SANGUIS TOOLCHAIN KIT

PROGRAM MAKER MANUAL (4.0.0-RC1-P)

Version 4.0.0-RC1-P-toolchain · Issued November 2026
Seed 15eb534381eaae4d · Linker 85836ef29df63802

Introduction

Sanguis is a small language and a complete toolchain for making programs that run on a real machine. The compiler in this kit is itself a Sanguis program, frozen at build time, and it can rebuild itself byte for byte. The tools here compile source, pack compiled programs into modules, and load those modules from any language that can call a function. A smoke test included with the kit proves all of it on your machine in about two minutes.

Programs are made of cells. A cell has a name, parameters, and one result.

The language is deliberately small: a full reference lives in README-PREVIEW-TOOLCHAIN.md. This manual covers what the kit does and how to use it. Where behavior was measured on the shipped floor, the manual reports the measurement. AGENTS.md in the kit root states the same rules for software agents, in directive form; an agent pointed at that file has the syntax, the integer law, the test protocol and the failure recipes without reading prose.

What Shipped

```
seed/            the compiler, frozen: compiler.bc, and its SEED.sha256
toolchain/lib/   compiler.sanguis, the compiler in source, and linker.bc
toolchain/boot/  bc_exec.c and its headers: the floor, built by gcc
tools/           hostfloor.sh, sg.sh, mkblob.sh, mklayout.sh
examples/        hello.sanguis          first program
                 cellsdemo.sanguis      cells, layouts, machine ops
                 roles.sanguis          all five roles, live
                 shader.sanguis         a pixel shader in fixed point
                 divide_probe.sanguis   prints the measured integer answers
                 mandel-0..2.bmp        what the shader made on this machine
                 shader.mod             the shader packed and ready
blobtest/        blobrun.c              known-answer driver for cellsdemo
                 shade.c                the image driver, a template for yours
_smoke.sh        the full test sequence, automated (Linux)
windows\         the same kit on native Windows: sg.bat, mkblob.ps1,
                 smoke.bat, build-exes.bat, and the three built tools in
                 windows\bin (bch.exe, blobrun.exe, shade.exe)
WINDOWS.txt      the Windows quickstart
AGENTS.md        the same rules for software agents, directive form
manual/          this manual and its formatted print
README-PREVIEW-TOOLCHAIN.md   the long form: provenance, hashes, decisions
```

System Requirements

On Windows: nothing. Not even WSL. The kit ships its three tools built, in windows\bin, and windows\smoke.bat runs the whole verification with the PowerShell and cmd that Windows already has. On Linux: gcc, any recent version, bash. Roughly 50 MB of disk either way. Nothing else: no drivers, no services, no registration.

Getting Started

Unzip the kit anywhere. Open a terminal in the top folder. The first command is cheap and settles whether the files you have are the files that shipped:

```
sha256sum -c SHA256SUMS.txt
```

Every line should answer OK.

Then run the whole verification. On Windows:

```
windows\smoke.bat
```

On Linux and WSL:

```
bash _smoke.sh
```

Five steps, about a minute: the cell demo, hello, the roles battery, the shader against its C reference, and the seed reproducing itself. The run ends FIXED_POINT_OK, then a line reading: smoke done. The Quick Start below does the same steps one at a time, so a failure arrives with the piece that failed.

The kit also carries installers. On Windows: windows\install.bat copies the kit to a fixed install home and puts command shims in its bin, and the /path flag also adds those shims to your command line. On Linux and WSL: sh install.sh --prefix=/usr/local does the same for POSIX, defaulting to $HOME/.local. The unpacked folder keeps working as it is either way, and the uninstaller ships beside what it installed. The Windows edition also ships the installer as a binary, windows\sanguis-setup.exe, with the same verbs plus --dry-run and --uninstall, and on POSIX a single-file .run carries the whole kit and calls install.sh itself.

Quick Start

Build the floor, the C harness that runs everything else:

```
bash tools/hostfloor.sh
```

Compile and run the first program:

```
bash tools/sg.sh examples/hello.sanguis examples/hello.bc

./tools/bin/bch examples/hello.bc main
```

The second command prints one line:

```
hello from the 4.0.0-RC1 toolchain kit
```

Next, machine cells. Compile them, pack them into a module, and run the known-answer driver against the module:

```
bash tools/sg.sh examples/cellsdemo.sanguis examples/cellsdemo.bc

bash tools/mkblob.sh examples/cellsdemo.bc examples/cellsdemo.mod

gcc -O2 -o blobtest/blobrun blobtest/src/blobrun.c -lm

./blobtest/blobrun examples/cellsdemo.mod
```

On Windows the same three steps with the shipped tools:

```
windows\sg.bat examples\cellsdemo.sanguis examples\cellsdemo.bc

powershell -NoProfile -ExecutionPolicy Bypass -File windows\mkblob.ps1 examples\cellsdemo.bc
examples\cellsdemo.mod

windows\bin\blobrun.exe examples\cellsdemo.mod
```

Expected:

```
blobrun: 10 tests, 0 failures
```

The roles battery runs the same way through bch:

```
bash tools/sg.sh examples/roles.sanguis examples/roles.bc

./tools/bin/bch examples/roles.bc main
```

The output includes six lines named t1 through t6, each ending in ok, plus summon arrived and closer ended the guard.

The image example compiles cells, packs them, and writes three image files:

```
bash tools/sg.sh examples/shader.sanguis examples/shader.bc

bash tools/mkblob.sh examples/shader.bc examples/shader.mod
```

```
gcc -O2 -o blobtest/shade blobtest/src/shade.c -lm
cd examples && ../blobtest/shade shader.mod 160 120 3 && cd ..
```

The driver checks three complete frames against an independent C implementation of the same mathematics, then writes mandel-0.bmp, mandel-1.bmp and mandel-2.bmp. If the driver's last line answers probe mismatches 0, the module's arithmetic matched the reference on every checked pixel. Open the .bmp in any image viewer.

A single command runs all of the above in the right order:

```
bash _smoke.sh
```

A passing run ends with:

```
ROLES_OK
SHADER_OK
FIXED_POINT_OK
smoke done
```

The Tools

```
hostfloor.sh    Build the floor. Produces tools/bin/bch and its helpers.
                Run once after every unzip.
sg.sh           Compile a .sanguis source file to a .bc.
mkblob.sh       Pack a .bc into a module: text the host can load.
mklayout.sh     Layout helper.
```

Call the tools from the kit root with examples/ style paths, as the smoke script does. Relative paths from inside a subfolder fail.

Writing Programs

Cells

Sanguis carries two kinds of cell. They compile with the same tool and cooperate only at one boundary, described later in this section.

```
Kind:       float
Runs under: bch, interpreted
Speaks:     text, lists, strings, and ordinary float arithmetic
```

```
cell main() -> float {
    print("hello from the 4.0.0-RC1 toolchain kit");
    output 0.0;
}
```

The compiler waits for the output line. Values leave through it.

```
Kind:       u64 and i64
Runs as:    native machine code
Speaks:     exact 64-bit integers and machine operations
```

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

Sys cells run from modules at native speed. Their common operations are add, mul, div, band, bxor, shl, shr, store64, load64 and store8. The exact semantics of several of these differ from what you would guess, and they are all listed under Technical Notes. Read that section once before writing integer code.

Layouts

A layout is a record: a field list with a size the compiler computes. Two accessors come with every field, get and set.

```
layout Demo2 {
    a: u32;
    b: u64;
}
```

On this floor, Demo2 measures 16 bytes. blobtest/blobrun.c exercises both accessors on both fields among its ten checks. Its source is short, and it is the working pattern for known-answer tests of your own.

Modules

A module is a plain text file with three parts. BLOB1 gives the code byte

count. BLOBOFF gives one line per cell: its name and its offset. BLOB2 carries the machine code in hex. Loading takes three steps:

1. read the file, take the count, the offsets, the hex
2. map the code bytes as readable, writable, executable
3. call base + offset. Arguments pass in the machine's integer registers, the answer returns in rax.

That is the whole interface. blobrun.c and shade.c are two working C loaders, thirty lines each, and the same shape loads from Python, Rust, Go, C# or any other language that can map a file and call an address. Start from one of them. The one rule at this boundary is the 2^53 note in Known Traps.

Images

The kit renders frames of pixels as files.

```
Buffer:    three bytes per pixel, blue, green, red, rows top-down
Driver:    blobtest/shade.c: flips rows, writes standard 24-bit .bmp
Shader:    examples/shader.sanguis
```

A shader cell is a pure function of position and a frame number:

```
mandel_color(px, py, w, h, frame) -> u64
shade_frame(buf, w, h, frame) -> u64
```

Values are Q16 fixed point: one unit is 65536. The shader's arithmetic is exact integer work end to end. shade_frame fills an entire frame in one native call, and the driver writes it to disk. The shipped example draws a zooming Mandelbrot across three frames at 160x120. Nine probe pixels per frame are compared against an independent implementation, and the driver stops with an error on any disagreement. Compute expected values outside the module, the way this driver does, and check against them.

There is no live window in this kit. Frames are written to disk, and any host language can display them.

The Five Roles

Five kinds of cell take a job title in their declaration, and the compiler builds the machinery behind it. No new builtins, no new verbs. The roles battery in examples/roles.sanguis runs all five and grades its own output.

A stem hands its calls to another cell, and takes them back:

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

```
let d = differentiate("st", "other");
let u = undifferentiate("st");
```

A tagger walks a list, passes each item through a gate you wrote, and returns pairs. A pruner keeps the items its tagger graded above one half:

```
let pairs = tag_tg([1.0, 5.0]);
let kept = prune_pp([1.0, 5.0]);
```

A helper is a small named value or call-in:

```
cell assign helper motd = "a long text";
```

An immune cell is a guard:

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

awaken_g() puts it on duty as its own task. summon_g(42.0) wakes it through the floor's mailbox, and patrol(2) runs its body every two ticks. A guard does not stop on its own, so post a closer with it:

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

spawn closer(t) with the guard's id, and run_all() returns to main once every task has ended. The battery uses this pattern for its guard. Give every guard a closer.

Checking Your Work

The fixed point proves the compiler in the kit reproduces itself byte for byte:

```
bash tools/sg.sh toolchain/lib/compiler.sanguis _fixedpoint.bc
cmp seed/compiler.bc _fixedpoint.bc && echo FIXED_POINT_OK
```

The checksums prove every file matches what shipped:

```
sha256sum -c SHA256SUMS.txt
```

Run both before handing work to anyone else.

Known Traps

Each entry states the rule, what a violation does, and the correction. None is hypothetical: every one here was measured on this floor.

1. Infix shift operators do not exist in the sys path. A program that uses infix `>>` does not compile. Write shr(a, 2) and shl(a, 2).
2. WARNING: division in the sys path acts on the stored 64-bit pattern as an unsigned value. Dividing a negative value returns the unsigned quotient of the wrapped pattern, for example 10 * 2 * (0 - 1) / 4 = 4611686018427387899, which is exactly (2^64 - 20) / 4. To divide a possibly negative value by a power of two, use the bias construction:

```
let p = zr * 2 * zi;
let p = p + 0x8000000000000000;
let p = shr(p, 16);
let p = p - 0x800000000000;
```

   That yields floor of the signed division. The shipped shader uses it throughout and matches an independent C reference at every probe. The measured answers are printed by examples/divide_probe.sanguis.
3. Numbers past 2^53 lose precision if they pass through a float. Carry them as u64 in modules, end to end.
4. Division by zero answers 0, with no error. Guard every denominator that input data can reach.
5. The two kinds of cell do not call each other directly. Float programs run under bch. Sys cells are called through modules from the host language. Plan the boundary; do not bridge it casually.
6. run_all() returns only after every task has ended. A guard ends only through kill, so a guard without a closer holds the program open forever. The pattern is in The Five Roles.
7. task_run_all is recognized and not implemented in this floor. Use run_all() and a closer. The omission is recorded on the toolchain's own notes as an open item.
8. WARNING: the privileged instructions (port I/O, MSRs, rdtsc, cpuid, cli, sti, hlt) compile anywhere and execute only under the Sanguis OS. A module that runs one on an ordinary host faults the host. Keep them out of modules unless the machine is yours.
9. The seed compiler and the committed floor are frozen. If toolchain/lib/compiler.sanguis is edited locally, the fixed point breaks and the kit is no longer the kit. Either restore the seed or run the

fixed-point check before trusting a rebuild.

10. Run a module only after its checksums verify. The kit's own check is sha256sum -c SHA256SUMS.txt from the kit root; SEED.sha256 pins the compiler alone.
11. Author source files with Unix (LF) line endings. The smoke script strips carriage returns from the shipped examples only.
12. Do not report success from one green line. The gate runs ten known-answer checks, five roles, three probe-checked frames and a fixed point. If any step fails, the script stops at that step and names it.

If Something Goes Wrong

`bch: No such file or directory`	The floor is not built. hostfloor.sh.
`mkblob: FAILED ... cannot open`	The call passed bare names from a subfolder. Call from the kit root with examples/ prefixed paths, as the smoke script does.
`SPIN at blob offset ..., in cell NAME`	The alarm fired at sixty seconds and named the cell. The cell does not return. Usual cause is a loop whose exit condition can never go false.
`FIXED_POINT_OK missing`	The compiler source moved. The seed is the master copy. Investigate the rebuild before trusting it.
`probe mismatch N`	Module arithmetic and the reference disagree at one pixel. The frame number and pixel are printed. Integer rounding is the usual suspect.

Technical Notes

Measured on this floor:

```
64-bit add, sub, mul:        wrap, exact in the pattern
divide, remainder by 0:      answer 0
i64 minimum / -1:            stays the minimum
i64 remainder by -1:         answers 0
shift of 64 or more:         answers 0 (u64 and shl), sign fills for i64 shr
integer type of a cell:      the type of its ->; it types every variable,
                             operation, comparison and division in the cell
```

That last line is the load-bearing rule, measured with receipts. A cell is one integer type, its `->` declares it, and that type's divide runs for the whole cell: in a `u64` cell the divide acts on the stored 64-bit pattern, unsigned, and compares answer unsigned; in an `i64` cell the divide emits the signed form (with the special cases above) and compares answer signed. `a < 0` answers 1 for `a = -20` in an `i64` cell and 0 in a `u64` cell; `-20 / 4` answers -5 in an `i64` cell, even through a `let` and an added zero. Practical consequences: signed fixed-point work in a `u64` cell uses the 2^63 bias construction (shown in the shader's source), and a computation that needs true signed division or a signed compare belongs in a cell that declares `-> i64`. The design record and the backend's comment block carry the same rule now.

Two more items are known and open. The working-tree hosted build of the floor was mid-edit when this kit was cut, so the kit ships the last committed floor instead, with its one local fix documented in the README. And the float side of the language (lists, strings, records) runs in the interpreter under bch: fully usable, and not part of the module path.

Glossary

bc: a compiled program. sg.sh produces it.

blob: the machine-code stage packed into a module for native calling.

cell: the named unit of a Sanguis program. Takes values in, hands one value back.

closer: an ordinary task that ends another task. The way run_all() returns.

fixed point: the byte-for-byte reproduction of the seed by itself.

floor: the C harness under everything. On a host: files, stdout, memory, software ticks. Under the OS: hardware, real clocks, privilege.

guard: an immune cell on duty. Answers summons, patrols on ticks, ends by kill.

layout: the record type. A field list with a computed size and two accessors per field.

module: a .bc packed for calling: a count, a table of offsets, and the code in hex.

probe check: a driver that compares the module's answers at sampled points against a reference computed outside the module.

Q16: fixed point with sixteen binary places. One unit is 65536.

roles: helper, stem, tagger, pruner, immune. Declared at the cell, wired by the compiler.

seed: the frozen compiler: compiler.bc, pinned by hash.

summon: a message that wakes a guard's body on the mailbox.

sys cell: the integer species. Runs as native code from a module.

tick: the floor's unit of time. Software on a host, hardware under the OS.

toolchain: everything that turns source into running programs.

About This Kit

```
Version:        4.0.0-RC1-P-toolchain
Issued:         November 2026
Built:          from the Sanguis OS toolchain family, seed pinned
Seed hash:      15eb534381eaae4d6105d6a91b0060190fc9f7073395a0bf07e582554f1a399f
Linker hash:    85836ef29df638024a63b849d51d6f6124f19d4a176c5ee75c6416c1fc6cdeba
```

The README beside this manual is the long form: provenance, the module contract in full, and every shipped hash. Read it second. Run the smoke first.