, vx)

Set the velocity of sprite *#id*. *vy* is vertical speed (positive = down), and *vx* is horizontal speed (positive = right). Values range from -128 to 127. The sprite position is updated automatically each frame by the ISR.

```
CALL MOTION(#1,0,4)
```

CALL MAGNIFY(n)

Set the sprite magnification mode for all sprites:

1	8×8	1 char each
2	8×8 doubled	1 char each
3	16×16	4 chars each
4	16×16 doubled	4 chars each

```
CALL MAGNIFY(3)
```

CALL PATTERN(#id, pat)

Change the pattern of sprite *#id* to *pat* without affecting position or color.

```
CALL PATTERN(#1,37)
```

CALL LOCATE(#id, y, x)

Move sprite *#id* to absolute pixel position (*y*, *x*).

```
CALL LOCATE(#1,0,0)
```

CALL POSITION(#id, y, x)

Read the current position of sprite *#id* into variables *y* and *x*.

```
CALL POSITION(#1,SY%,SX%)
```

CALL COINC(#id1, #id2, tol, flag)

Check if two sprites overlap within *tol* pixels. *flag* is set to -1 (true) if they coincide, 0 otherwise.

```
CALL COINC(#1,#2,8,HIT%)
```

Use '`CALL COINC(ALL,tol,flag)`' to check if any two sprites overlap within *tol* pixels.

CALL DISTANCE(#id1, #id2, dist)

Calculate the squared distance between two sprites. *dist* receives (dy*dy + dx*dx).

```
CALL DISTANCE(#1,#2,D%)
```

SPRITE LIMITS

- The TMS9918A supports up to 32 sprites, but only 4 per scanline.
- Sprites with lower IDs have higher display priority.
- Y position 208 (0xD0) terminates sprite processing for all subsequent sprites.
- Sprite coordinates are in pixels: Y 0–191, X 0–255.

EXAMPLES

```
! Create a 16x16 bouncing ball
CALL MAGNIFY(3)
CALL CHAR(96,"3C7EFFFFFFFF7E3C")
CALL CHAR(97,"0000000000000000")
CALL CHAR(98,"0000000000000000")
CALL CHAR(99,"0000000000000000")
CALL SPRITE(#1,96,7,96,128)
CALL MOTION(#1,2,3)
```

SEE ALSO

xb-screen(7), xb-tiles(7), xb-vdp(7)

Source: `resources/man/man7/xb-sprites.7`

xb-string

xb-string(7)

XB BASIC built-in string functions

NAME

`xb-string` — XB BASIC built-in string functions

DESCRIPTION

String manipulation functions. String variables use the '`$`' suffix.

LEN(s$)

Return the length of string *s$*.

```
L%=LEN("HELLO")! L% = 5
```

SEG$(s$, pos, len)

Extract a substring starting at position *pos* (1-based) with length *len*.

```
X$=SEG$("HELLO",2,3)! X$ = ELL
```

CHR$(n)

Return the character with ASCII code *n*.

```
A$=CHR$(65)! A$ = A
```

ASC(s$)

Return the ASCII code of the first character of *s$*.

```
N%=ASC("A")! N% = 65
```

STR$(n)

Convert number *n* to its string representation.

```
S$=STR$(42)! S$ = 42
```

VAL(s$)

Convert string *s$* to a number. Returns 0 if the string is not a valid number.

```
N%=VAL("42")! N% = 42
```

POS(haystack$, needle$, start)

Find the position of *needle$* within *haystack$* starting at position *start* (1-based). Returns 0 if not found.

```
P%=POS("HELLO","LL",1)! P% = 3
```

RPT$(s$, n)

Repeat string *s$* *n* times.

```
BAR$=RPT$("=",20)
```

Concatenation (&)

Join two strings.

```
FULL$=FIRST$ & & LAST$
```

SEE ALSO

xb-math(7), xb-io(7), xb-variables(7)

Source: `resources/man/man7/xb-string.7`

xb-sub

xb-sub(7)

XB BASIC subroutine definition and invocation

NAME

`xb-sub` — XB BASIC subroutine definition and invocation

DESCRIPTION

User-defined subroutines with parameters and shared variables.

SUB name(params) / SUBEND

Define a subroutine. Parameters are passed by reference.

```
SUB MOVE_BALL(BX%,BY%,DX%,DY%)
  BX%=BX%+DX%
  BY%=BY%+DY%
SUBEND
```

CALL name(args)

Invoke a subroutine. Arguments are matched by position to the SUB's parameter list.

```
CALL MOVE_BALL(BALLX%,BALLY%,BDX%,BDY%)
```

SHARED var1, var2, ...

Declare variables as shared between the main program and all SUBs. Must appear inside a SUB body.

```
SUB UPDATE_SCORE(PLAYER%,POINTS%)
  SHARED SCORE1%,SCORE2%
  IF PLAYER%=1 THEN SCORE1%=SCORE1%+POINTS%
  IF PLAYER%=2 THEN SCORE2%=SCORE2%+POINTS%
SUBEND
```

NOTES

- SUBs are compiled as separate functions with their own stack frame.
- Integer (%) and string ($) parameters are passed as 16-bit pointers.
- Float parameters (no suffix) are passed as 32-bit values.
- SUBs may call other SUBs and use recursion.

SEE ALSO

xb-control(7), xb-variables(7)

Source: `resources/man/man7/xb-sub.7`

xb-tiles

xb-tiles(7)

XB BASIC tile placement and reading commands

NAME

`xb-tiles` — XB BASIC tile placement and reading commands

DESCRIPTION

Place or read characters on the 32×24 screen grid. Row values are 1–24, column values are 1–32.

CALL HCHAR(row, col, char [, count])

Place character *char* at position (*row*, *col*) horizontally. If *count* is specified, repeat the character *count* times, wrapping to the next row at column 32.

```
CALL HCHAR(1,1,42,32)
```

CALL VCHAR(row, col, char [, count])

Place character *char* at position (*row*, *col*) vertically. If *count* is specified, repeat downward, wrapping to the next column at row 24.

```
CALL VCHAR(1,1,42,24)
```

CALL GCHAR(row, col, var)

Read the character code at position (*row*, *col*) into integer variable *var*.

```
CALL GCHAR(12,16,C%)
```

ADDRESSING

Screen positions map to VDP Name Table addresses:

```
addr = (row - 1) * 32 + (col - 1)
```

Row 1 is the top row, column 1 is the leftmost column. The name table starts at VDP address 0x0000 by default.

EXAMPLES

```
! Draw a border around the screen
CALL HCHAR(1,1,42,32)    ! Top row
CALL HCHAR(24,1,42,32)   ! Bottom row
CALL VCHAR(2,1,42,22)    ! Left column
CALL VCHAR(2,32,42,22)   ! Right column
```

SEE ALSO

xb-screen(7), xb-sprites(7), xb-vdp(7)

Source: `resources/man/man7/xb-tiles.7`

xb-variables

xb-variables(7)

XB BASIC variable types, arrays, and declarations

NAME

`xb-variables` — XB BASIC variable types, arrays, and declarations

DESCRIPTION

Variable naming conventions, data types, and array dimensions.

VARIABLE TYPES

Variable type is determined by its suffix:

%	Integer	16-bit	-32768 to 32767
$	String	pointer	up to 255 chars
(none)	Float	32-bit	IEEE 754 single

```
X%=42
```

```
NAME$="PONG"
```

```
PI=3.14159
```

DIM var(size [, size2])

Declare an array. By default, indices start at 0 (see '`OPTION BASE`').

```
DIM BOARD%(24,32)    ! 2D array
DIM NAMES$(10)       ! 1D string array
```

The actual number of elements is *size* + 1 (indices 0 through *size*).

OPTION BASE n

Set the minimum array index. *n* must be 0 or 1. Default is 0.

```
OPTION BASE 1
```

VARIABLE SCOPE

- All variables in the main program are global.
- Variables inside a SUB are local unless declared SHARED.
- Arrays inside SUBs are local.

TYPE COERCION

- Assigning a float to an integer variable truncates the value.
- Integer expressions in float context are promoted automatically.
- String and numeric types cannot be mixed.

SEE ALSO

xb-sub(7), xb-control(7), xb-data(7)

Source: `resources/man/man7/xb-variables.7`

xb-vdp

xb-vdp(7)

XB BASIC VDP memory access and bulk transfer commands

NAME

`xb-vdp` — XB BASIC VDP memory access and bulk transfer commands

DESCRIPTION

Direct VDP (Video Display Processor) memory access for bulk data transfers, pattern loading, and low-level hardware control. These are extensions to standard TI Extended BASIC.

CALL VDPMEMCPY(addr, label, len)

Copy *len* bytes from a BYTES data block labeled *label* to VDP memory starting at address *addr*.

```
CALL VDPMEMCPY(>0800,@PATTERN_DATA,64)
```

addr can be specified in decimal or hex with the '`>`' prefix. *label* must reference a '`BYTES`' data block defined in the source.

CALL VDPMEMSET(addr, byte, len)

Fill *len* bytes of VDP memory starting at *addr* with the value *byte*.

```
CALL VDPMEMSET(>0000,32,768)
```

CALL PEEK(addr, var)

Read one byte from CPU memory address *addr* into variable *var*.

```
CALL PEEK(>8802,V%)
```

CALL POKE(addr, val)

Write byte *val* to CPU memory address *addr*.

```
CALL POKE(>8C02,>05)
```

VDP MEMORY MAP

The default VDP memory layout for Graphics I mode:

Address	Size	Contents
>0000	768	Name Table (32×24 screen)
>0380	32	Color Table (1 byte per set)
>0800	2048	Pattern Table (8 bytes per char)
>0300	128	Sprite Attribute Table
>1800	2048	Sprite Pattern Table

BYTES DIRECTIVE

The '`BYTES`' directive defines raw binary data blocks in ROM:

```
@PATTERN_DATA:
BYTES >3C,>7E,>FF,>FF,>FF,>FF,>7E,>3C
```

Each value is a single byte (0–255 or >00→FF). Labels referenced by VDPMEMCPY must use the '`@`' prefix.

EXAMPLES

```
! Load custom character patterns from ROM
@MY_CHARS:
BYTES >00,>7E,>42,>42,>42,>42,>7E,>00
BYTES >18,>3C,>7E,>FF,>FF,>7E,>3C,>18
```

```
CALL VDPMEMCPY(>0800+42*8,@MY_CHARS,16)
```

SEE ALSO

xb-screen(7), xb-tiles(7), xb-sprites(7)

Source: `resources/man/man7/xb-vdp.7`

The ti99-tidepod-gamedev agent skill

AI agents

Tidepod ships an *agent skill*: a set of Markdown files that tells AI coding assistants how to work on a Tidepod project. Without it, assistants tend to edit the generated Makefile, invent compiler or packer flags, and call library functions that don't exist. With it they use the CLI, the real build and the real emulator options.

Where it is

`tidepod new` copies the skill into every new project, in the two locations most assistants look for project skills:

Project layout

```
mygame/
├── .agents/skills/ti99-tidepod-gamedev/
│   ├── SKILL.md
│   ├── reference.md
│   └── examples.md
└── .cursor/skills/ti99-tidepod-gamedev/
    └── (same three files)
```

The master copy is in the Tidepod install under `resources/agent-skills/ti99-tidepod-gamedev/`. For existing projects, copy that directory into the project yourself. If your assistant uses another location, copy or link it there; the files are plain Markdown with a YAML header.

Contents

File	What it covers
`SKILL.md`	When to use the skill, the hard rules, the canonical new/add/build/run loop, CLI table, emulator and GDB usage, PEB/disk configuration, hang forensics with `--dump-regs`, headless verification, banking and F18A essentials, a pitfall checklist, and pointers to the example projects.
`reference.md`	Memory map, project-file details (disk mounts, PEB cards), packer outputs and options, VDP and F18A notes, current emulator flags, tool paths, debug/profiler helpers.
`examples.md`	Short recipes: new banked cartridge, persistent code plus a resource, F18A GPU image, headless GDB session, disk/EA5, and which example project to open for what.

When an assistant uses it

The skill's description makes assistants load it for anything involving Tidepod projects: building banked cartridges, EA5 disks or XB projects, VDP/F18A graphics, packer or banking problems, code placement, emulator debugging, and adding or removing sources and resources.

Keeping it current

The skill is versioned with Tidepod. After updating Tidepod, refresh the copies in your projects from `resources/agent-skills/` so assistants see the current CLI and emulator options. If you have project rules of your own, put them in a separate file rather than editing the skill, so they survive the refresh.

Rules the skill enforces

AI agents

These are the rules the skill gives to assistants. They are good rules for humans too; if an assistant breaks one, point it at the skill.

Hard rules

1. **Never edit the generated Makefile.** It starts with `# [GENERATED BY TI-99 IDE - DO NOT EDIT MANUALLY]` and is overwritten by the next sync. Extra rules go in `Makefile.custom`.
2. **Change the build list only with the CLI**: `tidepod add`, `tidepod remove`, or edit `srcFiles` in the `.project` file and run `tidepod sync`. In particular, never add a resource by editing `RESOURCE_OBJS`.
3. **Build with `make` / `make debug`.** There is no `tidepod build`.
4. **Don't edit generated files**: `Makefile`, `autobank.ld`, the trampolines, `autobank_mapping.h`, `include/resource_defs.h`, `include/gpu_*_map.h`, `bank_report.json`.
5. **Prefer the Tidepod CLI over hand-made packer or linker flags.**
6. **`tidepod run` starts the stand-alone emulator, not the IDE.** It uses the layered configuration (Preferences, project `"emu"` block, disk mounts). For headless runs, call `ti994a-emu` directly.
7. **No large local variables.** The stack is small. Buffers of about 64 bytes or more (sector buffers, file records, tables) must be `static` or global, never locals.

Canonical loop

Terminal

```
tidepod new mygame --type banked --path ./mygame
cd mygame
# edit main.c, tidepod add new files / resources
tidepod sync && make debug
tidepod run --debug                       # emulator + GDB on :3333

# verify boot the way a user sees it (no --autorun)
ti994a-emu --cart mygame.rpk --f18a --headless \
  --keys "WAIT:300 SPACE WAIT:120" --screenshot-at 450 --frames 480
```

Pitfall checklist

Before handing over a build, the skill has the assistant check:

- VR30 set after unlocking the F18A.
- Interrupt / vsync strategy is intentional.
- Programs using disk files: scratchpad area starts at `0x8380` or higher.
- No source file larger than one bank.
- Makefile and `RESOURCE_OBJS` never edited by hand.
- No edits to generated packer, resource or GPU map files.
- `make debug` + `--gdb` used for debugging rather than a home-made debug protocol.
- Boot verified without `--autorun`, using `--keys` and `--screenshot-at`.
- `bank_report.json` headroom, trampolines and stack depth compared with the last good build.

The background for each item is in this documentation: Banking, Code placement, F18A GPU code, bank_report.json and Headless testing.