Sanguis Toolchain 4.0.2: Program Maker Manual

Sanguis is a small language with a complete toolchain. The compiler is itself a Sanguis program, and it rebuilds itself from its own source byte for byte. This manual covers installing the toolchain, the `sanguis` command, and the parts of the language you need to write programs with it.

Installing

Windows (x86-64, Windows 10 version 1903 or newer). Run `sanguis-4.0.2-setup.exe` and press Install. It installs for your account only, needs no administrator rights, adds `sanguis` to your PATH (you can untick that) and appears in your installed apps, where it can be removed.

Linux (x86-64). Run:

```
sh sanguis-4.0.2-linux-x86_64.run
```

It installs to `~/.local` and links the `sanguis` command into `~/.local/bin`. Use `--prefix=DIR` for another place. Remove it with `sanguis-uninstall`. No compiler or other package is needed. The `sanguis` command and its floor are statically linked; the grid engine, which runs the circuit commands, uses the system's GNU C library (2.17 or newer, as on mainstream distributions; not on musl-only systems such as Alpine).

Without installing. Unpack the zip or tar.gz for your system anywhere and run `bin/sanguis` from it.

Both installers delete only files they installed. They refuse a folder that holds other files, and files you save inside an install folder survive removal.

After installing, open a new terminal and check the install:

```
sanguis verify
```

`verify` compares every file with the manifest, rebuilds the compiler and the linker from their sources and compares the bytes, then compiles and runs a test program, packs a native module, checks that a fleet gives the same bytes on one core and on several, and links a program with the standard library. It ends with `VERIFIED`.

Your First Program

Save this as `hello.sanguis`:

```
cell main() -> float {
    print("hello from Sanguis");
    output 0.0;
}
```

```
sanguis drip hello.sanguis
```

It prints the greeting, then `0`: the floor prints the answer of the cell it ran.

The Command

```
sanguis new name                          start a project
sanguis bleed prog.sanguis [out.bc]       compile one file to BloodCode
sanguis link main.sanguis [out.bc]        compile a program and its imports, and join them
sanguis drip prog.sanguis|prog.bc [cell]  run (cell: main unless named)
sanguis test prog.sanguis                 run the file's test blocks
sanguis fmt prog.sanguis                  trim line ends, only if the meaning stays
sanguis pulse prog.sanguis [cell]         rerun as you edit, only when the meaning changes
sanguis seal prog.sanguis [out.bc]        compile and write a receipt anyone can check
sanguis prove out.bc.seal [prog.sanguis]  rebuild from source and compare with the receipt
sanguis blob prog.bc out.mod              pack sys cells into a native x86-64 module
sanguis verify                            check every file, rebuild the compiler, run a test
sanguis version                           which toolchain this is
```

Every command exits 0 on success. A command that fails exits with another number, so scripts can rely on it. A usage mistake exits 2.

A build never writes over its own source, and a build that fails leaves the previous output as it was. Stopping `sanguis` also stops the program it is running. `drip`, `test` and `pulse` link a file that has `import` lines by themselves.

new

`sanguis new greeter` makes a folder with a `Sanguis.toml` and `src/main.sanguis`. Inside it, `sanguis drip`, `sanguis test` and `sanguis link` need no file name: they use the `entry` named in `Sanguis.toml`.

test

A `test` block holds checks. `need` stops the test, with its line, when its condition is false.

```
cell half(x: float) -> float {
    output x / 2.0;
}

cell main() -> float {
    print(half(8.0));
    output 0.0;
}

test "half of four is two" {
    need half(4.0) == 2.0;
}
```

`sanguis test file.sanguis` builds each test as its own program, with the file's `main` left out, and runs them one after another, so one failure does not stop the rest. The tests are compiled side by side. The command exits 0 only if every test passed.

link and imports

A file that other files import begins with `circuit name;`, and its file is `name.sanguis`. Its cells are reached through an organ, as in `shapes.area(w, h)` for the cell `area` in the organ `shapes`.

```
circuit app;
import math;
import str;

cell main() -> float {
    print(math.max(2.0, 9.0));
    output 0.0;
}
```

An imported name is looked for in this order: `name.sanguis` beside the file that imports it; a package; the standard library (`toolchain/lib/std`, which holds `math` and `str`). A package is a folder named in `Sanguis.toml`:

```
[deps]
shapes_kit = "../shapes_kit"
```

The modules are compiled side by side, one for each processor, and joined by the linker, which is itself written in Sanguis. The result is the same bytes however many are built at once. Packages are not fetched over the network.

fmt

`sanguis fmt` removes spaces at the ends of lines and ends lines with LF. It writes the file only if it compiles to the same bytes afterwards.

pulse

(Given a circuit file, `pulse` runs the grid: see Circuit Grids. Given a `.sanguis` file, it is the watch command below.)

`sanguis pulse game.sanguis` runs the program, then watches the file. Each time you save, it compiles. The program runs again only if the compiled result changed: comments, blank lines, indentation and line endings do not change it. A save that does not compile shows the error and the watch goes on. Ctrl+C stops it. While your program is running the watch waits for it to end.

seal and prove

The compiler is deterministic: the same meaning gives the same bytes, on Windows and on Linux, from any folder. `seal` uses this.

```
sanguis seal tool.sanguis
```

This writes `tool.bc` and `tool.bc.seal`. The seal is a short text file holding the SHA-256 of the compiler, the source and the program, and the two file names (never folder names). Whoever has the source, the program and the seal can check that the program really is that source, compiled:

```
sanguis prove tool.bc.seal
```

`prove` rebuilds the program from the source and compares. It answers `PROVEN` when the bytes are identical, and says so if only the text of the source (not its meaning) has changed. It answers `NOT PROVEN`, with the reason, if the program was replaced or changed, or the source now means something else. `prove` looks for both files beside the seal; name the source as a second argument if it is elsewhere.

A seal is not a signature. It does not say who made the program, and someone who can replace all three files can make a new seal. It answers one question: does this program come from this source?

Writing Programs

Programs are made of cells. A cell has a name, parameters and one result, which leaves through `output`. There are two kinds.

Kind	Declared	Runs as	Works with
float cell	`-> float`	interpreted, by `sanguis drip`	text, lists, floating-point numbers
sys cell	`-> u64` or `-> i64`	native x86-64 code, from a module	exact 64-bit integers and typed memory

A file given to `sanguis blob` holds sys cells only. The two kinds do not call each other directly.

Float cells

```
cell area(w: float, h: float) -> float {
    output w * h;
}

cell main() -> float {
    let sizes = [2.0, 3.0, 5.0];
    let total = 0.0;
    let i = 0.0;
    while i < len(sizes) {
        total = total + area(sizes[i], 2.0);
        i = i + 1.0;
    }
    gate total > 10.0 then { print(concat("large: ", total)); }
    output 0.0;
}
```

Fleets

A fleet runs one cell many times over, tick by tick. What a cell sends arrives on the next tick, in the order the cells sent it.

```
cell step(id: float, tick: float) -> float {
    let got = inbox();
    let value = id;
    for v in got { value = value + v; }
    send(mod(id + 1.0, 6.0), value);
    gate tick == 4.0 then { print(concat(concat("cell ", id), concat(" holds ", value))); }
    output 0.0;
}

cell main() -> float {
    fleet step cells 6 ticks 5;
    output 0.0;
}
```

Fleets use every core. When a fleet is long enough to be worth it, the floor splits its cells among worker processes. The output, the order of messages, the point where an error stops the program and the exit status are the same as when the cells run one at a time. If a cell, or anything it calls, could affect something outside itself (writing a file, emitting a value, reading the clock, spawning a task, swapping a stem), the fleet runs on one core.

Setting	Effect
unset	the first tick is timed, and workers start only if the rest is long enough to repay them
`SANGUIS_THREADS=1`	every fleet on one core
`SANGUIS_THREADS=8`	eight workers from the first tick
`SANGUIS_FLEET_STATS=1`	say on standard error what was decided, and why

Numbers, bytes and sound

`nn_gelu(x)`, `nn_softmax(list)`, `nn_argmax(list)`, `nn_matvec(rows, list)` and `nn_layernorm(x, weights, biases)` are small neural-network operations. `step_divide(a, b)` divides, and answers just under 0.1 (with the sign of the quotient) instead of dividing a non-zero number by zero. `read_bytes(path)` gives a file's bytes as a list of numbers, `write_wav(path, samples)` writes a one-channel 48 kHz floating-point WAV, and `now_ms()` is the time in milliseconds since 1970 (not available in web pages). A circuit input written `ext:name` reads as 0 when nothing sets it.

Sys cells and layouts

```
cell add(a: u64, b: u64) -> u64 {
    output a + b;
}
```

```
sanguis bleed math.sanguis
sanguis blob math.bc math.mod
```

Shifts are calls, `shl(x, n)` and `shr(x, n)`; there is no infix shift. A layout describes a record once and gives it accessors that check their bounds:

```
layout Point {
    x: u32;
    y: u64;
}
```

This defines `Point_size`, `Point_x(p)`, `Point_set_x(p, v)` and the same pair for `y`.

Modules

A module is a text file in three parts: `BLOB1` and the code size in bytes; one `BLOBOFF name offset` line per cell; `BLOB2` and the machine code in hexadecimal. To load one from a host language: read the three parts, copy the bytes into memory you can execute, and call the address `base + offset`. Arguments go in the integer registers of the System V AMD64 convention and the answer returns in `rax`. This convention is the same on both systems: a Windows host must call with it explicitly (in GCC and Clang, `__attribute__((sysv_abi))`).

Modules made by `blob` have no time or stack guards. A cell that loops forever or reaches outside its memory does so inside your process.

Roles

Five kinds of cell take a role in their declaration, and the compiler builds what the role needs: `stem`, `tagger`, `pruner`, `helper` and `immune`.

A helper is a named value or shortcut; a value helper is called like a cell, `motto()`. A stem runs its own body until `differentiate` hands its calls to another cell, and `undifferentiate` takes them back:

```
cell assign helper motto = "think in cells";

cell assign stem st() -> float {
    output 1.0;
}

cell other() -> float {
    output 2.0;
}

cell main() -> float {
    print(motto());
    print(concat("own body: ", st()));
    let a = differentiate("st", "other");
    print(concat("handed over: ", st()));
    let b = undifferentiate("st");
    print(concat("taken back: ", st()));
    output 0.0;
}
```

An immune cell is a guard that runs as its own task. `awaken_g()` starts it, `summon_g(value)` wakes its body, `patrol(n)` runs a block every `n` ticks, and `run_all()` runs every task until all have ended. A guard does not end by itself: give it a closer that calls `kill`.

```
cell assign immune g watching wbox(x: float) {
    print("summon arrived");
    patrol(2) {
        print("patrol tick");
    }
}

cell closer(tid: float) -> float {
    sleep_ticks(6.0);
    kill(tid);
    output 0.0;
}

cell main() -> float {
    let t = awaken_g();
    let s = summon_g(42.0);
    let c = spawn closer(t);
    let r = run_all();
    print("closed");
    output 0.0;
}
```

Known Traps

1. In a sys cell, the type after `->` types the whole cell: every variable, operation, comparison and division. In a `u64` cell, divide and compare are unsigned, so `0 - 7` divided by 2 is `9223372036854775804`. Use an `i64` cell when you need signed division or signed comparison.
2. In sys cells, values wrap. Division or remainder by zero gives 0. The smallest `i64` divided by -1 stays the smallest value. A shift of 64 or more gives 0, except that a signed right shift fills with the sign.
3. A number above 2^53 loses precision if it passes through a float. Keep it in `u64` from end to end.
4. `run_all()` returns only when every task has ended. A guard without a closer keeps the program open.
5. `task_run_all` is recognised but not implemented. Use `run_all()`.
6. Port I/O, model registers, `rdtsc`, `cpuid`, `cli`, `sti` and `hlt` compile anywhere but are meant to run only under the Sanguis operating system. Most of them crash an ordinary program.
7. `bleed` compiles one file. A program with `import` lines is built by `link` (or just run with `drip`).
8. `read_file` gives a file's bytes as they are, so a file with Windows line endings keeps its carriage returns.

Circuit Grids

A circuit grid is a network of switches that settles until nothing changes. The settled state is the answer. The grid engine runs on the processor and is the same one 3.9 used.

A circuit in VANA

A `.vana` file names its inputs with `ext`, its outputs with `out`, and drives wires with `wire`. A `wire` line sends its source to a destination when its conditions hold; `and` adds a condition, and `but` adds one that blocks. This is `half_adder.vana`:

```
ext a
ext b
out sum
out carry

wire 1 to sum when a = 1 and b = 0
wire 1 to sum when a = 0 and b = 1
wire 1 to carry when a = 1 and b = 1
```

`sanguis truth half_adder.vana` prints its whole table. `#` starts a comment. `tool`, `region` and `organism` blocks group lines by indentation (4 spaces or one tab).

Commands

```
sanguis wire file.vana                          settle every combination of the inputs
sanguis truth file.vana                         the truth table
sanguis trace file.vana [--input a=1 ...]       one settlement, step by step
sanguis inspect file.vana                       size, shape and depth
sanguis lint file.vana                          errors and warnings
sanguis vfmt file.vana                          print the file formatted (the file is not changed)
sanguis vana file                               load a grid and settle it
sanguis grid file [-o out.vgrid]                lower to a compiled grid, which loads with no parser
sanguis pulse file [batches] [sweeps]           run batch by batch and print the outputs; Ctrl+C stops
it
sanguis press file [batches] [sweeps] --out core.kcol   run a grid, then save it with its state
sanguis insert core.kcol [batches] [sweeps]     resume a pressed grid where it stopped
sanguis play file.vana [--steps N] [--maze F] [--persistent] [--trace N] [--quiet]
```

`vana`, `pulse`, `press` and `grid` also read `.netlist` and `.vgrid` files; `vana`, `pulse`, `press` and `insert` read pressed cores (`.kcol`). `pulse` takes `--perturb NAME,EVERY,HOLD` to drive an input high for HOLD batches out of every EVERY. Pressing a core again adds to its chain of receipts. `wire` and `vana` run every combination of the inputs, so with many inputs use `trace`.

Netlists

A netlist names each switch and wire by number. This one is a ring that never settles, so `pulse` shows it beating:

```
# name: oscillator
switch 0  inv  not 0  -> 1
wire 0  fb  sw:0  delay 2  -> 0
wire 1  out sw:0  ->
output 1 pulse
```

A `switch` line is its number, name, kind (`and`, `or`, `not`), input wires and output wire. A `wire` line is its number, name, source (`sw:N` for a switch, `ext:NAME` for an outside input), an optional `delay N`, and the switches it feeds.

The arena

`play` runs a circuit as the player of a maze game, without a window. Each step the circuit is given what it senses as inputs and steers with its outputs.

- Drive it with the outputs `go_up`, `go_down`, `go_left` and `go_right`.
- It senses `wall_up` and the other three walls, `pellet_up` and the other three directions (a pellet in line of sight), `pellets_left`, `moved_up` and the other three (the last move), and `moved2_up` and the other three (the move before).

A maze file is rows of equal length: `#` is a wall, `.` a pellet, `P` the start (with a pellet), and anything else open floor. The same circuit in the same maze makes the same run, and the receipt printed at the end shows it. `--persistent` lets the circuit keep its own state between steps.

Coming From 3.9

4.0 replaces the 3.9 Rust toolchain with a compiler written in Sanguis on a small C floor. Most programs run unchanged.

Speed. 3.9's `bridge` compiled programs to machine code as they ran. 4.0 has no such compiler: `drip` interprets BloodCode, so tight numeric loops on one core run several times slower, and in some cases far more. Long fleets use every core. Work that needs native speed can be written as sys cells and run from a module.

Stricter. 4.0 stops, with a message, where 3.9 carried on silently:

- reading a name nothing has given a value (3.9 read 0)
- `read_file` or `read_bytes` of a file that is not there (3.9 answered empty); use `file_exists(path)` to ask first
- `concat` with other than two values (3.9 dropped the extra ones, or answered an empty text)
- `nn_matvec` given something that is not a list of rows (3.9 answered an empty list)
- calling `module.Organ.method(...)` directly: make the organ with `new` and call the method on it

Circuit grids work as they did (see Circuit Grids), with these differences: the output is plain text without colours; `play` always runs without a window; `fmt` is for `.sanguis` files and `vfmt` for `.vana`; and `--gpu`, `graded`, `press-graded` and `insert-graded`, which need the graphics card, are not included.

Not in this toolchain. `live`, `jar`, `selectric`, `cant`, `tui`, `ad`, `bridge` (use `drip`) and `update`; the graphics-card and window builtins (`gpu_*`, `win_*`, `fb_*`, `eye_*`); and packages fetched with git.

Sanguis in Web Pages

The `web` folder holds the same floor built as WebAssembly, the same compiler, and `sanguis.js`. A page compiles and runs Sanguis and gets the same bytes as your computer. See `web/README.txt`.

Building From Source

The floor is `toolchain/boot/bc_exec.c` with its headers, and the command is `src/sanguis.c`. The grid engine is `toolchain/grid`, in Rust with no outside libraries: `cargo build --release` there makes `sanguis-grid`, which goes in `bin`. With Zig 0.16 for the rest:

```
zig cc -O2 -target x86_64-linux-musl -static -o bin/bch toolchain/boot/bc_exec.c -lm
zig cc -O2 -target x86_64-linux-musl -static -o bin/sanguis src/sanguis.c
zig cc -O2 -target x86_64-windows-gnu -o bin/bch.exe toolchain/boot/bc_exec.c src/utf8.rc
zig cc -O2 -target x86_64-windows-gnu -o bin/sanguis.exe src/sanguis.c src/utf8.rc
```

`src/utf8.rc` lets the Windows programs open files whose names are not in the system's old code page.

The compiler's source is `toolchain/lib/compiler.sanguis`. Keep it with LF line endings: a string in it spans a line break, so a CRLF copy builds a broken compiler. If you change it, `sanguis verify` reports that the compiler no longer rebuilds, which is what it is for.

Glossary

BloodCode (`.bc`): a compiled program. Its set of 29 instructions is frozen.

bleed: compile. **drip**: run.

floor: the C program (`bch`) that runs BloodCode.

fixed point: the compiler rebuilding itself into the same bytes.

module (`.mod`): sys cells packed as native machine code.

seed: the compiler as shipped, `seed/compiler.bc`.

seal: a receipt tying a program to the source it was built from.