ucode

A small ECMAScript-like scripting language for Linux systems

A comprehensive ucode programming manual

Reading order. Part I is a tutorial and reads straight through. Parts II–III are a reference; you can drop into any section. Part IV assumes you want to put ucode inside your own C program. Part V is about where ucode lives in the wild.

Part I: The language

1. **Introduction** · What ucode is, why it exists, how it differs from Lua, JavaScript and shell scripts; the shape of the implementation; license and authorship.
2. **Installing ucode** · Building with CMake, the feature toggles and what each costs, packaging in OpenWrt and Debian, cross-compiling for a router, the runtime library search path.
3. **A first program** · `print`, statements and blocks, comments, the `-e` one-liner, exit status, shebang lines, `ucc` and `utpl`.
4. **Values and types** · The value types, `type()` and its tenth name `resource`, truth and coercion, signed and unsigned integers, the number line's edge cases.
5. **Names, scope and bindings** · `let`, `const`, global variables, function scope, block scope, closures and upvalues, strict mode.
6. **Operators** · Arithmetic, comparison and the two notions of equality, logical and nullish operators, bitwise operators and unsigned results, assignment forms, `delete`, `in`, optional chaining, the precedence table, the division-by-zero and `**` associativity quirks.
7. **Control structures** · `if`, `while`, `for(;;)`, `for ... in` with one and two loop variables, `switch`, the brace-free alternative syntax, `break` and `continue`.
8. **Functions** · Declarations, expressions, arrow functions, methods, `this`, forward declarations, arity and missing arguments, recursion limits, tail calls, making values callable with `__call__`.
9. **Strings** · Literals and escapes, byte semantics and UTF-8, the length question, `sprintf` / `printf` and every format specifier, the string builtins.
10. **Arrays** · The array value, indexing and negative indices, holes and `length`, growth, the array builtins, copying and aliasing.

11. **Objects** · Ordered hash tables, key stringification and the null-byte rule, insertion order, literal syntax, merging with spread, `keys`, `values`, `exists`, `rawget` / `rawset` / `rawdelete`.
12. **Prototypes and metamethods** · The prototype chain, `proto()`, inheritance without classes, the five metamethods, metamethods as fallbacks, `in` and prototypes, the `this` binding.
13. **Regular expressions** · POSIX ERE, flags, `regexp()`, `match`, `replace`, `wildcard`, capture groups, why patterns are not Perl.
14. **Errors and exceptions** · Runtime errors, the exception types, the exception object, `try` / `catch` (there is no `throw` statement and no `finally`), `die`, `assert`, `warn`, `exit`, tracebacks.
15. **JSON and other notations** · `json()` for parsing (text and stream), `%J` for serialising, the ucode-literal round trip, what cannot be serialised, and why there is no `json.parse`.
16. **Templates** · Template mode (`-T`), expression / statement / comment tags, block bodies and their terminating words, whitespace control, `render()`, generating configuration files.
17. **Modules and program organisation** · `import` / `export`, dynamic `import()`, `require()`, `include()`, the search path and the `modules` registry, writing and shipping a script module.
18. **Memory** · Reference counting and the cycle collector, no finalisers, `gc()` and `-g`, roots, closures, measuring.
19. **Idiosyncrasies** · The behaviours that surprise people arriving from JavaScript, Lua or shell, and why they are that way; a migration appendix in miniature.

Part II: The standard library

1. **The core environment** · Predefined globals (`ARGV`, `SCRIPT_NAME`, …), how builtins are registered, the full builtin index.
2. **Strings and formatting** · Complete reference for the string builtins, with the `printf` specifier table reproduced in full.
3. **Arrays and objects as containers** · The functional toolkit (`map`, `filter`, `sort`, `slice`, `splice`, `push`, `uniq`, …) and the recipes built from it: chunking, flattening, grouping, de-duplicating, merging.
4. **Time** · `time`, `clock`, `sleep`, `localtime`, `gmtime`, `timelocal`, `timegm`, monotonic versus wall clock.
5. **math** · Every function, the exported constants, seeding, platform-dependent results.
6. **fs** · Paths, stat and the stat fields, directories, walking, file helpers, the `/proc` interface, `fs.proc` handles, the `O_*` constants.
7. **io** · File handles, the `io` module, read/write modes, buffering, seeking, `popen`, pipes, non-blocking handles.
8. **struct and binary data** · The `pack` / `unpack` format language, byte order, padding, the buffer object.
9. **digest, zlib, base64 and hex** · Hashing (incremental and one-shot), the available algorithms and how they are selected, compression and decompression streams, `b64enc` / `b64dec`, `hex`, `hexenc`.

10. **ffi: calling C without writing C** · Declaring symbols, type names, calling conventions, callbacks, memory, and the limits of the approach.
11. **Processes and signals** · `system()`, `signal()`, `fs.popen`, `uloop` processes, exit codes and the shell's involvement.
12. **log** · Writing to the system log from ucode, priorities and facilities, identity and backend selection.

Part III: Networking and system integration

1. **socket** · Creating sockets, the address table forms, the method set, datagrams and streams, timeouts and non-blocking operation, `recvfrom` / `accept` return shapes, the constant tables.
2. **resolv and netaddr** · Resolver calls, and the `netaddr` address/CIDR value model — constructing, validating, matching and containment.
3. **rtnl: routing and interfaces via netlink** · The object-schema model, links, addresses, neighbours, routes, rules, traffic control, and the mutating `set` calls.
4. **nl80211: wireless** · Interface dumps, station tables, scans, the capability/flag enumerations, how this module drove the new Wi-Fi stack.
5. **uloop: the event loop** · Timers, intervals, I/O watchers, signals, processes, running and stopping, and writing a daemon in ucode.
6. **ubus** · Connections, calls and status codes, objects and procedures, subscribe/notify, and publishing a service from ucode.
7. **uci** · Cursors, sections and options, iteration, changes and commits, arrays inside UCI, the delta against `uci show`.
8. **serial** · Serial ports, baud rates and line settings.

Part IV: Embedding ucode in C

1. **An embedding overview** · What to link, what to include, the shape of the API, a 40-line host program walked line by line.
2. **Values in C** · `uc_value_t`, the tagged representation, creation and access for every type, and the ownership rules that decide whether you must call `ucv_get`.
3. **The virtual machine state** · `uc_vm_t`, the scope stack, the registry, the globals table, running code and reading results.
4. **Compiling sources** · Sources and their names, `uc_compile`, the shape of a compile error, writing and loading a program.

5. **Native functions** · `uc_cfn_ptr_t`, arguments and their ownership, returning values, raising exceptions, registration tables, calling back into the script, and receivers.
6. **Resource types** · Declaring a type, the plain and extended shapes, data accessors and their type checks, value slots, and the release callback.
7. **Exceptions and signals in an embedder** · The five statuses and what each hands back, the exception record and its handler, break requests and resume, and the signal self-pipe.
8. **Programs, bytecode and precompilation** · The program API, the file format and its flags, what debug information costs and needs, the magic and version checks, precompiled modules, and the compile switches.
9. **Writing a native module** · The module ABI, `uc_module_init`, what the entry point is handed, building a `.so` against the installed headers without linking the library, and where a module's state lives.
10. **The six example programs** · A guided tour of `examples/`: `execute-string`, `execute-file`, `native-function`, `exception-handler`, `state-reuse` and `state-reset`, what each demonstrates, and what a VM costs to make.
11. **A worked embedding** · Three assembled hosts covering every run outcome including break recovery, three scope-lifetime policies around one script, and C-owned objects exposed through typed resources whose methods scripts call directly.
12. **Inside the interpreter** · Lexer to compiler to bytecode to VM: the pipeline, the instruction set in full, the calling convention, upvalues and closures, tail calls, the bytecode file format, and how to read a disassembly.

Part V: Deployment and ecosystem

1. **Deployment models** · The standalone CLI, scripts in `/bin` with a shebang, precompiled bytecode for flash-constrained devices, the interpreter as a library, persistent daemons, and a comparison of the four.
2. **uhttpd: ucode as a web backend** · Dispatch rules, the request environment, the `http` API, templates as pages, per-request interpreter semantics, configuration and hardening.
3. **uwsd: a persistent ucode web server** · Why a second server, its request routing and module API, async handlers on an event loop, TLS and CGI compatibility, and migrating a uhttpd script.
4. **rpcd: ucode as an ubus service** · Publishing ubus objects from `.uc` scripts, procedure tables, calling them from `ubus call` and LuCI.
5. **Case study: firewall4** · The largest ucode program in OpenWrt: UCI to nftables, the template pipeline, the module layout, and what it teaches about structuring ucode programs.
6. **Case study: the LuCI ucode runtime** · The `luci` module API, page templates and dispatch, the CBI successor, and how a Lua page becomes a ucode page.
7. **Case study: Wi-Fi** · The `wifi-scripts` detection and generation scripts, their use of `nl80211` and JSON schemas, and the ucode support that OpenWrt's `hostapd` package carries.

8. **The wider ecosystem** · Who else embeds or uses ucode: a survey of repositories, packaging across distributions, editor and tooling support, and how to find ucode in a codebase.
9. **The debugger** · `ucode -x` and `-X`, the `udbg` client, breakpoints, stepping, inspecting and disassembly, the wire protocol, and driving the debugger from a tool.
10. **Testing and tooling** · The test suite layout, writing tests in ucode, fuzzing, generating this book's own reference from C comments, and editor syntax files.

Appendices

Only G exists today; A to F are the ones still to write.

- **A. Keywords and grammar** · The reserved words and a syntax summary.
- **B. Operator precedence** · The table, verified against the compiler.
- **C. Builtin index** · Every builtin and module function in one alphabetical list.
- **D. C API index** · Every public symbol, grouped by header.
- **E. From Lua, JavaScript and shell** · A translation table for the three languages people arrive from.
- **F. Further reading** · Documentation, the reference site, mailing lists, related projects.
- **G. Source revisions and links** · The commit each quoted file was read at, and an address carrying it for every file and line span the book quotes.

Where to find the latest version

The latest version of this book is maintained in its own repository, `https://github.com/ucode-lang/manual`, and is published as a web page at `https://ucode-lang.github.io/manual`.

Introduction

ucode is a small scripting language with ECMAScript syntax, written in C, that runs on Linux systems — particularly on routers. It is an interpreter you can put in 30 KB, a library you can link into a daemon, and a template engine, and it speaks JSON natively. It is not a browser's JavaScript, and it is not an attempt to become one.

What ucode is

A ucode program is a sequence of statements in a syntax a JavaScript reader can follow without a translation layer: `let`, `const`, `function`, arrow functions, objects with identifier keys, template literals, `for` / `while` / `switch`, truthiness and coercion rules that mostly match. Under that syntax is a different, smaller thing: byte strings rather than character strings, no classes, no `throw` statement, no coroutines, and no asynchronous anything. Errors are exceptions and are catchable, but there is nothing to await, and a program cannot raise one itself except with `die()`. Values are JSON-shaped — null, boolean, integer, double, string, array, object — plus three kinds that JSON has no word for: regular expressions, functions, and resources (file handles, sockets, connections).

The language comes with a standard library in loadable modules rather than a monolithic runtime: file system, I/O, math, time, JSON, regular expressions, struct packing, digests, zlib, sockets, serial ports, a POSIX-regex engine, DNS resolution, netlink routing and wireless, the OpenWrt `ubus` message bus and `uci` configuration API, an event loop, and a C FFI. Which modules exist is a build-time choice (chapter 2); which you use is a `require()` away (chapter 17).

Three things characterise it in use:

- **It is synchronous.** There is one flow of control, and concurrency, when needed, comes from the event loop module (`uloop`, chapter 36) rather than from the language.
- **It is data-first.** `sprintf("%J", value)` renders any value as JSON and `json(text)` parses it back, so a script that talks to `ubus`, `rpcd`, `uhttpd` or a config file is not writing a serialiser.
- **It is a system tool.** Modules reach the operating system: routes, interfaces, wifi, DNS, serial consoles, processes, signals, and the raw netlink and ioctl level below them.

Why it exists

The development of ucode was motivated by the need to rewrite OpenWrt's firewall framework around nftables: `firewall4` needed to turn a declarative UCI configuration into nftables rules, which is a job for a language with real data structures and templates, not for shell scripts parsing text. ucode began as a template processor for exactly that and grew into the general system scripting language the modules above describe. Its design goals, in the order the source states them, are easy integration with C applications, efficient handling of JSON data and complex data structures, support for the `ubus` message bus, and a broad set of built-in functions in the spirit of Perl 5 — plus a small executable size.

That history explains most of the shapes that look unusual coming from JavaScript. Templates are a first-class mode of the interpreter itself — a separate Jinja-style template processing with its own command (`utpl`) and its own markup, distinct from the backtick template literals of the language itself (chapter 16). The builtins are Perl's vocabulary — `substr` , `index` , `rindex` , `splice` , `shift` , `unshift` , `split` , `join` , `trim` , `hex` , `ord` , `chr` — as functions, because on a device with 8 MB of flash, methods on values cost dictionary lookups the language can avoid by simply not having them. Synchronous flow, byte strings, and a value model that maps one-to-one onto JSON are the same trade: less machinery, predictable cost.

How it differs from its neighbours

From JavaScript. Same surface syntax, deliberately smaller semantics. Values are not objects: `"abc".length` and `arr.push(x)` are errors, not silent `undefined` s (chapter 9, chapter 22). There are no classes, no `throw` statement, no `async` / `await` /promises, no generators, no destructuring, no default parameters, and no `typeof` operator — `type(v)` is a function that returns a string. `try` / `catch` does exist and catches real exception objects; there is no `finally` clause (chapter 14). `==` on arrays and objects compares identity, as in JavaScript, so a copy of a structure is not equal to its original, and the integer type is 64-bit signed with `/` on two integers truncating toward zero (chapter 6). Chapter 19 lists the differences in one place; they are worth reading before writing anything longer than a line.

From Lua. ucode replaces Lua in the same niche that Lua occupied in OpenWrt, and the differences run the other way: the syntax is C-like rather than Algol-like, `keys()` / `values()` replace `pairs()` , `length()` replaces `#` , `+` concatenates strings, `!=` is inequality, and metamethods live in a value's *prototype* rather than in a side table passed to `setmetatable` (chapter 12). There are no coroutines. `printf` is available under its C name rather than as `string.format` .

From shell. Everything that makes shell painful for configuration logic — text as the only data type, word splitting, quoting, subshells as function calls — is absent. `system()` runs a command and returns its *exit status*, not its output; to capture output you `fs.popen()` and read from the handle. The argument-array forms of `system()` and `fs.popen()` bypass `/bin/sh` entirely, which is what you want whenever any part of the command comes from outside the script. In return you inherit a language where a typo in a name is a runtime error rather than a silently empty string, and where a missing quoting decision fails loudly.

The implementation

One repository builds four things:

```
libucode.so        the interpreter: lexer, compiler, VM, value layer, core
library
ucode              the command line interpreter
udbg               the debugger client
lib/ucode/*.so     the optional modules (fs, socket, ubus, ...)
```

The core is about 25,000 lines of C — `lib.c` (the builtin functions) 6,300, `compiler.c` 4,300, `vm.c` 3,800, `types.c` 3,100, `lexer.c` 1,400, the rest split between source handling, bytecode images and the command line. The modules add roughly 44,000 more, the largest being the netlink, socket, struct and event-loop bindings. A size-optimised build of `libucode.so` is around 170 KB of text and the `ucode` binary 16 KB — the interpreter is a shared library precisely so that several host programs can carry one copy of it.

The only hard external dependency is **json-c**, used for parsing and serialising JSON. Everything else is optional and is probed for a library at configure time: `zlib`, `libmd` (digests), `libffi`, `libubox`, `libubus` + `libblobmsg-json`, `libuci`, `libnl-tiny`. Regular expressions are compiled by the host C library through `regcomp()`, so their exact feature set is the host's POSIX ERE, not ucode's (chapter 13).

Execution runs lex → single-pass compile → bytecode → VM, with the bytecode image optionally written to a file by the compiler (`ucc`, chapter 3) so a device can load programs without compiling them. Values are reference-counted, with a periodic mark-and-sweep pass over unreclaimed objects to catch cycles; the pass runs on an allocation interval and can be triggered from a script with `gc()`. There are no threads in the language; concurrency is the event loop's business, and the C API's.

Embedding is a central design goal rather than an afterthought: `rpcd` and `uhttpd` both embed ucode, and the `examples/` directory of the source tree holds small C programs that execute a file, execute a string, add a native function, and install an exception handler. Part IV of this book describes that API.

Where things come from

ucode is written by Jo-Philipp Wich and released under the ISC license, a two-clause permissive licence; the whole tree, including the modules, carries it. Development happens in the `ucode-lang/ucode` repository, with releases tagged by date (`v0.0.20250529` and so on) and packaged into Debian (`ucode`, `ucode-modules`, `libucode`, `libucode-dev`) and into OpenWrt (`ucode`, `libucode`, and one `ucode-mod-*` package per module). The generated reference documentation lives at ucode.mein.io, derived from the same source comments this book cites.

The notable users define what the language is for: **firewall4**, the OpenWrt firewall and the origin of the language; **LuCI**, the OpenWrt web interface, whose `luci-lib-ucode` bindings and templates run on ucode; **rpcd** and **uhttpd**, which embed the interpreter; and the `hostapd` / `wpa_supplicant` ubus glue and `wifi-scripts` that implement modern OpenWrt wireless configuration. Part V of this book examines what each of them does with it.

A taste

The following script reads a UCI configuration and prints the interfaces that are up, which is the shape of most real ucode programs:

```
let rv = { ok: true, ifaces: [
        { name: "lo", up: true, mtu: 65536 },
        { name: "lan", up: true, mtu: 1500 },
        { name: "wan", up: false, mtu: 1500 }
] };

let up = filter(rv.ifaces, (i) => i.up);

printf("%d up: %s\n", length(up), join(", ", map(up, (i) => `${i.name}/${i.mtu}`)));
printf("%J\n", map(up, (i) => i.name));
```

```
2 up: lo/65536, lan/1500
[ "lo", "lan" ]
```

No imports were needed: arrays, objects, `filter`, `map`, `join`, `sprintf`, and template literals are all part of the core (chapter 20). Reading that structure from a real configuration file, from a `ubus` call, or from a JSON string takes three different one-liners, and each of them has its own chapter.

Conventions used in this book

Code examples appear in `ucode` blocks, showing the script as a ucode program, with the interpreter's output for the same example in the `text` block that follows it. Where an example is meant to be run at a shell prompt, a `console` block shows a shell session, with `$` denoting the prompt. Where a behaviour is version-specific — a POSIX regex feature, a libc `printf` quirk — the chapter says so explicitly rather than relying on the examples.

Summary

Question	Answer
What syntax?	ECMAScript-like, with a smaller semantics than JavaScript
What is a string?	bytes, not characters; UTF-8 by convention
What is data?	the seven JSON types plus regexp, function, resource
Concurrency?	synchronous; `uloop` event loop when needed
Standard library?	core builtins (Perl-flavoured) plus loadable modules
Dependencies?	json-c; everything else optional
Regexp engine?	the host libc's POSIX ERE
Who uses it?	firewall4, LuCI, rpcd, uhttpd, OpenWrt wireless scripts
Licence?	ISC

Installing ucode

What you need

ucode is written in C99 with GNU extensions, is built with CMake 3.13 or later, and relies on **json-c**, which is the one hard dependency: it is probed with `pkg_check_modules(JSONC REQUIRED json-c)` and the build stops without it. Everything else is optional and is enabled by finding the library at configure time.

ucode has been tested with glibc and musl libc on Linux, and on OS X. `_GNU_SOURCE` is always defined, and a libc beyond those is on its own. `libdl` is linked if `dlopen` is not already in libc and `libm` if `fmod` is not; the `math` module additionally links `libm` if `ceil` is missing from libc. Because regular expressions are compiled by the host C library, the `regexp` feature set is whatever the target's `regcomp()` implements (chapter 13) — a musl target and a glibc host will not agree on every construct.

Building

```
$ git clone https://github.com/ucode-lang/ucode && cd ucode
$ cmake -B build
$ cmake --build build -j4
$ sudo cmake --install build          # or: sudo make -C build install
```

CMake prints what it found and what it therefore will build. A configure run on a machine with only json-c produces the core, the CLI, the debugger and the modules that need nothing else:

```
$ cmake -B build
-- Found JSONC: ...
-- Configuring done
$ ls build/*.so
build/debug.so   build/io.so     build/resolv.so  build/socket.so
build/fs.so      build/log.so    build/math.so    build/struct.so
build/serial.so
```

That listing is the answer to "why is `require('uci')` failing on my build host" more often than any other explanation: the module was never built because its library was absent. The `modules` global and the failure message of `require` (chapter 17) tell you the same thing at run time.

Running from the build tree

The built `build/ucode` does not know about `build/*.so` unless the search path says so — pass `-L` :

```
$ build/ucode -L build -e 'const fs = require("fs"); printf("%J\n",
fs.access("/etc"))'
true
```

Without `-L build`, `require("fs")` searches the compiled-in default path — which on a development checkout points at an installation prefix that holds nothing. For scripts, the same `-L` works, and so does setting `REQUIRE_SEARCH_PATH` inside the program (chapter 20).

The feature toggles

Each module is a CMake option. Those with a fixed `ON` default always build; the others default to `ON` **only if** their library was found, which is why the same source tree configures differently on a laptop, a build container and a router SDK.

Option	Needs	Builds
`FS_SUPPORT`	—	`fs.so` (chapter 25)
`IO_SUPPORT`	—	`io.so` (chapter 26)
`MATH_SUPPORT`	— (`libm` if `ceil` absent)	`math.so` (chapter 24)
`STRUCT_SUPPORT`	— (`libm` optional)	`struct.so` (chapter 27)
`SOCKET_SUPPORT`	—	`socket.so` (chapter 32)
`SERIAL_SUPPORT`	—	`serial.so` (chapter 39)
`LOG_SUPPORT`	— (`libubox/ulog.h` optional)	`log.so` (chapter 31)
`RESOLV_SUPPORT`	— (`libresolv` optional)	`resolv.so` (chapter 33)
`DEBUG_SUPPORT`	— (`libubox/uloop.h` optional)	`debug.so`, the debugger support module
`ZLIB_SUPPORT`	zlib	`zlib.so` (chapter 28)
`DIGEST_SUPPORT`	libmd	`digest.so` (chapter 28)
`DIGEST_SUPPORT_EXTENDED`	libmd	the extra hash algorithms in `digest.so`
`FFI_SUPPORT`	libffi	`ffi.so` (chapter 29)
`UBUS_SUPPORT`	libubus + blobmsg_json	`ubus.so` (chapter 37)
`UCI_SUPPORT`	libuci + libubox	`uci.so` (chapter 38)
`ULOOP_SUPPORT`	libubox	`uloop.so` (chapter 36)
`RTNL_SUPPORT`	libnl-tiny + libubox, Linux only	`rtnl.so` (chapter 34)
`NL80211_SUPPORT`	libnl-tiny + libubox, Linux only	`nl80211.so` (chapter 35)

Three more options change the shape of the binaries rather than the set of modules:

Option	Default	Effect
`BUILD_OPTIMIZE_SIZE`	`ON`	size-optimising compile flags; this is what keeps `libucode` near 170 KB of text
`BUILD_FUNCTION_SECTIONS`	`ON`	`-ffunction-sections`, paired with `LINK_GC_SECTIONS` (`--gc-sections` at link time, off on Apple platforms)
`COMPILE_SUPPORT`	`ON`	compiles from source at run time (`loadstring`, `require` of `.uc` sources) and installs the `ucc` name; turning it off defines `NO_COMPILE`

To build a specific set, name the toggles on the configure line. This is the invocation that produces a bare router-side interpreter with no compiler and nothing but the three OpenWrt-facing modules:

```
$ cmake -B build \
    -DCOMPILE_SUPPORT=OFF \
    -DFFI_SUPPORT=OFF -DDIGEST_SUPPORT=OFF -DZLIB_SUPPORT=OFF \
    -DRTNL_SUPPORT=OFF -DNL80211_SUPPORT=OFF -DSERIAL_SUPPORT=OFF \
    -DUBUS_SUPPORT=ON -DUCI_SUPPORT=ON -DULOOP_SUPPORT=ON
```

What the toggles buy in bytes is worth knowing when the target is a small embedded device. On a size-optimised x86-64 build of the same revision, the modules range from 21 KB for `log.so` to about 105 KB for `nl80211.so`, `ffi.so` and `rtnl.so`, with the core library at 208 KB installed and the interpreter itself at 32 KB:

Artifact	Installed size
`ucode`	32 KB
`libucode.so.0`	208 KB
`log.so`, `zlib.so`	21–22 KB
`resolv.so`, `io.so`, `math.so`	27–28 KB
`serial.so`, `uci.so`	31–43 KB
`uloop.so`, `fs.so`, `struct.so`	43–49 KB
`ubus.so`, `digest.so`, `socket.so`	57–83 KB
`debug.so`, `nl80211.so`, `ffi.so`, `rtnl.so`	86–106 KB

The `debug` module and `udbg` are the pieces to drop first on a device that only runs precompiled scripts; `ffi` is the one to drop before anything that talks to the kernel, since it drags in a C declaration parser.

What gets installed

```
${bindir}/ucode            the interpreter
${bindir}/udbg             the debugger client (chapter 42)
${bindir}/ucc              symlink to ucode, compile mode (chapter 3)
${bindir}/utpl             symlink to ucode, template mode (chapter 16)
${libdir}/libucode.so.0    the VM, with a libucode.so symlink for developers
${libdir}/ucode/*.so       the enabled modules
${prefix}/include/ucode/*.h   the embedding API (part IV)
```

The two extra command names are plain symlinks created at build time — the interpreter reduces `argv[0]` to its basename and behaves accordingly, so a distribution that installs only `ucode` can still offer the other two modes with `-c` and `-T`.

The default module search path is a build-time string:

```
<prefix>/<libdir>/ucode/*.so : <prefix>/share/ucode/*.uc : ./*.so : ./*.uc
```

`-L dir` prepends `dir/*.so` and `dir/*.uc` to it (a `-L` argument containing `*` is taken verbatim), and `-L` may be repeated. Since the path is compiled in, moving an installation means rebuilding it or carrying `REQUIRE_SEARCH_PATH` in the environment of every program — the packaging in both distributions keeps the default consistent by installing to the standard prefix.

Testing the build

Tests are enabled by a CMake definition rather than an option, and run through CTest:

```
$ cmake -B build -DUNIT_TESTING=ON
$ cmake --build build -j4
$ ctest --test-dir build --output-on-failure
```

The suite is written in ucode itself — `tests/custom/` holds directories named `00_syntax`, `01_arithmetic`, `02_runtime`, `03_stdlib`, `04_modules`, `06_metamethods`, `99_bugs`, `99_debugger` and more, driven by `tests/custom/run_tests.uc` — with a handful of `cram` tests for the command line and a `tests/fuzz` corpus. When the compiler is Clang, `UNIT_TESTING` also builds a `ucode-san` binary with address, leak and undefined behaviour sanitisers, which is the binary to run the suite against after changing the value layer or the VM (part IV's chapter on the C API notes where those hooks are).

Debian

The source tree carries Debian packaging that produces four binary packages:

Package	Contains	Depends
`ucode`	`ucode`, `udbg`, `ucc`, `utpl`	`${shlibs}`, and `libucode` through the shared library link
`ucode-modules`	`${libdir}/ucode/*.so`	`libucode` at the same version; `Enhances: libucode`
`libucode`	`libucode.so.0`	`Recommends: ucode-modules`
`libucode-dev`	the headers under `/usr/include/ucode`	`libucode` at the same version, a libc dev package

The build dependencies are `debhelper-compat (= 13)`, `cmake`, `pkgconf`, `libjson-c-dev`, `libmd-dev` and `zlib1g-dev`, so a Debian build always has the `digest` and `zlib` modules; the OpenWrt-specific ones (`uci`, `ubus`, `uloop`, `rtnl`, `nl80211`) appear only if the additional libraries are installed, and `libnl-tiny` in particular is not in the Debian archive in the form the build expects.

```
$ dpkg -l 'ucode*' 'libucode*'
ii  libucode      ...  Tiny scripting and templating language (library)
ii  ucode         ...  Tiny scripting and templating language
ii  ucode-modules ...  Tiny scripting and templating language (modules)
$ ucode -e 'const fs = require("fs"); printf("%d modules\n",
length(fs.glob("/usr/lib/ucode/*.so")))'
9 modules
```

OpenWrt

On OpenWrt, ucode is preinstalled on modern releases — the base system needs it for the network and firewall scripts. The packaging in `openwrt/ucode/Makefile` splits it the same way: a `libucode` package depending on `libjson-c`, a `ucode` package depending on `libucode`, and one `ucode-mod-*` package per module, each with its own library dependency:

Package	Depends on
`libucode`	`libjson-c`
`ucode`	`libucode`
`ucode-mod-fs` , `-math` , `-resolv` , `-struct`	`ucode` alone
`ucode-mod-rtnl` , `-nl80211`	`ucode` , `libnl-tiny` , `libubox`
`ucode-mod-uloop`	`ucode` , `libubox`
`ucode-mod-ubus`	`ucode` , `libubus` , `libblobmsg-json`
`ucode-mod-uci`	`ucode` , `libuci`

The menu entry is `Languages → ucode` , and the modules are individually selectable in `make menuconfig` , which is the intended way to pay for only what a device uses. On a device image, the relevant check is not `which ucode` but whether the modules a script requires are present:

```
# ucode -e 'const fs = require("fs"); printf("%J\n", sort(fs.glob("/usr/lib/ucode/*.so")))'
[ "/usr/lib/ucode/fs.so", "/usr/lib/ucode/math.so", "/usr/lib/ucode/uci.so" ]
```

Cross-compiling

For an OpenWrt target the package builds inside the SDK or the source tree with the toolchain file that OpenWrt generates; the CMake probes then find the target's `libubus` , `libuci` and `libnl-tiny` in the staging directory rather than on the host, which is what turns the corresponding modules on. Outside OpenWrt, a CMake toolchain file works as it would for any project, with two cautions specific to ucode: the `rtnl` and `nl80211` modules are guarded on `LINUX` , so a non-Linux target silently loses them, and the Apple build path links modules with `-undefined dynamic_lookup` and installs to the Homebrew prefix when one is detected, which is rarely what a cross build wants — pass `-DCMAKE_INSTALL_PREFIX` explicitly.

A common embedded configuration is: no `COMPILE_SUPPORT` (so no `ucc` and no run-time compilation), no `ffi` , no `debug` , and the modules the device's scripts actually `require` — although the OpenWrt package instead ships the standard feature set, with all modules available as `ucode-mod-*` packages. With `BUILD_OPTIMIZE_SIZE` and `LINK_GC_SECTIONS` at their defaults, that configuration is a few hundred kilobytes of library plus the modules chosen.

Checking an installation

Four commands, in order, tell you whether the interpreter you just built is the one your scripts will see:

```
$ which ucode ucc utpl udbg
$ ucode -p '2 ** 8'
256
$ ucode -e 'printf("%J\n", REQUIRE_SEARCH_PATH)'
[ "/usr/lib/ucode/*.so", "/usr/share/ucode/*.uc", "./*.so", "./*.uc" ]
$ ucode -e 'const m = require("fs"); printf("%J\n", type(m))'
"object"
```

A failed `require` raises a `Runtime error` naming the module — `No module named 'fs' could be found` — and the message is the same whether the file is absent from the search path or was never built. The `cmake -B build` output, or `fs.glob("/usr/lib/ucode/*.so")` at run time, is what tells the two apart.

Summary

Need	How
Build it	`cmake -B build && cmake --build build -j4 && cmake --install build`
Hard dependency	json-c (`pkg-config`); C99+GNU, CMake ≥ 3.13
Optional libraries	zlib, libmd, libffi, libubox, libubus+blobmsg_json, libuci, libnl-tiny
Turn a module off	`-D<NAME>_SUPPORT=OFF` at configure time
Run from the tree	`build/ucode -L build script.uc`
Search path	compiled-in default, `-L dir` prepends, `REQUIRE_SEARCH_PATH` at run time
Tests	`-DUNIT_TESTING=ON` then `ctest --test-dir build` ; `ucode-san` under Clang
Debian packages	`ucode` , `ucode-modules` , `libucode` , `libucode-dev`
OpenWrt packages	`libucode` , `ucode` , one `ucode-mod-*` per module
Smallest useful build	no `debug` , no `ffi` , `COMPILE_SUPPORT=OFF` , only required modules

A first program

Getting started

This chapter gets a program on screen and off again. It is deliberately narrow: the interpreter's command line, the two output functions, how statements end, how a program reports its status, and the ways a program can be handed to the interpreter. Everything the language itself offers is a chapter away.

Printing

`print(...)` writes its arguments to standard output, one after another, with no separators and **no trailing newline** — a program that wants lines has to write them itself:

```
print("Hello, World!\n");
print("a"); print("b"); print("\n");
print("x", 1, "\n");
```

```
Hello, World!
ab
x1
```

Values are rendered by the rules of chapter 4 — an array or object in its JSON form, a function as its source text — with one exception worth knowing early: a `null` (or `undefined`) argument contributes nothing at all, it is not even written as `null`:

```
let missing = null;

print("before", missing, "after", "\n");
printf("printf %%s: [%s]\n", missing);
printf("printf %%J: [%J]\n", missing);
```

```
beforeafter
printf %s: [(null)]
printf %J: [null]
```

The same value through `printf` with `%s` becomes `(null)`, and through `%J` becomes `null`; only `print` and `printf`'s implicit rendering omit it.

`printf(fmt, ...)` is the C function, and like the C function it adds nothing of its own — the format string carries the newline. The `%J` conversion renders a ucode value in its JSON form and is the most useful single thing in the language for inspecting data:

```
printf("%s: %d\n", "count", 3);
printf("routes: %J\n", ["default", "lan", "wan"]);
printf("%J %J %J\n", { port: 67, on: true }, null, 1.5);
```

```
count: 3
routes: [ "default", "lan", "wan" ]
{ "port": 67, "on": true } null 1.5
```

`sprintf()` is the same function returning a string instead of writing it, and `warn(...)` writes to standard error without a newline, which makes it the place diagnostics go:

```
let line = sprintf("%J\n", { ok: 1 });

printf("formatted %d bytes\n", length(line));
warn("this goes to stderr\n");
```

```
formatted 12 bytes
```

The whole formatting family — widths, padding, the integer conversions, the `%.J` pretty-printing variant — is chapter 21.

Statements and blocks

A statement is terminated by a semicolon, and the semicolon is required between two statements:

```
let a = 1; let b = 2; printf("%d\n", a + b);
```

```
3
```

Writing the last statement of a block, or of the whole program, without its semicolon is allowed, which is why the single-statement examples in this book look unterminated:

```
let sum = 0;
for (let i = 1; i <= 4; i++) {
        sum += i
}
printf("%d\n", sum)
```

```
10
```

Blocks are written in braces, and a block is a scope of its own: `let`, `const` and function declarations inside it are not visible outside (chapter 5). There are no labels in ucode, so `outer: { ... }` is a syntax error, and a bare `{` at the start of a statement always opens a block — an object literal has to be made part of an expression instead, by wrapping it in parentheses, assigning it, or passing it to a function:

```
({ a: 1 });
let o = { b: 2 };
printf("%J %J\n", { c: 3 }, o)
```

```
{ "c": 3 } { "b": 2 }
```

Comments are `//` to the end of the line and `/* … */` across lines. Block comments do **not** nest, so commenting out a region that already contains a `/* … */` ends the comment at the first `*/` and leaves the rest of the region as code:

```
// a line comment
printf("after comments\n"); /* and a trailing
                               block comment */
```

```
after comments
```

There is no comment-out-a-region convention beyond deleting the text or wrapping it in `if (false)`.

Command line

The interpreter is one binary with several personalities:

```
ucode [options] [script.uc [args...]]
ucode -e "expression" [args...]
ucc [options] -o out.uc script.uc
utpl [options] template.uc
```

Run a script file, with the arguments after the script name reaching the program as `ARGV`. The script's own path is in `SCRIPT_NAME`:

```
printf("script=%J argv=%J\n", SCRIPT_NAME, ARGV);
```

Run an expression straight off the command line — the form this book uses for short examples:

```
$ ucode -e 'printf("%J\n", 2 ** 10)'
1024
```

`-p` is `-e` with the value of the expression printed afterwards, which makes it a calculator. It adds no newline either:

```
$ ucode -p '2 ** 10'
1024$
```

Read the program from standard input by naming the script `-`, which is what lets a pipeline carry a program and still pass arguments to it:

```
$ echo 'printf("%J\n", ARGV)' | ucode - one two
[ "one", "two" ]
```

A file is opened once, and only the **first** file name is taken as the program; every later argument is script data, not another source file to run, despite what `ucode -h` suggests. To run several files as one program, `require` them (chapter 17) or concatenate them.

Other options worth knowing at this stage:

Option	Effect
`-S`	strict mode: undeclared assignment and a few other laxities become errors (chapter 5)
`-D name=value`, `-D name`	define a global from the command line, `value` parsed as JSON
`-F name=path`	define a global from the contents of a JSON file
`-U name`	remove a global
`-l name=lib`, `-L dir`	preload a module; add a directory to the module search path (chapter 17)
`-t`	trace every opcode to standard error while running
`-c`	compile to bytecode instead of running (see "Precompiling" below)
`-T`, `-R`	treat the input as a template, or as plain code (chapter 16)
`-x`, `-X`	start inside the debugger, or enable it for later use (chapter 42)

The full list is in `ucode -h`, and chapter 20 covers the environment these options feed.

Exit status

A program ends by running out of statements, by `exit(code)`, or by `die(message)`:

```
printf("about to exit\n");
exit(3);
printf("never printed\n");
```

```
about to exit
```

The status is what the shell sees:

```
$ ucode -e 'exit(3)'; echo $?
3
$ ucode -e 'print("done\n")'; echo $?
done
0
```

An uncaught runtime error — a call to something that is not a function, an out-of-range array access, a failed `require` — ends the program with status **254** and a report on standard error, and a syntax error with status **255**. `die(message)` is the deliberate version of the same thing: the message on standard error without a newline, status 254:

```
$ ucode -e 'let o = null; o.field'; echo $?
Reference error: left-hand side expression is null
In [-e argument], line 1, byte 17:

 `let o = null; o.field`
  Near here ------^

254
```

Those error reports come with the offending source line and a caret; the line text is available even for `-e` arguments, because the source stays resident. The error types themselves are chapter 14, and `exit` / `die` / `assert` are in chapter 20.

Shebang scripts

A ucode file can be an executable the way a shell script is. The first line is a comment as far as ucode is concerned, so nothing special is needed to make it work:

```
$ cat > leases.uc
#!/usr/bin/env ucode
let fs = require("fs");
let path = ARGV[0] ?? "/tmp/dnsmasq.leases";

if (!fs.access(path))
        die(`no leases at ${path}\n`);

printf("%d leases\n", length(filter(split(fs.readfile(path), "\n"), (l) =>
l != "")));
$ chmod +x leases.uc
$ ./leases.uc /tmp/dnsmasq.leases
12 leases
```

`#!/usr/bin/env ucode` is the portable form; a hard-coded `#!/usr/bin/ucode` breaks on systems where the interpreter lives elsewhere, which on a build host it usually does. On a router it does not.

Precompiling

`ucode -c` (or the `ucc` name for the same binary) turns sources into a bytecode image. The output begins with an interpreter line, so the compiled file is executable in place:

```
$ ucc -o leases.bin leases.uc       # or: ucode -c -o leases.bin leases.uc
$ head -1 leases.bin
#!/usr/bin/env ucode
$ chmod +x leases.bin
$ ./leases.bin /tmp/dnsmasq.leases
12 leases
```

A compiled file is loaded by parsing the image rather than by lexing and compiling the source, which is the difference that matters on a device with a few hundred scripts on it: it saves the compile time and the text of the source. `loadfile(path)` reads either form and returns a function which, called, runs the program and returns whatever it returned:

```
let fs = require("fs");

fs.writefile("/tmp/uc3-prog.uc", "return { ok: true };");
let prog = loadfile("/tmp/uc3-prog.uc");
printf("loaded %s as %s, value %J\n", "file", type(prog), prog());
fs.unlink("/tmp/uc3-prog.uc");
```

```
loaded file as function, value { "ok": true }
```

Compile options go after `-c` as a comma-separated list: `-c,no-interp` omits the interpreter line, `-c,interp=/usr/bin/ucode` overrides it, `-c,module` produces a loadable module rather than a program, and `-s` leaves out the debug information that error reports and the debugger use — which is how the compiled form of a script gets small enough to be worth its flash space:

```
$ ucode -c,s -o /tmp/tiny.uc /tmp/prog.uc
```

Templates

The same binary also renders templates, and the `utpl` name enables that mode without an option — `utpl page.uc` is `ucode -T page.uc`. A template is a file of text with `{{ expression }}` to interpolate and `{% statement %}` to control it:

```
$ cat > iface.tpl
Interface {{ name }} is {{ up ? "up" : "down" }}.
{% for (let i = 0; i < 2; i++): %}address {{ i }}
{% endfor %}
$ ucode -T -D name=lan -D up=true iface.tpl
Interface lan is up.
address 0
address 1
```

The block tags use the alternative block syntax of chapter 7 — the opening tag ends in a colon and the closing tag is `{% endfor %}`, `{% endif %}` and so on — and a name with no value renders as the empty string. Data comes from `-D name=value` options or from a JSON file via `-F name=path`; the details, including the whitespace-control flags, are in chapter 16. This `{{ }}` markup is a different feature from the backtick template literals of the language itself (chapter 9).

Summary

Task	Form
Write to stdout	`print(...)` (no newline), `printf(fmt, ...)`
Inspect a value	`printf("%J\n", v)`, pretty with `printf("%.J\n", v)`
Write to stderr	`warn(...)`
Run a file	`ucode script.uc args...` — `ARGV` holds `args...`
Run a snippet	`ucode -e '...'`, or `-p` to print the result
Read a program from stdin	`ucode -`
Define a global	`-D name=json`, `-F name=path.json`
Succeed / fail	fall off the end (0), `exit(n)`, `die(msg)` (254), syntax error (255)
Make it executable	first line `#!/usr/bin/env ucode`, `chmod +x`
Compile it	`ucc -o out.uc in.uc`, `-c,module` for a module, `-s` to strip debug info
Render a template	`utpl file.tpl`, `ucode -T` (chapter 16)

Values and types

ucode has nine kinds of value. Four of them — `null`, booleans, numbers and strings — are scalars you meet in any language. Three more — arrays, objects and functions — are the structures you build things out of. A regular expression is a value of its own, and the interpreter carries a handful of internal *resources* (open files, sockets, ubus connections) that behave like objects but are not part of the language's own inventory.

`type()`

The builtin `type()` names the kind of a value:

```
let values = [null, true, 1, 1.5, "text", [1], { a: 1 }, type, regexp("a")];

for (let i = 0; i < length(values); i++)
    printf("[%s] ", type(values[i]));

print("\n");
```

```
[(null)] [bool] [int] [double] [string] [array] [object] [function] [regexp]
```

The first entry is printed as `(null)` because that is how `printf()` renders a missing string argument; `print()` renders it as nothing at all.

Note the last line's `null`: `type(null)` returns `null` rather than the string `"null"`, which is why it prints as nothing at all. `undefined` is not a separate value in ucode — an unset variable, a missing object key and an out-of-range array index all yield the same `null`:

```
print(undefined === null, " ", nosuchvariable === null, "\n");
```

```
true true
```

Reading a name that was never declared gives `null` in the default *sloppy* mode; under `-S` (strict mode) it raises a reference error instead. Writing to an undeclared name creates a global variable, in both modes unless `-S` says otherwise.

Truth

Every value is either true or false when the interpreter needs a decision. The falsy values are `null`, `false`, `0`, `-0` and the empty string:

```
let vals = [null, false, true, 0, 1, "", "x", [], [0], {}, json("{}")];

for (let i = 0; i < length(vals); i++)
    printf("%s ", vals[i] ? "true" : "false");

print("\n");
```

```
false false true false true false true true true true true
```

(There are ten values in the list; the last `true` is the object returned by `json("{}")`.)

The two results worth memorising are that `[]` and `{}` are **true**. In Lua an empty table is true, and in JavaScript it is as well, so ucode agrees with both here; and, unlike shell, the *existence* of a container says nothing on its own — a list with no elements still branches into the true arm.

Numbers

Numbers come in two flavours, and the difference only shows up in edge cases.

Integers are 64-bit. They are written in decimal, or in hexadecimal, octal or binary:

```
print(0xff, " ", 017, " ", 0b1011, " ", 08, "\n");
```

```
255 15 11 8
```

A leading `0` means octal, as in C, so `017` is fifteen; a digit sequence that cannot be octal is read as decimal, which is why `08` is eight. `0o17` is not a number at all — it is a syntax error — and neither are digit separators: `1_000_000` does not parse.

Doubles are IEEE 754 64-bit floats, recognised by a decimal point, an exponent, or by the operation that produced them.

Division is the operator that behaves least like JavaScript. When both operands are integers, `/` performs *integer* division, truncating towards zero:

```
print(7 / 2, " ", -7 / 2, " ", 1 / 3, " ", 7.0 / 2, "\n");
```

```
3 -3 0 3.5
```

`%` follows the sign of its left operand, and works on doubles as well:

```
print(7 % 2, " ", -7 % 2, " ", 7.5 % 2, "\n");
```

```
1 -1 1.5
```

`**` exponentiation yields an integer when it can and a double when it must:

```
print(2 ** 10, " ", type(2 ** 10), " ", 2 ** -1, " ", type(2 ** -1), "\n");
```

```
1024 int 0.5 double
```

Doubles print without trailing zeros, and switch to exponential notation for large and small magnitudes. The conversion is round-trip friendly but not exhaustive, which is why `0.1 + 0.2` looks better in ucode than in a browser console:

```
print(3.0, " ", 0.1 + 0.2, " ", 1e20, " ", 1e-9, " ", 1e300 * 1e300, "\n");
```

```
3 0.3 1e+20 1e-09 Infinity
```

Signed and unsigned integers

Here is the part with no equivalent in JavaScript, where all numbers are doubles.

An integer value in ucode is either *signed* or *unsigned*, and the flavour is part of the value. When the operands of an arithmetic operation are all positive, the calculation is done with unsigned operands and the result is inferred to be unsigned too, which lets a computation hold magnitudes larger than `2**63 - 1`:

```
let big = 2 ** 63;

print(big, " ", type(big), " ", big + 1, "\n");
```

```
9223372036854775808 int 9223372036854775809
```

`9223372036854775808` does not fit in a signed 64-bit integer, and ucode does not fall back to a double for it — it stays an integer, rendered in full unsigned magnitude. Bring a negative operand into the same expression and the arithmetic turns signed:

```
print(0 - 2 ** 63, " ", sprintf("%d", 2 ** 63), "\n");
```

```
-9223372036854775808 -9223372036854775808
```

The same bits, two readings: `sprintf()` with `%d` converts to a signed integer, while the natural rendering of an unsigned value prints its full magnitude. Bitwise operators produce unsigned results, which is how you meet this behaviour first:

```
print(~6, " ", ~6 == -7, "\n");
```

```
18446744073709551609 false
```

`~6` is the two's complement of seven, `0xfffffffffffffff9`. Read as a signed value that is `-7`; ucode reads it as an unsigned value and prints `18446744073709551609`, and the equality with `-7` is false because the two values are compared as the numbers they denote, not as the bit patterns that happen to implement them. Likewise:

```
print(2 ** 63 < 0, " ", 2 ** 63 == -9223372036854775808, "\n");
```

```
false false
```

Overflow wraps silently. There is no exception and no automatic widening to a double:

```
print(2 ** 64, " ", (2 ** 63) * 2, "\n");
```

```
0 0
```

Three rules keep you out of trouble. If you mix a negative operand in, you get signed arithmetic. If you print with `%d`, you get the signed reading. And if a computation can reach `2**63`, check the sign of the operands on both sides of every comparison, because `x == -1` is false for the unsigned value whose bits are all ones.

Converting to a number

`int()` converts a value to an integer, and unlike the operators it is strict about the text it accepts:

```
print(int("42"), " ", int("42abc"), " ", int("010"), " ", int("0x1f"), " ",
int("zz"), "\n");
```

```
42 42 10 0 NaN
```

The conversion is decimal only: a leading `0` does not request octal, a `0x` prefix is not understood (yielding `0`), trailing garbage is silently dropped, and text with no leading digits at all produces `NaN`. To parse a hexadecimal numeral, including an optional `0x` prefix, as a number, use `hex()`; `hexdec()` does something different — it decodes a hex-encoded digit sequence into a binary string.

Arithmetic on strings coerces as JavaScript does — `+` concatenates if either side is a string, the other operators convert:

```
print("3" + 4, " ", "3" * "4", " ", true + 1, " ", null + 1, " ", [] + 1,
"\n");
```

```
34 12 2 1 NaN
```

An array or object has no numeric value, so arithmetic on one yields `NaN` rather than raising. Comparisons are strict about type and never coerce; `==` compares values of different types as unequal, and `1 < 2 < 3` is `true` only because `1 < 2` yields the boolean `true`, which then compares as `1`.

Strings are byte strings

A string is a sequence of bytes with a length, not a sequence of characters. ucode does not interpret the bytes as UTF-8 and does not care whether they decode: `substr()` cuts at byte offsets and can split a multi-byte character in half. Treat string indices and lengths as byte counts and everything works as documented.

The practical consequence is that `"héllo"` has a length of six, not five, and that sorting, matching and slicing behave exactly the same on non-ASCII text as on ASCII — bytewise.

Notes

- `type()` is the only type inspection the language offers; there is no `typeof` operator and no `instanceof`.

- Handles to external things — the file returned by `fs.open()`, a socket, a ubus connection — are *resources*, and `type()` names them as such:

```
import * as fs from "fs";

let f = fs.open("/dev/null", "r");

print(type(f), " ", type(fs), " ", type(fs.open), "\n");
```

```
resource object function
```

A resource is not one of the language's own types, but it carries a prototype whose methods (`read()`, `close()`, and so on) are reached with ordinary field access.

- Small integers do not allocate: they are encoded directly in the value representation (see *Inside the interpreter*), so loops over integers are cheap.
- `NaN` is not equal to anything, including itself; test with `x != x`.

Names, scope and bindings

Identifiers

An identifier is made of ASCII letters, digits and the underscore, and may not start with a digit. There is no `$`, and non-ASCII characters are not identifier characters — a name is a run of bytes, and the lexer refuses anything outside its alphabet:

```
let _private = 1, max_lease_count = 2, ifname2 = 3;

printf("%d %d %d\n", _private, max_lease_count, ifname2);
```

```
1 2 3
```

The following words are reserved and cannot be used as names:

```
break   case    catch   const   continue  default  delete  else
elif    endfor  endif   endwhile  endfunction  export  false  for
function  if    import  in      let       null     return  switch
this    true    try     while
```

Case matters: `if_`, `IF`, `If` and `ifname` are all ordinary names.

```
let If = 1, IF_ = 2, ifname = 3;

printf("%d %d %s\n", If, IF_, ifname);
```

```
1 2 3
```

Bindings

A binding is introduced by `let`, `const` or `function`. `let` may be left without a value, in which case the name holds `null`; several names may be declared in one statement:

```
let a;
let b = 2, c = "three";

printf("%J %d %s\n", a, b, c);
```

```
null 2 three
```

`const` requires an initializer, and rebinding the name is refused — by the compiler, so the program does not run at all:

```
const limit = 1500;

printf("%d\n", limit);
```

```
1500
```

```
const limit = 1500;
limit = 9000;
```

A constant protects the binding, not the value behind it. An object or array held in a `const` stays as mutable as any other:

```
const ports = [80, 443];

push(ports, 8080);
ports[0] = 8000;
printf("%J\n", ports);
```

```
[ 8000, 443, 8080 ]
```

A `function` declaration binds the name to the function; that binding is writable and may be re-declared, which is what makes monkey-patching possible. `function name;` — a forward declaration — is the opposite case: the binding becomes a constant that only a definition may fill. Chapter 8 covers both.

Declaring a name a second time in the same scope is accepted, and the later declaration wins. This holds for `let` and `const` alike, and for a `let` / `const` over a function declaration (strict mode rejects it):

```
let mode = "fast";
let mode = "safe";

printf("%s\n", mode);
```

```
safe
```

The exception is a name that was forward-declared with `function`: after `function foo;`, a `let foo`, a `const foo`, a second `function foo;` or a second definition of `foo` are all syntax errors (chapter 8).

`delete` removes object properties, not bindings — naming a variable is a syntax error:

```
let temp = { a: 1 };

printf("%J %J\n", delete temp.a, temp);
```

```
true { }
```

Scope

Bindings belong to the innermost enclosing block — the braces of a function body, a conditional, a loop, or a bare block — and are invisible outside it. An inner block may shadow an outer name; the outer binding is untouched and remains visible to code outside the shadow:

```
let scope_name = "outer";

function show() {
        let scope_name = "inner";

        return scope_name;
}

{
        let scope_name = "block";
        printf("%s\n", scope_name);
}

printf("%s %s %s\n", scope_name, show(), scope_name);
```

```
block
outer inner outer
```

Function parameters are bindings of the function's own scope, and the loop variables of a `for` header belong to the loop — two loops in one function can reuse the same name, and neither leaks out:

```
function twice(list) {
        let out = [];

        for (let i = 0; i < length(list); i++) {
                push(out, list[i]);
        }

        for (let i = length(list) - 1; i >= 0; i--) {
                push(out, list[i]);
        }

        return out;
}

printf("%J %J\n", twice([1, 2]), (function () { try { return i; } catch (e)
{ return null; } })());
```

```
[ 1, 2, 2, 1 ] null
```

A function declared inside a block or inside another function is scoped to it (chapter 8), which means the same name can serve as a private helper at several levels:

```
function outer() {
        function helper() { return "outer helper"; }

        function inner() {
                function helper() { return "inner helper"; }

                return helper();
        }

        return [helper(), inner()];
}

printf("%J\n", outer());
```

```
[ "outer helper", "inner helper" ]
```

Closures and upvalues

A function refers to the bindings of the scopes around it, not to copies of their values at the moment the function was created. Reads and writes are shared with everyone else who sees that binding, and they stay shared for as long as the function lives — which is why the same name can be read through a closure after the code that created it finished:

```
let counter = 0;

function bump() {
        counter = counter + 1;

        return counter;
}

bump();
bump();
printf("%d\n", counter);
```

```
2
```

Holding the state inside a function, rather than at the top level, is the standard way to make a private variable:

```
function make_gate(name) {
        let passes = 0;

        return {
                through: function () { passes++; return [name, passes]; },
                count: () => passes
        };
}

let gate = make_gate("wan");

gate.through();
gate.through();
printf("%J %J\n", gate.through(), gate.count());
```

```
[ "wan", 3 ] 3
```

The one subtlety is the `for` header variable: it is a single binding for the whole loop, so closures created in the body share it and see the value the loop finished with. A binding declared inside the body is created fresh by each iteration:

```
let from_header = [], from_body = [];

for (let i = 0; i < 3; i++) {
        let snapshot = i;

        push(from_header, () => i);
        push(from_body, () => snapshot);
}

printf("%J %J\n", map(from_header, (f) => f()), map(from_body, (f) => f()));
```

```
[ 3, 3, 3 ] [ 0, 1, 2 ]
```

Before the declaration

One difference from JavaScript: ucode has no hoisting. A binding is not known before its declaration statement, so what a read of the name before that resolves to depends on strict mode. Outside strict mode the read falls through to the global object, finds nothing there, and returns `null`, as if the name were undeclared:

```
printf("%J ", value);

let value = 42;

printf("%J\n", value);
```

```
null 42
```

In strict mode (`-S`), the same read is a `Reference error` naming the variable, and the program stops:

```
$ ucode -S -e 'printf("%.J\n", x)'
Reference error: access to undeclared variable x
In [-e argument], line 1, byte 17:

 `printf("%.J\n", x)`
  Near here ------^
```

The same applies to reading a name inside its own declaration — the difference is that the compiler catches this at compile time and rejects the program, in either mode, so no `try` can catch it. A name may not be referenced inside its own initializer, not even from a function that would run later:

```
let node = {
        toString: () => node
};

print(node);
```

The message is `Can't access lexical declaration 'node' before initialization`. Declaring the name first and assigning afterwards moves the reference out of the initializer (chapter 8 uses the same technique to let a published object call back into itself).

The global object

Names that no lexical binding provides are looked up in the **global object**, reachable under its own name `global`. It holds the builtin functions and constants, and it is where a name that has never been declared ends up living. A lookup that finds nothing answers `null`, so reading an undeclared name is a perfectly ordinary `null`, not an error — the error comes when the result is used as something it is not:

```
printf("%J\n", definitely_not_declared);
printf("%J\n", (function () { try { return definitely_not_declared.field; }
catch (e) { return "error: " + e; } })());
```

```
null
"error: left-hand side expression is null"
```

A caught exception carries the message on its own; the `Reference error:` prefix seen above is part of how an *uncaught* error is reported, not part of the message (chapter 14).

Assigning to an undeclared name creates a key on the global object, from inside a function as much as at the top level (strict mode refuses both the read and the write):

```
function remember() {
        remembered = "value";
}

remember();
printf("%J %J\n", remembered, exists(global, "remembered"));
printf("%J\n", delete global.remembered);
printf("%J\n", exists(global, "remembered"));
```

```
"value" true
true
false
```

The reverse does not hold: a top-level `let`, `const` or `function` declaration is a binding of the script's outermost lexical scope, **not** a key of the global object. Two scripts therefore cannot step on each other's top-level names, and `keys(global)` shows only what the interpreter installed plus what was assigned without declaring:

```
let top_level = 1;
implicit_global = 2;

printf("%J %J\n", exists(global, "top_level"), exists(global,
"implicit_global"));
```

```
false true
```

`global` itself is an ordinary object with the usual key operations available, and it refers to itself under `global.global`. Besides the functions of chapter 19 and following, it carries a few names of interest to the embedding and module machinery — `ARGV`, `REQUIRE_SEARCH_PATH`, `modules`, `NaN` and `Infinity`:

```
printf("%J %J %J\n", global.global == global, type(global.print),
length(keys(global)) > 50);
printf("%J %J\n", type(ARGV), type(REQUIRE_SEARCH_PATH));
```

```
true "function" true
"array" "array"
```

To ask whether a name exists rather than read it, use `exists(global, name)`. Note the difference to the `in` operator, which also consults prototype chains that `exists()` ignores, and to `rawget()`, which follows the prototype chain but skips `__get__` metamethods (chapter 12):

```
let base = { shared: 1 };
let derived = proto({}, base);

printf("%J %J %J\n", exists(derived, "shared"), "shared" in derived,
rawget(derived, "shared"));
```

```
false true 1
```

Code that runs through `call()` gets a global environment of its own, and a module or a template has its own scope as well; in all three cases the surrounding globals are inherited unless explicitly replaced. Chapters 8, 16 and 17 deal with each.

Strict mode

The rules above are the permissive ones. Two switches turn them off: the `-S` option, which compiles every source of a run strictly, and a `"use strict";` pragma, which strictens the function it appears in. Strictness is lexical — a nested function inherits it — and since the top level of a file is itself a function body, a pragma in the first line strictens the whole file.

Strict mode changes four things, and nothing else:

Permissive	Strict
reading an undeclared name answers `null`	`Reference error: access to undeclared variable X`
assigning to an undeclared name creates a global	the same reference error
`++` / `--` on an undeclared name creates a global	the same reference error
re-declaring a name in the same scope	`Syntax error: Variable 'x' redeclared`

```
"use strict";

let counter = 0;
counter++;

printf("%J %J\n", counter, (function () { try { return counter2; } catch (e)
{ return "error: " + e; } })());
```

```
1 "error: access to undeclared variable counter2"
```

The pragma is recognised only as the first statement of a body. Anywhere else — after another statement, or as the first thing inside a plain block — it is an ordinary string expression with no further meaning:

```
let prepared = 1;
"use strict";

printf("%J\n", still_permissive);
```

```
null
```

The reference errors are ordinary exceptions, so a `try` block sees them like any other (chapter 14), including from inside a callback:

```
"use strict";

function lookup(name) {
        try {
                return config_for[name];
        } catch (e) {
                return "default";
        }
}

printf("%s\n", lookup("lan"));
```

```
default
```

Compiling at run time takes the same switch as a member of the parse configuration object accepted by `loadstring()` and `loadfile()` (chapter 20), and the C API's script configuration carries the equivalent `strict_declarations` field (chapter 41 onward). In a template, the pragma goes in the first `{% ... %}` block that opens the file.

What strict mode does *not* touch is everything else in this chapter: a constant's value stays mutable, `delete` still cannot remove a binding, a binding is still visible from the top of its block before its declaration runs, and `global` remains a rebindable name.

Operators

ucode's operators are largely the ones you expect from ECMAScript, with a handful of differences that matter. Arithmetic on integers stays in the integer domain, the power operator associates to the left, division by zero is a value rather than an error, and bitwise operators hand back unsigned values. This chapter is the inventory, together with the precedence table the parser actually uses.

Arithmetic

```
print(7 / 2, " ", -7 / 2, " ", 1 / 3, " ", 7.5 / 2, "\n");
print(7 % 2, " ", -7 % 2, " ", 7.5 % 2, "\n");
```

```
3 -3 0 3.75
1 -1 1.5
```

When both operands of `/` are integers the result is an integer, truncated towards zero — `1 / 3` is `0`, not `0.333`. When one operand of a binary operation is a double, the operation is carried out in the double domain — but only from that operation on: the operands of an earlier operation keep the domain they arrived in, so `4 / 3 * 1.0` is `1.0` (the division yields the integer `1`, which the multiplication then converts), not `1.3333`. `%` takes the sign of its left operand and works on doubles too, since it is implemented with `fmod()`.

Dividing by zero does not raise, and follows IEEE-754: the infinity carries the sign the division rules give it, and zero over zero is `NaN`:

```
print(5 / 0, " ", -5.0 / 0, " ", 0.0 / 0, " ", 5 % 0, " ", NaN != NaN, "\n");
```

```
Infinity -Infinity NaN NaN true
```

The double path (`vm.c:1947-1949`) is the hardware division itself, so `-5.0 / 0` is negative infinity and `0.0 / 0` is `NaN`. The integer path (`vm.c:2042-2048`) has no hardware answer to lean on and reproduces the same result itself: signed infinity with the sign of the numerator, `NaN` for `0 / 0`. The last two columns show where this leaves you: `5 % 0` goes through `fmod()` and really is `NaN`, and `NaN != NaN` is true only because it is computed by a path that handles `NaN` properly. Overflow, likewise, wraps silently; there is no automatic promotion to double:

```
print(2 ** 64, " ", (2 ** 63) * 2, " ", (0 - 2 ** 63) / -1, "\n");
```

```
0 0 9223372036854775808
```

The third result deserves a moment. `INT64_MIN / -1` would trap with `SIGFPE` in C, so the interpreter special-cases it and returns `2**63` as an *unsigned* integer — the value does not fit in the signed range, so it does not claim to. Note how the signed `INT64_MIN` had to be manufactured with `0 - 2 ** 63`; the expression `2 ** 63 / -1` is something else again, because the left operand there is the *unsigned* two to the sixty-third, which saturates to `INT64_MAX` when the mixed-signedness division needs a signed operand:

```
print(2 ** 63 / -1, " ", (0 - 2 ** 63) / -1, "\n");
```

```
-9223372036854775807 9223372036854775808
```

`**` is right-associative in JavaScript. In ucode it is a plain left-associative binary operator, and unary minus binds tighter than it does:

```
print(2 ** 3 ** 2, " ", -2 ** 2, " ", 2 ** -1, "\n");
```

```
64 4 0.5
```

`2 ** 3 ** 2` is `(2 ** 3) ** 2`, and `-2 ** 2` is `(-2) ** 2`, which is why it comes out positive. JavaScript rejects the second form outright; ucode accepts it and means something slightly different from what a JS reader would guess. A negative exponent produces a double, since the integer result would be zero.

Bitwise

`&`, `|`, `^`, `<<`, `>>` and `~` work on the 64-bit integer representations:

```
print(5 << 2, " ", 5 >> 1, " ", 5 & 3, " ", 5 | 3, " ", 5 ^ 3, " ", ~5,
"\n");
```

```
20 2 1 7 6 18446744073709551610
```

The last column is the signed/unsigned behaviour of values and types at work: `~5` is all the bits of five inverted, which is `-6` read as signed and `18446744073709551610` read as unsigned, and bitwise negation yields the unsigned flavour. There is no unsigned right shift `>>>`, and no `<<<`.

Because the operands are integers, applying a bitwise operator to a double converts it by truncation.

Equality and comparison

`==` performs coercion between numbers and strings, `===` does not:

```
let a = [1, 2];

print("1" == 1, " ", "1" === 1, " ", "3" < 4, "\n");
print(a == [1, 2], " ", a === a, " ", { a: 1 } == { a: 1 }, "\n");
```

```
true false true
false true false
```

Composite values are compared by identity, never by structure: two arrays holding the same elements are unequal, and the only way to compare contents is to write the loop yourself (or compare the `%J` renderings of both, which is cheaper to type but fragile).

Relational operators coerce a string operand to a number when compared against a number, as the `"3" < 4` result shows. Comparison chaining is *not* the mathematical relation — it is two separate comparisons, exactly as in JavaScript:

```
print(1 < 2 < 3, " ", 3 > 2 > 1, "\n");
```

```
true false
```

`1 < 2` yields `true`, which counts as `1` in `1 < 3`, so the first expression is true. The second computes `3 > 2` → `true` → `1`, and `1 > 1` is false. Write `x > low && x < high`.

Logical and nullish operators

`||`, `&&` and `??` return one of their operands rather than a boolean, and all three short-circuit:

```
printf("[%s] [%s] [%s] [%s]\n", 0 || 1, true && 7, null ?? "default", "" ??
"default");
```

```
[1] [7] [default] []
```

`||` and `&&` test for truthiness, so `0 || 1` is `1`; `??` tests only for `null`, so an empty string or a zero passes through untouched. That difference is the main reason to prefer `??` when reading values that may legitimately be zero.

Optional chaining reaches through values that may be missing:

```
let o = { a: { b: 2 } };
let p = null;

printf("[%s] [%s] [%s]\n", o.a.b, o.x?.y, p?.a?.b);
```

```
[2] [(null)] [(null)]
```

(The `(null)` renderings are what `printf()` makes of a missing string argument; with `print()` they would print as nothing.)

Reading a key an object does not have already gives `null`, so `?.` matters only where a further dereference would fail — `o.x.y` raises a reference error because it dereferences `null`, while `o.x?.y` stops and yields `null`.

Assignment

Plain `=` assigns, and the compound forms cover every arithmetic, bitwise and logical operator:

```
let n = 5;

n += 3;
n **= 2;
n -= 8;
n /= 4;

printf("%d ", n);

let z = null;
z ??= 9;

let s = "";
s ||= "set";

printf("[%s] [%s]\n", z, s);
```

```
14 [9] [set]
```

Assignment is an expression, yielding the assigned value, which is why `print(r = f())` works. The logical assignment operators are short-circuiting: `x ||= y` evaluates `y` only when `x` is falsy, so `x ||= expensive()` can be used as a lazy default.

Increment and increment-by exist in both prefix and postfix position:

```
let i = 0;

print(i++, " ", i, " ", ++i, " ", -i, "\n");
```

```
0 1 2 -2
```

`delete` removes a key from an object and reports whether anything went away. It applies to object keys only; deleting an array index raises an error, because an array has no removable gaps, and a named key of an array can be handled only by a metamethod the array's prototype carries (chapter 12):

```
let o = { a: 1, b: 2 };

print("a" in o, " ", delete o.a, " ", "a" in o, "\n");
```

```
true true false
```

`in` is true when the key exists on the object or anywhere along its prototype chain. The comma operator evaluates both sides and yields the right one; it is rarely worth the confusion.

Precedence

The table below is the order coded in the parser (`include/ucode/internal/compiler.h`), loosest first. Each row binds less tightly than the rows below it:

Precedence	Operators
comma	`,`
assignment	`=` `+=` `-=` `*=` `/=` `%=` `<<=` `>>=` `&=` `^=` `|=` `||=` `&&=` `**=` `??=`
conditional	`?:`
logical or	`||` `??`
logical and	`&&`
bitwise or	`|`
bitwise xor	`^`
bitwise and	`&`
equality	`==` `!=` `===` `!==`
comparison	`<` `<=` `>` `>=` `in`
shift	`<<` `>>`
additive	`+` `-`
multiplicative	`*` `/` `%`
exponentiation	`**`
unary	`!` `~` `+` `-` `++x` `--x`
postfix increment	`x++` `x--`
call and member	`.` `[` `(`
primary	`(…)`

Two consequences are easy to trip over. Bitwise operators bind *less* tightly than comparison, so `flags & FLAG == 0` parses as `flags & (FLAG == 0)` and does not test what you meant — parenthesise. And `**` sits below unary, which is why `-2 ** 2` is `4`:

```
print(4 & 1 == 0, " ", (4 & 1) == 0, "\n");
```

```
0 true
```

```
print(1 + 2 * 3, " ", (1 + 2) * 3, " ", 2 * 3 ** 2, " ", 1 + 2 < 4 && true,
"\n");
```

```
7 9 18 true
```

The conditional operator is right-associative, so a chain of them reads like nested `if` s:

```
print(0 ? 1 : 2 ? 2 : 3, "\n");
```

```
2
```

Absent operators

Knowing what is *not* there saves time:

- No `typeof` and no `instanceof` . Use the `type()` builtin.
- No `>>>` or `<<<` .
- No `throw` ; raise an exception with the `die()` builtin and catch it with `try { … } catch (e) { … }` .
- No `new` , no `class` , no `super` , no `this` in a class sense. Callable objects are built with a `__call__` metamethod.
- No operator overloading of any kind, and no metamethod for arithmetic or comparison.
- No `await` , no generators, no `yield` , no arrow-function `=>` currying beyond the single-arrow form.

Notes

- `-0.0` is falsy and `NaN != NaN` . The globals `NaN` and `Infinity` exist (`vm.c:150-151`), alongside `global` , `modules` and `REQUIRE_SEARCH_PATH` .
- Because `==` coerces numbers and strings but composite values compare by identity, the safest habit is to write `===` everywhere and let the type errors tell you where a conversion was actually wanted.
- `sprintf("%J", v)` renders infinity as `1e309` and `NaN` as the string `"NaN"` , so JSON output never contains the non-standard `Infinity` token (see types.c:1968-1973).

Control structures

ucode's control structures are the C family ones — `if`, `while`, `for`, `switch` — with a couple of differences worth knowing up front: `for ... in` iterates containers, `switch` compares strictly and falls through like C's, and every block-taking statement has a second spelling with a colon and a terminator keyword, designed for templates:

```
let ports = [22, 80, 443];

for (let p in ports) {
        if (p < 1024) {
                printf("%d is privileged\n", p);
        }
}

if (length(ports) == 0):
        print("nothing to do\n");
endif
```

```
22 is privileged
80 is privileged
443 is privileged
```

Truth and control flow

Every control structure tests its condition the same way, and the values that test false are `null`, `false`, the number `0` (`0.0` included) and the empty string `""`. Everything else — including empty containers, the string `"0"`, and objects — tests true:

```
for (let v in [0, 0.0, "", null, false, [], {}, "0", 1]) {
        printf("%J -> %s\n", v, v ? "true" : "false");
}
```

```
0 -> false
0.0 -> false
"" -> false
null -> false
false -> false
[ ] -> true
{ } -> true
"0" -> true
1 -> true
```

Because a condition is an ordinary expression, an assignment can stand in it. ucode has no "assignment inside a condition" warning, and no declarations are allowed there — `if (let x = 5)` is a syntax error:

```
let line = "hello";

if (line = "yes") {
        printf("assigned, and truish: %s\n", line);
}
```

```
assigned, and truish: yes
```

Conditional execution

The `if` statement takes a condition in parentheses, a block, and optional `else if` and `else` branches. When a branch is a single statement the braces may be left out — but then only that one statement belongs to the branch:

```
function grade(n) {
        if (n >= 90)
                return "A";
        else if (n >= 80)
                return "B";
        else
                return "C";
}

printf("%s %s %s\n", grade(95), grade(85), grade(10));
```

```
A B C
```

A conditional expression is available as `?:`, and unlike `if` it accepts expressions only; it is covered in chapter 6.

Loops

`while` tests before each iteration; `for` has the C form, with an initializer, a condition and a post statement:

```
let i = 0;

while (i < 3) {
        printf("%d ", i);
        i++;
}

for (let j = 0, n = 3; j < n; j += 1) {
        printf("[%d]", j);
}

printf("\n");
```

```
0 1 2 [0][1][2]
```

The initializer of a `for` loop declares its variables in the loop's own scope, so `j` above is not visible after the loop, and the same name can be reused by a second loop. Any of the three `for` clauses may be empty; `for (;;)` loops until something `break`s out of it:

```
let n = 0;

for (;;) {
        n++;

        if (n > 2) {
                break;
        }
}

printf("n=%d\n", n);
```

```
n=3
```

There is no `do ... while` — a loop whose body must run at least once is written as `while (true) { ... }` with a `break`, or as a `for` loop.

Iterating containers

`for ... in` walks a container. Its meaning depends on the kind of container, and on how many loop variables are declared:

Written	Iterates over
`for (let v in array)`	the values, in index order
`for (let i, v in array)`	the index in `i`, the value in `v`
`for (let k in object)`	the key names, in insertion order
`for (let k, v in object)`	the key in `k`, the value in `v`
`for (let v in string)`	nothing at all
`for (let v in null)`	nothing at all

```
for (let v in [10, 20]) {
        printf("%J ", v);
}

for (let i, v in [10, 20]) {
        printf("%d:%J ", i, v);
}

for (let k, v in { lan: "eth0", wan: "eth1" }) {
        printf("%s=%J ", k, v);
}

printf("\n");
```

```
10 20 0:10 1:20 lan="eth0" wan="eth1"
```

Strings are not iterable; index over them, or `split()` them first:

```
let s = "abc";

for (let i = 0; i < length(s); i++) {
        printf("%s ", substr(s, i, 1));
}

printf("\n");

for (let part in split("a:b:c", ":")) {
        printf("%s ", part);
}

printf("\n");
```

```
a b c
a b c
```

Walking an array keeps up with modifications made during the walk, because iteration works by index against the live array. Adding elements while iterating therefore extends the walk, and removing the element just visited makes the next iteration skip one:

```
let a = [1, 2, 3];
let seen = [];

for (let v in a) {
        push(seen, v);

        if (length(a) < 6) {
                push(a, v * 10);
        }
}

printf("seen=%J final=%J\n", seen, a);
```

```
seen=[ 1, 2, 3, 10, 20, 30 ] final=[ 1, 2, 3, 10, 20, 30 ]
```

Object walks take a snapshot of the key set as they go: a key deleted during the walk is simply not visited later, and adding keys during a walk is not something to rely on. When a loop both searches and edits, collect first and apply afterwards:

```
let m = { a: 1, b: 2 };

for (let k in m) {
        printf("visiting %s\n", k);

        if (k == "a") {
                delete m.b;
        }
}

printf("%J\n", m);
```

```
visiting a
{ "a": 1 }
```

`break` leaves the innermost loop, `continue` starts its next iteration. Both are checked statically: `break` must sit lexically inside a loop or a `switch`, and `continue` inside a loop, so neither can be used to escape from a callback — a `return` from the callback, or a flag variable, is the way to stop a walk that a function performs for you:

```
let found = null;

for (let v in [1, 2, 3, 4]) {
        if (v == 2) {
                found = v;
                break;
        }
}

printf("found=%J\n", found);
```

```
found=2
```

Loops have no labels, so leaving two loops at once needs a helper function (whose `return` exits all of them) or a condition on the outer loop:

```
function first_pair(list, test) {
        for (let a in list) {
                for (let b in list) {
                        if (test(a, b)) {
                                return [a, b];
                        }
                }
        }

        return null;
}

printf("%J %J\n", first_pair([1, 2, 3], (a, b) => a * b > 4), first_pair([1,
2], (a, b) => false));
```

```
[ 2, 3 ] null
```

Switch

`switch` compares the subject against each `case` label with **strict** equality — the `===` comparison of chapter 6 — so a string label never matches a number subject. Execution enters at the matching label and runs on through the following labels until a `break`, which is C's fallthrough:

```
function describe(v) {
        let out = [];

        switch (v) {
                case 1:
                        push(out, "one");
                case 2:
                        push(out, "two");
                        break;
                case "3":
                        push(out, "three");
                        break;
                default:
                        push(out, "other");
        }

        return out;
}

printf("%J %J %J %J\n", describe(1), describe(2), describe("3"),
describe(3));
```

```
[ "one", "two" ] [ "two" ] [ "three" ] [ "other" ]
```

`default` may appear anywhere among the labels and is itself subject to fallthrough, and several `case` labels may share one body by writing them consecutively:

```
function classify(v) {
        switch (v) {
                case 2: case 3: case 5: case 7:
                        return "small prime";
                default:
                        return "not a small prime";
        }
}

printf("%s %s\n", classify(5), classify(8));
```

```
small prime not a small prime
```

Inside a loop, `break` in a `switch` leaves the `switch` only, while `continue` skips to the next loop iteration — the loop is not affected by the `switch` ending:

```
let log = [];

for (let v in [1, 2, 3, 4]) {
        switch (v) {
                case 2:
                        continue;
                case 3:
                        break;
        }

        push(log, v);
}

printf("%J\n", log);
```

```
[ 1, 3, 4 ]
```

The alternative block syntax

Every block-taking statement except `switch` has an alternative spelling: a colon where the opening brace would be, and a terminator keyword where the closing brace would be. The terminators are `endif`, `endwhile`, `endfor` and `endfunction`, and `if` chains gain a single-word `elif`:

```
let x = 5;

if (x == 0):
        print("zero\n");
elif (x == 5):
        print("five\n");
else
        print("another value\n");
endif

let i = 0;

while (i < 2):
        printf("%d ", i);
        i++;
endwhile

for (let v in ["a", "b"]):
        printf("%s ", v);
endfor

function double(v):
        return v * 2;
endfunction

printf("\n%d\n", double(21));
```

```
five
0 1 a b
42
```

The statements inside such a block are ordinary statements and need their semicolons; it is only the braces that the colon and terminator replace. Note that `else` takes no colon, that `switch` has no alternative form, and that `elif` belongs to this syntax and cannot be used where braces are used — `if (c) { … } elif (c2) { … }` is a syntax error, and the braced chain is spelled `else if`.

The reason for the second syntax is templating. A template file interleaves markup with ucode delimited by `{% … %}`, in which braces already carry meaning, so the block statements are written with colons and terminators instead:

```
{% if (length(ifaces) > 0): %}
iface count: {{ length(ifaces) }}
{% else %}
no interfaces
{% endif %}
```

The template renderer itself is chapter 16; the block syntax is ordinary ucode and can be used in plain scripts, as above — most code uses braces, and mixing the two styles within one statement is not possible.

Functions

Functions are ordinary values in ucode. They can be stored in variables and containers, passed to and returned from other functions, and created at run time. There is a single callable kind — there are no classes, no constructors and no bound method objects — and three ways to write one: a declaration, a function expression and an arrow function:

```
function add(a, b) { return a + b; }

let sub = function (a, b) { return a - b; };
let mul = (a, b) => a * b;

printf("%J %J %J\n", add(1, 2), sub(5, 3), mul(3, 4));
printf("%s %s\n", type(add), type(mul));
```

```
3 2 12
function function
```

A function value knows nothing about its own source: printing one yields a placeholder built from its name, and functions carry no properties — an attempt to store one is a type error:

```
function named() { return 1; }

printf("%s\n", sprintf("%s", named));
printf("%J\n", (function () { try { named.tag = 1; return named.tag; } catch
(e) { return "error: " + e; } })());
```

```
function named() { ... }
"error: attempt to set property on closure value"
```

Two function values are equal only if they are the same function; two separately written functions with identical bodies are not:

```
function f() { return 1; }

printf("%J %J\n", f == f, f == function () { return 1; });
```

```
true false
```

Since functions are not containers, `length()` and `keys()` have nothing to say about them and return `null`. An anonymous function can be called immediately after writing it, which is the usual way to give a block of code its own scope:

```
let secret = (function () {
        let hidden = 42;

        return function () { return hidden; };
})();

printf("%J\n", secret());
```

```
42
```

Arguments

A function receives the arguments it declared; missing ones are `null` and extra ones are discarded. There is no `arguments` object — an undefined name simply reads as `null`:

```
function two(a, b) { return [a, b]; }

printf("%J %J\n", two(1), two(1, 2, 3));
printf("%J\n", arguments);
```

```
[ 1, null ] [ 1, 2 ]
null
```

There are no default parameter values; a parameter with a default is given one in the body, where `??` distinguishes a missing argument from a `0` or `""` one:

```
function repeat(s, n) {
        n = n ?? 2;

        return join("", map([0, 1], (i) => i < n ? s : ""));
}

printf("%s %s\n", repeat("ab"), repeat("ab", 1));
```

```
abab ab
```

A final rest parameter collects whatever is left, and spread syntax passes a container's elements as separate arguments. A rest parameter has to be the last one:

```
function collect(first, ...rest) { return [first, rest]; }

printf("%J\n", collect(1, 2, 3));
printf("%J\n", collect(...[9, 8, 7]));
```

```
[ 1, [ 2, 3 ] ]
[ 9, [ 8, 7 ] ]
```

Spread works on any iterable value and raises on anything else:

```
printf("%J\n", (function () { try { let f = (a) => a; return f(...5); }
catch (e) { return "error: " + e; } })());
```

```
"error: (5) is not iterable"
```

Declarations are not hoisted

A function is not visible before its declaration. The name exists from the declaration statement onwards, in the enclosing block's scope, and calling it earlier is a type error:

```
printf("%J\n", (function () {
        try {
                return later();
        }
        catch (e) {
                return "error: " + e;
        }
})());

function later() { return "defined"; }
```

```
"error: left-hand side is not a function"
```

Functions declared inside a block are local to it, and a function declared inside a function is local to that function:

```
function outer() {
        function inner() { return "inner"; }

        return inner();
}

printf("%J %J\n", outer(), (function () { try { return inner(); } catch (e)
{ return "error: " + e; } })());
```

```
"inner" "error: left-hand side is not a function"
```

A name that a later declaration will fill in can be announced ahead of time with a **forward declaration** — `function` followed by a name and a semicolon. It behaves much like `let name;`, with the constraint that the binding is then a constant: only a function definition may give it a value.

```
function is_even;
function is_odd;

function is_even(n) {
        if (n == 0) {
                return true;
        }

        return is_odd(n - 1);
}

function is_odd(n) {
        if (n == 0) {
                return false;
        }

        return is_even(n - 1);
}

printf("%J %J %J\n", is_even(0), is_even(10), is_odd(11));
```

```
true true true
```

The constraints exist so that a forward-declared name cannot be quietly re-bound: after a forward declaration, assigning to it, incrementing it, declaring it again with `let` or `const`, declaring it forward twice, defining it twice,

or forward-declaring a name that is already defined are all syntax errors. A function declared the ordinary way has a writable binding and may be redefined:

```
function redef() { return 1; }

redef = function () { return 2; };
printf("%J\n", redef());
```

```
2
```

Until a forward declaration is filled, its name reads as `null`, so calling it fails the same way calling any non-function does — with a type error, not with a message about the missing definition.

Closures

A function keeps the scope it was written in alive, and reads and writes of the enclosing variables are shared with whoever else sees that scope:

```
function counter() {
        let n = 0;

        return function () { return ++n; };
}

let c = counter();

printf("%d %d %d\n", c(), c(), c());
```

```
1 2 3
```

One detail distinguishes ucode from JavaScript: a loop variable declared in a `for` header is a **single** binding, not a fresh one per iteration, so closures made inside a loop all see the value the loop ended with. Declaring a variable inside the loop body gives each iteration its own:

```
let shared = [], per_iteration = [];

for (let i = 0; i < 3; i++) {
        let copy = i;

        push(shared, () => i);
        push(per_iteration, () => copy);
}

printf("%J %J\n", map(shared, (f) => f()), map(per_iteration, (f) => f()));
```

```
[ 3, 3, 3 ] [ 0, 1, 2 ]
```

A lexical binding cannot be named inside its own initializer, not even from a closure that would only run after initialization — the compiler rejects such a program, so this is a compile-time error, not something a `try` can catch:

```
let self_ref = {
        get: () => self_ref
};

print(self_ref.get());
```

The reported message is `Can't access lexical declaration 'self_ref' before initialization`. The workaround is to declare the name first and assign afterwards, which puts the definition outside the initializer:

```
let self_ref = null;

self_ref = {
        get: () => self_ref
};

printf("%J\n", self_ref.get() == self_ref);
```

```
true
```

Methods and `this`

A function stored in an object and called with dot syntax is a method, and inside it `this` refers to the receiving object. In a plain function call — and at the top level of a script — `this` is `null`. A method that is pulled out of its object loses its receiver and fails when it uses `this`:

```
let host = { name: "router", get() { return this.name; } };
let detached = host.get;

printf("%s\n", host.get());
printf("%J\n", (function () { try { return detached(); } catch (e) { return
"error: " + e; } })());
printf("%J\n", this);
```

```
router
"error: left-hand side expression is null"
null
```

Arrow functions have no `this` of their own; they use the one from the scope they were written in. That makes them the right choice for callbacks created inside a method, and the wrong choice for the method itself:

```
let host = { name: "router", names: ["eth0", "eth1"] };

host.describe = function () {
        let me = this;

        return map(this.names, function (n) { return me.name + "." + n; });
};

printf("%J\n", host.describe());
```

```
[ "router.eth0", "router.eth1" ]
```

`map()` calls the callback without a receiver, so an inner `function` would see `this` as `null`; capturing `this` in a variable, or writing the callback as an arrow function, are both standard.

Calling a function value directly

`call()` invokes a function value with an explicit `this`, an explicit global scope, and the arguments to pass on. Both optional parameters come before the arguments, so a plain invocation with arguments needs the two `null` placeholders:

```
printf("%J\n", call(function (a, b, c) { return [a, b, c]; }, null, null, 1,
2, 3));
printf("%J\n", call(function () { return this.x; }, { x: 7 }));
```

```
[ 1, 2, 3 ]
7
```

The third parameter gives the function a different global environment. A plain object is added on top of the surrounding scope — the function sees the usual globals, with the scope object's own keys shadowing them — which is a convenient way to inject a value:

```
global.greeting = "hello";

printf("%J %J\n", call(function () { return greeting; }), call(function () {
return greeting; }, null, { greeting: "hi" }));
```

```
"hello" "hi"
```

To get an environment that is truly closed, give the scope object an explicit prototype with `proto()` (chapter 12) — an explicit prototype replaces the implicit inheritance from the current scope, so names that are not in the object read as `null`, builtins included. A scope that is neither an object nor `null` makes `call()` return `null` without invoking anything:

```
global.greeting = "hello";

printf("%J %J\n", call(function () { return greeting; }, null, proto({},
{})), call(function () { return printf; }, null, proto({}, {})));
printf("%J %J\n", call(function () { return x; }, null, proto({ x: 1 },
{})), call(function () { return 1; }, null, 5));
```

```
null null
1 null
```

Recursion

A function can call itself by name, whether it was declared, written as a named expression, or forward declared. The name of a function expression is visible only inside that function:

```
let fact = function self(n) { return n <= 1 ? 1 : n * self(n - 1); };

printf("%J %J\n", fact(5), (function self2(n) { return n <= 1 ? 1 : n *
self2(n - 1); })(5));
printf("%J\n", (function () { try { return self(5); } catch (e) { return
"error: " + e; } })());
```

```
120 120
"error: left-hand side is not a function"
```

Whether recursion is bounded at all depends on the shape of the recursive expression:

```
function down(n) { return n <= 0 ? 0 : down(n - 1); }
function count(n, acc) { return n <= 0 ? acc : count(n - 1, acc + 1); }

printf("%J %J\n", down(3000000), count(1000000, 0));
```

```
0 1000000
```

Both of these return the result of the recursive call unchanged, which makes the call a **tail call**, and the virtual machine performs a tail call by reusing the current stack frame instead of pushing a new one. Such functions recurse as deeply as their logic requires and use no extra stack; the frames of completed calls are gone, so nothing accumulates.

A call whose result is used by an enclosing expression is not a tail call, and those frames do accumulate, up to a fixed limit of 1000 nested calls:

```
function sum(n) { return n <= 0 ? 0 : n + sum(n - 1); }

try {
    printf("%J\n", sum(10000));
} catch (e) {
    printf("%s: %s\n", e.type, e.message);
}
```

```
Runtime error: Too much recursion
```

The limit is reported as an ordinary `Runtime error`, so a recursive descent over untrusted depth can be run under `try`, and the usual remedy is to move the accumulated work into the arguments so that the call returns directly:

```
function sum(n, acc) { return n <= 0 ? acc : sum(n - 1, acc + n); }

printf("%J\n", sum(10000, 0));
```

```
50005000
```

Two consequences are worth keeping in mind. A tail-recursive function with no reachable base case runs indefinitely rather than failing, since nothing grows. And a tail call is recognised only where the call is the whole returned value: `return f(x);` and `return cond ? f(x) : g(x);` are tail positions, while `return f(x) + 1;`, `return [f(x)];`, and a call made for its side effect before the `return` are not.

Functions as data

Because functions are values, the container toolkit of chapter 22 accepts them, and small functions assemble into larger ones:

```
let ops = {
        double: (n) => n * 2,
        inc: (n) => n + 1
};

let pipeline = [ops.double, ops.inc, ops.double];
let n = 3;

for (let f in pipeline) {
        n = f(n);
}

printf("%d\n", n);
```

```
14
```

Used as an object key, a function is converted to its string form, like any other non-string key:

```
let f = (x) => x;
let m = {};

m[f] = "value";
printf("%J\n", keys(m));
```

```
[ "(x) => { ... }" ]
```

Strings

Literal strings

A string literal is written between double or single quotes; the two forms are equivalent and both allow the other quote unescaped inside. A literal may not span a raw newline — write `\n` instead:

```
let a = "hello", b = 'it\'s', c = "she said \"hi\"";

printf("%J %J %J\n", a, b, c);
```

```
"hello" "it's" "she said \"hi\""
```

The escapes are the familiar C ones:

```
printf("%J %J %J\n", "tab:\tx", "newline:\nx", "backslash: \\");
```

```
"tab:\tx" "newline:\nx" "backslash: \\"
```

`\xHH` represents one byte by two hexadecimal digits, `\NNN` represents one byte by up to three octal digits, and `\uHHHH` takes a Unicode code point and encodes it as UTF-8:

```
printf("%J %J %J %J\n", "\x41", "\101", "\u0041", "\u00e9");
```

```
"A" "A" "A" "é"
```

Any other character after a backslash is taken literally, so `\d` is `d`. To write a literal backslash — a Windows-style path, a regex fragment — double it. There is no raw-string form; a regular expression is usually written as a `regexp()` argument (chapter 13), where the pattern is still a string and still needs doubled backslashes.

A string is a byte sequence and may contain zero bytes; length counts bytes:

```
printf("%J %J\n", length("a\0b"), length("\u00e9"));
```

```
3 2
```

Template literals

A string written between backticks is a *template literal*. It may span raw newlines, it understands the same escapes as a quoted literal, and `${ … }` splices the value of an expression into it:

```
let username = "jow";

printf("%s\n", `hello ${"world"}`);
printf("%s\n", `2 + 2 = ${2 + 2}`);
printf("%s\n", `user: ${username ? `user ${username}` : "anonymous"}`);
printf("%s\n", `first
second`);
```

```
hello world
2 + 2 = 4
user: user jow
first
second
```

Interpolation is an expression, not a statement: `${}`, `${ if (x) {} }` and `${ let y = 1 }` are syntax errors, while conditionals, function calls, property access and further template literals are fine — the braces open a fresh lexical context, so a placeholder can contain quotes and backticks of its own, escaped as `\`` and `\${` when they should be literal text:

```
let o = { a: 1 }, list = [1, 2];

printf("%s %s %s\n", `${o}`, `${list}`, `${"}"}`);
printf("%s\n", `cost: \${1}, escaped backtick: \` and interpolated ${o.a + 1}
`);
```

```
{ "a": 1 } [ 1, 2 ] }
cost: ${1}, escaped backtick: ` and interpolated 2
```

A template literal compiles to a sequence of `+` concatenations, so the interpolated values follow the coercion rules of the addition operator (chapter 6): strings pass through, objects and arrays render as JSON, `null` becomes `null`, and a `regexp` or `resource` yields `""`.

Adjacent literals are never joined by juxtaposition, in either syntax — `"a" "b"` is a syntax error and so is `` `a` `b` `` (with its own message, "Adjacent template literals are not implicitly concatenated"). Write `+`, or put both parts in one literal.

Template literals are a feature of the language and are unrelated to the template *files* processed by `uhttpd`, `utpl` and `ucode -T`, which use `{{ … }}` and `{% … %}` markup instead (chapter 16). Inside such a file, backtick literals can still be used within `{{ }}` expressions.

Byte strings

Strings hold bytes, not characters. `length()` counts bytes, `substr()` and `index()` count bytes, and `ord()` and `chr()` deal with single byte values:

```
printf("%J %J %J\n", length("äb"), ord("äb", 0), substr("äb", 2));
printf("%J %J %J\n", ord("A"), ord("A", 0), chr(65, 66));
```

```
3 195 "b"
65 65 "AB"
```

The `ä` takes two bytes, so `substr("äb", 1, 1)` hands back a lone continuation byte instead of a character — nothing complains, but the result is not valid UTF-8 on its own. Case mapping works on ASCII letters only and passes everything else through unchanged:

```
printf("%J %J\n", uc("ätherNet"), lc("ÄThERnet"));
```

```
"äTHERNET" "Äthernet"
```

`uchr()` is the counterpart of `ord()` at the code-point level: it takes numbers and produces UTF-8. Values outside `0` .. `0x10FFFF` become the replacement character:

```
printf("%J %J %J\n", uchr(0xe9), uchr(0x41, 0x2d, 0x1f600),
length(uchr(0xe9)));
```

```
"é" "A-😀" 2
```

Values are not objects

A string is a value, not an object: it has no properties and no methods, and it cannot be indexed with brackets. Every string operation is a function of the language or of a module:

```
printf("%J\n", (function () { try { return "abc".upper(); } catch (e) {
return "error: " + e; } })());
printf("%J\n", (function () { try { return "abc"[1]; } catch (e) { return
"error: " + e; } })());
printf("%J %J\n", substr("abc", 1, 1), index("abc", "b"));
```

```
"error: left-hand side expression is not an array or object"
"error: left-hand side expression is not an array or object"
"b" 1
```

Where a language with string indexing would write `s[i]`, ucode asks for a slice or for the byte value — `substr(s, i, 1)` is the piece at position `i` and `ord(s, i)` is its numeric value, and both accept a negative position counted from the end:

```
printf("%J %J %J\n", substr("abc", 1, 1), ord("abc", 1), substr("abc", -1));
```

```
"b" 98 "c"
```

For the same reason, strings are not containers: `map()`, `filter()`, `sort()`, `keys()` and `values()` answer `null` on them, a `for` loop over a string produces nothing, and neither `in` nor `exists()` holds:

```
printf("%J %J %J %J\n", map("abc", (c) => c), keys("abc"), "a" in "abc",
exists("abc", "a"));
printf("%J\n", (function () { let out = []; for (let c in "abc") push(out,
c); return out; })());
```

```
null null false false
[ ]
```

To iterate the characters or bytes of a string, take a slice per position:

```
let s = "abc", chars = [], bytes = [];

for (let i = 0; i < length(s); i++) {
        push(chars, substr(s, i, 1));
        push(bytes, ord(s, i));
}

printf("%J %J\n", chars, bytes);
```

```
[ "a", "b", "c" ] [ 97, 98, 99 ]
```

Concatenation and conversion

`+` concatenates whenever either operand is a string; no operand is ever converted to a number by `+`. The other arithmetic operators go the other way and convert strings numerically, yielding `NaN` for a string that does not start with a number:

```
printf("%J %J %J\n", "a" + "b", "count: " + 5, "abc" + 0);
printf("%J %J %J %J\n", "5" - 2, "5" * 2, "5" / 2, "5" % 2);
```

```
"ab" "count: 5" "abc0"
3 10 2 1
```

Non-string values become strings through concatenation, through `sprintf()` (chapter 21) or through the `%J` conversion, which renders a value the way the language itself prints it:

```
printf("%s | %s | %s\n", null, [1, 2], { a: 1 });
printf("%J | %J | %J\n", null, [1, 2], { a: 1 });
```

```
(null) | [ 1, 2 ] | { "a": 1 }
null | [ 1, 2 ] | { "a": 1 }
```

Note the difference for `null`: `%s` on a null pointer prints `(null)`, while `%J` prints `null`. `%J` is what to use for building or inspecting JSON-ish text (chapter 15).

Comparison

Strings compare byte by byte, which orders the uppercase letters before the lowercase ones and makes any prefix sort before the longer string. Equality is by content — two separately built strings with the same bytes are equal — and no string equals `null`:

```
printf("%J %J %J\n", "a" < "b", "Z" < "a", "10" < "9");
printf("%J %J %J\n", "ab" == "a" + "b", "" == null, "abc" < "abcd");
```

```
true true true
true false true
```

Case-insensitive comparison is done by mapping both sides first:

```
let a = "WLAN0", b = "wlan0";

printf("%J %J\n", a == b, lc(a) == lc(b));
```

```
false true
```

Substrings and searching

`substr(str, off[, len])` extracts a piece. A negative `off` counts from the end, an omitted `len` runs to the end, a negative `len` drops that many bytes from the end, and an `off` beyond the string yields the empty string:

```
printf("%J %J %J %J\n", substr("hello", 1), substr("hello", 1, 3),
substr("hello", -2), substr("hello", -4, -1));
printf("%J %J\n", substr("hello", 99), substr("hello", 0, 99));
```

```
"ello" "ell" "lo" "ell"
"" "hello"
```

`index()` and `rindex()` return the byte offset of the first and last occurrence, or `-1`. Both take an optional offset argument that limits the search (chapter 21):

```
printf("%J %J %J\n", index("hello world", "o"), rindex("hello world", "o"),
index("hello world", "z"));
printf("%J %J\n", index("hello world", "o", 5), rindex("hello world", "o",
5));
```

```
4 7 -1
7 4
```

`split()` cuts a string into an array, on a literal string or on a regular expression, and an optional limit caps the number of pieces, so the remainder lands in the last element whole:

```
printf("%J %J\n", split("a,b,,c", ","), split(",a,", ","));
printf("%J %J\n", split("a:1:b:2", ":") , split("a:1:b:2", ":", 2));
printf("%J\n", split("a1b22c", regexp("[0-9]+")));
```

```
[ "a", "b", "", "c" ] [ "", "a", "" ]
[ "a", "1", "b", "2" ] [ "a", "1:b:2" ]
[ "a", "b", "c" ]
```

Matching and replacing

`match(subject, re)` returns the match and its capture groups as an array, or `null`. The pattern argument has to be a regular expression value — a plain string is matched literally, and metacharacters in it are then part of the pattern:

```
printf("%J\n", match("10.0.0.1/24", regexp("^([0-9.]+)/([0-9]+)$")));
printf("%J %J\n", match("iface:   lan", regexp(":[[:space:]]*([[:alnum:]]
+)")), match("abc", regexp("q")));
printf("%J %J\n", match("iface: lan", /l.n/), match("iface: lan", "l.n"));
```

```
[ "10.0.0.1/24", "10.0.0.1", "24" ]
[ ":   lan", "lan" ] null
[ "lan" ] null
```

The last line shows that a pattern has to be a regular expression — written as `regexp(...)` or as a `/.../` literal (chapter 13). A plain string is not compiled on the way in and never matches; the answer is `null`, which is indistinguishable from a regular expression that found nothing. `replace()` and `split()` are the two functions that also accept a plain string as their pattern.

A global regular expression makes `match()` return one array per match instead:

```
printf("%J\n", match("a1 b22 c333", regexp("[0-9]+", "g")));
```

```
[ [ "1" ], [ "22" ], [ "333" ] ]
```

`replace(subject, pattern, replacement[, limit])` takes either a plain string or a regular expression, and the two behave differently: a string replaces **every** occurrence, a regular expression replaces the **first** match unless it carries the `g` flag. `limit` caps the number of replacements either way. With a regular expression, `$1` up to `$9` in the replacement refer to its capture groups; `$0` is not special. A function may be given as the replacement, and it is called once per match:

```
printf("%J %J\n", replace("a-b-c", "-", "+"), replace("a-b-c", "-", "+", 1));
printf("%J %J\n", replace("a1 b22", regexp("[0-9]+"), "N"), replace("a1
b22", regexp("[0-9]+", "g"), "N"));
printf("%J\n", replace("2024-01-02", /([0-9]+)-([0-9]+)-([0-9]+)/,
"$3/$2/$1"));
printf("%J\n", replace("a1 b22 c3", /([0-9]+)/g, (m, n) => "<" + n + ">"));
printf("%J\n", replace("a1 b22 c3", /[0-9]+/g, "N", 2));
```

```
"a+b+c" "a+b-c"
"aN b22" "aN bN"
"02/01/2024"
"a<1> b<22> c<3>"
"aN bN c3"
```

A subject that is not a string is converted first, and a subject or pattern without a match comes back unchanged:

```
printf("%J %J\n", replace(1212, "1", "x"), replace("abc", regexp("q"), "X"));
```

```
"x2x2" "abc"
```

`wildcard(subject, pattern[, ignorecase])` matches shell-style globs — `*`, `?` and `[...]` — and is the convenient test when a full regular expression would be overkill:

```
printf("%J %J %J %J\n", wildcard("eth0.100", "eth*"), wildcard("eth0",
"eth?"), wildcard("wlan0", "WLAN?"), wildcard("wlan0", "WLAN?", true));
```

```
true true false true
```

Trimming

`trim()`, `ltrim()` and `rtrim()` strip whitespace from both ends, the left end and the right end respectively. A second argument gives a set of characters to strip instead of whitespace — a set, not a substring:

```
printf("%J %J %J\n", trim("  x  "), ltrim("  x  "), rtrim("  x  "));
printf("%J %J\n", trim("--x--", "-"), trim("abccba", "abc"));
```

```
"x" "x  " " "  x"
"x" ""
```

Bytes and hexadecimal

`chr()` turns numbers into bytes (values above 255 are truncated to 255, negative ones become zero bytes) and `ord()` reads a byte back:

```
printf("%J %J\n", chr(65, 66, 67), ord("abc", 1));
printf("%J %J\n", ord(chr(256)), ord(chr(-1)));
```

```
"ABC" 98
255 0
```

Three functions deal with hexadecimal, and they do different things. `hex()` converts a string of hex digits to a number — it is the counterpart of `int(x, 16)`, not a way to render a number as hex. `hexenc()` encodes a byte string as hex digits and `hexdec()` decodes them back, tolerating (and by default skipping) spaces and newlines:

```
printf("%J %J %J\n", hex("ff"), hex("0xff"), hex("zz"));
printf("%J %J\n", hexenc("Hi"), hexdec("4869"));
printf("%J\n", hexdec("48 69\n"));
```

```
255 255 "NaN"
"4869" "Hi"
"Hi"
```

An optional `0x` prefix is accepted by `hex()`, and a value that is not a number is reported as `NaN` rather than as an error.

To render a number as hex, use `sprintf()` with `%x` or `%X` (chapter 21):

```
printf("%J %J %J\n", sprintf("%x", 255), sprintf("%08X", 255), int("ff",
16));
```

```
"ff" "000000FF" 255
```

One caution about `int()`: it parses in base 10 unless told otherwise, and it stops at the first character that is not a digit in that base, so a `0x` prefix does not switch it to hexadecimal:

```
printf("%J %J %J\n", int("0x10"), int("0x10", 16), int("12abc"));
```

```
0 16 12
```

Strings in practice

Reading a `CIDR` address with one regular expression and no indexing:

```
let m = match("192.168.1.10/24", regexp("^([0-9.]+)/([0-9]+)$"));

if (m) {
        printf("addr=%s prefix=%s net=%s\n", m[1], m[2],
                join(".", slice(split(m[1], "."), 0, 3)) + ".0");
}
```

```
addr=192.168.1.10 prefix=24 net=192.168.1.0
```

Turning the output of a command into structured lines:

```
let text = "lan  up  10.0.0.1\nwan  down  0.0.0.0\n";

let rows = filter(
        map(split(trim(text), "\n"),
                (line) => filter(split(line, regexp("[[:space:]]+")), (f)
=> f != "")),
        (row) => length(row) > 0);

printf("%J\n", rows);
```

```
[ [ "lan", "up", "10.0.0.1" ], [ "wan", "down", "0.0.0.0" ] ]
```

To quote a value for a shell command, wrap it in single quotes, with each embedded single quote turned into an escaped one:

```
function shell_quote(s) {
        return "'" + replace(s, "'", "'\\''") + "'";
}

printf("%s\n", shell_quote("it's here"));
```

```
'it'\''s here'
```

Base 64

`b64enc()` and `b64dec()` convert between byte strings and their base64 representation. Both work on bytes, so binary data and multi-byte text round-trip unchanged:

```
printf("%J %J\n", b64enc("Hi"), b64dec("SGk="));
printf("%J %J\n", b64enc(chr(0, 255, 128)), hexenc(b64dec("AP+A")));
printf("%J\n", b64enc("héllo"));
```

```
"SGk=" "Hi"
"AP+A" "00ff80"
"aMOpbGxv"
```

The decoder ignores whitespace anywhere in its input and answers `null` for any character outside the base64 alphabet — a corrupt value and a missing pad are reported the same way, without a message. Unlike `hexdec()`, `b64dec()` takes no argument naming extra characters to skip; only whitespace is tolerated:

```
printf("%J %J %J\n", b64dec("SG k="), b64dec("S G k ="), b64dec("SGk="));
printf("%J %J\n", b64dec("SGk"), b64dec("!!!"));
```

```
"Hi" "Hi" "Hi"
null null
```

The input must otherwise be well formed: it has to be a whole number of four-character quanta once whitespace is removed, the padding has to be canonical (the unused bits of the last quantum must be zero, which is why `SG==` is refused while `YQ==` decodes), and nothing but whitespace may follow the pad:

```
printf("%J %J\n", b64dec("SGk="), b64dec("YQ=="));
printf("%J %J %J\n", b64dec("SGk"), b64dec("SGk=SGk="), b64dec("SG=="));
```

```
"Hi" "a"
null null null
```

Reference: string functions

Function	Result
`length(s)`	byte length; `null` for numbers
`substr(s, off[, len])`	substring, negative `off` / `len` count from the end; `substr(s, i, 1)` is `s[i]` elsewhere
`index(s, needle[, off])`	first byte offset, `-1` if absent
`rindex(s, needle[, off])`	last byte offset at or before `off`, `-1` if absent
`split(s, sep[, limit])`	array of pieces; `sep` may be a `regexp`
`join(sep, arr)`	string of `arr` elements separated by `sep`
`replace(s, pat, repl[, limit])`	string pattern: all occurrences; regexp: first match unless it has `g`; `limit` caps either; `$1` .. `$9` or a function as `repl`
`match(s, re)`	match plus groups, `null` if none; array of arrays with the `g` flag; a plain string pattern never matches
`wildcard(s, pat[, icase])`	`true` / `false` glob match
`trim(s[, set])` / `ltrim` / `rtrim`	strip whitespace, or the given character set
`uc(s)` / `lc(s)`	ASCII case mapping
`chr(n...)` / `ord(s[, i])`	bytes from numbers; byte at position `i` — with `substr(s, i, 1)`, the stand-in for indexing
`b64enc(s)` / `b64dec(b)`	base64 of a byte string, back from base64 (whitespace ignored, `null` if undecodable)
`uchr(cp...)`	UTF-8 from code points
`hex(s)`	number from a hex string, `NaN` if not hex
`hexenc(s)` / `hexdec(h[, skip])`	byte string to hex, hex string to bytes
`sprintf(fmt, ...)` / `printf` / `print`	formatting and output (chapter 21)
`` `text ${expr} text` ``	template literal: concatenation with interpolation, raw newlines allowed

None of these modify the subject: every one returns a new string, since a string value never changes.

Arrays

An array in ucode is a dense, ordered list of values with an integer index and a length. It is not an object with numeric keys; it does not inherit from anything, and it has no methods — the operations on arrays are the builtin functions `push()`, `slice()`, `sort()` and their relatives, which take the array as their first argument.

Literals and indexing

```
let a = [1, "two", null, [3]];

printf("[%s] [%s] [%s]\n", a[0], a[2], a[3][0]);
```

```
[1] [(null)] [3]
```

Indexing is by integer. Reading outside the populated range yields `null` rather than raising, which keeps defensive code short:

```
let a = [1, 2];

printf("[%s] [%s]\n", a[5], a[1]);
```

```
[(null)] [2]
```

Negative indices count from the end. This has no equivalent in JavaScript, where `a[-1]` would set a property named `"-1"`; in ucode it is a real index:

```
let a = [1, 2, 3];

print(a[-1], " ", a[-2], " ", a[-4], "\n");
```

```
3 2
```

A negative index too far off the front is simply out of range and reads `null`. The same holds for assignment, so `a[-1] = 9` overwrites the last element instead of lengthening the array.

Growth and holes

Assigning past the end grows the array, filling the skipped positions with `null`:

```
let a = [1];

a[4] = 2;

print(a, " ", length(a), "\n");
```

```
[ 1, null, null, null, 2 ] 5
```

There is no distinction between a hole and an explicit `null` — the storage holds a `null` either way, and `length()` counts it. There is no `in`-style "does this index exist" question for arrays: `1 in a` is false even for a populated index, because `in` is an object operator.

Named properties do not stick to arrays. The assignment is not an error, but nothing is stored:

```
let a = [1];

a.name = "list";

printf("[%s]\n", a.name);
```

```
[(null)]
```

What you can do with an array is give it a *prototype*, which is how standard-module resources and user-defined array-ish types get their behaviour (Prototypes and metamethods).

`length()` is a function

```
print(length("héllo"), " ", length({ a: 1, b: 2 }), " ", length([1, null]),
" ", length(42), "\n");
```

```
6 2 2
```

`length()` works on strings (bytes), objects (own keys) and arrays (elements), and returns `null` for anything else — including `null`. There is no `.length` property and no `#` operator.

The array builtins

All of these take the array first. The differences from ECMAScript's `Array.prototype` methods are worth memorising, because the names are the same but the details are not:

Function	Result
`push(arr, v…)`	appends; returns the **last value pushed**
`unshift(arr, v…)`	prepends; returns the **last value added**
`pop(arr)` / `shift(arr)`	remove and return the last / first element, `null` if empty
`splice(arr, off, len, …v…)`	removes `len` elements at `off`, inserts the rest; **returns the modified input array**
`slice(arr, [off], [end])`	returns a **new** array of the range; empty if `off > end`
`sort(arr, [fn])`	sorts **in place**, returns the same array
`reverse(arr)`	returns a **new** reversed array
`uniq(arr)`	returns a **new** array of unique values
`map(arr, fn)`	new array of `fn(value, index)`
`filter(arr, fn)`	new array of elements where `fn(value, index)` is truthy
`join(sep, arr)`	concatenates; **separator first**
`min(v…)` / `max(v…)`	variadic over values, not over an array

Three of them differ from their JavaScript equivalents. `splice()` does not return the removed elements — it returns the array it modified:

```
let a = [1, 2, 3];

print(splice(a, 1, 1), " ", a, "\n");
```

```
[ 1, 3 ] [ 1, 3 ]
```

`join()` takes the separator before the array:

```
print(join(", ", ["a", "b", "c"]), "\n");
```

```
a, b, c
```

`min()` and `max()` compare their arguments, so to find the extremes of an array you spread it:

```
let vals = [3, 1, 2];

print(min(...vals), " ", max(...vals), "\n");
```

```
1 3
```

Ordering

`sort()` without a comparator uses ucode's value ordering: numbers numerically, strings bytewise, and mixed types by type. It is *not* the "convert everything to a string" rule that makes JavaScript's default sort put `100` before `9`:

```
print(sort([10, 9, 1]), " ", sort(["10", "100", "9"]), "\n");
```

```
[ 1, 9, 10 ] [ "10", "100", "9" ]
```

The strings sort bytewise, which happens to give the same answer JavaScript's default would. A comparator is called with two values and must return a negative, zero or positive number:

```
let users = [{ n: 30 }, { n: 10 }, { n: 20 }];

print(sort(users, (a, b) => a.n - b.n), "\n");
```

```
[ { "n": 10 }, { "n": 20 }, { "n": 30 } ]
```

`sort()` also accepts an **object**, in which case it sorts the object's keys in place:

```
let o = { b: 1, a: 2 };

sort(o);

print(keys(o), "\n");
```

```
[ "a", "b" ]
```

That is the idiom for getting ordered output out of a table: collect into an object, `sort()` it, then iterate `keys()`.

`uniq()` compares values with ucode's notion of equality, which is type-sensitive, so the number `1` and the string `"1"` both survive:

```
print(uniq([1, 1, "1", null, null]), "\n");
```

```
[ 1, "1", null ]
```

Aliasing and copying

Arrays are values held by reference. Assignment copies the reference, and a nested array in a copy is the *same* nested array:

```
let a = [[1], 2];
let b = a;

b[1] = 9;

print(a, " ", slice(a)[0] === a[0], "\n");
```

```
[ [ 1 ], 9 ] true
```

`slice(a)` gives a shallow copy: a new outer array sharing the inner ones. To copy properly, copy each level, or round-trip through JSON if the data is plain and you do not mind the cost:

```
let a = [[1], 2];
let b = json(sprintf("%J", a));

b[0][0] = 99;

print(a, " ", b, "\n");
```

```
[ [ 1 ], 2 ] [ [ 99 ], 2 ]
```

The `%J` / `json()` round trip is the standard deep-copy trick in ucode scripts. It loses functions, and it turns the number `1` and the string `"1"` into distinguishable things again only because JSON keeps type tags; anything exotic (a resource, a regexp) will not survive.

Iterating

Three ways, in order of how often you want them:

```
let a = ["x", "y", "z"];

for (let i = 0; i < length(a); i++)
    print(i, ":", a[i], " ");

print("\n");

for (let v in a)
    print(v, " ");

print("\n");

for (let i, v in a)
    print(i, "=", v, " ");

print("\n");
```

```
0:x 1:y 2:z
x y z
0=x 1=y 2=z
```

There is no `for … of` in ucode: `for … in` with a single variable gives the values, and with two variables gives index and value. Iterating a string with `for … in` yields nothing, and iterating `null` is a no-op rather than an error — handy when the value came from a lookup that may have missed.

`map()` and `filter()` build new arrays and pass the index as the callback's second argument:

```
print(map(["a", "b"], (v, i) => i + ":" + v), "\n");
```

```
[ "0:a", "1:b" ]
```

Recipes

Removing elements while keeping the array identity (other variables referencing it see the change):

```
let a = [1, 2, 3, 4];

splice(a, 1, 2);

print(a, "\n");
```

```
[ 1, 4 ]
```

Summing, with no `reduce()` in the language:

```
let vals = [1, 2, 3, 4];
let sum = 0;

for (let v in vals)
    sum += v;

print(sum, "\n");
```

```
10
```

Grouping rows by a key, the shape almost every config generator needs:

```
let rows = [["net", "eth0"], ["net", "wlan0"], ["dns", "1.1.1.1"]];
let groups = {};

for (let k, row in rows) {
    if (!exists(groups, row[0]))
        groups[row[0]] = [];

    push(groups[row[0]], row[1]);
}

print(keys(groups), " ", groups.net, "\n");
```

```
[ "net", "dns" ] [ "eth0", "wlan0" ]
```

`exists(table, key)` is the way to test for a key without tripping over a stored `null`, and the insertion order of `groups` is preserved, which is what lets a generated file come out in a stable order.

Notes

- Passing a non-array where an array is expected generally returns `null` rather than raising, so a mistyped argument often shows up later as a `null` in a `print()`.
- `sort()` is stable for its purpose here but is not documented as stable; if equal order matters, include a tiebreaker in the comparator.
- Arrays print with `[` … `]` and `null` elements visible, which makes debugging output honest about holes.
- `min()` / `max()` on an empty spread (`min(...[])`) has no sensible answer; do not rely on it.

Objects

An object in ucode is an ordered hash table from strings to values. That one sentence covers most of what you need: insertion order is preserved, keys are strings, and there is no hidden storage of any kind. Objects are the workhorse value — configuration, parsed JSON, ubus replies, command tables are all objects, and their ordering property is what makes generated files come out the same way twice.

Literals and keys

```
let o = { name: "eth0", mtu: 1500, "ifname": "eth0" };

print(o.name, " ", o["mtu"], " ", keys(o), "\n");
```

```
eth0 1500 [ "name", "mtu", "ifname" ]
```

A key may be an identifier or a quoted string; both produce the same kind of key. Computed keys and identifier shorthand are supported in literals:

```
let k = "count";
let abc = 123;
let o = { [k]: 1, ["max" + "_val"]: 9, abc };

print(keys(o), " ", o.abc, "\n");
```

```
[ "count", "max_val", "abc" ] 123
```

The unqualified `abc` is shorthand for `abc: abc` — a key named by an identifier whose value is that identifier's binding, the same as in ECMAScript object literals.

A numeric literal key is not accepted (`{ 1: "x" }` is a syntax error) — write `{ "1": "x" }`. At runtime, though, any value used as a key is coerced to a string:

```
let o = {};

o[1] = "num";
o[true] = "bool";
o[null] = "nul";

print(keys(o), "\n");
```

```
[ "1", "true" ]
```

The `null` key is silently dropped, because there is no string to store under. Two further consequences follow from keys being C strings internally: a key containing a NUL byte is truncated at that byte, so `"a\0b"` and `"a\0c"` collide, and objects cannot key on composite values.

Reading and writing

Reading a key that is not there gives `null` — no error, no distinction between "absent" and "present but null":

```
let o = { a: null };

printf("[%s] [%s] %s\n", o.a, o.missing, exists(o, "a"));
```

```
[(null)] [(null)] true
```

`exists(table, key)` is the only way to tell the two apart, and it looks at own keys — for the prototype-aware version, use `in`.

Assigning creates the key at the end of the order, and deleting and re-adding it moves it to the end:

```
let o = { a: 1, b: 2, c: 3 };

delete o.b;
o.b = 22;

print(keys(o), "\n");
```

```
[ "a", "c", "b" ]
```

That matters when you render a file from a table and the diff should be small. If you want a key's *position* kept, overwrite it rather than deleting and re-adding.

Order, and what reads it

`keys()` and `values()` return arrays in insertion order, and `for … in` walks in that order too:

```
let o = { first: 1, second: 2, third: 3 };

for (let k, v in o)
    print(k, "=", v, " ");

print("\n");
```

```
first=1 second=2 third=3
```

With one loop variable, `for … in` yields the *keys* for an object (values for an array) — the asymmetry is a frequent source of confusion, so when in doubt, write both variables and ignore the one you do not need:

```
let o = { a: 1, b: 2 };
let one = [];

for (let k in o)
    push(one, k);

print(one, "\n");
```

```
[ "a", "b" ]
```

`length(o)` counts own keys. Nothing counts inherited ones.

Spread and merging

Object literals spread another object's own keys, and the last occurrence of a key wins, which is the idiom for overriding defaults:

```
let defaults = { host: "0.0.0.0", port: 80, tls: false };
let cfg = { ...defaults, port: 8080, name: "api" };

print(cfg, "\n");
```

```
{ "host": "0.0.0.0", "port": 8080, "tls": false, "name": "api" }
```

Note that the override keeps `port`'s *original position*, since the key already existed when the spread inserted it — the value changes, the order does not. Spread copies the top level only; nested objects stay shared, exactly as with arrays.

Spreading works in function calls too, and it flattens arrays into argument lists:

```
let join3 = (a, b, c) => a + "|" + b + "|" + c;
let parts = ["x", "y", "z"];

print(join3(...parts), "\n");
```

```
x|y|z
```

Building objects incrementally

Because key order is insertion order, the standard shape of a generator program is: start with `{}`, insert in the order you want output, then serialise:

```
let rule = {};

rule.name = "allow-dhcp";
rule.proto = "udp";
rule.dest_port = 67;

print(sprintf("%J", rule), "\n");
```

```
{ "name": "allow-dhcp", "proto": "udp", "dest_port": 67 }
```

`sprintf()` with `%J` writes JSON, with keys in that same order. `%J` has a formatting detail invisible to plain `print()`: it pretty-prints with a newline per key when given a precision, and emits a compact single line otherwise.

Nested tables are usually built by reading back what you stored, which is why the `if (!exists(...))` guard appears so often in real ucode programs:

```
let tree = {};

for (let i, row in [["a", 1], ["b", 2], ["a", 3]]) {
    if (!exists(tree, row[0]))
        tree[row[0]] = [];

    push(tree[row[0]], row[1]);
}

print(keys(tree), " ", tree.a, " ", tree.b, "\n");
```

```
[ "a", "b" ] [ 1, 3 ] [ 2 ]
```

Both loop variables are mandatory in the two-variable form — `for (let, row in rows)` is a syntax error ("Expecting label after `local`"), so bind the index to a name even if you never read it.

Comparing and copying

Objects compare by identity. Two objects with the same contents are unequal, and there is no structural comparison operator:

```
print({ a: 1 } == { a: 1 }, " ", { a: 1 } === { a: 1 }, "\n");
```

```
false false
```

For a deep copy, the `%J` / `json()` round trip is the accepted idiom; for a shallow one, spread is what you want:

```
let src = { a: { n: 1 }, b: 2 };
let shallow = { ...src };
let deep = json(sprintf("%J", src));

shallow.a.n = 99;

print(src.a.n, " ", deep.a.n, "\n");
```

```
99 1
```

The shallow copy shares `a`, so mutating through it is visible in the original; the deep copy is independent. Anything that JSON cannot express — functions, resources, regular expressions, `NaN` — does not survive the round trip; `NaN` in fact comes back as the string `"NaN"`.

The `in` operator and the prototype chain

`in` is true when the key exists on the object or anywhere above it in the prototype chain, while `keys()`, `values()`, `length()` and `delete` see own keys only:

```
let base = { shared: 1 };
let o = proto({ own: 2 }, base);

print("own" in o, " ", "shared" in o, " ", keys(o), " ", length(o), "\n");
```

```
true true [ "own" ] 1
```

Metamethods in the prototype are also invisible to `in`, since they synthesise values rather than storing keys — a property that only exists through `__get__` never makes `"that" in o` true. The whole subject of prototypes, `rawget` / `rawset` and the five metamethods is in Prototypes and metamethods.

Recipes

Renaming keys while keeping order:

```
let src = { old_name: 1, keep: 2 };
let dst = {};

for (let k, v in src)
    dst[k == "old_name" ? "new_name" : k] = v;

print(keys(dst), "\n");
```

```
[ "new_name", "keep" ]
```

Selecting a subset with a key list, in the list's order rather than the source's:

```
let row = { id: 7, name: "eth0", mtu: 1500, up: true };
let out = {};

for (let k in ["id", "name"])
    out[k] = row[k];

print(out, "\n");
```

```
{ "id": 7, "name": "eth0" }
```

Testing whether an object is empty, which `!o` will never do since objects are always truthy:

```
print(length({}) == 0, " ", !{} ? "truthy" : "falsy", "\n");
```

```
true falsy
```

Notes

- Object storage is a hash table with an insertion-ordered key list, so key lookup is fast and iteration order is stable, but iterating is not free: for very large tables built in a loop, prefer arrays.
- Key strings are copied into the object's own key storage; there is no restriction on bytes other than the NUL truncation described above.
- `delete` returns true when a key was removed. On an object it never fails, so it is a convenient way to make an optional key absent.
- Assigning a key whose name is one of the five dunder metamethod names is allowed and has no special meaning on that object; see Prototypes and metamethods for why.

Prototypes and metamethods

Every value in ucode except `null` has a type, and values of type object, array, and resource can carry a *prototype*: another object consulted whenever the value itself has nothing to offer. The prototype is the whole of ucode's object system. There are no classes, no constructors and no inheritance keywords, because a prototype chain, plus five special names that hook the interpreter's property accesses, turns out to be enough.

The prototype chain

An object holds its own keys in an ordered hash table. Property access starts there and walks outward:

```
let base = {
    greet: function () {
        return "hello, " + this.name;
    }
};

let obj = { name: "world" };

proto(obj, base);

print(obj.greet(), "\n");
```

```
hello, world
```

The lookup of `greet` fails in `obj`, succeeds in `base`, and the resulting function is called with `this` bound to `obj` — not to `base`. That single rule is what makes shared methods possible: `greet` is stored once, but every object it is reached through reads its own `name`.

The chain may be as deep as you like, and the nearest definition wins. Reading `keys()` reports only own keys, never inherited ones.

Reading and setting a prototype

`proto()` is both a getter and a setter:

```
let base = { tag: "base" };
let obj = {};

print(proto(obj) === null, "\n");

let r = proto(obj, base);

print(r === obj, " ", proto(obj) === base, "\n");
```

```
true
true true
```

Called with one argument, it returns the current prototype, or `null` when there is none. Called with two, it installs the second argument as the prototype and returns the *object*, so prototype installation can be written inline:

```
let base = { tag: "base" };
let obj = proto({ present: "own" }, base);

print(obj.tag, " ", proto(obj) === base, "\n");
```

```
base true
```

Arrays accept prototypes exactly like objects do, which is the only way to give an array behaviour of its own:

```
let a = proto([1, 2], {
    twice: function () {
        return length(this) * 2;
    }
});

print(a.twice(), "\n");
```

```
4
```

Note that `a.length` is still not a number. A prototype supplies *named* properties; it does not turn an array into an object with index properties (see Arrays).

Prototypes are ordinary values, and they may be replaced at any time. Nothing is copied when a prototype is set — the object holds a reference, so mutating the prototype later changes every object that shares it. This is a feature worth knowing about, and worth being careful with.

The five metamethods

A metamethod is a function stored under a reserved *dunder* name that the interpreter itself invokes at specific moments. ucode has exactly five:

Metamethod	Invoked when
`__get__(key)`	reading a property that a raw lookup did not find
`__set__(key, val)`	writing a key the object does not have yet
`__delete__(key)`	deleting a key the object does not have yet
`__call__(...)`	calling the value as a function
`__tostring__()`	converting the value to a string

Arrays take part in `__get__`, `__set__` and `__delete__` as well, but only for keys that are not indices: an array's prototype can serve named reads, writes and deletes, while integer keys always address elements and never reach a metamethod.

```
let store = {};
let a = proto([], {
        __set__: function (key, val) { store[key] = val; },
        __get__: function (key) { return store[key]; },
        __delete__: function (key) { return delete store[key]; }
});

a.label = "lan";
printf("%s %J %J\n", a.label, a, keys(store));
printf("%s %J %J\n", delete a.label, a.label, keys(store));
```

```
lan [ ] [ "label" ]
true null [ ]
```

There are no metamethods for arithmetic or comparison. `+`, `<` and friends always work on the values themselves and never consult a user-supplied function; if you need sums of objects, write a function and call it.

The first thing to understand is when a metamethod is *not* consulted, because it is a fallback rather than an accessor. An own key always wins, and a key found anywhere in the prototype chain also wins:

```
let base = {
    __get__: function (key) {
        return "virtual " + key;
    }
};

let obj = proto({ present: "own" }, base);

print(obj.present, " ", obj.missing, "\n");
```

```
own virtual missing
```

`__get__` may return anything, and that value becomes the result of the property read unchanged. A returned object or array is handed over as the value, it is not looked into any further:

```
let defaults = { host: "0.0.0.0", port: 80 };

let obj = proto({}, {
    __get__: function (key) {
        return defaults;
    }
});

print(type(obj.anything), " ", obj.anything == defaults, "\n");
```

```
object true
```

Delegation is spelled differently: put the object itself in the prototype slot instead of a function. A metamethod slot holding an object or array makes the operation re-dispatch on that value with the same key, which is how a table of defaults is expressed:

```
let defaults = { host: "0.0.0.0", port: 80 };

let obj = proto({}, { __get__: defaults });

print(obj.host, ":", obj.port, " [", obj.nope, "]\n");
```

```
0.0.0.0:80 []
```

When `obj.host` is read, it has no own key, the `__get__` slot holds an object, and the key is looked up *inside* `defaults`, yielding `0.0.0.0`. A key present in neither place reads back as `null`, as `obj.nope` shows. The value found in the slot is looked up like any other object, so it may hold its own keys, and its own metamethods:

```
let virtual = proto({}, {
    __get__: function (key) {
        return "virtual " + key;
    }
});

let obj = proto({}, { __get__: virtual });

print(obj.anything, "\n");
```

```
virtual anything
```

`__set__` behaves like Lua's `__newindex`: it fires only for keys the object does not already have, and its return value is discarded. Storing through it takes an explicit write:

```
let audit = [];

let obj = proto({}, {
    __set__: function (key, val) {
        push(audit, key + " = " + val);
    }
});

obj.a = 1;
obj.b = 2;

print(keys(obj), " ", audit, "\n");
```

```
[ ] [ "a = 1", "b = 2" ]
```

Because `__set__` never stored anything, `obj` still has no keys and every further assignment keeps firing the metamethod. A validating or recording setter therefore has to write the key itself, using `rawset()`.

An object-valued `__set__` slot redirects the store, which needs no function at all:

```
let store = {};
let obj = proto({}, { __set__: store });

obj.a = 1;
obj.b = 2;

printf("%J %J %J\n", keys(obj), keys(store), store);
```

```
[ ] [ "a", "b" ] { "a": 1, "b": 2 }
```

The writes land in `store`, `obj` gains no own keys, and reads through `obj` stay metamethod lookups: they consult `__get__`, not `__set__`, so `obj.a` reads back as `null` unless `__get__` delegates to the same table.

`__delete__` follows the same shape: it is consulted only for keys that are not there, and its return value says whether the delete should report success. An object-valued slot re-dispatches the delete, removing the key from that object:

```
let backing = { a: 1, b: 2 };
let obj = proto({}, { __delete__: backing });

delete obj.a;

printf("%J %s\n", keys(backing), "a" in backing ? "present" : "gone");
```

```
[ "b" ] gone
```

```
let obj = proto({ keep: 1 }, {
    __delete__: function (key) {
        print("refusing to delete ", key, "\n");
        return false;
    }
});

print(delete obj.keep, " ", delete obj.absent, " ", keys(obj), "\n");
```

```
refusing to delete absent
true false [ ]
```

Arrays and resources have no own key storage, so for them `__delete__` is the only route by which a key can be removed at all. Where neither storage nor a metamethod can take the key, the delete raises `Reference error: left-hand side expression is not an object` — the error a plain array raises too — rather than answering

`false`, which would read as though the key simply had not been there. Array indices stay outside the whole arrangement: elements are positional and `splice()` removes them, so `delete a[0]` is refused before any metamethod is consulted.

`__call__` makes a value callable. It receives the call's arguments in the ordinary way, with `this` bound to the callable value:

```
let adder = proto({ base: 10 }, {
    __call__: function (...args) {
        let sum = this.base;

        for (let i = 0; i < length(args); i++)
            sum += args[i];

        return sum;
    }
});

print(adder(1, 2, 3), " ", adder(), "\n");
```

```
16 10
```

`__tostring__` controls how a value renders when a string is needed — in `print()`, in concatenation, and in `%s`:

```
let point = proto({ x: 1, y: 2 }, {
    __tostring__: function () {
        return sprintf("(%d, %d)", this.x, this.y);
    }
});

print(point, " ", "point is " + point, "\n");
```

```
(1, 2) point is (1, 2)
```

A plain `tostring` key is honoured as a legacy alias, which is convenient because it lets an object expose a method named `tostring` that the interpreter will also use by itself.

Metamethods live in the prototype

The lookup rule is fixed: **a metamethod is looked up starting at the object's prototype, never on the object itself.** An own property called `__get__` is just data.

```
let obj = { __get__: "not a metamethod" };

print(obj.__get__, "\n");
print(obj.other, "\n");
```

```
not a metamethod
```

Reading `obj.other` does not call anything — `obj` has no prototype, so there is no metamethod to find, and the read yields `null`. Only the second line's emptiness distinguishes this from a hit; the `__get__` string sitting in the object is inert, as the first line proves.

The reason is safety. If own keys counted, then any ordinary property write could change how a value behaves: a loop copying fields from untrusted data, a mixin, a function someone stores under a dunder name, and suddenly property reads on that object go through code. Making the mechanism prototype-level only means customisation is always an explicit act — you hand an object a prototype that defines the behaviour, and you can see that you did.

Two consequences follow. To customise a single object, give it its own prototype, even a throwaway one:

```
let obj = proto({ x: 1 }, { __get__: function (k) { return "parent"; } });

print(obj.anything, "\n");

obj.__get__ = function (k) { return "own"; };

print(obj.anything, " ", obj.__get__, "\n");
```

```
parent
parent function(k) { ... }
```

and a metamethod is found by walking the *entire* chain from the direct prototype outwards, first hit winning — so a middle prototype can shadow a distant one.

Only closures and native functions count as metamethods. A non-function value under a dunder name is skipped during the walk rather than being called.

The raw escape hatches

A metamethod that needs the underlying storage must ask for it directly, or it will call itself again. `rawget()`, `rawset()` and `rawdelete()` perform the same lookup and the same write the interpreter would, but with metamethods switched off:

```
let obj = proto({}, {
    __get__: function (key) {
        let val = rawget(this, key);

        return val !== null ? val : "default";
    }
});

rawset(obj, "real", 1);

print(obj.real, " ", obj.missing, " ", keys(obj), "\n");
```

```
1 default [ "real" ]
```

`rawget()` works on any value you can name, which also makes it the way to read a property from an object while ignoring the behaviour its prototype provides.

Arrays, indices, and what `in` sees

Array indices are exempt from metamethods. A key that is a valid index never dispatches, because an array's storage is dense and the interpreter is not willing to let user code obscure it. Non-index keys on an array do dispatch:

```
let arr = proto([1, 2], {
    __get__: function (key) {
        return "meta:" + key;
    }
});

print("[", arr[0], "] [", arr[5], "] [", arr.length, "]\n");
```

```
[1] [] [meta:length]
```

The index reads return `1` and `null` as usual; `length`, not being an index, goes through `__get__`.

The `in` operator looks only at real keys, own or inherited. It never runs a metamethod, so a purely virtual property stays invisible to it:

```
let obj = proto({}, {
    visible: 1,
    __get__: function (key) { return "virtual"; }
});

print("visible" in obj, " ", "phantom" in obj, " ", keys(obj), "\n");
```

```
true false [ ]
```

Keep that asymmetry in mind when you write code that iterates: `for (k in obj)` and `exists()` will not see what `obj.anything` happily synthesises.

Pattern: a type without classes

The idiomatic ucode "class" is a plain object holding methods, used as the prototype of the objects it creates. There is no `new` operator; a factory function does the job, and `this` inside `new` is the prototype itself because the method is called on it:

```
let Counter = {
    new: function (start) {
        return proto({ count: start }, this);
    },

    incr: function (n) {
        this.count += (n != null ? n : 1);

        return this.count;
    },

    tostring: function () {
        return "count=" + this.count;
    }
};

let c = Counter.new(5);

print(c.incr(), " ", c.incr(10), " ", c, "\n");
```

```
6 16 count=16
```

The last line prints `count=16` because printing `c` consults `tostring`, found in its prototype `Counter`. Everything here is visible and mutable: `Counter` can be extended, another prototype can be placed above it, and instances that already exist see the changes immediately.

Notes

- Metamethod dispatch applies to resources as well as to objects and arrays; the standard modules build much of their behaviour on prototypes, and a resource's prototype can carry a `__tostring__` like any other.
- A metamethod that raises leaves the exception to propagate normally; wrap the property access in `try { ... } catch (e) { ... }` if you want to catch it.
- Because `__get__` fires on *every* missing key, a careless implementation that builds a table on each call can turn a typo into a hidden allocation. Return `null` early for keys you do not know.
- Prototypes are not copied into objects, so an object's memory cost does not grow with the depth of its chain.

Regular expressions

Regular expressions in ucode are POSIX extended regular expressions, compiled and matched by the C library the interpreter is built against. That single fact governs everything else in this chapter: ucode contributes the syntax for writing a pattern and the shape of the results, but the matching engine itself belongs to glibc, musl or whichever libc the build targets. The practical consequence is that the feature set is POSIX, not Perl — and that it can vary between the machine you develop on and the router your script ships to.

The full reference for the core builtins is generated from the source and published at ucode-lang.org: *the core functions reference*.

Compiling a pattern

A pattern is compiled into a regexp value, either by calling `regexp()` or by writing a literal:

```
printf("%J %J\n", match("port 8080", regexp("[0-9]+")), match("abc", /ab/));
```

```
[ "8080" ] [ "ab" ]
```

`match()` needs a compiled pattern. Passing a plain string as the second argument does not raise; it returns `null`, which is indistinguishable from "no match", and is the way this mistake usually reveals itself:

```
printf("%J\n", match("port 8080", "[0-9]+"));
```

```
null
```

A literal is convenient but ambiguous with division — the parser has to guess from context, and `x / 2 / 3` is arithmetic, not a pattern. A pattern containing a slash is easier to read in `regexp()` form, where the string escaping is also explicit:

```
printf("%J %J\n", match("foo/bar", regexp("foo/(.+)")), match("2/3", /\/(.+)
\//));
```

```
[ "foo/bar", "bar" ] null
```

Literals are compiled; `regexp()` is not

The two forms are not interchangeable in cost. A literal is a compile-time constant: the compiler emits the compiled pattern into the script's constant pool (`compiler.c:76` , `uc_compiler_emit_regexp()` at `compiler.c:676`), so it is built once when the script is loaded and each evaluation merely references it. `regexp()` is an ordinary function call — it runs `regcomp()` every time control passes over it (`types.c:1485`).

The difference is invisible for a single match but expensive in a loop. Matching a fixed string 300,000 times on the development host took 0.51 s with a literal and 0.62 s with `regexp()` in the loop — about a fifth of the runtime spent recompiling a pattern that never changed, on top of the function-call overhead itself. On the slower flash-and-RAM parts these scripts usually run on, that proportion is worse.

So the guidance is not a stylistic preference. Use a literal when the pattern is static. When the pattern is built at runtime — assembled from a configuration value, say — compile it once outside the loop and reuse the value:

```
let pattern = regexp(sprintf("^%s$", "eth[0-9]"));
let n = 0;

for (let i = 0; i < 3; i++) {
        if (match("eth1", pattern)) {
                n++;
        }
}

printf("%d matched\n", n);
```

```
3 matched
```

Recompiling inside the loop would call `regcomp()` again on every iteration for a pattern that cannot have changed. The same reasoning applies to any function whose arguments are loop-invariant; `regexp()` is simply the case where the wasted work is a compile rather than an arithmetic operation.

POSIX, not Perl

The most frequently missing Perl constructs are the character class shorthands, with one twist: the lexer rewrites them in regexp **literals**, but only **outside character classes**, so `\d` in a literal compiles to the digit class and matches digits, while in a string-constructed pattern — or inside a bracket expression — `\d` is just `d` :

```
printf("%J %J %J\n", match("a1b2", /\d+/), match("a1b2", /[\d]/),
match("a1b2", regexp("\\d+")));
```

```
[ "1" ] null null
```

The same applies to `\D`, `\w`, `\W`, `\s` and `\S`: in a literal outside a character class each is spelled out as the matching POSIX class, and anywhere else each is the ordinary letter. Other escapes are not part of the rewrite — in a literal `\b` is a bare `b`, not a word boundary. The safe habit for any pattern that will be built from a string is to write `[0-9]`, `[a-z]`, `[:space:]` and friends explicitly. Lookaround is absent as well, and unlike a non-matching pattern, it is rejected outright at compile time:

```
try {
        regexp("(?=b)c");
}
catch (e) {
        printf("%s: %s\n", e.type, e.message);
}
```

```
Syntax error: Invalid preceding regular expression
```

Alternation follows POSIX's leftmost-longest rule rather than Perl's leftmost-first, which changes the answer for patterns where the order of alternatives was thought to matter:

```
printf("%J\n", match("ab", regexp("(a|ab)")));
```

```
[ "ab", "ab" ]
```

Perl would report `a`, because it takes the first alternative that lets the overall match succeed. POSIX takes the longest match, so the second alternative wins even though it is listed later. This is worth checking whenever a pattern was ported from another language; the fix, when a preference is genuinely wanted, is to make the alternatives mutually exclusive rather than to order them.

Flags

Three flags are recognised, and only three: `i` for case-insensitive matching, `s` for dot-matches-newline, and `g` for global matching. Anything else raises:

```
try {
        regexp("a", "m");
}
catch (e) {
        printf("%s: %s\n", e.type, e.message);
}
```

```
Type error: Unrecognized flag character 'm'
```

There is no `m` flag because anchors are already newline-aware: `^` matches after a newline and `$` before one, with no flag needed — the engine is compiled with `REG_NEWLINE`, except when `s` is in effect, in which case `.` is allowed to cross a line boundary:

```
printf("%J %J\n", match("one\ntwo", regexp("^two")), match("line1\nline2",
regexp("line2$")));
printf("%J %J\n", match("a\nb", regexp("a.b")), match("a\nb", regexp("a.b",
"s")));
```

```
[ "two" ] [ "line2" ]
null [ "a\nb" ]
```

`i` behaves as expected:

```
printf("%J\n", match("HELLO", regexp("hello", "i")));
```

```
[ "HELLO" ]
```

Results

`match()` returns an array holding the whole match followed by each capturing group, or `null` when the pattern does not match. A group that participated in nothing comes back as `null` in its position:

```
printf("%J\n", match("abc123", regexp("([a-z]+)([0-9]+)")));
printf("%J\n", match("b", regexp("(a)|(b)")));
```

```
[ "abc123", "abc", "123" ]
[ "b", null, "b" ]
```

Because there are no named groups, elements are addressed by position, and index `0` is the whole match. An empty match is still a match — `match("", regexp(".*"))` returns a one-element array holding the empty string — so testing the result for `null` is the only reliable way to ask "did it match?".

With `g`, the result is one array per match, nested in an outer array:

```
printf("%J\n", match("a1b2c3", regexp("[0-9]", "g")));
printf("%J\n", match("ABC abc", regexp("[a-z]+", "g")));
```

```
[ [ "1" ], [ "2" ], [ "3" ] ]
[ [ "abc" ] ]
```

The second example shows `g` without `i` skipping the uppercase run entirely. Note what is *not* in the result: no offsets, no match lengths. To know where a match began, the usual approach is to `match()` and then `index()` against the original string, or to structure the pattern so that the interesting part arrives as a group.

Replacing

`replace()` accepts a compiled pattern in place of a literal search string, and captures are referenced in the replacement as `$1`, `$2` and so on:

```
printf("[%s]\n", replace("2024-01-15", regexp("([0-9]+)-([0-9]+)-([0-9]+)"),
"$3/$2/$1"));
```

```
[15/01/2024]
```

Besides the numbered groups, the replacement text recognises a few special sequences: `$&` is the whole match, `$`` is the text before it, `$'` is the text after it, and `$$` is a literal dollar sign. What does **not** expand is `$0` — despite appearing natural, it is left as a literal in the output. To include the matched text, capture it as a group and write `$1`:

```
printf("[%s]\n", replace("2024-01-15", regexp("([0-9]+)"), "<$0>"));
```

```
[<$0>-01-15]
```

A pattern without `g` replaces only the first occurrence — the opposite of the default for a literal search string, which replaces all of them, so the count semantics invert depending on which form of the second argument was passed:

```
printf("[%s] [%s]\n", replace("a1b2", regexp("[0-9]"), "X"), replace("a1b2",
"[1]", "X"));
printf("[%s] [%s]\n", replace("a1b2c3", regexp("[0-9]", "g"), "X"),
       replace("aaaa", regexp("a", "g"), "b", 2));
```

```
[aXb2] [a1b2]
[aXbXcX] [bbaa]
```

The optional fourth argument caps the number of replacements, and only has a visible effect alongside `g`.

The second call on the first line found nothing, and that is the underlying rule rather than a further exception: a string second argument is matched literally, never as a pattern, so `"[1]"` means the three characters `[`, `1`, `]` and not a bracket expression. Metacharacters come into existence only when a compiled regexp is passed — which is also why a bracket expression in a literal search string silently matches nothing instead of raising.

Portability

Since the engine is the host C library's, the features beyond core POSIX ERE are a matter of which libc was linked in, and should not be relied on across builds. Backreferences are the clearest example — a build linked against the GNU C library answers `\1`:

```
printf("%J\n", match("abab", regexp("(a)(b)\\1")));
```

```
[ "aba", "a", "b" ]
```

but backreferences are not part of POSIX ERE, and a musl-based build — the norm on OpenWrt, which is where most ucode scripts end up running — can reject the pattern or match it differently. The same caution applies to any construct that seems to work and has no POSIX pedigree.

The safe discipline for anything that must run on a device is to stick to what POSIX guarantees: bracket expressions with character classes, `*`, `+`, `?`, `{n,m}`, groups, alternation and anchors. Everything else should be tested on the target, not on the build host — and where a pattern must be portable, prefer splitting the work across several simple matches over one clever expression.

Errors and exceptions

ucode distinguishes sharply between two ways a program can go wrong. A *fault* — a type mismatch, a null dereference, a failed `require`, a call on something that is not a function — raises an exception object that the program can intercept with `try` / `catch`. A *failure* — a read that hit EOF, a socket call that the kernel refused, a UCI node that does not exist — is reported by returning `null` and leaving a message behind for the module's `error()` function. Deciding which of the two a given function belongs to is most of what this chapter is about; the language mechanism is small.

Catching

`try` runs a block; if it raises, `catch` runs a block with the exception bound to a name:

```
let text = "{ not json at all";

try {
    let data = json(text);
    printf("parsed %J\n", data);
} catch (e) {
    printf("refused: %s\n", e.message);
}
```

```
refused: Failed to parse JSON string: quoted object property name expected
```

The `catch` block is mandatory — `catch (e);` is a syntax error — and the binding is optional; omitting it is the right form when the message is not wanted:

```
try {
    die("bad");
} catch {
    print("recovered\n");
}
```

```
recovered
```

There is no `finally` clause. Cleanup therefore has to be written twice, or arranged around the `try`:

```
try {
    print("work\n");
} catch (e) {
    print("failed\n");
} finally {
    print("always\n");
}
```

The usual shape in ucode is a plain `try` whose `catch` does the cleanup, with the resource acquired before it so that acquisition failure cannot leave a half-open handle behind:

```
import { open } from "fs";

let fh = open("/tmp/ucode-ch14-demo", "w+");

if (fh) {
    try {
        fh.write("payload\n");
        fh.seek(0);
        printf("read back: %s", fh.read("line"));
    } catch (e) {
        printf("io problem: %s\n", e.message);
    }

    fh.close();
}
```

```
read back: payload
```

The exception object

The value a `catch` receives is not a string. It is an object with exactly three properties:

```
try {
    die("no such interface");
} catch (e) {
    printf("keys=%J\n", keys(e));
    printf("type=%J message=%J\n", e.type, e.message);
}
```

```
keys=[ "type", "message", "stacktrace" ]
type="Error" message="no such interface"
```

`type` is one of a short list of category names, `message` is the bare message without the category prefix, and `stacktrace` is an array of frames, innermost first. Each frame describes one call that was on the way to the fault:

```
function inner() {
    die("deep");
}

function outer() {
    inner();
}

try {
    outer();
} catch (e) {
    printf("frames=%d\n", length(e.stacktrace));
    printf("%J\n", keys(e.stacktrace[0]));
    printf("%J\n", map(e.stacktrace, (f) => [f.line, f.function ??
"(toplevel)"]));
}
```

```
frames=3
[ "filename", "line", "byte", "function", "context" ]
[ [ 2, "inner" ], [ 6, "outer" ], [ 10, "(toplevel)" ] ]
```

`filename` is the source the frame belongs to — the script path, or `[-e argument]` for code given on the command line — `line` and `byte` locate it, `function` names the enclosing function or is `null` at top level, and `context` holds the formatted source excerpt: the same text the interpreter writes to stderr for an uncaught exception, including the caret line. A log message can therefore include the offending line without the program reading the source file itself.

An exception renders as its message, not as its structure. `%s` , `+` and `%J` all use the message:

```
try {
    die("x");
} catch (e) {
    printf("as string: %s\n", e);
    printf("concatenated: %s\n", "pre " + e + " post");
    printf("as JSON: %J\n", e);
}
```

```
as string: x
concatenated: pre x post
as JSON: "x"
```

That last one surprises people who expect `%J` to serialise the object; when a handler wants to pass the whole exception along — to a log, or over `ubus` — build the structure explicitly, `json({ type: e.type, message: e.message, frames: length(e.stacktrace) })` say. There is no `tostring()` builtin to call either; string conversion happens through `+`, `sprintf` and `printf`.

```
try {
    die("x");
} catch (e) {
    printf("%s\n", tostring(e));
}
```

The categories

The interpreter raises four categories of its own, and `die()` and `assert()` add a fifth:

`e.type`	Raised by	Typical message
`Error`	`die()`, `assert()`	the message given, or `Died` / `Assertion failed`
`Type error`	an operation applied to the wrong kind of value	`left-hand side is not a function`, `unable to convert object to number`
`Reference error`	dereferencing through `null`, or a key operation the value kind cannot carry	`left-hand side expression is null`, `left-hand side expression is not an object`
`Syntax error`	compilation, including `loadstring()` and `loadfile()`	`Unexpected token`, `Expecting ';'`
`Runtime error`	the loader and the file-level machinery	`No module named 'x' could be found`, `Unable to open source file ...`

Worth noticing is what is *not* in the list. Arithmetic and comparison never fail: `"a" * 2` and `1 + {}` coerce and produce a number, as chapter 6 describes, and indexing past the end of an array returns `null` rather than raising. A call to an undefined name is a `Type error` about the left-hand side not being a function, not a `Reference error` about the name, because in lax mode an undefined name reads as `null` (chapter 5) and calling `null` is a type error:

```
try {
    nosuchfunction();
} catch (e) {
    printf("%s: %s\n", e.type, e.message);
}
```

```
Type error: left-hand side is not a function
```

Under `-S` the same line fails earlier and more honestly, naming the missing symbol:

```
$ ucode -S -e 'nosuchfunction();'
Reference error: access to undeclared variable nosuchfunction
```

Raising

There is no `throw` statement. The way to raise is `die()`, which takes any value and raises it as an `Error` whose message is that value rendered:

```
for (let v in [null, 42, { a: 1 }, [1, 2]]) {
    try {
        die(v);
    } catch (e) {
        printf("%J\n", e.message);
    }
}
```

```
"Died"
"42"
"{ \"a\": 1 }"
"[ 1, 2 ]"
```

`null` and no argument at all both mean "no message", and anything else goes through the value renderer, which is the one chapter 15 describes. That rendering is a one-way trip: a handler receives text, and a structure that has to survive the trip needs to be serialised on the way out and parsed on the way in (chapter 15 shows the idiom).

`assert()` is `die()` with a condition and a default message, and returns true when it does not raise:

```
printf("assert passed: %J\n", assert(1 + 1 == 2, "math is broken"));

try {
    assert(false);
} catch (e) {
    printf("%s: %s\n", e.type, e.message);
}
```

```
assert passed: true
Error: Assertion failed
```

Raising the caught value again loses its category, because the only way to raise is `die()` and `die()` always raises `Error`. A handler that wants to pass a fault upward unwrapped has to settle for the message:

```
try {
    let x = null;
    x.field;
} catch (e) {
    printf("original: %s\n", e.type);

    try {
        die(e.message);
    } catch (e2) {
        printf("re-raised: %s %s\n", e2.type, e2.message);
    }
}
```

```
original: Reference error
re-raised: Error left-hand side expression is null
```

Uncaught exceptions

A fault that no `catch` intercepts ends the program. The message goes to stderr in a fixed shape — the category and message, then the source position, then the source line with a caret marking the position that raised — and the exit status is 254. A `die()` prints its message without the `Error` prefix:

```
$ ucode -e 'let x = null; x.field'
Reference error: left-hand side expression is null
In [-e argument], line 1, byte 17:

 `let x = null; x.field`
  Near here ------^

$ echo $?
254
$ ucode -e 'die("boom")'; echo "status=$?"
boom
In [-e argument], line 1, byte 11:

 `die("boom")`
           ^-- Near here

status=254
```

A compilation failure — the main script containing a syntax error — is reported the same way but with status 255, because nothing ran. That is the only case in which a `catch` in the same file cannot help: the file is compiled before execution starts, so the handler was never reached:

```
$ ucode -e 'let = 1;'
Syntax error: Expecting variable name
In line 1, byte 5:

 `let = 1`
      ^-- Near here

$ echo $?
255
```

`exit()` is not an exception and cannot be caught; it terminates with the status given:

```
try {
    exit(3);
} catch (e) {
    print("this line never runs\n");
}

print("nor this one\n");
```

Errors that are not exceptions

The system-facing modules report most failures by *returning* a value the caller is expected to test — usually `null`, sometimes `false` — and remembering a message that the module's own `error()` function returns once. This is a deliberate difference: a script that reads a file which may not exist is not in an exceptional situation, and writing `try` / `catch` around every probe would obscure the normal path.

```
import { open } from "fs";
import { error } from "fs";

let fh = open("/definitely/not/here", "r");

printf("result=%J\n", fh);
printf("error=%J\n", error());
```

```
result=null
error="No such file or directory"
```

The convention, module by module, is: `fs`, `io`, `socket`, `serial`, `resolv`, `rtnl`, `nl80211`, `uci`, `ubus` and `digest` report this way, while `json()`, `loadstring()`, `loadfile()`, `require()`, `include()`, `regexp()` and the arithmetic and string builtins raise. `error()` is consume-once in the modules that have it: reading it clears the stored status, so a later success leaves `null` behind, and two reads of the same failure give the message and then nothing. The `require()` loader is the case that catches people: a missing module is an exception, not a `null` return, so probing for a module's presence means a `try`:

```
let have_rtnl = true;

try {
    require("definitely-not-a-module");
} catch (e) {
    printf("not available: %s\n", e.message);
}
```

```
not available: No module named 'definitely-not-a-module' could be found
```

Compilation errors at run time

`loadstring()` and `loadfile()` compile source while the program is running, and a failure there is an ordinary catchable exception — category `Runtime error`, with the inner `Syntax error` report embedded in the message:

```
try {
    let fn = loadstring("let x = ;");
} catch (e) {
    printf("%s\n", e.type);
    printf("%s\n", match(e.message, /^[^\n]*/)[0]);
}
```

```
Runtime error
Unable to compile source string:
```

`loadfile()` reports a file it cannot open the same way, and `include()` reports a file it cannot find:

```
try {
    loadfile("/definitely/missing.uc");
} catch (e) {
    printf("%s: %s\n", e.type, match(e.message, /^[^\n]*/)[0]);
}
```

```
Runtime error: Unable to open source file /definitely/missing.uc: No such
file or directory
```

This is what makes a precompilation strategy useful: `ucc` turns the source into a bytecode image at build time, when a syntax error is a build failure, and the device only loads a result (chapter 3).

Exceptions across calls and callbacks

An exception propagates out of every function and callback frame until a handler is found — including the callbacks the container functions invoke:

```
try {
    map([1, 2, 3], function (v) {
        if (v == 2) {
            die("bad element");
        }
    });
} catch (e) {
    printf("caught out of map: %s\n", e.message);
}
```

```
caught out of map: bad element
```

The interesting boundary is the event loop. A fault inside a timer, handle, signal or process callback is raised *while the loop is running*, where there is no caller frame left to propagate to; `uloop` therefore routes it to a guard function installed with `uloop.guard()`, and without one the loop terminates. That is described in chapter 36; what matters here is that a `try` around `uloop.run()` does not catch anything from inside the callbacks it runs, because by the time the fault happens `uloop.run()` is not the frame the fault is in.

Structuring error handling

Three habits cover most of what ucode needs.

Test the return value of the modules, catch the things that compile. A script that opens files checks `null`; a script that parses text it did not write wraps `json()` in `try`. Mixing the two — wrapping `open()` in `try`, or testing `json()` for `null` — produces code that silently does nothing on the failure path, since neither call fails the other way.

Keep `try` blocks small. The `catch` cannot tell which line raised without consulting the stack trace, so a block that parses, validates and writes three files in one `try` has one handler for three unrelated failures. `e.stacktrace[0].line` and `.context` exist precisely so that the small-block version can name the failure and the big-block version cannot be bothered to.

Turn the module style into exceptions when the normal path wants them. A helper that must not continue on failure can raise what the module reported, which is the one place where the two conventions meet:

```
import { readfile, error as fserror } from "fs";

function must_read(path) {
    let data = readfile(path);

    if (data === null) {
        die(sprintf("cannot read %s: %s", path, fserror()));
    }

    return data;
}

try {
    must_read("/definitely/not/here");
} catch (e) {
    printf("%s: %s\n", e.type, e.message);
}
```

```
Error: cannot read /definitely/not/here: No such file or directory
```

Summary

Question	Answer
Catch syntax	`try { } catch (e) { }` ; binding optional; block required; no `finally`
Raise	`die(value)` or `assert(cond[, msg])` ; there is no `throw`
Caught value	object with `type` , `message` , `stacktrace`
`type` values	`Error` , `Type error` , `Reference error` , `Syntax error` , `Runtime error`
`stacktrace[i]`	`{ filename, line, byte, function, context }` , innermost first
Renders as	its message (`%s` , `+` , `%J` all give the message)
Uncaught status	254; a compilation failure of the main script is 255
`exit()`	terminates, is not catchable
System modules	return `null` and remember a message for `error()` , which is consume-once
`require` , `loadfile` , `loadstring` , `include`	raise a catchable `Runtime error`
Callbacks	propagate to the caller; inside `uloop` , to `uloop.guard()`

JSON and other notations

ucode has one JSON entry point in each direction: `json()` to parse, and `sprintf()` with the `%J` conversion to serialise. There is no `json` module — `require("json")` fails with `No module named 'json' could be found`, and `import json from "json"` fails the same way, because `json` is an ordinary core function registered alongside `print()` and `sprintf()` (`lib.c:6276`). That placement is deliberate: serialisation is common enough that no `require` should stand in its way.

The full reference for the core builtins is generated from the source and published at ucode-lang.org: *the core functions reference*.

Parsing

`json(text)` takes exactly one argument and no options:

```
let cfg = json('{ "name": "eth0", "mtu": 1500, "up": true, "addrs":
["10.0.0.1"] }');

print(cfg.mtu, " ", type(cfg.mtu), " ", cfg.addrs[0], " ", keys(cfg), "\n");
```

```
1500 int 10.0.0.1 [ "name", "mtu", "up", "addrs" ]
```

JSON integers become ucode integers and JSON numbers with a fraction or exponent become doubles, which is visible in `type()`:

```
print(type(json("123")), " ", type(json("12.5")), " ", type(json("1e3")),
"\n");
```

```
int double double
```

`null` in the document is `null` in ucode, `true` / `false` are booleans, and objects and arrays nest recursively. Objects keep the order in which json-c reports the keys, which is the order they appear in the document, so parsed JSON round-trips its own key order — and a *duplicate* key keeps the first occurrence and silently drops the rest.

The argument must be a string, or an object or resource with a callable `read()` method. Anything else raises a type exception, and the message comes straight from the C source:

```
json(42);
```

The parser is more forgiving than JSON

Under the hood, parsing is json-c's `json_tokener`, which accepts several things the JSON specification forbids:

```
print(json("{'a': 1}"), " ", json("[1, 2,]"), "\n");
```

```
{ "a": 1 } [ 1, 2 ]
```

Single-quoted strings and a trailing comma in an array are both accepted. Unquoted object keys, however, are not — those get rejected by json-c with its own wording, wrapped in ucode's message:

```
json("{ a: 1 }");
```

So a document that parses may not be strict JSON. If you are writing JSON out for someone else's parser, always produce it with `%J` rather than by hand, and if you need to *validate* input rather than merely swallow it, round-trip it: `sprintf("%J", json(text))` will fail on anything the parser rejects.

Parse failures are exceptions, so a script that reads untrusted input should catch them. The exact messages are fixed strings in `lib.c`, plus json-c's own error descriptions:

Situation	Exception	Message
Argument not a string/object/array/ resource	type	`Passed value is neither a string nor an object`
Trailing non-whitespace after the document	syntax	`Trailing garbage after JSON data`
Document cut short	syntax	`Unexpected end of string in JSON data`
Anything else json-c rejects	syntax	`Failed to parse JSON string:` *desc*

Trailing whitespace is fine; `"[1,2,]\n"` parses happily. A `syntax` exception means the text was not parseable, which is why the guard is a `try` / `catch` and not an `if`:

```
let text = "{ not json";
let val = null;

try {
    val = json(text);
} catch (e) {
    printf("rejected: %s\n", e);
}

printf("value=[%s]\n", val);
```

```
rejected: Failed to parse JSON string: quoted object property name expected
value=[(null)]
```

A failed parse is an ordinary exception, so JSON handling composes with the language's error handling:

```
try {
    let cfg = json("{ status: ok }");
} catch (e) {
    printf("not JSON: %s\n", e.message);
}
```

```
not JSON: Failed to parse JSON string: quoted object property name expected
```

What the caught value is — an object with `type`, `message` and `stacktrace` — what the categories mean, and how to raise, chain or guard against exceptions is chapter 14's subject. Two facts matter where JSON is concerned.

Exception messages are strings: `tostring` of the caught object renders it the way the interpreter reports an uncaught error, and the message itself is text. `die({ code: 42 })` reaches the handler as the message `{ "code": 42 }` — ucode's own rendering of the value, which looks like JSON but is not something a handler can index into. When a script needs to hand real structure to its caller, serialise on the way out and parse on the way in, which makes the contract explicit instead of incidental:

```
function read_config(path) {
    die(sprintf("%J", { code: 2, path: path }));
}

try {
    read_config("/etc/config/firewall");
} catch (e) {
    let err = json(e.message);

    printf("code=%d path=%s\n", err.code, err.path);
}
```

```
code=2 path=/etc/config/firewall
```

`die(null)` produces the default message `Died`; any other value is stringified into `message`, including numbers and doubles.

Parsing a stream

Because `json()` accepts any value with a `read()` method, a file handle or socket can be fed to it directly and the parser pulls as it needs, stopping at the end of the first complete document (`lib.c:3699-3789`). If the object has no `read()` you get a type exception — `Input object does not implement read() method` — and if `read()` itself raises, that exception propagates out of `json()` unchanged. One consequence worth remembering: a stream that ends in the middle of a document is an error, not a partial value, so the message you get from a truncated file is `Unexpected end of string in JSON data` rather than something about the file.

Serialising with `%J`

The `%J` conversion writes JSON for any value, using ucode's insertion-ordered keys:

```
let rule = { name: "allow-dhcp", proto: "udp", ports: [67, 68], enabled:
true };

printf("%J\n", rule);
```

```
{ "name": "allow-dhcp", "proto": "udp", "ports": [ 67, 68 ], "enabled":
true }
```

The compact form still puts a space after each colon and inside the brackets; that is valid JSON, just not the tightest output. Precision controls indentation, and a pretty print with it:

```
printf("%.4J\n", { a: 1, b: [1, 2] });
```

```
{
    "a": 1,
    "b": [
        1,
        2
    ]
}
```

`%.J` — precision given but empty — indents with a tab instead of spaces, which is what most ucode-generated config files use. There is no `json.stringify`; anything you can write is one `sprintf()` away, and `%J` is usable anywhere a format string is, including `print(sprintf(...))` and template output.

Values without a JSON spelling

Three kinds of ucode value have no direct JSON equivalent, and each gets a defined treatment rather than an error:

```
printf("%J %J %J\n", NaN, Infinity, -Infinity);
```

```
"NaN" 1e309 -1e309
```

`NaN` becomes the *string* `"NaN"`, since JSON has no NaN literal; infinities become `1e309`, a numeric literal that parses back as a double too large to represent and so lands on infinity again. Functions are rendered as their own source text, which keeps a dump of a mixed table readable:

```
printf("%J\n", { handler: () => 1, n: 42 });
```

```
{ "handler": "() => { ... }", "n": 42 }
```

Prototypes are not followed — only own keys are emitted, matching `keys()`. Arrays with `null` holes come out as explicit `null` elements, so an array's length survives a round trip.

Round trips and deep copies

The idiom for a deep copy of plain data is to serialise and reparse:

```
let src = { list: [1, { n: 2 }] };
let copy = json(sprintf("%J", src));

copy.list[1].n = 99;

print(src.list[1].n, " ", copy.list[1].n, "\n");
```

```
2 99
```

It is worth knowing what such a round trip changes. Integers stay integers and doubles stay doubles across the wire, but a string containing an embedded NUL byte does survive parsing (length-preserving `ucv_string_new_length`, `types.c:1708`) while a *key* containing one was truncated when it was stored, so key truncation is permanent. `NaN` returns as the string `"NaN"`. Prototypes, resources and the identity of nested objects are all lost, and any function in the tree comes back as an ordinary string.

For comparing two structures, `%J` gives you a cheap structural comparison, since key order is deterministic:

```
print(sprintf("%J", { a: 1, b: 2 }) == sprintf("%J", { a: 1, b: 2 }), " ",
      sprintf("%J", { a: 1, b: 2 }) == sprintf("%J", { b: 2, a: 1 }), "\n");
```

```
true false
```

That is not a general-purpose equality — it is order-sensitive by construction — but it is enough for the common "did this config change?" test, and it never needs a recursion guard.

Other notations

ucode's own literal syntax is a superset of JSON in the ways you would expect — unquoted identifier keys, single-quoted strings, trailing commas — and `%J` is the inverse mapping back to strict JSON. `%s` is the *display* rendering, which is close to the literal syntax rather than to JSON: unquoted keys, `null` elements visible, strings in double quotes. `%J` is the interchange rendering. The two differ in exactly the places where JSON and JavaScript disagree, so the rule of thumb is `%s` for humans and `%J` for machines — including when the machine on the other end is `json()`.

A `printf("%J", x)` on a value containing a reference cycle terminates cleanly: the serialiser marks the structures it visits and replaces a reference it has already seen — a loop back to an ancestor — with `null`, so the output is finite:

```
let o = {};

o.foo = true;
o.abc = o;
printf("%J\n", o);
```

```
{ "foo": true, "abc": null }
```

Tables that point back at themselves are rare in practice, but if you build one deliberately (a parent pointer in a parsed tree, say), serialise a projection of it rather than the tree itself.

Templates

Alongside ordinary ucode source, the interpreter can process a file in **template mode**: most of the file is literal text to be emitted, and a few tagged regions are ucode that decides what the text says. Templates are how a ucode program generates a configuration file, an HTML page or a DHCP lease block, and the mode is a property of the source file, not of the program.

```
$ cat > hosts.tpl
127.0.0.1 localhost
{% for (let h in ["router", "switch"]): %}
10.1.0.1 {{ h }}
{% endfor %}
$ ucode -T hosts.tpl
127.0.0.1 localhost
10.1.0.1 router
10.1.0.1 switch
```

`-T` switches the input files to template mode; without it — and with `-R`, which restates the default — the same file is parsed as raw ucode, and the tags above are syntax errors. There is no file extension magic: a `.tmpl` file is raw code unless `-T` says otherwise.

The `-T` option's flag list must be attached to the option (`-Tno-lstrip`, not `-T no-lstrip`, because `getopt` only accepts an attached argument for an optional one):

Flag	Effect
`no-lstrip`	keep whitespace between the start of a line and a statement or comment tag
`no-rtrim`	keep the newline following a block tag

Both behaviours are on by default in template mode and are described under Whitespace below.

Expression tags

`{{ expression }}` evaluates the expression and writes its value to the output. Values are rendered the way `sprintf("%J", …)` renders them, with two differences that matter when generating text: `null` and undefined produce nothing at all, and strings are written as themselves rather than quoted and escaped.

```
$ cat > values.tpl
n={{ 42 }} f={{ 1.5 }} b={{ true }} nil=[{{ null }}]
arr={{ [1, 2] }} obj={{ { a: 1 } }}
html={{ "<b>note</b>" }}
$ ucode -T values.tpl
n=42 f=1.5 b=true nil=[]
arr=[ 1, 2 ] obj={ "a": 1 }
html=<b>note</b>
```

Arrays and objects therefore arrive as JSON with the spacing `%J` uses, which makes an expression tag a convenient way to embed structured data in generated JavaScript or in JSON-ish configuration. Note the space inside `[ 1, 2 ]`; generated documents that must be byte-exact should be written against that spacing, or serialised explicitly with `sprintf("%J", …)` / `sprintf("%.J", …)` where the compact or indented form is wanted.

Nothing is escaped for HTML. `{{ "<b>" }}` emits the tag, and a template producing a web page is responsible for escaping its own data: there is no autoescaping mode and no escaping helper in the core language. The convention is a helper function defined at the top of the template:

```
$ cat > esc.tpl
{% function esc(v) {
    return replace(replace(replace(replace("" + v,
        "&", "&amp;"), "<", "&lt;"), ">", "&gt;"), "\"", "&quot;")
} %}<p>{{ esc('a < b & "c"') }}</p>
$ ucode -T esc.tpl
<p>a &lt; b &amp; &quot;c&quot;</p>
```

The ampersand has to be replaced first, or the entities introduced by the later replacements get escaped in turn.

Statement tags

`{% statement %}` runs ucode and contributes nothing to the output. Control structures use the colon form of the body — the same form that lets a multi-line statement fit on one line in raw code (chapter 7):

```
{% for (let i = 1; i <= 3; i++): %}
line {{ i }}
{% endfor %}
```

With the default whitespace handling, this emits exactly three lines, because the lines holding the tags disappear. The closing tag is the ordinary terminator — `endfor` for a loop, `endif` for `if`, `endwhile` for `while` — and `else` / `elif` work between them:

```
{% if (length(items) == 0): %}
# no entries
{% else %}
# {{ length(items) }} entries
{% endif %}
```

A statement that is not a block needs no terminator, and one with a brace body can be written inline, which is how a template defines helpers it uses later:

```
$ cat > list.tpl
{% function li(text) { return "  - " + text + "\n" } %}
{% for (let i = 0; i < 2; i++): %}{{ li("item" + i) }}{% endfor %}
$ ucode -T list.tpl

  - item0
  - item1
```

Function declarations, `let`, `const`, assignments and expression statements all work, and everything the core provides — `sprintf()`, `join()`, `length()`, `trim()`, `replace()`, the regexp functions — is available inside both kinds of tag. Modules are the exception: since everything outside a tag is literal text, an `import` written on a line of its own is printed into the output rather than executed. Put it inside a statement block:

```
$ cat > mod.tpl
{% import { readfile } from "fs" %}
{% let m = require("math") %}
lines={{ length(split(trim(readfile("./mod.tpl")), "\n")) }}
ceil={{ m.ceil(1.2) }}
$ ucode -T mod.tpl
lines=3 ceil=2
```

`require()` behaves there as it does in raw code, including searching the module path with `-L`.

Comments

`{# … #}` is removed from the output. Line comments are not valid outside tags, so a template comment is the only way to annotate the text:

```
{% let port = 8080 %}
server {
    listen {{ port }}; {# default; overridden with -D port=… #}
}
```

Whitespace

Template mode applies two whitespace adjustments by default, and the tags carry modifiers that override them per tag.

- **Whitespace between the start of a line and a block tag is stripped**, so an indented statement tag does not leave its indent in the output. `-Tno-lstrip` keeps it everywhere; `{%+` keeps it for one tag.
- **The newline following a block tag is trimmed**, so a tag alone on a line does not leave a blank line behind. `-Tno-rtrim` keeps those newlines.
- **`-` at either end of any tag strips the whitespace on that side**, across newlines. This is what makes a loop's output contiguous:

```
$ cat > tight.tpl
{%- for (let i = 1; i <= 3; i++): -%}
line {{ i }}
{%- endfor %}
$ ucode -T tight.tpl
line 1line 2line 3$
```

Every newline in the source around the tags is consumed — including the one at the end of the file, which is why the output above ends without a newline and the shell prompt returns on the same line. The text itself has to supply whatever separation is wanted: a space before `line`, or a `{{ "\n" }}` inside the loop body.

The modifiers apply to expression tags as well, and since leading-whitespace stripping concerns only block tags, they are the only way to trim whitespace in front of a `{{` tag:

```
$ cat > mod2.tpl
A   {{- 1 }}   B
C   {{ 2 -}}   D
$ ucode -T mod2.tpl
A1   B
C   2D
```

Comments take the same modifiers, `{#-` and `-#}`.

Literal text, braces and errors

Outside tags, the document is text, and a `{` that does not begin `{{`, `{%` or `{#` is emitted verbatim, so JSON and CSS braces need no handling of their own:

```
server { port = {{ port }}; }
```

A literal `{{` in the output is produced by emitting it from an expression — `{{ "{" }}` — which is also how a template that generates JavaScript writes a placeholder of its own. Text that merely looks like a tag inside a string literal is safe for the same reason: `{{ "a {{ b }} c" }}` prints `a {{ b }} c`.

A malformed tag is a compile-time syntax error naming the position:

```
$ ucode -T broken.tpl
Syntax error: Unterminated template block
In line 1, byte 7:
```

Blocks may not appear inside other blocks:

```
$ ucode -T nested.tpl
Syntax error: Template blocks may not be nested
In line 1, byte 8:
```

Rendering a template from a program

`render(path[, scope])` loads a file as a template, runs it, and returns everything it printed as a string. It is how a raw ucode program uses a template without starting another interpreter:

```
import { writefile } from "fs";

writefile("/tmp/ucode-ch16-iface.tpl", "iface {{ name }}\n");

printf("[%s]\n", trim(render("/tmp/ucode-ch16-iface.tpl", { name: "lan" })));
```

```
[iface lan]
```

The optional second argument is a scope: its properties become global variables inside the template, layered on top of the caller's own global scope, so a template sees both the values handed to it and whatever the program holds globally:

```
import { writefile } from "fs";

writefile("/tmp/ucode-ch16-scope.tpl", "given={{ name }} ambient={{ zone }}
\n");

zone = "fw";

printf("%s", render("/tmp/ucode-ch16-scope.tpl", { name: "lan" }));
```

```
given=lan ambient=fw
```

Note how `zone` reached the template: a plain assignment creates a global variable, and a template's scope is the caller's global scope. A `let` declaration would not have been visible, because the template reads the scope object, not the caller's lexical environment.

`render()` parses the file it is given in template mode, whatever mode the caller is in, while `require()` and `include()` keep the mode they were called from (chapter 17). Inside a template, then, `include()` inlines another template — the mechanism by which a page picks up a shared header — and `require()` loads a raw ucode module. Relative paths for all three resolve against the directory of the file doing the loading, so a template tree moves as a unit; an absolute path always works.

A scope object with an empty prototype, made with `proto()`, leaves the template with nothing but the values listed in it:

```
import { writefile } from "fs";

writefile("/tmp/ucode-ch16-sandbox.tpl", "only={{ only }} other={{ keys ==
null }}\n");

printf("%s", render("/tmp/ucode-ch16-sandbox.tpl", proto({ only: 1 }, {})));
```

```
only=1 other=true
```

`render()` also accepts a function instead of a path. Called that way, it invokes the function with the remaining arguments, captures its output and returns it, discarding the function's own return value — the form for a fragment assembled in code rather than in a file:

```
let out = render(function (name, count) {
    for (let i = 0; i < count; i++)
        printf("%s %d\n", name, i);

    return "ignored";
}, "item", 2);

printf("captured=%J\n", out);
```

```
captured="item 0\nitem 1\n"
```

The capture works by redirecting the VM's output stream, so everything the called code writes ends up in the returned string: output from a template's statement blocks, from a raw module it `require()`s, and from nested `render()` calls. Code that should print instead of returning simply does not call `render()`.

Compiling templates

A template can be compiled to bytecode ahead of time like any source file (chapter 41), and the mode is resolved at compile time, so the compiled file needs no `-T`:

```
$ ucode -T -c -o /etc/firewall/config.uc.o config.tpl
$ ucode /etc/firewall/config.uc.o
```

Flags combine as usual (`-Tno-lstrip -cno-interp`), and a template importing a module that is not installed on the build host compiles with `-cdynlink=name`, described in chapter 17.

Summary

Enable template mode	`ucode -T file`, flags attached: `-Tno-lstrip,no-rtrim`
Raw mode (default)	no option, or `-R`
Emit a value	`{{ expr }}` — JSON-ish for arrays/objects, nothing for `null`
Run code	`{% stmt %}`; control structures take the colon body form
Comment	`{# … #}`
Trim whitespace on one side	`-` on that side of the tag; `+` on a statement tag keeps it
Strip leading whitespace / trim trailing newline	on by default; `no-lstrip` / `no-rtrim`
Import a module	inside a statement block: `{% import … %}`
Render from a program	`render(path[, scope])`, `render(fn, …)` → captured string
Escape HTML	not done automatically; templates escape their own data

Modules and program organisation

A ucode program lives in one file and a ucode *module* lives in another, and the language has two mechanisms for crossing that boundary: the `import` statement, which is resolved when the program is compiled, and the `require()` / `include()` functions, which are resolved while it runs. They are not interchangeable. Knowing which one a piece of code needs is the first question of program organisation in ucode, and the answer is nearly always `import`.

Importing

`import` brings names from another file into the current one:

```
import * as math from "math";

printf("PI=%J, ceil=%J\n", math.PI > 3, type(math.ceil));
```

```
PI=true, ceil="function"
```

`import * as ns` binds the module's exported namespace to one name. `import { a, b }` picks individual names, and `as` renames them locally — the form to use when two modules export the same name, or when a short local name reads better:

```
import { writefile, readfile as slurp, access } from "fs";

writefile("/tmp/ucode-ch17-hosts", "lan\nwan\n");

printf("present=%J, content=%J\n", access("/tmp/ucode-ch17-hosts"),
trim(slurp("/tmp/ucode-ch17-hosts")));
```

```
present=true, content="lan\nwan"
```

An import with no bindings runs the module for its side effects only. The module's own top level stays private, so the visible effects are what its body does — computes, registers with `uloop`, opens something — and nothing else:

```
import "math";

printf("modules=%J\n", keys(modules));
```

```
modules=[ ]
```

A bare import executes the module but does not register it in `modules`, while the two forms that create bindings do register a native module. That asymmetry is only observable through `modules`, which is a registry of loaded native modules plus whatever `require()` has loaded; it is not something to build on, but it is worth knowing when a program inspects `modules` to find out what is available.

What a module exports

A module is an ordinary source file whose top-level declarations are private unless marked `export`. This file — call it `greeter.uc` — is the running example of this chapter:

```
export function hello(name) {
    return `hello ${name}`;
}

export const version = "1.0";

let secret = 42;

function helper() {
    return secret;
}

export function peek() {
    return helper();
}

printf("[greeter loaded]\n");
```

A consumer sees exactly the three exported names:

```
$ cat consumer.uc
import { hello, version } from "greeter";
import * as g from "greeter";

printf("hello=%s version=%s peek=%d\n", hello("world"), version, g.peek());
printf("secret visible=%J\n", g.secret);
$ ucode -L . consumer.uc
[greeter loaded]
hello=hello world version=1.0 peek=42
secret visible=null
```

The exported forms are `export function`, `export const`, `export let` and a trailing `export { a, b, c }` list for naming declarations after the fact:

```
function a() { return "a"; }
function b() { return "b"; }

export { a, b };

export const c = 3;
```

`secret` above is not invisible by accident: a module's unexported top level is as private as a function body, which is what makes a module a unit rather than a global namespace. `g.secret` reads `null` because undefined names read as `null` in lax mode, not because the file failed to load.

`export` is only allowed at the top level of a module file. Putting one inside a function, or in a file that is loaded by `require()` rather than imported, is rejected with `Exports may only appear at top level of a module`.

How imports are resolved

Names are resolved at compile time, against the module search path (see below). Both failure modes of a name are therefore compile-time errors, and neither can be caught:

```
$ ucode -e 'import * as m from "does-not-exist-at-all";'
Syntax error: Unable to resolve path for module 'does-not-exist-at-all'
In [-e argument], line 1, byte 43:

 $ ucode -e 'import { nope } from "greeter";'
Syntax error: Module /tmp/mods/greeter.uc does not export 'nope'
```

The first says the name matched nothing in the search path; the second says the file was found but the name is not among its exports. A typo in an imported name thus fails when the program is compiled, not halfway through running — one of the practical advantages of `import` over `require()`.

Absolute paths may be used, which is how code shared between programs on a system refers to itself:

```
import { is_equal, phy_open } from "/usr/share/hostap/common.uc";
```

The file is found at that path directly, without consulting the search path. OpenWrt's wireless scripts do exactly this, since `wdev.uc` and `wifi-detect.uc` both sit next to `common.uc` but are started from unpredictable working directories.

A module is loaded and executed **once**, no matter how many files import it, and it is executed before the importing file's own statements run. Two imports of `greeter.uc` above printed `[greeter loaded]` once. Consequently, a module body can do setup work — compute a table, open a handle, register a constant — without a lazy-initialization flag, and two consumers cannot see different versions of it.

Circular imports are rejected at compile time rather than handled with partially initialised modules:

```
$ ucode -L /tmp/mods c19.uc
Syntax error: Unable to compile module '/tmp/mods/circ1.uc':

  | Syntax error: Unable to compile module '/tmp/mods/circ2.uc':
  |
  |   | Syntax error: Circular dependency
  |   | In /tmp/mods/circ2.uc, line 1, byte 21:
```

The messages nest, one level per module, because the whole import graph is compiled together. The reporting is not subtle, which is a fair trade for the alternative.

The search path

The path that `import` and `require()` consult is a list of glob patterns, compiled in at build time (chapter 2) and exposed to programs as `REQUIRE_SEARCH_PATH`:

```
printf("patterns=%J, first is a .so pattern=%J\n",
       length(REQUIRE_SEARCH_PATH) > 1, match(REQUIRE_SEARCH_PATH[0], /
\.so$/) != null);
```

```
patterns=true, first is a .so pattern=true
```

`-L dir` on the command line prepends `dir/*.so` and `dir/*.uc` — a `-L` argument containing no `*` is added twice, once for each suffix, and one containing `*` is used verbatim. A program can extend the path itself, which is occasionally useful for a script that ships plugins:

```
import { writefile } from "fs";

writefile("/tmp/ucode-ch17-widget.uc",
"return { build: function (n) { return n * 2 } };\n");
push(REQUIRE_SEARCH_PATH, "/tmp/ucode-ch17-*.uc");

let widget = require("widget");

printf("built=%J\n", widget.build(21));
```

```
built=42
```

Note the difference between that example and everything above it: it uses `require()`, because the file did not exist when the program was compiled.

Dynamic import

`import` is also an expression. Called with a computed or late name, it loads a module while the program runs:

```
let name = "math";
let m = import(name);

printf("type=%J, PI present=%J, modules=%J
", type(m), m.PI > 3, keys(modules));
```

```
type="object", PI present=true, modules=[ "math" ]
```

Dynamic `import()` is the runtime loader for **script** modules: the file is compiled in module mode, its body runs, and the returned namespace object holds exactly its exports. The name is resolved through the search path (an absolute path is not accepted), a missing module raises a catchable `Runtime error`, and the module is registered in `modules`:

```
import { writefile } from "fs";

writefile("/tmp/ucode-ch17-state.uc",
          "let n = 0;\n\nexport function bump() { return ++n }
\n\nprint('[state loaded]\n');\n");

push(REQUIRE_SEARCH_PATH, "/tmp/ucode-ch17-*.uc");

let a = import("state");
let b = import("state");

printf("a=%d a=%d b=%d, same object=%J\n", a.bump(), a.bump(), b.bump(), a
== b);
```

```
[state loaded]
a=1 a=2 b=3, same object=false
```

The body ran once, the state is shared — `b.bump()` continued where `a.bump()` left off — and the functions inside the two namespace objects are identical; only the namespace objects themselves differ, so comparing them with `==` is meaningless.

Loading the same file by both a static `import` statement and a dynamic `import()` produces **two independent instances**: the static import graph and the runtime loader keep separate module tables, the body runs once for each, and they do not share state. Mixed loading of one file is therefore something to avoid rather than reason about. `import()` and `require()` do share through `modules`, which is described with `require()` below.

require()

`require(name)` loads a module at run time and returns a value. What it returns depends on what was found, and this is the single most confusing thing about the function:

- A **native module** (`fs` , `math` , `socket` , …) returns the module's scope object, and the module also appears in the `modules` registry.
- A **script module** (`.uc`) executes the file in its own scope and returns the value of its top-level `return` statement — `null` if the file has none. Its top-level names, exported or not, are not returned; the file is registered in `modules` under the name it was requested by.

That last bullet is worth restating, because it differs from `import` : `require()` registers whatever it loads, script file or native, whereas `import` registers native modules only when the form creates bindings, and never registers a script file.

```
import { writefile } from "fs";

writefile("/tmp/ucode-ch17-plain.uc",
"let hidden = 1;\nfunction f() { return 2; }\n");
push(REQUIRE_SEARCH_PATH, "/tmp/ucode-ch17-*.uc");

printf("no return -> %J\n", require("plain"));
printf("modules after require: %J\n", keys(modules));

require("math");

printf("modules after a native require: %J\n", keys(modules));
```

```
no return -> null
modules after require: [ "fs", "plain" ]
modules after a native require: [ "fs", "plain", "math" ]
```

(`fs` is there because the example imported it at the top; the script file `plain` was registered by `require()`, exactly as the native `math` was.)

A script meant to be used with `require()` therefore ends with a `return` of the object it wants to hand out — the pattern that predates `import` and still works, and is the only way to get a value out of a file loaded at run time:

```
import { writefile } from "fs";

writefile("/tmp/ucode-ch17-retn.uc", "return { a: 1, b: function () { return
2 } };\n");
push(REQUIRE_SEARCH_PATH, "/tmp/ucode-ch17-*.uc");

let m = require("retn");

printf("a=%J b()=%J\n", m.a, m.b());
```

```
a=1 b()=2
```

`require()` resolves names through the search path only — it does not accept the absolute paths that `import` accepts, and it does not look next to the running script. Failure is a catchable `Runtime error` (chapter 14), which makes `require()` the option of choice when a module is genuinely optional:

```
let nl = null;

try {
    nl = require("definitely-not-a-module");
} catch (e) {
    printf("optional module absent: %s\n", e.message);
}

printf("nl=%J\n", nl);
```

```
optional module absent: No module named 'definitely-not-a-module' could be
found
nl=null
```

Native modules loaded by `require()` can also be loaded by `import`, and vice versa; the `import * as fs from "fs"` and `const fs = require("fs")` styles are interchangeable for them. The choice is stylistic — `import` is checked at compile time and cannot be conditional, `require()` can be wrapped in `try` and can take a computed name.

The two runtime loaders compile script files in different modes, and a file can only be loaded by the one its contents agree with: `export` is a syntax error under `require()`, and a top-level `return` is a syntax error under `import`. A module written with `export`s is therefore `require()`-able by nothing:

```
$ ucode -L . -e 'require("greeter")'
Runtime error: Unable to compile source file '/tmp/mods/greeter.uc':

  | Syntax error: Exports may only appear at top level of a module
  | In line 1, byte 1:
```

Both loaders write into the same `modules` registry, and `import()` reads from it before it loads anything: the value registered under a name is what a later `import()` of that name returns. A script loaded by `require()` registers the value of its top-level `return`, so a program that mixes the two forms on one file gets that value back from `import()`, not a namespace of exports:

```
push(REQUIRE_SEARCH_PATH, "/tmp/mods/*.uc");

let r = require("retn");
let m = import("retn");

printf("require -> %J\n", r);
printf("import  -> %J\n", m);
printf("same value=%J, same function=%J\n", r == m, r.b == m.b);
```

```
require -> { "a": 1, "b": "function() { ... }" }
import  -> { "a": 1, "b": "function() { ... }" }
same value=false, same function=true
```

The two results are separate wrappers around one loaded value: the objects are not equal, but the function inside them is the same function, so state changes made through either are visible through both.

include()

`include(path)` runs another source file. It is the weakest of the three mechanisms and is used for running a file, not for organising a program:

```
import { writefile } from "fs";

writefile("/tmp/ucode-ch17-tick.uc", "print('[tick]\n');\nlet counter =
1;\n");

include("/tmp/ucode-ch17-tick.uc");
include("/tmp/ucode-ch17-tick.uc");
print("done\n");
```

```
[tick]
[tick]
done
```

Each call re-reads and re-executes the file — unlike `import`, it is not cached — and its return value is discarded. Its own `let` declarations stay inside the included file, so `counter` above is not visible afterwards.

`include()` takes an optional second argument, a scope object. With one, the included file sees the object's properties as globals, with the caller's own global scope behind them; giving the object an empty prototype with `proto()` leaves it with nothing but the listed properties, which is how untrusted files are included:

```
import { writefile } from "fs";

writefile("/tmp/ucode-ch17-inside.uc",
          "print(given, \"; hidden=\", host_path == null, \"\\n\");\n");

host_path = "/usr/lib/ucode";

include("/tmp/ucode-ch17-inside.uc", proto({ given: "only this", print:
print }, {}));
```

```
only this; hidden=true
```

Without the second argument, the included file runs in the caller's scope and sees `host_path`; with the `proto()` scope it sees only what is listed — even `print` has to be handed over explicitly, since an empty prototype cuts off the core functions too.

Unlike `require()`, `include()` does not consult the search path for names: a bare relative name is taken relative to the including file, so an include that works when run from the module's directory may fail from elsewhere. Absolute paths always work, and a missing file raises a catchable `Runtime error` with the message `Include file not found`.

A file is compiled in the mode its loader is in: `require()` always uses raw mode, `include()` uses the mode of the file calling it, and `render()` always uses template mode (chapter 16). That is why a template can `include()` another template and `require()` a module, and why `include()`ing raw ucode from a template produces that code as literal text.

A load that fails leaves its name registered, which changes what a second attempt reports:

```
import { writefile } from "fs";

writefile("/tmp/ucode-ch17-bad.uc", "export function f() { return 1 }\n");
push(REQUIRE_SEARCH_PATH, "/tmp/ucode-ch17-*.uc");

try {
    require("bad");
} catch (e) {
    printf("first attempt: %s\n", match(e.message, /^[^\n]*/)[0]);
}

let v = import("bad");

printf("after the failure: %J, %J\n", keys(modules), type(v));
```

```
first attempt: Unable to compile source file '/tmp/ucode-ch17-bad.uc':
after the failure: [ "fs", "bad" ], "function"
```

The name `bad` is in the registry after all, holding a value from the aborted load, so a retry of a failed `require()` can report success while returning something that is not the module, and `import()` hands back the same leftover. Code that loads modules conditionally should test for the module once and keep the result, rather than retrying under a different loader.

Precompiled modules

A `.uc` file may hold compiled bytecode instead of text: the type is read from the magic at the front of the file, after any shebang line, and the same entry point handles both. A module deployed that way keeps the name it would have as text, because the search path only ever reaches the two suffixes its templates end in, `.so` and `.uc` — a name such as `util.uc.o` cannot be reached by any template, whatever the template is:

```
$ ucode -L '*.uc.o' app.uc
Runtime error: No module named 'm' could be found
```

What changes is *when* the module is taken in. Resolution still happens while the importer is compiled, so the module has to be present and findable at that moment; but a precompiled module is not linked into the program being compiled — the import is turned into a load at run time, so the module has to be findable then too:

```
$ ucode -cmodule -L 'mods/*.uc' -o app.tmp app.uc && mv app.tmp app.uc
$ ucode -L 'mods/*.uc' app.uc
printed: mark-value
$ ucode app.uc
Runtime error: No module named 'util' could be found
```

That is the opposite of the text case, where precompiling an importer compiles the module in with it and leaves a file that stands by itself. So an application that has been precompiled is self-contained to exactly the extent that its modules are text; `-cdynlink=name` and precompiled modules are the two ways to keep something out of it. `-L` entries go in front of the built-in search path, whose last entries are `./*.so` and `./*.uc`, so a deployment path overrides a module in the working directory rather than the other way round. Dotted names keep mapping to directories, and a precompiled module can import a precompiled module; and as ever the route is `import` rather than `require()`, which on a precompiled module simply yields the null value. Chapter 47 has the format, the tools and the deployment notes.

Organising a program

The rules above produce a small number of decisions, and the codebases in part V have converged on the same answers.

One module per concern, imported by name. A file that parses nftables fragments, a file that owns UCI access, a file of string helpers: each exports its interface, each is imported with `import { ... } from`, and nothing reaches the global namespace except the entry point's own declarations. Compile-time checking of both the file name and the exported names is the benefit worth having.

Absolute paths for shared, installed code. `/usr/share/…/common.uc` style imports make a program's installed layout explicit and independent of the working directory. Relative names are fine for a tree that is always run from its own directory.

`require()` only where the name or the presence is not known at compile time. Optional drivers, plugins, and code that must degrade when a module was not built (chapter 2) are the legitimate uses. Everything else gains from being an `import`, since an unresolvable `import` is a compile-time message naming the module instead of a `Runtime error` in production.

`include()` for running a file, not for sharing code. It has no exports, no caching, no scope visibility, and its path resolution is the least predictable of the three. A script that needs to hand something to a caller should `return` it and be loaded with `require()`.

Compile against modules that are only present on the target. A program that imports `uci` cannot be compiled on a host where `uci.so` is not in the search path — resolution happens at compile time. `ucc` and `ucode -c` take `dynlink=name` (attached, as in `-cdynlink=uci`) to declare that imports of that name refer to a shared extension to be loaded at run time, which lets the file be compiled anywhere and run where the module exists (chapter 41).

Keep the module body cheap and side-effect-light. A module body runs before its first importer's first statement, once per program — so work in it is program-initialisation work, and a `uloop` handler or an open socket registered there is registered for every program that imports the module, including those that only wanted one helper function from it.

Summary

	`import` statement	`import()` expression	`require()`	`include()`
Resolved	at compile time	at run time, search path	at run time, search path	at run time, relative to the file
Absolute path accepted	yes	no	no	yes
Returns	names bound locally / `* as ns`	namespace of exports	native: scope object; script: its `return` value, else `null`	`null`
Script compile mode	module (`export` yes, top-level `return` no)	module	script (top-level `return` yes, `export` no)	script
Executes	once, before the importer	once per name	once per name	once per call
Registers in `modules`	binding forms: native only	yes, native and script	yes, native and script	no
Failure	compile error, not catchable	catchable `Runtime error`	catchable `Runtime error`	catchable `Runtime error`
Circular	rejected at compile time	detected	—	—
Uses `modules` as a cache	no	yes	yes	—
Typical use	organising a program	late/computed script modules	optional native modules, legacy code	running another file

Memory

ucode manages memory in two ways at once. Every value carries a reference count, and a value whose count reaches zero is freed immediately; on top of that, the virtual machine has a mark and sweep collector which finds the structures that reference counting cannot reach. Neither of them runs on a schedule you can miss: the first happens as a matter of course, and the second is only what you make it — it is off until you turn it on.

Values are passed around by reference. Assigning a container, or passing it to a function, shares it rather than copying it:

```
let a = [1, 2];
let b = a;

push(b, 3);
printf("%J %J\n", a, a === b);
```

```
[ 1, 2, 3 ] true
```

To get an independent structure you have to build one: `slice()` for a flat array, or a round trip through JSON for a tree:

```
let o = { a: [1, 2], b: { c: 3 } };
let copy = json(sprintf("%J", o));

printf("%J %J\n", copy == o, copy.a == o.a);
```

```
false false
```

Reference counting

A value is freed as soon as the last reference to it goes away — when the variable holding it is reassigned or goes out of scope, when the entry holding it is deleted, when the container holding it is freed. The point is that release is deterministic: you can reason about when it happens, and for values that own something outside the VM, that matters. A file handle closes when the last reference to it is dropped, not at some later collection:

```
import { open, lsdir } from "fs";

let count = function () {
    return length(lsdir("/proc/self/fd"));
};

let before = count();
let f = open("/dev/null", "r");
let held = count();

f = null;

printf("descriptor taken=%J released=%J\n", held - before, held - count());
```

```
descriptor taken=1 released=1
```

There is no finaliser. No metamethod is consulted when an object is freed — unlike Lua, which calls a `__gc` metamethod on collected objects, ucode has no such mechanism — and putting one on a prototype has no effect:

```
let ran = false;
let p = proto({ __gc: function () { ran = true; } }, {});
let o = proto({}, p);

o = null;
gc();
printf("called=%J\n", ran);
```

```
called=false
```

Code that must release something on a known schedule does it explicitly — a `close` call, or a wrapper function that clears the reference. Embedders get the other half of the mechanism: a resource type registered in C has a free callback which runs when the resource is destroyed, and that is where handles, descriptors, and library contexts are released (chapter 45).

Deleting an entry releases whatever it held, which is the ordinary way to let a large member of a long-lived object go:

```
let cache = { config: { retries: 3 }, body: [1, 2, 3, 4, 5] };

delete cache.body;
printf("%J %J\n", keys(cache), cache.body);
```

```
[ "config" ] null
```

Cycles

What reference counting cannot collect is a cycle: two values referring to each other, referenced by nothing else. Each still has a count of one, held by its partner, so neither is ever freed:

```
let base = gc("count");

function makepair() {
    let a = {};
    let b = {};

    a.o = b;
    b.o = a;
}

for (let i = 0; i < 100; i++) {
    makepair();
}

let after = gc("count");

gc();
printf("freed_all_pairs=%J baseline_restored=%J\n",
       after - gc("count") == 200, gc("count") - base <= 1);
```

```
freed_all_pairs=true baseline_restored=true
```

Four hundred objects, none of them reachable, all of them still allocated until `gc()` marks everything reachable from the running program and frees the rest. A cycle collector is the reason self-referential structures are safe to build in ucode: a parent with child links back to the parent, a memo table that caches closures over itself, a tree of objects each holding its siblings, all become collectable again once the program stops pointing at them, provided something runs the collector.

The collector

The collector is a function, `gc`, with four operations:

Call	Effect
`gc()` or `gc("collect")`	runs a collection cycle, returns `true`
`gc("start"[, n])`	enables periodic collection every `n` allocations (1..65535, default 1000), returns `true` if that changed anything
`gc("stop")`	disables periodic collection, returns `true` if it was enabled
`gc("count")`	returns the number of values the collector tracks

```
printf("stop=%J start=%J start_again=%J stop=%J\n",
       gc("stop"), gc("start"), gc("start"), gc("stop"));
printf("bad_operation=%J bad_interval=%J zero_interval=%J\n",
       gc("bogus"), gc("start", 70000), gc("start", 0));
```

```
stop=false start=true start_again=false stop=true
bad_operation=null bad_interval=null zero_interval=true
```

The first `gc("stop")` returns `false` because **periodic collection is off by default** — a program that creates cycles and never collects leaks them, and it will do so quietly. At startup, `ucode -g interval` turns periodic collection on with the given interval; `gc("start", 0)` means "start with the default interval", which is 1000 allocations. With collection enabled, the same cyclic loop above no longer grows:

```
$ ucode -e 'function mk() { let a = {}, b = {}; a.o = b; b.o = a; } let c0 =
gc("count"); for (let i = 0; i < 200; i++) mk(); printf("retained=%J\n",
gc("count") - c0);'
retained=400
$ ucode -g 50 -e
'function mk() { let a = {}, b = {}; a.o = b; b.o = a; } let c0 =
gc("count"); for (let i = 0; i < 200; i++) mk(); printf("retained=%J\n",
gc("count") - c0);'
retained=8
```

`gc("count")` counts the values the collector keeps in its list — the container-like ones: arrays, objects, closures, resources, programs. Strings and numbers are reference-counted but not tracked, so a loop that builds and discards strings barely moves the count, while closures do:

```
let base = gc("count");
let keep = [];

for (let i = 0; i < 500; i++) {
    print(type("string" + i) == "string" ? "" : "");
}

let afterStrings = gc("count");

for (let i = 0; i < 100; i++) {
    push(keep, function () { return i; });
}

printf("strings_tracked=%J closures_tracked=%J\n",
       afterStrings - base <= 1, gc("count") - afterStrings == 100);
```

```
strings_tracked=true closures_tracked=true
```

Roots

A value is kept if it is reachable from a root. The roots are the VM's global scope, the module registry, the signal handler table, the stack trace of a pending exception, the arguments and locals of every active call, the operand stack, the prototypes of the registered resource types, and resources the VM itself keeps alive.

Two of those are worth spelling out.

Everything loaded by name stays loaded for the life of the process — the registry is a root, so a module required in a loop is instantiated once and never freed, however far away the loop is from anything that uses it:

```
let base = gc("count");
let m = require("math");

printf("tracked_after_require=%J\n", gc("count") - base);
```

```
tracked_after_require=2
```

Second, a resource the VM has been told to keep is kept even when no script variable points at it. This is why an event source armed and then forgotten still works — a timer whose handle was never stored still fires:

```
import * as uloop from "uloop";

function arm() {
    uloop.timer(10, function () {
        print("fired\n");
        uloop.end();
    });
}

arm();
uloop.run();
```

```
fired
```

The other side of that coin is that such a resource holds its callback, and through the callback the scope the callback was written in, until the loop releases it. Long-running programs should cancel or free event sources that are no longer wanted rather than forgetting them.

Closures hold their scope

A function keeps the scope in which it was written alive for as long as the function itself is alive, which is exactly what makes a counter work and exactly what makes a closure expensive:

```
let base = gc("count");

function make() {
    let rows = [{ id: 1 }, { id: 2 }, { id: 3 }];

    return function () {
        return length(rows);
    };
}

let first = make();
let whileAlive = gc("count");

first = null;
gc();

printf("scope_held_while_alive=%J scope_released_after=%J\n",
       whileAlive - base > 2, gc("count") < whileAlive);
```

```
scope_held_while_alive=true scope_released_after=true
```

A closure over a large table keeps the whole table. Where that is a problem, keep the closure small and explicit about what it captures, or pass the data as an argument to a plain function instead.

Measuring

`gc("count")` is the in-script measure of how many tracked values are alive; on Linux the resident set is readable too, which is the honest measure of what a program actually costs:

```
import { readfile } from "fs";

let resident = function () {
    return int(match(readfile("/proc/self/status"), /VmRSS:\s+(\d+)/)[1]);
};

let before = resident();
let rows = [];

for (let i = 0; i < 300000; i++) {
    push(rows, { i: i });
}

printf("grew=%J tracked=%J\n", resident() > before, length(rows) > 0);
```

```
grew=true tracked=true
```

The numbers themselves are not stable across builds and machines, so a program that watches its own memory should compare against its own baseline rather than against a constant.

Stack, not heap

The stack is a separate budget and it is small. A call made in a tail position reuses the current frame and so costs nothing that has to be reclaimed, but an ordinary nested call consumes a frame, and nesting stops at a thousand frames with a `Runtime error: Too much recursion` — chapter 8 has the shapes that are tail positions and the accumulator style that keeps deep recursion cheap. A deep structure held in variables is a heap question and is fine; a deep chain of pending calls is a stack question and needs restructuring.

If an allocation cannot be satisfied, the VM writes `Out of memory` to standard error and exits; there is no recoverable path, and no memory limit is configured or enforced by ucode itself — a runaway program grows until the system says no.

In practice

- Release what owns descriptors explicitly. Reference counting makes release prompt, but only the last reference counts, and a forgotten alias in a long-lived table will keep a socket or a file open for the life of the program.
- Programs that build and discard cyclic structures — graphs, trees with parent links, object registries — should either call `gc()` after a phase of churn or run with periodic collection enabled (`-g 1000`, or `gc("start")` at startup). A daemon that does neither leaks its garbage.
- A daemon's memory growth is usually a lingering reference, not the collector's fault: a closure kept in a global table, a registered signal handler, an event source nobody cancels, a module required with a name assembled in a loop.
- Prefer plain data where plain data will do. Strings, numbers, arrays and objects are cheap to create and are freed the moment they become unreachable; the expensive patterns are the ones that keep things alive for a long time by accident.

Idiosyncrasies

ucode looks like JavaScript, was deliberately influenced by Lua, is written in C for a place where you count bytes and kilobytes, and refuses to be either language. Most of its differences from those languages follow from one decision — keep the surface small enough to fit in a router — and the rest from choices particular to ucode. They are set out here rather than left to be discovered.

This chapter is a checklist. Each entry states the behaviour, names what a newcomer would have expected, and points at the chapter that treats the subject properly.

Numbers do integer things

Integer division truncates. `7 / 2` is `3`, and `-7 / 2` is `-3`, because dividing two integers yields an integer. In JavaScript and Lua 5.3+ you would get `3.5`; to divide "properly" in ucode, write a double literal on one side: `7.0 / 2`. (Operators, Values and types.)

The power operator associates to the left. `2 ** 3 ** 2` is `64`, not `512`. JavaScript makes this a syntax error precisely to avoid the ambiguity; ucode picked an order.

Unary minus binds tighter than `**`. `-2 ** 2` is `4`. In JavaScript it is a syntax error.

Division by zero is a value, not an error. `5 / 0` is `Infinity`, `-5.0 / 0` is `-Infinity` and `0.0 / 0` is `NaN` — the IEEE-754 results, which the integer path reproduces itself. In C this would be a signal and in Python an exception. `5 % 0` is `NaN`, because `%` goes through `fmod()`. (Operators.)

Overflow wraps silently. `2 ** 64` is `0`. There is no widening to double, no error, no warning.

Integers come in signed and unsigned flavours. `type()` says `int` for both, but `~6` prints `18446744073709551609` and `~6 == -7` is false. An expression whose operands are all positive may be computed in unsigned arithmetic and can therefore hold values above `2**63 - 1`; mixing in a negative operand switches back to signed, and an out-of-range unsigned operand saturates to `INT64_MAX` in the process. This is the single most unusual thing about ucode's numbers.

`int()` is decimal-only. `int("0x1f")` is `0`, not `31`; `int("42abc")` is `42`; `int("zz")` is `NaN`. Use `hex()` for a hexadecimal numeral; `hexdec()` decodes hex digits into a byte string.

Integer literals are forgiving, but not JS-forgiving. `0xff`, `0b1011` and `017` (octal, fifteen) all work; a literal with a leading zero that cannot be octal falls back to decimal, so `08` is eight. `0o17` and `1_000_000` are syntax errors.

Doubles print shortly. `3.0` prints `3`, and `0.1 + 0.2` prints `0.3` — the formatter is not round-trip exact, so do not use printed output to compare floating- point values bit by bit.

The syntax you expect is not there

Semicolons are mandatory. There is no automatic semicolon insertion:

```
let a = 1
let b = 2
print(a + b, "\n");
```

`var` does not exist. Use `let`, `const`, or a bare assignment to create a global.

There is no `throw` statement, though there is `try`. `try` / `catch` catches both interpreter faults and `die()`; what is missing is a statement to raise an exception yourself — which is what the `die()` builtin is for — and a `finally` clause (chapter 14). Since `die()` is an ordinary function call, it raises an exception the way any runtime fault does:

```
try {
    die("no good");
} catch (e) {
    print("caught\n");
}
```

```
caught
```

There is no `new`, `class`, `extends` or `super`; no `do { … } while (…)`; no labelled statements; no destructuring (`let {a, b} = o` and `let [x, y] = a` are both rejected); no default parameter values; no getters or setters in object literals; no `for … of`. Numeric keys in object literals are rejected too — `{ 1: "x" }` is a syntax error, while `{ "1": "x" }` is fine.

Everything on that list has a workaround that takes two lines, which is the argument for leaving it out.

Values are not objects

Nothing has methods. Not strings, not arrays, not numbers:

```
let a = [1, 2];

print(a.length, "\n");
```

```

```

`a.length` is `null` because `length` is a *builtin function* in ucode, not a property, and a field read never synthesises one. Write `length(a)`. Likewise `push(a, 3)`, not `a.push(3)`.

```
let a = [1, 2];
print(a.concat([3]), "\n");
```

Calling a non-function raises `Type error: left-hand side is not a function`, which is the error you will see most often in your first week with ucode: it fires for every method call you write out of habit.

Indexing a string raises rather than returning a character:

```
print("hello"[0], "\n");
```

Use `substr("hello", 0, 1)`.

Assigning a named property to an array is silently ignored — no error, no stored value, because an array's storage holds indexed values only:

```
let a = [1];
a.foo = 1;

printf("keys=[%s] foo=[%s]\n", keys(a), a.foo);
```

```
keys=[(null)] foo=[(null)]
```

An array has no named keys to list, which is why `keys()` answers `null` rather than an empty list — a detail worth remembering, since `keys(o)` on a value you *thought* was an object is a common way to discover it was not.

`null`, and the missing

There is no `undefined`. `null === undefined` is `true`; the token `undefined` is just an undeclared name, which reads as `null`.

Reading an undeclared name yields `null` in default (sloppy) mode. Under `-S` it is a reference error. **Assigning to an undeclared name creates a global** in sloppy mode and errors under `-S`.

A missing object key is `null`, and so is an out-of-range array index. Negative array indices count from the end, so `a[-1]` is the last element — and `a[-1] = x` assigns to the last element rather than creating a key named `-1`.

`printf()` renders `null` as `(null)` while `print()` renders it as nothing. The same missing value therefore looks different depending on which output function you used:

```
printf("[%s]\n", null);
print("[", null, "]\n");
```

```
[(null)]
[]
```

`type(null)` returns `null`, not the string `"null"`, and there is a tenth type name beyond the language's nine: handles to files, sockets and ubus connections report `resource`.

`delete` works on object keys only. `delete a[1]` raises a reference error; to remove an array element use `splice()`.

Truth

`0` and `""` are false; `[]` and `{}` are true. The first agrees with JavaScript and differs from Lua, in which every number and string is true; the second agrees with both Lua and JavaScript, and differs from the shell, where an empty string tests false.

Functions and closures

A loop variable is not fresh per iteration. `for (let i = 0; i < 3; i++)` shares one `i`, so closures built inside the loop all see its final value:

```
let fs = [];

for (let i = 0; i < 3; i++)
    push(fs, function () { return i; });

print(fs[0](), " ", fs[1](), " ", fs[2](), "\n");
```

```
3 3 3
```

JavaScript would print `0 1 2`. If you need per-iteration capture, pass the value as an argument to an immediately-invoked function.

Tail calls are optimised. A self-recursive call in tail position does not grow the stack, so an accumulator-style factorial runs to `100000`. Move the same call out of tail position and you get `Too much recursion` after a few thousand frames.

`this` is only bound by method calls. In a plain function it is `null`. Arrow functions capture `this` lexically, which means an arrow written in an object literal sees `null` — it takes `this` from the surrounding script, not from the object:

```
let o = {
    tag: "T",
    arrow: () => "[" + this + "]",
    fn: function () { return "[" + this.tag + "]"; }
};

print(o.arrow(), " ", o.fn(), "\n");
```

```
[null] [T]
```

There is no `Function.prototype.call`, but there is a `call()` builtin with the same purpose and a wider remit: `call(fn, this, scope, ...args)` lets you replace both the receiver and the global scope the function sees.

Objects and prototypes

Object key order is insertion order, and `keys()` and `values()` follow it. This is a guarantee, not an accident, and it is what makes ucode comfortable for generating configuration files where line order matters.

Keys are strings, and a key containing a NUL byte is truncated at that byte — the dictionary is keyed by C strings, so `"a\0b"` and `"a\0c"` are the same key. And a key that is not a string at all is coerced, or dropped: storing under `null` silently stores nothing.

`in` follows the prototype chain; `keys()` does not. An inherited method makes `"method" in o` true while staying out of `keys(o)`.

Metamethods are looked up starting at the prototype, never on the object itself. An own `__get__` property is plain data. This is deliberate and is explained in Prototypes and metamethods; the practical rule is "to customise an object, give it a prototype".

Dereferencing through a missing value raises. `o.a.b` when `o.a` is `null` is a reference error, not `null`. Use `o.a?.b`.

Strings and patterns

Strings are byte strings. `length("héllo")` is `6`, and `substr()` can cut a UTF-8 sequence in half. There is no character type.

`match()` with a plain string pattern returns `null`. A pattern must be a regexp value:

```
print("[", match("abcabd", "b"), "] [", match("abcabd", regexp("b")), "]\n");
```

```
[] [[ "b" ]]
```

The result of a successful match is an array whose first element is the whole match and whose remaining elements are the capture groups; with the `g` flag you get an array of those arrays. `replace()` replaces every occurrence, not just the first, and `split()` with a limit keeps the *remainder* of the string in its last element rather than discarding it: `split("a,b,,c", ",", 3)` gives `["a", "b", ",c"]`.

`wildcard(subject, pattern)` is a glob matcher; neither JavaScript nor Lua has one — the reason it exists is visible in every firewall4-style config generator.

JSON

`json` is a *function*, not a module, and there is no `json.parse` or `json.stringify`:

```
let o = json("{\"a\": 1}");
print(o.a, " ", sprintf("%J", o), "\n");
```

```
1 { "a": 1 }
```

Parsing is `json(text)` — or `json(handle)` to parse a stream incrementally — and serialising is `sprintf("%J", value)`. Malformed input raises a syntax error, which is catchable.

Modules

`import fs from "fs"` does not work: the standard modules have no default export. The form that does is

```
import * as fs from "fs";

print(type(fs), " ", type(fs.open), "\n");
```

```
object function
```

`require()` loads ucode scripts, not built-in C modules; `require("json")` fails because `json` is a core function. The globals `modules`, `REQUIRE_SEARCH_PATH` and `global` are provided by the interpreter (`vm.c:145-154`).

Errors

Runtime errors are exceptions, and they carry a type: type errors, reference errors, runtime errors, syntax errors, and the user-raised kind from `die()`. All are catchable with `try` / `catch`, including errors raised inside builtins. The interpreter's exit status distinguishes a syntax error (255) from a runtime error (254) from a script's own `exit(n)` — which is why a wrapper script must check the status rather than assume zero.

Where you came from

If you write **Lua**: forget `pairs`, `ipairs`, `setmetatable`, `#t`, `..`, `~=` and `self`. `keys()` and `values()` replace `pairs`, `length()` replaces `#`, `+` concatenates strings, `!=` is inequality, and metamethods go in the *prototype*, not in a metatable argument. There is no coroutine, and `string.format` becomes `sprintf`.

If you write **JavaScript**: forget methods on values, `for…of`, destructuring, default parameters, `class`, `Array.isArray`, template tag functions (template *literals* do exist, chapter 9), automatic semicolon insertion, `>>>`, `typeof` — the function `type(v)` returning a string is all there is — and `throw`, since `try` / `catch` exists but script code cannot raise an exception except through `die()`. Integer division truncates — `7 / 2` is `3` — and a double operand lifts only the operation it is an operand of, not the value, so `4 / 3 * 1.0` is `1.0`; `==` on arrays and objects compares identity, as in JavaScript, not contents (chapter 6).

If you write **shell**: `system()` returns the command's exit status as an integer, not its output. To capture output, use `fs.popen()` and read from the handle. Both `system()` and `fs.popen()` take an argument array as an alternative to a shell command string, which is the safer form whenever part of the command comes from outside the script. `$?`-style truthiness does not exist, and `0` means false in ucode where it means success in shell — the single most dangerous one-line difference in this book.

There is no destructuring assignment. `let [a, b] = pair;` and `let {x, y} = obj;` are both rejected with `Expecting variable name`, even though `const` and multi-variable `let a = 1, b = 2;` are fine. Index the result instead:

```
const parts = split("eth0.1:1500", ":");
const name = parts[0];
const mtu = parts[1];

printf("name=%s mtu=%s\n", name, mtu);
```

```
name=eth0.1 mtu=1500
```

Functions that return two values therefore hand back an array, and the caller names the positions by hand. The doc comments in the tree follow the same rule; where an example in a comment looks like destructuring, treat it as shorthand until the parser says otherwise.

The core environment

What is always there

A ucode program starts with a single global object, a set of builtin functions, five predefined names and — depending on how the interpreter was started — a script path and some arguments. Everything else in the language lives in modules and has to be asked for (chapter 17). This chapter is the inventory of what does not, and of the command-line machinery that feeds it.

The full reference for the core builtins is generated from the source and published at ucode-lang.org: *the core functions reference*.

The predefined names

Name	Kind	Value
`ARGV`	array	the arguments after the script name, as strings
`SCRIPT_NAME`	string	the script file name, or the interpreter name in `-e` mode
`REQUIRE_SEARCH_PATH`	array	the glob patterns `require` searches (chapter 17)
`modules`	object	the modules loaded so far, keyed by the name they were loaded under
`global`	object	the global object itself
`NaN`, `Infinity`	double	the floating-point specials (chapter 4)

`ARGV` is what a script's arguments become. It is filled in after option parsing, so the options belong to the interpreter and everything past the script name belongs to the program:

```
$ ucode -e 'printf("%J\n", ARGV)' one two
[ "one", "two" ]
```

`SCRIPT_NAME` is defined only when a file was loaded; reading it in `-e` mode yields `null`, because an undefined name reads as `null` outside strict mode (chapter 5):

```
$ ucode -e 'printf("%J\n", SCRIPT_NAME)'
null
$ printf 'printf("%%J\\n", SCRIPT_NAME)\n' > /tmp/p.uc
$ ucode /tmp/p.uc
"/tmp/p.uc"
```

`REQUIRE_SEARCH_PATH` is an ordinary, writable array of glob patterns — the ones the `-L` option adds, followed by the compiled-in defaults. Assigning to it, or to elements of it, changes where `require` looks (chapter 17):

```
printf("patterns: %J, first ends in .so: %J\n", length(REQUIRE_SEARCH_PATH)
> 0,
       match(REQUIRE_SEARCH_PATH[0], /\.so$/) != null);
```

```
patterns: true, first ends in .so: true
```

`modules` is the registry of loaded modules; it starts empty and grows as modules are required. A module name becomes a property of `modules`, but not a global name (chapter 17):

```
require("math");

printf("modules=%J global.math=%J\n", keys(modules), type(global.math));

let m = require("math");
printf("bound locally=%J, PI present=%J\n", type(m), m.PI > 3);
```

```
modules=[ "math" ] global.math=null
bound locally="object", PI present=true
```

`global.math` stays `null`: requiring a module registers it under `modules` and *returns* the module scope, but does not introduce a name of its own. The `import` statement is the better habit — it binds the names it asks for locally and registers the module in one statement — while binding a `require()` return value is the legacy form: `const m = require("math")`, `global.math = require("math")`, or, in lax mode, a bare `math = require("math")` which becomes an implicit global. Under `-S` the bare form is a `Reference error` instead — chapters 5 and 17.

Note that a module loaded with `import` is a different animal from a file loaded with `require()`: the compiler treats an imported source as a *module* (top-level `return` forbidden, `export` allowed), while a `require`d `.uc` source is compiled as a *script* and its return value — the file's final `return` — is what the loader hands back. A native module's scope object happens to be that return value, which is why `require("math")` yields the namespace the `import` form binds from (chapter 17).

`modules` is also the module cache: any of `import`, `import()` or `require()` that is asked for a name already in the cache gets the cached scope back, and the module is not recompiled or re-evaluated. Deleting the entry forces a reload the next time the name is requested.

`global` is the global object. Names are looked up through it, so a builtin can be replaced through an assignment to the name — and inspected through the object:

```
let count = 0;
let std_print = global.print;

print = function(...args) { count++; std_print(...args) };

print("counted\n");
printf("count=%d\n", count);
```

```
counted
count=1
```

Deleting a name goes through the object too, since `delete` needs a property access:

```
printf("%J ", type(ARGV));
delete global.ARGV;
printf("%J\n", exists(global, "ARGV"));
```

```
"array" false
```

The builtin functions

The functions in this section are what the standard function set exposes when a script runs under the ucode interpreter or a host that loads the standard library into the globals — which is how the `ucode` command itself starts every script. They are functions like any other — `type(print)` is `"function"` — and none of them are methods on anything. A different host program can start a script with a smaller or otherwise shaped globals, so treat this inventory as the set the interpreter provides rather than a feature of the language itself. The grouping here is by what they operate on; the detailed behavior is in the chapter named in the right column.

Output and diagnostics — chapter 3, chapter 21

`print(...)`	write values to stdout, no separators, no newline, `null` omitted
`printf(fmt, ...)`	write formatted output; returns the number of bytes written
`sprintf(fmt, ...)`	the same, returning a string
`warn(...)`	write to stderr, no newline
`trace(level)`	turn VM opcode tracing on (`1`) or off (`0`)

Control of the program — chapter 3, chapter 14

`exit([code])`	stop, with status 0 when called with no argument
`die([msg])`	stop with status 254 after writing `msg` to stderr
`assert(cond[, msg])`	`die` unless `cond` is true
`sleep(seconds)`	suspend; may be interrupted by a signal
`call(fn[, this[, scope[, ...]]])`	invoke a function with an explicit scope (chapter 8)
`loadstring(s)` , `loadfile(p)`	compile to a function without running it
`render(tpl, data)`	render a template string (chapter 16)
`signal(name[, handler])`	install a signal handler (chapter 40)
`system(cmd)`	run a command, returning its exit status (chapter 30)

Types and values — chapter 4

`type(v)`	the type name, one of ten: `null` , `bool` , `int` , `double` , `string` , `array` , `object` , `regexp` , `function` , `resource`
`exists(obj, key)` , `keys(o)` , `values(o)`	presence and contents of objects and arrays
`proto(o[, p])` , `rawget` , `rawset` , `rawdelete`	prototype chain and unhooked access (chapter 12)
`gc()`	force a garbage collection pass
`json(text)`	parse JSON text; it does not serialise — use `sprintf("%J", v)` (chapter 15)

The names `type()` returns are not the same words the reference manual uses for the types: an integer is `int`, a boolean is `bool`, and both a script function and a function implemented in C are `function`. A value of a type the name list does not cover cannot appear in a script.

```
let vals = [null, true, 1, 1.5, "s", [1], {}, regexp("a"), print];
let names = [];
for (let v in vals) {
        push(names, type(v));
}
let fs = require("fs");
let fh = fs.open("/tmp/ucode-ch20-types", "w");
push(names, type(fh));
print(join(" ", names), "\n");
```

```
null bool int double string array object regexp function resource
```

Strings — chapter 9, chapter 13

`length`, `substr`, `index`, `rindex`, `split`, `join`	the string basics
`trim`, `ltrim`, `rtrim`, `uc`, `lc`	stripping and ASCII case
`chr`, `ord`, `uchr`	bytes and code points
`hex(s)`, `hexenc`, `hexdec[, skip]`, `b64enc`, `b64dec`	textual number forms; `b64dec` ignores whitespace and yields `null` for undecodable input
`regexp(flags)`, `match`, `replace`, `wildcard`	patterns
`iptoarr`, `arrtoip`	dotted-quad strings to four-element byte arrays and back
`sourcepath()`	the path of the currently running source file

Those two are the only address helpers in the core; for validating, parsing, or doing CIDR arithmetic on addresses — IPv4, IPv6 and MAC alike — the `netaddr` module (chapter 33) is the full tool.

Containers — chapter 22

`map`, `filter`, `sort`, `reverse`, `uniq`	whole-container transforms
`slice`, `splice`, `push`, `pop`, `shift`, `unshift`	element-level edits
`min`, `max`	extremes of an array or of the arguments
`int(v[, base])`	numeric conversion, with an explicit base for strings

Time — chapter 23

`time()`, `clock()`	wall clock and CPU clock
`localtime`, `gmtime`, `timelocal`, `timegm`	the broken-down time conversions

Modules — chapter 17

`require(name)`, `include(name)`	load a module or source file

Nothing in this list is defined in a module, so nothing here needs an `import`; conversely, anything not in this list — file access, sockets, UCI, math — needs `require`.

How a program reaches the VM

The command line is processed in a fixed order, and understanding it explains a few behaviors that look odd otherwise.

`ucode` decides what it is from `argv[0]`. Named `ucc` it defaults to compilation mode, named `utpl` it turns on template mode, and any other name runs programs. The names are symlinks to the same binary, installed by the build.

Option parsing happens in two passes, with the module search path initialised between them — that is why `-L` directories precede the compiled-in defaults when `require` searches.

Then the global object is filled: `REQUIRE_SEARCH_PATH`, `modules`, `NaN`, `Infinity` and `global` come from the VM, the builtin functions come with them, and `ARGV` and `SCRIPT_NAME` are added by the CLI. `ARGV` is registered as an empty array before the second option pass and filled after it, which is what makes `-U ARGV` able to remove it before the program starts.

Only then is the source read — a file named on the command line, the script from standard input when the file is `-`, or the `-e` string — compiled, and executed. An unresolvable name, a failed compilation or a runtime error ends the program with 255 or 254 (chapter 3, chapter 14).

Strict mode

The `-S` flag turns on the checks that the language otherwise leaves open: assigning to an undeclared name becomes an error instead of an implicit global definition, and a few other laxities are tightened. The details, and the per-source `strict_declarations` form, are in chapter 5. What matters for the environment is that under `-S`, `SCRIPT_NAME` in an `-e` program is an error rather than `null`, since the name was never defined.

The interpreter's own paths

```
$ ucode -e 'printf("%s\n", sourcepath())'
null
```

`sourcepath()` returns the file a piece of code was compiled from, or `null` for code that came from `-e` or `loadstring()`. A module can use it to find data files next to itself; the compiled form of a module reports the compiled path.

Summary

Name / group	What it is
`ARGV`, `SCRIPT_NAME`	script arguments and script name
`REQUIRE_SEARCH_PATH`, `modules`	module loading state
`global`, `NaN`, `Infinity`	the global object and the float specials
output	`print`, `printf`, `sprintf`, `warn`
control	`exit`, `die`, `assert`, `sleep`, `call`, `loadstring`, `loadfile`, `render`, `signal`, `system`
values	`type`, `exists`, `keys`, `values`, `proto`, `rawget`, `rawset`, `rawdelete`, `gc`, `json`
strings	`length`, `substr`, `index`, `rindex`, `split`, `join`, `trim`, `ltrim`, `rtrim`, `uc`, `lc`, `chr`, `ord`, `uchr`, `hex`, `hexenc`, `hexdec`, `b64enc`, `b64dec`, `regexp`, `match`, `replace`, `wildcard`, `iptoarr`, `arrtoip`, `sourcepath`
containers	`map`, `filter`, `sort`, `reverse`, `uniq`, `slice`, `splice`, `push`, `pop`, `shift`, `unshift`, `min`, `max`, `int`
time	`time`, `clock`, `localtime`, `gmtime`, `timelocal`, `timegm`
modules	`require`, `include`

Strings and formatting

String handling in ucode is a handful of short-named functions in the core namespace plus the two formatting functions `printf()` and `sprintf()`. Everything operates on bytes, not characters — there is no encoding layer anywhere in this part of the system, and that fact accounts for most of the behaviour described below.

The full reference for the core builtins is generated from the source and published at ucode-lang.org: *the core functions reference*.

Formatting

`sprintf(format, ...)` returns a string; `printf(...)` writes it to standard output and returns the number of characters written:

```
print(sprintf("%s/%d", "a", 7), " ", type(sprintf("%s", "z")), "\n");

print(printf("x=%d\n", 1), "\n");
```

```
a/7 string
x=1
4
```

The conversion set is the C one, plus ucode's `%J`:

```
printf("1 [%d] [%u] [%x] [%X] [%o]\n", 255, 255, 255, 255, 255);
printf("2 [%f] [%.2f] [%e] [%g]\n", 3.14159, 3.14159, 3.14159, 3.14159);
printf("3 [%s] [%10s] [%.3s] [%c] [%%] [%J]\n", "abcdef", "abcdef",
"abcdef", 65, { a: 1 });
```

```
1 [255] [255] [ff] [FF] [377]
2 [3.141590] [3.14] [3.141590e+00] [3.14159]
3 [abcdef] [    abcdef] [abc] [A] [%] [{ "a": 1 }]
```

Supported conversions are `d u x X o f e g s c J %`. There are no C length modifiers: `%zu`, `%ld`, `%hd` and the like are not recognised and come out as literal text, so a count from `length()` or a size from `fs.stat()` is formatted with `%d`. The usual flags work — field width, `-` to left-align, `+` for a signed value, `0` to zero-pad, and precision, which means decimal places for floats and a maximum length for strings:

```
printf("[%5d] [%-5d] [%+d] [%05d]\n", 42, 42, 42, 42);
```

```
[   42] [42   ] [+42] [00042]
```

`%c` takes an integer and emits the byte; `%J` serialises any value as JSON, which is covered in chapter 15. `%s` on a composite value gives the display rendering, so objects arrive in literal form:

```
printf("[%s] [%s]\n", { a: 1 }, null);
```

```
[{ "a": 1 }] [(null)]
```

`print()` and `warn()` take a list of values rather than a format string, and they omit `null` and `undefined` arguments entirely — neither `null` nor `(null)` appears — which is the one place the rendering rules of chapter 4 do not apply:

```
print("[", null, undefined, "]", "\n");
print("[ ", null, undefined, " ]\n");
```

```
[]
[  ]
```

Missing and invalid arguments

Formatting never raises. A conversion with no corresponding argument gets a default — zero for numeric conversions, and the `(null)` placeholder for `%s`:

```
printf("[%d][%s]\n", 1);
```

```
[1][(null)]
```

An unrecognised conversion specifier is emitted verbatim and consumes no argument, so a typo produces visibly wrong output rather than an error:

```
printf("%z|%d\n", 1);
```

```
%z|1
```

`%u` reinterprets the bits as unsigned, exactly as C does — and because ucode integers are 64-bit throughout, a negative value becomes a 64-bit unsigned quantity rather than a 32-bit one:

```
printf("%d %u\n", -5, -5);
```

```
-5 18446744073709551611
```

That is `2^64 - 5`, not the `4294967291` a C programmer half-expecting a 32-bit `unsigned` might guess. Expressions can produce an unsigned integer flavour on their own — `2 ** 63` is one, as chapter 4 shows — so `%u` is how you render values that would be negative when read as signed, which is common when handling bitmasks read from `struct` or `ioctl`.

The string functions

Fifteen functions live in the core namespace: `length`, `trim`, `ltrim`, `rtrim`, `uc`, `lc`, `index`, `rindex`, `substr`, `split`, `join`, `match`, `replace`, `regexp` and `wildcard`. The names are short and the first argument is almost always the string, with two important exceptions noted below.

```
printf("[%s] [%s] [%s]\n", trim("  x  "), ltrim("  x  "), rtrim("  x  "));
printf("[%s] [%s]\n", uc("aB-c"), lc("aB-c"));
```

```
[x] [x  ] [  x]
[AB-C] [ab-c]
```

`uc` and `lc` are the uppercase and lowercase operations; `trim` without a second argument strips whitespace from both ends.

Length and indexing are in bytes

`length()` counts bytes, so a multi-byte character contributes more than one:

```
print(length("héllo"), "\n");
```

```
6
```

`index()` and `rindex()` return a zero-based byte offset, or **-1** when the needle is absent — not `null`, which matters because `if (index(s, t))` is true for every found position *and* is truthy for `-1`. Always compare against `-1`:

```
printf("%d %d %d\n", index("hello world", "o"), rindex("hello world", "o"),
       index("hello world", "z"));
```

```
4 7 -1
```

Both functions work on arrays as well as strings, returning the element index, and both return `-1` when the value is absent.

Each takes an optional third argument — a byte offset for strings, an element index for arrays — so a string can be scanned progressively without copying the remainder on every pass. A negative offset counts from the end, like `substr()`, and an out-of-range offset is clamped. For `rindex()` the offset is an upper bound: only indices at or below it are considered.

```
printf("%d %d %d\n", index("hello world", "o"), index("hello world", "o", 5),
       index("hello world", "l", -3));
printf("%d %d\n", rindex("hello world", "o", 5), index([1, 2, 3, 2, 1], 2,
2));
```

```
4 7 9
4 3
```

That makes the usual scanning loop read naturally, with the position advancing past each hit:

```
let s = "one two three", pos = 0, n = 0;

while ((pos = index(s, " ", pos)) >= 0) {
        n++;
        pos++;
}

printf("scan found %d separators\n", n);
```

```
scan found 2 separators
```

Substrings and splitting

`substr(string, start [, length])` takes a byte offset and an optional byte count. A negative start counts from the end, and a missing length runs to the end:

```
printf("[%s] [%s] [%s]\n", substr("hello", 1, 3), substr("hello", 2),
substr("hello", -2));
```

```
[ell] [llo] [lo]
```

`split(string, separator [, limit])` returns an array of the pieces; the limit caps the number of pieces and leaves the remainder of the string intact in the last one:

```
printf("%J %J\n", split("a,b,c", ","), split("a,b,c", ",", 2));
```

```
[ "a", "b", "c" ] [ "a", "b,c" ]
```

join() takes the separator first

`join(separator, array)` — separator first, array second, which is the reverse of the JavaScript method and of every other function in this chapter. Getting it wrong is silent:

```
printf("[%s] [%s]\n", join("-", [1, 2, 3]), join([1, 2, 3], "-"));
```

```
[1-2-3] [(null)]
```

The second call returns `null` rather than raising, which is how this mistake usually presents itself: an empty field in some output far away from the cause.

Matching and replacing

`match(string, regexp)` needs a real regular expression object as its second argument. Passing a plain pattern string does not raise — it returns `null`, which is indistinguishable from "no match":

```
printf("%J %J\n", match("port 8080", "[0-9]+"), match("port 8080",
regexp("[0-9]+")));
```

```
null [ "8080" ]
```

A successful match returns an array of the captured groups, and regular expressions are POSIX extended, not Perl-style — character classes and repetition work, but the Perl-specific shorthands do not. Chapter 13 covers them properly.

`replace(string, from, to [, count])` replaces every occurrence by default, unlike JavaScript's `String.prototype.replace` with a string pattern, which replaces only the first:

```
printf("[%s] [%s] [%s]\n", replace("a-b-c", "-", "+"), replace("aaa", "a",
"b", 2),
       replace("hello world", "o", "0", -1));
```

```
[a+b+c] [bba] [hello world]
```

Note the third case: a count of `-1` replaces *nothing* rather than everything, because the count is used as an unsigned limit and a negative value is not special-cased. To replace all occurrences, omit the argument.

`wildcard(string, pattern)` matches a shell-style glob against a whole string — again subject first, pattern second, the opposite of what the name suggests:

```
printf("%s %s\n", wildcard("index.uc", "*.uc"), wildcard("index.js",
"*.uc"));
```

```
true false
```

Working with bytes in practice

Because every one of these functions counts bytes, `substr` and `length` can cut a multi-byte character in half, producing invalid UTF-8 that a terminal will render as a replacement glyph. For ASCII configuration data this never matters. For user-facing text — device hostnames, DHCP client names, LuCI page output — the safe patterns are to split on separators rather than fixed offsets, and to trim with `substr(line, 0, 80)` only where a broken trailing character is acceptable. The same applies to `index()`: it locates byte offsets, so an offset meant for display should never be reported as a character column.

The functions themselves are cheap and allocation-light, which is why they are globals rather than string methods: a template rendering a few hundred DHCP leases in `trim()` and `split()` does no more than copy bytes, and no object is created per string in the process.

Arrays and objects as containers

Chapter 10 and chapter 11 introduced arrays and objects as values; this chapter is about the functions that move data through them. None of these functions are methods — a string, an array and an object have no methods at all — they are ordinary builtins that take a container as their first argument:

```
let hosts = ["router", "switch", "ap"];

printf("%d %s\n", length(hosts), join(",", hosts));
printf("%s\n", join(", ", map(hosts, (h) => uc(h))));
```

```
3 router,switch,ap
ROUTER, SWITCH, AP
```

The toolkit is small enough to hold in the head: `map`, `filter`, `sort`, `reverse`, `slice`, `splice`, `push`, `pop`, `shift`, `unshift`, `uniq`, `join`, `keys`, `values`, `exists`, `index`, `rindex`, `length`, `min`, `max`, `split`, `delete`. There is no `reduce` and no `concat`; a loop or a spread covers both, and the recipes at the end of the chapter show the idioms.

The full reference for the core builtins is generated from the source and published at ucode-lang.org: *the core functions reference*.

Two things about this family are worth learning before using it. The first is that a container function given the wrong kind of value — `map()` on an object, `join()` on an object, `push()` on an object — returns `null` rather than raising. The second is that some of them modify the container they are given and some do not, which the table below states for each.

Function	Signature	Modifies	Returns
`length`	`length(container)`	—	number of elements or characters
`map`	`map(array, cb(value, index, array))`	—	new array of results
`filter`	`filter(array, cb(value, index, array))`	—	new array of kept elements
`sort`	`sort(container[, cb(a, b)])`	yes	the same container
`reverse`	`reverse(array)`	—	new array, reversed
`slice`	`slice(container, start[, end])`	—	new array or string
`splice`	`splice(array, start[, count[, ...values]])`	yes	the same array
`push`	`push(array, ...values)`	yes	last value pushed
`pop`	`pop(array)`	yes	removed element, `null` if empty
`shift`	`shift(array)`	yes	removed element, `null` if empty
`unshift`	`unshift(array, ...values)`	yes	last value inserted
`uniq`	`uniq(array)`	—	new array, duplicates removed
`join`	`join(separator, array)`	—	string
`keys`	`keys(object)`	—	new array of key names
`values`	`values(object)`	—	new array of values
`exists`	`exists(object, key)`	—	boolean
`index` / `rindex`	`index(container, needle[, offset])`	—	position or `null`
`min` / `max`	`min(a, b, ...)`	—	the smallest/largest argument

Mapping and filtering

`map()` builds a new array by applying a function to every element. The function is called with the value, its index, and the whole array, and may declare as many of those as it needs:

```
import * as math from "math";

let n = [4, 9, 16];

printf("%J\n", map(n, (v) => math.sqrt(v)));
printf("%J\n", map(n, (v, i) => i));
printf("%J\n", map(n, function (v, i, arr) { return v == arr[i]; }));
```

```
[ 2.0, 3.0, 4.0 ]
[ 0, 1, 2 ]
[ true, true, true ]
```

`filter()` keeps the elements for which the function returns something truish, and renumbers the result:

```
let ports = [22, 80, 443, 8080];

printf("%J\n", filter(ports, (p) => p < 1024));
printf("%J\n", filter(ports, (p, i) => i % 2 == 1));
```

```
[ 22, 80, 443 ]
[ 80, 8080 ]
```

"Truish" is the same test `if` uses, so the elements that disappear under a bare identity filter are `null`, `false`, `0`, `0.0` and `""` — note that the string `"0"` and the empty array survive:

```
printf("%J\n", filter([0, "0", "", [], null, false, 1], (x) => x));
```

```
[ "0", [ ], 1 ]
```

Neither function works on an object; use `keys()` with a loop or `map()` over `keys()` instead:

```
let conf = { lan: "eth0", wan: "eth1" };

printf("%J %J\n", map(conf, (v) => v), keys(conf));
printf("%J\n", map(keys(conf), (k) => [k, conf[k]]));
```

```
null [ "lan", "wan" ]
[ [ "lan", "eth0" ], [ "wan", "eth1" ] ]
```

Sorting

`sort()` sorts in place and returns the array it was given, so the argument and the result are the same object. Numbers sort numerically, strings bytewise, and arrays element by element:

```
let a = [10, 9, 2];
let b = sort(a);

printf("%J %s\n", a, a == b);
printf("%J\n", sort(["b", "A", "a", "B"]));
printf("%J\n", sort([[2, "z"], [1, "a"]]));
```

```
[ 2, 9, 10 ] true
[ "A", "B", "a", "b" ]
[ [ 1, "a" ], [ 2, "z" ] ]
```

A comparator may be supplied. It is called with two elements and should return a negative number, zero or a positive number; anything truish other than zero is taken as "swap", so the difference — the natural choice in a language whose booleans are numbers — works, and returning `null` leaves the pair in place:

```
printf("%J\n", sort([1, 3, 2], (x, y) => y - x));
printf("%J\n", sort([1, 3, 2], (x, y) => y > x));
printf("%J\n", sort([2, 1], () => null));
```

```
[ 3, 2, 1 ]
[ 3, 2, 1 ]
[ 2, 1 ]
```

Sorting objects is supported and sorts their **keys**, values moving with them, which is the way to get an ordered walk out of an object whose insertion order was wrong:

```
let conf = { wan: "eth1", lan: "eth0" };

sort(conf);
printf("%J %J\n", conf, keys(conf));
```

```
{ "lan": "eth0", "wan": "eth1" } [ "lan", "wan" ]
```

Sorting values of different types against each other is not meaningful — there is no defined order between a number and a string — so the result of sorting a mixed array depends on the internals of the sort. Keep arrays homogeneous, or supply a comparator that projects to a comparable value:

```
let entries = [{ name: "wan", metric: 20 }, { name: "lan", metric: 5 }];

sort(entries, (x, y) => x.metric - y.metric);
printf("%J\n", map(entries, (e) => e.name));
```

```
[ "lan", "wan" ]
```

Because `sort()` modifies its argument, sorting something you do not own means copying it first. `slice()` is the copy:

```
let original = [3, 1, 2];
let top = slice(original);

sort(top, (x, y) => y - x);
printf("%J %J\n", original, top);
```

```
[ 3, 1, 2 ] [ 3, 2, 1 ]
```

Adding and removing elements

`push()` appends, and answers with the last value it inserted — not with the new length, which is `length(q)` afterwards. `pop()` and `shift()` remove from the end and from the front respectively, and answer with the element they removed, or `null` when the array is empty:

```
let q = ["a", "b"];

printf("push -> %J, array %J\n", push(q, "c", "d"), q);
printf("pop -> %J, array %J\n", pop(q), q);
printf("shift -> %J, array %J\n", shift(q), q);
printf("pop on empty -> %J\n", pop(pop(q)));
```

```
push -> "d", array [ "a", "b", "c", "d" ]
pop -> "d", array [ "a", "b", "c" ]
shift -> "a", array [ "b", "c" ]
pop on empty -> null
```

`unshift()` prepends, and answers the same way `push()` does — with the last value inserted:

```
let q = [1];

printf("unshift -> %J, array %J\n", unshift(q, 5, 6), q);
```

```
unshift -> 6, array [ 5, 6, 1 ]
```

`splice()` is the general insertion and removal: it starts at an index, removes `count` elements, and inserts any further arguments there. It returns the array it modified — **not** the elements it removed, which is the opposite of the convention in JavaScript:

```
let l = [1, 2, 3, 4, 5];
let r = splice(l, 1, 2, "x");

printf("%J %s\n", l, r == l);
printf("%J\n", splice([1, 2], 1, 0, 9, 8));
printf("%J\n", splice([1, 2, 3], 1));
```

```
[ 1, "x", 4, 5 ] true
[ 1, 9, 8, 2 ]
[ 1 ]
```

With no `count`, everything from the start index to the end goes. A start index past the end does nothing.

Copies and sharing

Arrays and objects are references. Assigning one to another name does not copy it, and copying a container copies the container, not what it holds:

```
let x = [1, [2]];
let y = slice(x);

y[0] = 9;
y[1][0] = 9;
printf("%J %J\n", x, y);
```

```
[ 1, [ 9 ] ] [ 9, [ 9 ] ]
```

The same holds for objects built with a spread, which is how objects get merged:

```
let defaults = { timeout: 5, retry: { n: 1 } };
let conf = { ...defaults, timeout: 10 };

conf.retry.n = 7;
printf("%J %J\n", conf, defaults);
```

```
{ "timeout": 10, "retry": { "n": 7 } } { "timeout": 5, "retry": { "n": 7 } }
```

`reverse()` is one of the functions that leaves its argument alone:

```
let a = [1, 2, 3];

printf("%J %J\n", reverse(a), a);
```

```
[ 3, 2, 1 ] [ 1, 2, 3 ]
```

Keys, values and membership

`keys()` and `values()` work on objects only; on an array they return `null`, because an array's indices are not entries. An array has no room for named properties either: writing one is silently discarded, reading it back answers `null`, and `delete` refuses the array outright. A prototype carrying `__set__`, `__get__` and `__delete__` is what gives an array named properties — chapter 12:

```
let a = [10, 20];

printf("%J %J %J\n", keys(a), values(a), length(a));
a.note = "extra";
printf("%J %J\n", a.note, length(a));
printf("%J\n", (function () { try { return delete a.note; } catch (e) {
return "error: " + e; } })());
```

```
null null 2
null 2
"error: left-hand side expression is not an object"
```

For objects, `keys()` answers in insertion order, which is the order `for ... in` walks, and both are unaffected by anything but an explicit `sort()` of the object:

```
let m = { zulu: 1, alpha: 2 };

printf("%J\n", keys(m));
printf("%J\n", values(m));
printf("%J\n", sort({ ...m }));
```

```
[ "zulu", "alpha" ]
[ 1, 2 ]
{ "alpha": 2, "zulu": 1 }
```

Membership has two spellings, `exists(object, key)` and the `in` operator, and both are object operations: an array reports `false` even for an index it holds, so ask about array membership through `index()` or by comparing to `length()`:

```
let m = { lan: "eth0" };
let a = [10, 20];

printf("%J %J\n", exists(m, "lan"), "lan" in m);
printf("%J %J %J\n", exists(a, 0), 0 in a, 0 < length(a));
printf("%J %J\n", index(a, 20) != null, "lan" in a);
```

```
true true
false false true
true false
```

`exists()`, unlike a direct read, does not consult the prototype chain — the distinction is chapter 12's.

One element, or none

`index()` and `rindex()` search arrays and strings and answer with a position, or with `-1` when the needle is absent (chapter 21). Both take an optional offset: for `index()` it is where to start looking, for `rindex()` it is the highest position that will be considered, and a negative offset counts from the end:

```
let a = ["lan", "wan", "lan"];

printf("%J %J\n", index(a, "lan"), rindex(a, "lan"));
printf("%J %J\n", index(a, "lan", 1), rindex(a, "lan", 1));
printf("%J %J\n", index(a, "wlan"), index("hello world", "o", 5));
```

```
0 2
2 0
-1 7
```

`uniq()` removes consecutive duplicates by value and type, so `1` and `"1"` are both kept:

```
printf("%J\n", uniq([1, "1", 1, 2, null, null]));
```

```
[ 1, "1", 2, null ]
```

`min()` and `max()` take any number of arguments of any type and return one of them unchanged; they are not the numeric fast paths, which are `math.fmin()` and `math.fmax()` (chapter 24):

```
printf("%J %J %J\n", min(3, 1, 2), max("a", "b"), min());
```

```
1 "b" null
```

Recipes

The functions above compose into most of what a container needs.

The recipes lean on `for ... in`, whose loop variables mean different things for the two container kinds: over an array one variable receives the **values** and a second receives them alongside their index, while over an object one

variable receives the **keys**. Chunking a list — the shape of every "process in batches" loop, and the one place an index is wanted:

```
let list = [1, 2, 3, 4, 5];

for (let i, v in list) {
        printf("%d:%d ", i, v);
}

printf("\n");
```

```
0:1 1:2 2:3 3:4 4:5
```

```
function chunk(list, size) {
        let out = [];

        for (let i = 0; i < length(list); i += size) {
                push(out, slice(list, i, i + size));
        }

        return out;
}

printf("%J\n", chunk([1, 2, 3, 4, 5], 2));
```

```
[ [ 1, 2 ], [ 3, 4 ], [ 5 ] ]
```

Flattening one level, with no `concat` to reach for:

```
let nested = [[1, 2], [3], [], [4, 5]];
let flat = [];

for (let inner in nested) {
        for (let v in inner) {
                push(flat, v);
        }
}

printf("%J\n", flat);
```

```
[ 1, 2, 3, 4, 5 ]
```

Grouping, the operation a `GROUP BY` performs, built from a plain object accumulator:

```
let leases = [
        { ifname: "lan", mac: "aa:bb" },
        { ifname: "wan", mac: "cc:dd" },
        { ifname: "lan", mac: "ee:ff" }
];

let by_ifname = {};

for (let lease in leases) {
        let key = lease.ifname;

        if (!exists(by_ifname, key)) {
                by_ifname[key] = [];
        }

        push(by_ifname[key], lease.mac);
}

printf("%J\n", by_ifname);
```

```
{ "lan": [ "aa:bb", "ee:ff" ], "wan": [ "cc:dd" ] }
```

Counting the same way, and taking the most frequent element:

```
let words = ["eth0", "eth1", "eth0", "eth0", "eth1"];
let counts = {};

for (let w in words) {
        counts[w] = (counts[w] || 0) + 1;
}

let order = sort(keys(counts), (x, y) => counts[y] - counts[x]);

printf("%J %J\n", counts, order[0]);
```

```
{ "eth0": 3, "eth1": 2 } "eth0"
```

Converting between an object and an array of pairs, which is how an object gets through a function that only speaks arrays:

```
let m = { a: 1, b: 2 };
let pairs = map(keys(m), (k) => [k, m[k]]);
let back = {};

for (let pair in pairs) {
        back[pair[0]] = pair[1];
}

printf("%J %J\n", pairs, back);
```

```
[ [ "a", 1 ], [ "b", 2 ] ] { "a": 1, "b": 2 }
```

Merging objects left to right is a spread; merging deeply takes a function, because the spread is shallow at every level:

```
function merge(base, extra) {
        let out = { ...base };

        for (let k in extra) {
                if (type(out[k]) == "object" && type(extra[k]) == "object")
{
                        out[k] = merge(out[k], extra[k]);
                }
                else {
                        out[k] = extra[k];
                }
        }

        return out;
}

let defaults = { a: 1, nested: { x: 1, y: 2 } };

printf("%J\n", merge(defaults, { b: 2, nested: { y: 9 } }));
printf("%J (unchanged)\n", defaults);
```

```
{ "a": 1, "nested": { "x": 1, "y": 9 }, "b": 2 }
{ "a": 1, "nested": { "x": 1, "y": 2 } } (unchanged)
```

A max-by-key, which is what `sort()` plus indexing gives you in one line, and a total, which is what a loop gives you in the absence of `reduce`:

```
let st = [ { name: "eth0", bytes: 120 }, { name: "eth1", bytes: 900 } ];
let best = sort(slice(st), (x, y) => y.bytes - x.bytes)[0];
let total = 0;

for (let s in st) {
        total += s.bytes;
}

printf("%s %d\n", best.name, total);
```

```
eth1 1020
```

Sparse arrays — those with a hole where no value was ever stored — are read as `null` at the gap, so `map()` and `filter()` see `null` there and `length()` counts the hole:

```
let a = [1];
a[3] = 4;

printf("%d %J %J\n", length(a), map(a, (v) => v == null ? "-" : v),
slice(a));
```

```
4 [ 1, "-", "-", 4 ] [ 1, null, null, 4 ]
```

Time

Seven functions cover time in ucode: `time()`, `clock()`, `sleep()`, `localtime()`, `gmtime()`, `timegm()` and `timelocal()`. There is no date type, no duration type and no formatting function — the representation of a moment is an integer count of seconds, and the representation of a broken-down moment is a plain object. The convenience of working with time in ucode is therefore exactly the convenience of working with objects and `sprintf()`.

The full reference for the core builtins is generated from the source and published at ucode-lang.org: *the core functions reference*.

Reading the clock

`time()` returns seconds since the epoch:

```
print(type(time()), " ", time() > 1700000000, "\n");
```

```
int true
```

`clock()` returns a two-element array of seconds and nanoseconds, which is more precise than `time()` and is the function to reach for when measuring how long something took. Its argument selects the clock: falsy (or omitted) gives the realtime clock, which jumps when the system clock is adjusted, and truthy gives the monotonic clock, which does not:

```
let start = clock(true);
let sum = 0;

for (let i = 0; i < 100000; i++) {
        sum += i;
}

let stop = clock(true);

printf("sum=%d tuple=%s went-backwards=%s\n", sum, type(stop), stop[0] <
start[0]);
```

```
sum=4999950000 tuple=array went-backwards=false
```

The two elements are seconds and nanoseconds; nanoseconds make the difference between two readings precise without any floating point, which is what makes the tuple worth its awkwardness.

The seconds-since-boot figure is only as meaningful as the platform's monotonic clock — on a device that has not resynced since power-up, `clock()` without an argument reads 1970. The difference between two monotonic readings is always valid, which is the point.

sleep() counts milliseconds

This is the single most common source of bugs in small ucode scripts. The argument to `sleep()` is a count of **milliseconds**, converted with `ucv_to_integer()`, so a fractional value meant as seconds is truncated to a fraction of a millisecond and rounds down to no delay at all:

```
printf("%s %s\n", sleep(0.01), sleep(1));
```

```
false true
```

The return value reports whether the call did anything: `false` when the argument was invalid or not greater than zero, `true` after an actual delay. Note that `sleep(1)` returns `true` — and sleeps for one millisecond, not one second. Two seconds is `sleep(2000)`.

The implementation is a `select()` with no file descriptors, so a signal can cut the delay short and the return value will still be `true`. Anything waiting on an interval in a long-running script should re-read the clock rather than trust that the full delay elapsed.

Broken-down time

`localtime(secs)` and `gmtime(secs)` return an object with nine fields:

```
printf("%J\n", localtime(0));
```

```
{ "sec": 0, "min": 0, "hour": 1, "mday": 1, "mon": 1, "year": 1970, "wday":
4, "yday": 1, "isdst": 0 }
```

The field names are deliberately familiar, and four of their values are deliberately not what `struct tm` would give, because the C conventions are a constant source of off-by-one errors:

Field	Meaning	Deviation from `struct tm`
`sec`, `min`	0–59, 0–59	—
`hour`	0–23	—
`mday`	day of month	1-based, as in C
`mon`	month	**1–12**, not 0–11
`year`	year	**four digits**, not years since 1900
`wday`	day of week	**1 = Monday … 7 = Sunday**, not 0 = Sunday
`yday`	day of year	**1–366**, not 0–365
`isdst`	daylight saving in effect	integer `0` or `1`, not a boolean

The day-of-week numbering is verifiable at either end of a week — 1970-01-04 was a Sunday and 1970-01-05 a Monday:

```
printf("Sun=%d Mon=%d Thu=%d\n", localtime(3 * 86400).wday,
       localtime(4 * 86400).wday, localtime(0).wday);
```

```
Sun=7 Mon=1 Thu=4
```

`localtime()` obeys the system timezone setting, so the same epoch value can render an hour apart across a daylight-saving boundary.

Going back to seconds

`timegm()` interprets a broken-down object as UTC; `timelocal()` interprets it as local time. Matching pairs round-trip exactly:

```
let t = 1000000000;
printf("%d %d\n", timegm(gmtime(t)), timelocal(localtime(t)));
```

```
1000000000 1000000000
```

Mixing them shifts the result by the UTC offset — and the shift is not always the offset you expect, because the `isdst` field travels with the object and `timelocal()` believes it:

```
let t = 1000000000;
printf("%d %d\n", timelocal(gmtime(t)), timegm(localtime(t)));
```

```
999996400 1000007200
```

The first value is three hours and twenty minutes off from the second, not a clean hour: `gmtime()` produced an object with `isdst: 0`, `timelocal()` took that literally and used the standard-time offset, while the instant itself fell inside daylight saving. The rule worth memorising is the simple one — build the object with `gmtime()`, hand it to `timegm()`; build it with `localtime()`, hand it to `timelocal()`. When constructing an object by hand, set `isdst` explicitly, since `timelocal()` will otherwise assume standard time.

Formatting, or the lack of it

There is no `strftime()` — the name is simply not defined — so a timestamp is assembled from the broken-down fields with `sprintf()`:

```
function iso8601(ts) {
        let t = gmtime(ts);

        return sprintf("%04d-%02d-%02dT%02d:%02d:%02dZ",
                       t.year, t.mon, t.mday, t.hour, t.min, t.sec);
}

printf("%s\n", iso8601(1000000000));
```

```
2001-09-09T01:46:40Z
```

The padding is worth writing out by hand rather than abbreviating, since a mon or hour of `9` would otherwise produce `2001-9-9`, which sorts before `2001-10-01` in a log file. Because the pieces are just integers in a plain object, the same object can be serialised straight to JSON for a log record — one of the few places where the absence of a date type is an advantage rather than an inconvenience.

math

The `math` module is a flat namespace of about forty functions plus fifteen named constants: the trigonometric, exponential and rounding operations of the C math library, and a handful of sign and comparison helpers. It adds no types and it never raises. Two of its conventions are worth noting at the outset: the argument order of `clamp()`, and the fact that a name the module does not export reads back as `null` rather than raising.

The full reference — every function, constant and resource method — is generated from the module source and published at ucode-lang.org: *the math module*.

```
import * as math from "math";
```

Constants

The module exports fifteen constants, all doubles, with the values of the corresponding `M_*` macros of the C library:

Constant	Value
`PI`	3.141592653589793
`PI_2`, `PI_4`	pi/2, pi/4
`E`	2.718281828459045
`SQRT2`, `SQRT1_2`	sqrt(2), 1/sqrt(2)
`LN2`, `LN10`	natural logs of 2 and 10
`LOG2E`, `LOG10E`	base-2 and base-10 log of e
`LOG2_10`, `LOG10_2`	log base 2 of 10, log base 10 of 2
`INV_PI`, `INV_2PI`, `INV_SQRT2PI`	1/pi, 1/(2 pi), 2/sqrt(pi)

```
import * as math from "math";

printf("%.6f %.6f %.6f\n", math.PI, math.E, math.SQRT2);
printf("%s\n", type(math.PI));
```

```
3.141593 2.718282 1.414214
double
```

The names follow the C macros loosely rather than any one convention: halves are `PI_2` and `PI_4`, reciprocals are prefixed `INV_` — `INV_PI` is 1/pi, not 1/PI — and `INV_SQRT2PI` is 2/sqrt(pi) rather than 1/sqrt(2 pi). There is no `TAU` and no `MAX_INT` — and moreover no `min` or `max` either — those two are core functions rather than module members, as described below:

```
import * as math from "math";

printf("%s %s\n", math.PI, type(math.PI));
printf("%J %J\n", math.TAU, math.MAX_INT);
```

```
3.1415926535898 double
null null
```

The constants arrive with the module as properties of its namespace, so a script that wants a particular value at a precision the module's double does not give is free to define its own:

```
const PI = 3.141592653589793;

printf("%.15f\n", PI);
```

```
3.141592653589793
```

```
import * as math from "math";

printf("%.6f %.6f\n", math.deg2rad(180), math.rad2deg(1));
```

```
3.141593 57.295780
```

For minimum and maximum, the module offers `fmin()` and `fmax()`, each taking two numeric arguments. The variadic forms are in the core language and need no import at all:

```
printf("%J %J %J\n", min(3, 1, 2), max(3, 1, 2), max(1, 9, -4, 7));
printf("%J %J\n", min(2.5, 2), max("a", "b", "aa"));
```

```
1 3 9
2 "b"
```

`min()` and `max()` take any number of arguments of any types and return the element the comparison picks, unchanged in type and flavor: `min(2.5, 2)` is the integer `2`, and `max("a", "b", "aa")` is a string, ordered by byte comparison. Two details are worth remembering. With no arguments at all they return `null`, and a `null` argument outranks every value, so `min(null, 1)` is `null` rather than `1` — a list built by appending conditionally can therefore poison a result that looks numeric. Passing a single array argument returns that array, not its smallest element; the elements have to be spread into the call, which means the loop form is usually the clearer one:

```
let vals = [4, 7, 2, 9];
let best = vals[0];

for (let v in vals) {
        best = min(best, v);
}

printf("min=%J count=%d\n", best, length(vals));
```

```
min=2 count=4
```

`fmin()` and `fmax()` are the numeric fast paths. Each takes exactly two arguments, coerces them to double, and always returns a double — including for arguments that are not numbers, where coercion follows the usual module rules of `null` as zero and an unparseable string as `NaN`:

```
import * as math from "math";

printf("%J %J %J\n", math.fmin(3, 1), math.fmax(3, 1), math.fmin(2.5, 2));
printf("%J %J\n", math.fmin(null, 1), math.fmin("a", 1));
```

```
1.0 3.0 2.0
0.0 "NaN"
```

`NaN` has no JSON spelling, which is why `%J` emits it quoted — `%f` renders it as `nan` and `sprintf("%s", ...)` as `NaN`.

So the choice between the two pairs is: core `min()` / `max()` for any number of arguments of mixed type, returned unchanged; `math.fmin()` / `math.fmax()` for exactly two numbers when a double result is wanted and the coercion of chapter 4 is acceptable.

Integer and floating-point division

Arithmetic operators are in the language, not the module, but this is where their behaviour matters most. Integer division truncates **toward zero**, so it is not floor division for negative operands:

```
printf("%s %s %s\n", 7 / 2, -7 / 2, 7.0 / 2);
```

```
3 -3 3.5
```

Division by zero does not raise, and follows IEEE-754: the sign the division rules give it survives, and zero over zero is `NaN`:

```
printf("%s %s %s\n", 1 / 0, -1 / 0, 0.0 / 0);
```

```
Infinity -Infinity NaN
```

`-1 / 0` is `-Infinity`, and `-1.0 / 0` and `1 / -0.0` agree with it. `0.0 / 0` is IEEE-indeterminate and is `NaN`, as in C, so `isnan()` reports `true` for it. The operators chapter treats the division rules in full.

Rounding

```
import * as math from "math";

printf("%s %s %s %s\n", math.floor(2.7), math.ceil(2.1), math.trunc(-2.7),
math.floor(-2.7));
printf("%s %s\n", math.round(2.5), math.round(3.5));
```

```
2 3 -2 -3
3 4
```

`round()` rounds halves **away from zero** rather than to nearest even, so `2.5` becomes `3` and `3.5` becomes `4`. `trunc()` chops toward zero while `floor()` goes to the next lower integer — the two differ on every negative non-integer.

Types are preserved where they can be

`abs()` keeps the flavour of its argument: an integer in, an integer out; a double in, a double out.

```
import * as math from "math";

printf("%s %s\n", math.abs(-5), type(math.abs(-5)));
printf("%s %s\n", math.abs(-5.5), type(math.abs(-5.5)));
```

```
5 int
5.5 double
```

The one place this goes somewhere unexpected is the most negative integer, whose absolute value does not fit in the signed range. It does not wrap around — the result comes back as an unsigned integer:

```
import * as math from "math";

printf("%s\n", math.abs(-9223372036854775807 - 1));
```

```
9223372036854775808
```

Format such a value with `%u`, as chapter 21 discusses, and remember it cannot be compared as a signed quantity.

clamp() takes the maximum first

`clamp(x, max, min)` — the upper bound is the second argument and the lower bound the third. The documented example makes the convention plain, if unidiomatic:

```
import * as math from "math";

printf("%s %s %s\n", math.clamp(190, 200, 180), math.clamp(1000, 200, 180),
       math.clamp(-1000, 200, 180));
```

```
190 200 180
```

Call it the way `clamp(x, min, max)` would suggest and the function still returns a plausible number — it always returns the second argument whenever the value is anywhere outside the reversed interval, so a limit written the wrong way round pins everything to the minimum:

```
import * as math from "math";

printf("%s %s\n", math.clamp(50, 0, 100), math.clamp(75, 0, 100));
```

```
0 0
```

Both readings of `clamp(v, 0, 100)` should bracket the input between 0 and 100; instead, every value collapses to `0`, because `0` is read as the upper bound. The arguments are documented as `max` before `min`, the order used by scaling code that names the full-deflection limit first. Bounds passed in the opposite order do not raise: the second argument is taken as the maximum, so the result is that value whenever it is below the third.

Sign functions

```
import * as math from "math";

printf("%s %s %s %s\n", math.sign(-7), math.sign(0), math.sign(3),
math.copysign(3, -1));
printf("%s %s %s\n", math.signbit(-0.0), math.signbit(0.0),
math.signnz(-0.0));
```

```
-1 0 1 -3
-1 0 1
```

`signbit()` returns `-1` or `0`, not a boolean, so it composes arithmetically but should not be tested with `if (math.signbit(x))` on the assumption that `0` means positive and anything else means negative — which happens to work here, though only by luck of the encoding. `signnz()` ("non-zero") reports `1` for a negative zero rather than `0`, which is the whole reason it exists alongside `signbit()`.

Random numbers

`rand()` returns an integer in the range `[0, 2^31)`. It is seeded automatically on first use, so successive program runs give different sequences; `srand(seed)` pins the sequence for reproducibility:

```
import * as math from "math";

math.srand(42);
let a = [math.rand(), math.rand()];

math.srand(42);
let b = [math.rand(), math.rand()];

printf("%s\n", a[0] == b[0] && a[1] == b[1]);
```

```
true
```

The module records whether `srand()` has been called in the VM registry under the key `math.srand_called`, which is what lets the first `rand()` seed itself lazily and a later explicit seeding still take effect. To get a floating-point value in `[0, 1)`, divide by `2147483648.0` — writing `2147483647` gives a range that includes `1.0`, and dividing by the integer `2147483647` instead keeps the expression in integer arithmetic and yields `0`.

No errors, only conversions

Every function coerces its arguments to a number rather than rejecting bad input. `null` becomes `0`, and an operation with no mathematical answer yields `NaN` or an infinity:

```
import * as math from "math";

printf("%s %s %s %s\n", math.sqrt(null), math.abs(null), math.sqrt(-1),
math.log(0));
```

```
0 NaN NaN -Infinity
```

Note the asymmetry between `sqrt(null)` and `abs(null)`: the same `null` becomes `0` in one and `NaN` in the other, because the underlying conversion differs between code paths. Neither raises, so a mistaken argument surfaces as a `NaN` several operations downstream — `isnan()` at the point of use is the only reliable guard.

The accuracy helpers are the reason to prefer this module over hand-rolled arithmetic: `log1p(x)` and `expm1(x)` stay precise for tiny `x` where `log(1+x)` and `exp(x)-1` lose every significant digit, and `hypot()` and `cbrt()` avoid the intermediate overflow a naive `sqrt(x*x + y*y)` or `x ** (1/3)` runs into.

```
import * as math from "math";

printf("%.6f %.6f %.1f %.1f\n", math.expm1(1), math.log1p(1),
math.log2(1024), math.hypot(3, 4));
```

```
1.718282 0.693147 10.0 5.0
```

fs

The full reference — every function, constant and resource method — is generated from the module source and published at ucode-lang.org: *the filesystem module*.

The `fs` module is the filesystem: paths, metadata, whole-file reads, handles, directories, and the small set of process-spawning helpers. It is a compiled module, so a program has to name it before using it, and its functions are reachable through the module object:

```
import * as fs from "fs";

print(fs.basename("/etc/hostname"), " ", fs.dirname("/etc/hostname"), "\n");
```

```
hostname /etc
```

Like any standard library module, `fs` exports no default, so `import fs from "fs"` fails with `Module does not export default`; `require("fs")` works and gives the same object. Because the functions are members of the module namespace and not globals, `stat()` alone is a `Type error: left-hand side is not a function` — write `fs.stat()`.

Errors: null plus fs.error()

Almost every `fs` function reports failure the same way: it returns `null` and stores the `errno` value in the VM-registry key `fs.last_error`, which `fs.error()` retrieves and clears (`lib/fs.c:119-122`):

```
import * as fs from "fs";

let ok = fs.access("/nope", "f");
let err = fs.error();

printf("access-is-null=%s errno=%J\n", ok === null, err);
printf("second read=%s\n", fs.error());
```

```
access-is-null=true errno="No such file or directory"
second read=(null)
```

Two consequences. A `null` return tells you that something failed but not what, so if the reason matters, ask `fs.error()` immediately — a later successful call has not touched it, but a later failing one has. And `fs.error()` clears, so a second read returns nothing, as above; keep the value if you need it twice.

```
import * as fs from "fs";

let f = fs.open("/definitely-not-here", "r");

printf("handle=%s errno=%s\n", f, fs.error());
```

```
handle=(null) errno=No such file or directory
```

The practical rule is that `errno` belongs to the call that set it: retrieve it immediately after the call you care about, and keep the value if you need it more than once.

`fs.access()` is the exception worth memorising. It takes an optional mode string built from the letters `r`, `w`, `x`, `f` (mapped to `R_OK`, `W_OK`, `X_OK`, `F_OK`), defaults to `F_OK` — existence only — requires *all* the requested modes to hold, and returns either `true` or `null`. It never returns `false`, and any character outside `rwx f` is an `EINVAL` failure:

```
import * as fs from "fs";

printf("etc=%s missing-is-null=%s bad-mode-is-null=%s\n", fs.access("/etc",
"rx"),
       fs.access("/nope") === null, fs.access("/nope", "d") === null);
```

```
etc=true missing-is-null=true bad-mode-is-null=true
```

Note the third call: `"d"` is not a mode letter, so that is a bad argument rather than a "does not exist" answer — which is exactly how a mistyped mode tends to look in practice. There is no `fs.exists()`; `fs.access(path)` with no mode is the existence test.

Whole files

`fs.readfile(path)` returns the contents as a string and `fs.writefile(path, data)` writes and returns the number of bytes written:

```
import * as fs from "fs";

let path = "/tmp/ucode-manual-demo.txt";

printf("wrote=%d read=%J\n", fs.writefile(path, "hello\n"),
fs.readfile(path));

fs.unlink(path);
```

```
wrote=6 read="hello\n"
```

This pair covers most scripting needs and avoids handle bookkeeping entirely. Both return `null` on failure, so a failed write is distinguishable from a zero-byte write only via `fs.error()`.

Handles

`fs.open(path, mode)` returns a resource of type `resource` — `type()` reports `resource`, not `object` — or `null`:

```
import * as fs from "fs";

let f = fs.open("/etc/hostname", "r");

printf("type=%s tty=%s fd>2=%s\n", type(f), f.isatty(), f.fileno() > 2);
f.close();
```

```
type=resource tty=false fd>2=true
```

Methods are called on the handle itself, in `stdio` style:

Method	Notes
`read(length)`	reads up to `length` bytes, returns a string
`write(data)`	returns the byte count written
`seek(offset, position)`	`position` selects the origin; returns `true`
`tell()`	current offset
`flush()` / `close()`	return `true`
`fileno()`	the underlying descriptor
`isatty()`	boolean
`truncate(offset)`	boolean
`lock(op)`	advisory locking
`error()`	per-handle error retrieval
`ioctl(direction, type, num, value)`	the raw ioctl escape hatch

A handle is closed when it is garbage-collected, and the standard descriptors are deliberately exempt — the closer skips file numbers 2 and below, so a collection can never take your stdout away (`lib/fs.c`, `close_file()`). `fs.fdopen(fd, mode)` wraps an existing descriptor, and `fs.pipe()` gives you a reader/writer pair.

Directories

`fs.opendir(path)` returns a directory handle whose `read()` yields one entry name per call, ending with `null`. `.` and `..` are returned like any other entry, so filtering them out is the caller's job; `tell()` and `seek(offset)` let you restart the walk:

```
import * as fs from "fs";

let d = fs.opendir("/usr/bin/nothing");

printf("opendir missing=%s errno=%s\n", d, fs.error());
```

```
opendir missing=(null) errno=No such file or directory
```

Here `fs.error()` is meaningful, because `opendir()` does record its `errno`.

For the common case there is `fs.lsdir(path)`, which returns the names as an array and spares you the handle.

stat

`fs.stat(path)` follows symlinks and `fs.lstat(path)` does not. The result is an object with these keys, in this order:

```
import * as fs from "fs";

printf("%J\n", keys(fs.stat("/etc")));
```

```
[ "dev", "perm", "inode", "mode", "nlink", "uid", "gid", "size", "blksize",
"blocks", "atime", "mtime", "ctime", "type" ]
```

`dev` is itself an object `{ major, minor }`, `type` is a short string naming the file kind, and `perm` is not an octal number but twelve booleans, which makes conditions readable at the cost of having to know the names:

```
import * as fs from "fs";

printf("%J\n", keys(fs.stat("/etc").perm));
```

```
[ "setuid", "setgid", "sticky", "user_read", "user_write", "user_exec",
"group_read", "group_write", "group_exec", "other_read", "other_write",
"other_exec" ]
```

So `fs.stat(p).perm.user_write` rather than the `mode & 0200` you would write in C, and `fs.stat(p).type` rather than a `S_ISDIR` test. The times are seconds, matching the `st_atime` / `st_mtime` / `st_ctime` of the underlying `struct stat`.

`fs.statvfs(path)` reports the filesystem, with the raw `statvfs` fields plus two conveniences, `freesize` and `totalsize`, computed from block size and block counts:

```
import * as fs from "fs";

printf("%J\n", keys(fs.statvfs("/")));
```

```
[ "bsize", "frsize", "blocks", "bfree", "bavail", "files", "ffree", "favail",
"fsid", "flag", "namemax", "freesize", "totalsize", "type" ]
```

`flag` is a number; the `ST_*` constants exported by the module are the bits to test it against (read-only, no-dev, no-suid and the rest), and the ioctl direction constants are exported as `IOC_*`.

Creating, removing, changing

`fs.mkdir(path, [mode])`, `fs.rmdir(path)`, `fs.symlink(target, path)`, `fs.unlink(path)`, `fs.rename(from, to)`, `fs.chmod(path, mode)`, `fs.chown(path, uid, gid)` and `fs.readlink(path)` all map one-to-one onto their system calls and return `null` on failure. `fs.mkstemp(template)` and `fs.mkdtemp(template)` take a trailing-X template and return the created name:

```
import * as fs from "fs";

let d = fs.mkdtemp("/tmp/ucode-XXXXXX");

printf("type=%s accessible=%s\n", type(d), fs.access(d, "x"));

fs.rmdir(d);
```

```
type=string accessible=true
```

`fs.chdir(path)` and `fs.getcwd()` change and report the working directory, and `fs.realpath(path)` resolves a path to its canonical absolute form:

```
import * as fs from "fs";

printf("%s %s\n", fs.realpath("/etc"), type(fs.getcwd()));
```

```
/etc string
```

`fs.glob(pattern)` returns an array of matches, and — worth stating, since it differs from the pattern in this book elsewhere — an empty array is not an error:

```
import * as fs from "fs";

printf("%J\n", fs.glob("/usr/bin/nothing*"));
```

```
[ ]
```

Processes

`fs.popen(command, [mode])` runs a command and returns a handle you can `read()` from or `write()` to, so command output becomes a stream rather than a temporary file:

```
import * as fs from "fs";

let p = fs.popen("echo ucode");

printf("%s %J", type(p), p.read(6));

p.close();
```

```
resource "ucode\n"
```

The `fs.proc` resource has six prototype methods covering reading, writing, closing, waiting for exit status, accessing the descriptor, and reading the error. `fs.pipe()` is the unidirectional sibling, returning reader and writer handles for in-process use. For anything needing more control over the child — arguments as a list rather than a shell string, environment, exit callbacks — use the `uloop` module instead.

Notes

- `fs` functions take paths as bytes; there is no encoding layer, so a filename with non-UTF-8 bytes round-trips unchanged.
- `fs.open` modes are the `fopen` letters (`r` , `w` , `a` , `r+` , `w+` , `a+`) plus the exported `O_*` flags where the C module provides them.
- Handles are resources: printing them with `%s` shows the resource name, `%J` renders them as far as it can, and neither is meaningful — use `fileno()` when you need to talk about the descriptor itself.
- The module is Linux-centric: `statfs` , several `ST_*` bits and `ioctl` behaviour are compiled conditionally (`HAVE_IOCTL` , `HAS_MAC_IOCTL` , `__linux__`), so a script that leans on them is portable only within Linux.

io

`io` is the raw POSIX layer under the filesystem: unbuffered file descriptors, `fcntl` and `ioctl`, terminal attributes, and pseudo-terminals. `fs` (chapter 25) sits on top of the same primitives with buffered, whole-file helpers, and the two are not interchangeable.

The full reference — every function, constant and resource method — is generated from the module source and published at ucode-lang.org: *the io module*.

```
import * as io from "io";

printf("%J\n", keys(io));
```

```
[ "pipe", "from", "open", "new", "error", "O_RDONLY", "O_WRONLY", "O_RDWR",
"O_CREAT", "O_EXCL", "O_TRUNC", "O_APPEND", "O_NONBLOCK", "O_NOCTTY",
"O_SYNC", "O_CLOEXEC", "O_DIRECTORY", "O_NOFOLLOW", "SEEK_SET", "SEEK_CUR",
"SEEK_END", "F_DUPFD", "F_DUPFD_CLOEXEC", "F_GETFD", "F_SETFD", "F_GETFL",
"F_SETFL", "F_GETLK", "F_SETLK", "F_SETLKW", "F_GETOWN", "F_SETOWN",
"FD_CLOEXEC", "TCSANOW", "TCSADRAIN", "TCSAFLUSH", "IOC_DIR_NONE",
"IOC_DIR_READ", "IOC_DIR_WRITE", "IOC_DIR_RW" ]
```

Six functions, and the rest of the module is constants: the `O_*` open flags, the `SEEK_*` origins, the `F_*` `fcntl` commands, the `TCSA*` `tcsetattr` action selectors and the `IOC_DIR_*` ioctl directions. There is no `io.stdout`, `io.stdin` or `io.stderr` member — those belong to the interpreter's own streams and to `fs`.

Opening files: numeric flags, not mode letters

`io.open(path, flags, [mode])` takes the numeric `O_*` flags of `open(2)`, where `fs.open()` takes `fopen` mode letters. Passing a string mode where flags are expected does not raise, it just gives you `null`:

```
import * as io from "io";

let path = "/tmp/ucode-io-demo.txt";
let f = io.open(path, io.O_WRONLY | io.O_CREAT | io.O_TRUNC, 420);

printf("type=%s wrote=%d\n", type(f), f.write("l1\nl2\n"));
f.close();
io.open(path, io.O_RDONLY).close();
```

```
type=resource wrote=6
```

The third argument is the creation mode in octal — `420` is `0644`. Like `fs`, failures return `null` and record the error for `io.error()`, which retrieves and clears it:

```
import * as io from "io";

let f = io.open("/definitely-not-here", io.O_RDONLY);

printf("handle=%s error=%s\n", f, io.error());
```

```
handle=(null) error=No such file or directory
```

Reading: the length argument is not optional

The length argument separates the two forms. `read()` with no argument does not mean "read everything" — it returns `null`:

```
import * as io from "io";

let f = io.open("/tmp/ucode-io-demo.txt", io.O_RDONLY);

printf("read()=%J\n", f.read());
printf("read(3)=%J tell=%d\n", f.read(3), f.tell());
f.close();
```

```
read()=null
read(3)="l1\n" tell=3
```

Because nothing was consumed by the failed `read()`, the offset stayed at 0 and the following `read(3)` returned the first three bytes. Give `read()` an explicit byte count, and end your loop on the **empty string**, which is what end-of-file looks like in this module. `null` never means EOF — it means the length argument was missing:

```
import * as io from "io";

let f = io.open("/tmp/ucode-io-demo.txt", io.O_RDONLY);

let chunk = f.read(4);
let next = f.read(4);
let past = f.read(4);
let noarg = f.read();

printf("chunk=%J next=%J past=%J no-arg=%J\n", chunk, next, past, noarg);
printf("past-is-empty=%s past-is-null=%s\n", past === "", past === null);
f.close();
```

```
chunk="l1\nl" next="2\n" past="" no-arg=null
past-is-empty=true past-is-null=false
```

The idiom for breaking on either an error or end of file is to test the read's return value with `length()` — `length()` answers `null` for both a `null` read and an empty chunk, and `null` is falsy, so the same check covers both:

```
import * as io from "io";

let res = io.open("/etc/hostname", io.O_RDONLY);
let total = 0;

for (let s = res.read(512); length(s); s = res.read(512)) {
    total += length(s);
}

printf("read %d bytes\n", total);
res.close();
```

```
read 3 bytes
```

Testing for `null` instead of an empty chunk is the classic way to write a read loop that never terminates, and it is easy to do accidentally coming from `fs`, whose buffered reads behave differently.

Handles

An `io` handle is a resource with eighteen methods:

```
import * as io from "io";

let f = io.open("/tmp/ucode-io-demo.txt", io.O_RDONLY);

printf("%J\n", keys(proto(f)));
f.close();
```

```
[ "unlockpt", "grantpt", "tcsetattr", "tcgetattr", "ptsname", "error",
"close", "isatty", "ioctl", "fcntl", "fileno", "dup2", "dup", "tell", "seek",
"write", "read" ]
```

Reading and writing are `read(length)` and `write(data)`, the latter returning the byte count. Position is `tell()` and `seek(offset, origin)` with `io.SEEK_SET`, `SEEK_CUR` or `SEEK_END`. Beyond that sits the descriptor layer — `fileno()`, `dup()`, `dup2()`, `fcntl()`, `ioctl()`, `isatty()` — and the terminal and pty group: `tcgetattr()` and `tcsetattr()` for line-discipline and echo control, and `grantpt()`, `unlockpt()` and `ptsname()` for pseudo-terminals, which is how a ucode program drives another program the way `script` does:

```
import * as io from "io";

let f = io.open("/tmp/ucode-io-demo.txt", io.O_RDONLY);

printf("isatty=%s fileno>2=%s\n", f.isatty(), f.fileno() > 2);
f.close();
```

```
isatty=false fileno>2=true
```

A plain file is not a terminal, so `isatty()` is `false` and the `tcgetattr()` call you may be tempted to try first would fail — check it first.

Pipes

`io.pipe()` returns a two-element array of handles, reader first. ucode has no destructuring assignment, so you index it:

```
import * as io from "io";

let pair = io.pipe();
let reader = pair[0];
let writer = pair[1];

writer.write("ping");
writer.close();

printf("got=%J\n", reader.read(4));
reader.close();
```

```
got="ping"
```

`read(length)` returns as soon as *any* data is available; it never waits to fill `length`. It blocks only when the pipe is empty — and on an empty pipe with a writer still open, that block is the whole interpreter waiting with nothing to wake it, because a plain script has no scheduler and no other thread:

```
import * as io from "io";

let pair = io.pipe();
let reader = pair[0];
let writer = pair[1];

writer.write("abc");
writer.close();

printf("first=%J second=%J\n", reader.read(100), reader.read(100));
```

```
first="abc" second=""
```

The 100 requested bytes never arrive: the first read hands back the three that were available, the second gets the empty string — end-of-file, reported only because the writer was closed. Hold that writer open and the second read never returns at all, so either close every writer before reading to EOF, or set `io.O_NONBLOCK` on the reader.

Converting anything into a handle

`io.from(value)` adapts another value to an `io` handle. It accepts an integer file descriptor number, an `fs.file`, `fs.proc` or `socket` resource, or any object, array or resource carrying a `fileno()` method (`lib/io.c`, `uc_io_from()`):

```
import * as io from "io";

printf("fd1=%s string=%s\n", type(io.from(1)), io.from("not a descriptor"));
```

```
fd1=resource string=(null)
```

`io.from(1)` is the way to get at standard output through this module — the interpreter does not expose it as a member, but descriptor 1 is standard output, so `io.from(1).write("now\n")` writes it unbuffered and unformatted, bypassing `print()`. Strings are not accepted: `io.from` converts descriptors, it does not wrap text in a memory stream, and the `null` return is the only signal you get.

`io.new(fd)` is the sibling constructor for a descriptor you already own.

Which module to reach for

Use `fs` for ordinary file work: mode letters, buffered reads, `readfile`/`writefile`, `stat`, directory walking, `popen`. Use `io` when you need the descriptor itself — non-blocking I/O, `ioctl`, terminal settings, ptys, `dup2` onto a child's stdio — or when you need a handle on something that is not a file at all, such as a socket or a pipe. The two interoperate in one direction only in practice: `io.from()` takes an `fs` handle and gives you a raw one, so the richer terminal and descriptor API can be applied to anything `fs` opened.

Both modules report failure the same way — `null` plus a retrievable `errno` string — and both clear it on retrieval. Neither is buffered, which means output ordering between `print()` (buffered stdout) and `io.from(1).write()` (unbuffered) is not guaranteed; if you mix them and the interleaving matters, flush `fs`/stdout between writes.

struct and binary data

Network programming, parsing driver ioctl responses, reading a binary TLV frame — all of these need a way to move between ucode values and packed bytes. The `struct` module does that, and it does it in two layers: one-shot `pack()` and `unpack()` functions for the common case, and a stateful buffer object for sequential parsing. The module is named for the C `struct`, and its format strings follow the same conventions. There is no `sizeof` equivalent: sizes come from packing a value and measuring the result.

The full reference — every function, constant and resource method — is generated from the module source and published at ucode-lang.org: *the struct module*.

```
import * as st from "struct";
```

The module surface

Four functions are exported: `pack`, `unpack`, `new` and `buffer`. There is no `size`, so the byte length of a format is obtained by packing a value and measuring the result:

```
import * as st from "struct";

printf("%s\n", type(st.size));
```

```
(null)
```

Ask for a format's byte length by packing a dummy value and measuring the result:

```
import * as st from "struct";

printf("%d %J\n", length(st.pack("I", 0)), st.unpack("I", st.pack("I",
305419896)));
```

```
4 [ 305419896 ]
```

`unpack()` takes an optional third argument, an offset into the input string, so a header can be skipped without copying the buffer.

The format characters have the widths you would expect from C on a 32-bit-or-later platform, and there are signed and unsigned variants of each integer:

Format	Meaning	Bytes
`c`	signed char	1
`b` / `B`	signed / unsigned byte	1
`h` / `H`	signed / unsigned short	2
`i` / `I`	signed / unsigned int	4
`q` / `Q`	signed / unsigned long long	8
`f`	float	4
`d`	double	8
`Ns`	N raw bytes	N
`x`	padding byte	1 each

```
import * as st from "struct";

printf("c=%d b=%d h=%d i=%d q=%d f=%d d=%d\n",
       length(st.pack("c", 1)), length(st.pack("b", 1)), length(st.pack("h",
1)),
       length(st.pack("i", 1)), length(st.pack("q", 1)), length(st.pack("f",
1)),
       length(st.pack("d", 1)));
```

```
c=1 b=1 h=2 i=4 q=8 f=4 d=8
```

`unpack()` always returns an array, one element per format item — even for a single value. That is worth remembering when writing `let x = unpack("I", buf);`, which yields an array, not the integer.

Byte order must be stated

By default, values are packed in the native order of the machine, which on the usual development host is little-endian. Three prefixes override it: `<` for little-endian, `>` for big-endian and `!` for network order, which is big-endian:

```
import * as st from "struct";

printf("%J %J\n", st.unpack("!H", st.pack("!H", 258)), st.unpack(">H",
st.pack(">H", 258)));
```

```
[ 258 ] [ 258 ]
```

A mismatch between packing and unpacking is not an error — it is a silently wrong number:

```
import * as st from "struct";

printf("%J\n", st.unpack("I", st.pack(">I", 258)));
```

```
[ 33619968 ]
```

`33619968` is `258` with its bytes reversed, read as a native-order integer. Any protocol field therefore needs its order spelled out on both sides; stating the order on one side only produces correct values on a host of that order, and reversed ones on a host of the opposite order.

Compiling a format once

`struct.new(format)` returns a compiled format object, which exposes the same `pack` and `unpack` operations:

```
import * as st from "struct";

let f = st.new("!HHI");
let packet = f.pack(1, 2, 3);

printf("%s %J %d\n", type(f), f.unpack(packet), length(packet));
```

```
resource [ 1, 2, 3 ] 8
```

Compiled formats are `resource` values. Use them the same way you would use a compiled regular expression: when a script parses many records with the same layout, building the format once and reusing it avoids re-parsing the format string on every pass. In a loop over a few thousand frames the difference is measurable; in a one-off conversion it is noise, and the one-shot form reads better.

Reading a buffer sequentially

The buffer object is where `struct` becomes practical for parsing. `struct.buffer(data)` wraps a string; `get()` reads one value and advances the cursor; the read stops when the data runs out:

```
import * as st from "struct";

let buf = st.buffer(st.pack("!IIB", 1, 2, 3));

printf("%J %J %J %J\n", buf.get("!I"), buf.get("!I"), buf.get("B"),
buf.get("B"));
```

```
1 2 3 null
```

The fourth read returns `null` rather than raising. That single behaviour makes sequential parsing safe on truncated input — which is the normal condition for data arriving from a socket — and it is the reliable way to detect the end of a buffer. Checking `pos()` against `length()` works too, but requires arithmetic that has to agree with the width of every field read so far; letting `get()` report exhaustion does not.

`get()` also accepts a plain number, which reads that many raw bytes as a string instead of decoding a value:

```
import * as st from "struct";

let buf = st.buffer("hello world");

printf("%d [%s] %d\n", length(buf.get(5)), buf.get(1), buf.length());
```

```
5 [ ] 11
```

The combination of a format-read, a byte-count header read and then `get(count)` is the shape of every type-length-value parser; the example at the end of this chapter uses it.

`read(format)` is the multi-value sibling, and unlike `get()` it is strictly format-based — a number is rejected:

```
import * as st from "struct";

let buf = st.buffer(st.pack("!III", 10, 20, 30));

printf("%J pos=%d\n", buf.read("!III"), buf.pos());
```

```
[ 10, 20, 30 ] pos=12
```

It returns an array of all the decoded values and leaves the cursor after them. Passing a count — `read(4)` — raises `Type error: Format value not a string` ; if raw bytes are what is wanted, `get(n)` is the call.

Slicing, pulling and filling

`slice([from[, to]])` extracts bytes as a string without moving the cursor; with no arguments it returns the whole buffer:

```
import * as st from "struct";

let buf = st.buffer(st.pack("!HH", 1, 2));

printf("%d %d\n", length(buf.slice()), length(buf.slice(2, 4)));
```

```
4 2
```

`pull()` hands over the buffer's entire contents as a string and empties the buffer. It ignores the cursor — a `get()` before it makes no difference to what comes back:

```
import * as st from "struct";

let buf = st.buffer(st.pack("!HH", 1, 2));
buf.get("!H");

printf("%d %d\n", length(buf.pull()), buf.length());
```

```
4 0
```

The implementation reuses the buffer's own storage for the returned string instead of copying it, then clears `data`, `capacity`, `length` and `position`. That makes `pull()` the efficient way to finish: accumulate a message with `put()`, hand the bytes straight to `socket.send()` or `fs.write()`, and let the buffer go. Because the buffer is left empty rather than invalid, it can be reused for the next message immediately.

`set(byte[, from[, to]])` fills a byte range, in the manner of `memset`, and takes a byte value; a string argument contributes its **first character**:

```
import * as st from "struct";

let buf = st.buffer(st.pack("!II", 1, 2));

buf.set("A", 2, 4);
printf("%s\n", buf.slice(2, 4));
```

```
AA
```

Writing `buf.set("H", 4, 9)` does not store the number `9` as a 16-bit value at offset 4 — it fills bytes 4 through 8 with `H`, the first character of the format string, exactly as `buf.set(0x48, 4, 9)` would. Use `put()` to write values and `set()` only to fill bytes.

Chaining and cursors

The methods that modify the buffer return the buffer itself, so calls chain:

```
import * as st from "struct";

let buf = st.buffer();

buf.put("!H", 1).put("!H", 2).set(0, 0, 1);
buf.start();

printf("%J %d %s\n", buf.get("!H"), buf.pos(), buf.get("!H") == 2);
```

```
1 2 true
```

`set(0, 0, 1)` rewrote byte 0 as zero, which is a no-op here because the big-endian encoding of `1` already begins with a zero byte — a useful illustration of why `set()` takes a byte value and not a format: it has no idea what the bytes mean.

The cursor itself is reported by `pos()`. Two methods move it, and both return the buffer so they chain: `start()` rewinds to offset 0, `end()` seeks to the end of the data. Their implementations are four lines each — `buffer->position = 0` and `buffer->position = buffer->length` — and that is the whole of what they do:

```
import * as st from "struct";

let buf = st.buffer(st.pack("!III", 1, 2, 3));

printf("%J %d %d\n", buf.get("!I"), buf.pos(), buf.end().pos());
printf("%J %d\n", buf.start().get("!I"), buf.pos());
```

```
1 4 12
1 4
```

Neither function reads an argument, so `buf.start(4)` rewinds rather than seeking to byte 4. Ignoring arguments a function does not read is the convention throughout the standard library rather than a peculiarity of these two: the interpreter passes an argument count and each function looks at the positions it wants (see the discussion of arity in the functions chapter). `pos(4)` is the call that seeks. After `end()` the cursor is past the last byte, so subsequent `get()` calls return `null` until `start()` rewinds.

The cursor is a single number, and `pos()` is both its accessor and its seek. Called with no argument it reports the current offset; called with one it moves the cursor there and returns the buffer, so it chains:

```
import * as st from "struct";

let buf = st.buffer(st.pack("!III", 1, 2, 3));

buf.pos(4);
printf("%J %d\n", buf.get("!I"), buf.pos());
```

```
2 8
```

A negative offset counts back from the end of the data, which makes reading a trailing field easy without knowing the total length:

```
import * as st from "struct";

let buf = st.buffer(st.pack("!III", 1, 2, 3));

buf.pos(-4);
printf("%J\n", buf.get("!I"));
```

```
3
```

Seeking beyond the end is not an error. The buffer **grows** to cover the new position, and the skipped bytes become zeroes — which is how a message with a reserved region is built:

```
import * as st from "struct";

let buf = st.buffer();

buf.pos(3);
buf.put("B", 65);

printf("%s %d\n", buf.slice(3), buf.length());
```

```
A 4
```

The three bytes skipped over are present in the output as zeroes, so `length()` is 4 rather than 1. If the seek was meant to be a bounds check rather than an allocation, compare against `length()` first — `pos()` will not complain.

With `pos()` for absolute movement, `start()` and `end()` are conveniences for the two ends of the data, and the pair covers most parsing needs: rewind, read fields in order, and let `get()` report exhaustion with `null`.

A complete parser

Putting it together — a small type/length/value parser over a buffer built with `put()`. Nothing here depends on cursor arithmetic; each field is read in order and the loop ends on exhaustion or truncation:

```
import * as st from "struct";

function parse_tlv(data) {
        let buf = st.buffer(data);
        let out = [];

        while (true) {
                let type = buf.get("B");
                if (type == null) {
                        break;
                }

                let len = buf.get("!H");
                if (len == null) {
                        printf("truncated header for type %d\n", type);
                        break;
                }

                push(out, { type: type, value: buf.get(len) });
        }

        return out;
}

let data = st.buffer();
data.put("B", 1).put("!H", 4).put("4s", "abcd");
data.put("B", 7).put("!H", 2).put("2s", "hi");
data.put("B", 9).put("!H", 3).put("3s", "abc");
data.put("B", 11);

let tlvs = parse_tlv(data.slice());
printf("%d entries, pos=%d\n", length(tlvs), data.pos());
printf("%s %s %s\n", tlvs[0].type, tlvs[1].value, tlvs[2].value);
```

```
truncated header for type 11
3 entries, pos=19
1 hi abc
```

The trailing type byte had no length field after it, and the parse noticed and stopped rather than running off the end, while the three complete records were kept. That is the behaviour to aim for when parsing anything that came off a wire: ucode raises no exception here, it returns `null`, and noticing it is the programmer's job.

Note the use of `push(out, ...)` to accumulate results. The `arr[arr.length] = value` pattern familiar from other languages does not work here, because the length of an array is obtained by calling `length(arr)`, not by

reading a property — `arr.length` is `null`, so the assignment goes nowhere useful.

`arr[length(arr)] = value` does work, but `push()` is shorter, clearer, and the idiom to use.

digest, zlib, base64 and hex

Three separate pieces of machinery live under this heading because they are used together: the `digest` module for hashing, the `zlib` module for compression, and the base64 and hexadecimal conversion functions that sit in the core rather than in any module. All three work on strings holding arbitrary bytes; none of them has an encoding layer, and none of them inspects the bytes it is given.

The full references are generated from the module sources and published at ucode-lang.org: *the digest module* and *the zlib module*.

Hashing with digest

The `digest` module provides eight algorithms, each in two forms: one that hashes a string and one that hashes a file.

Algorithm	Function	File variant	Output length
MD5	`md5(s)`	`md5_file(path)`	32 hex chars
SHA-1	`sha1(s)`	`sha1_file(path)`	40 hex chars
SHA-256	`sha256(s)`	`sha256_file(path)`	64 hex chars
SHA-384	`sha384(s)`	`sha384_file(path)`	96 hex chars
SHA-512	`sha512(s)`	`sha512_file(path)`	128 hex chars
MD4	`md4(s)`	`md4_file(path)`	32 hex chars
MD2	`md2(s)`	`md2_file(path)`	32 hex chars
FNV-1a 64	`fnv1a64(s)`	`fnv1a64_file(path)`	16 hex chars

Every function takes exactly one argument and returns the digest as a lowercase hexadecimal string:

```
import * as dg from "digest";

printf("%s\n%s\n%s\n", dg.md5("hello"), dg.sha256("hello"),
dg.fnv1a64("hello"));
```

```
5d41402abc4b2a76b9719d911017c592
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
a430d84680aabd0b
```

These are the standard test vectors for the empty-free string `hello`, and the empty string likewise produces its published value:

```
import * as dg from "digest";

printf("%s\n", dg.md5(""));
```

```
d41d8cd98f00b204e9800998ecf8427e
```

Not every build provides every algorithm. `DIGEST_SUPPORT` controls whether the module is built at all; `DIGEST_SUPPORT_EXTENDED` adds MD2, MD4, SHA-384 and SHA-512, and is commonly left off builds intended for flash-constrained devices. The base set — MD5, SHA-1, SHA-256 and FNV-1a — is always present. A function that was not compiled in is simply absent, so looking it up yields `null` and calling it raises:

```
import * as dg from "digest";

printf("%s\n", type(dg.sha512));
```

```
function
```

`sha512` is there when the build enables the extended digests, the `DIGEST_SUPPORT_EXTENDED` option of chapter 2. On a build without them the same lookup prints `(null)`, which makes a pre-flight check worthwhile in scripts that are deployed across device profiles:

```
import * as dg from "digest";

if (type(dg.sha256) == "function") {
        printf("sha256 available\n");
} else if (type(dg.sha1) == "function") {
        printf("falling back to sha1\n");
} else {
        printf("no usable hash\n");
}
```

```
sha256 available
```

Arguments and failures

The argument must be a string. Numbers and other values are not converted, and no error is raised for them:

```
import * as dg from "digest";

printf("%s %s %s\n", dg.md5(123), dg.md5(null), dg.md5([1]));
```

```
(null) (null) (null)
```

Because `null` is the failure result, a hash of the empty string — which is a valid, distinct value — must be distinguished from a failed call by context rather than by testing the result for truthiness. `null` is false and a hex string is true, so `if (!dg.md5(x))` conflates "the argument was not a string" with "the hash could not be computed", and never with "the input was empty", which succeeds.

The file variants hash the file's contents and return `null` on any failure, including a missing file, an unreadable file or a directory:

```
import * as dg from "digest";
import * as fs from "fs";

let f = fs.open("/tmp/ucode-digest-demo", "w");
f.write("hello");
f.close();

printf("%s\n", dg.md5_file("/tmp/ucode-digest-demo"));
printf("%s\n", dg.md5_file("/nonexistent-xyz"));
```

```
5d41402abc4b2a76b9719d911017c592
(null)
```

The file digest of `hello` is the same value `md5("hello")` returns, which is the check to make when a script mixes the two forms. There is no errno and no error message: the module reports failure only by returning `null`.

What digest does not provide

The module is one-shot. Each call hashes one complete input and returns a complete digest; there is no `create()`, no `update()` and no `final()`, so a hash cannot be computed incrementally as data arrives. For a stream, the choices are to buffer the data and hash it in one call, or to hash the completed file with `md5_file()` or its equivalent.

There is no HMAC and no keyed-hash function of any kind in the module. A message authentication code has to be assembled from the primitives by hand — a prefix-construction `md5(key + message)` is available but is not a secure HMAC — or produced by an external tool. The available surface is deliberately narrow: eight hash functions and their file forms.

The return value is text, not bytes. A digest that must be embedded in a binary protocol needs converting first, which is what the hexadecimal functions below are for.

Compressing with zlib

The `zlib` module wraps the zlib library: two one-shot functions, two stream constructors, and the compression-level and flush constants from `zlib.h`.

```
import * as z from "zlib";

let s = "the quick brown fox jumps over the lazy dog, the quick brown fox.";
let c = z.deflate(s);

printf("raw=%d compressed=%d roundtrip=%s\n", length(s), length(c),
z.inflate(c) == s);
```

```
raw=65 compressed=55 roundtrip=true
```

The second argument to `deflate()` selects the container, not the compression level. With `true` the output carries a gzip header and trailer; the default produces a zlib stream:

```
import * as z from "zlib";

printf("zlib=%d gzip=%d\n", ord(z.deflate("hello"), 0),
ord(z.deflate("hello", true), 0));
```

```
zlib=120 gzip=31
```

The leading byte is `0x78` for a zlib stream and `0x1f` — the first byte of the gzip magic number — for a gzip stream. `inflate()` accepts either without being told which:

```
import * as z from "zlib";

printf("%s\n", z.inflate(z.deflate("auto-detected", true)));
```

```
auto-detected
```

The gzip flag must be a boolean. It is read as a boolean and an integer is rejected, so the common mistake of writing `deflate(data, 1)` is reported rather than silently treated as true:

```
import * as z from "zlib";

z.deflate("data", 1);
```

Small strings can come out larger than they went in, because the header and trailer cost bytes that short inputs cannot earn back. A script that compresses many short values should compare the sizes before shipping the compressed form.

Setting the level

The one-shot `deflate()` has no level argument. Level is settable on a stream, whose constructor takes the gzip flag first and the level second:

```
import * as z from "zlib";

let s = "compress me slowly, compress me slowly, compress me slowly.";
let fast = z.deflater(false, z.Z_BEST_SPEED);

printf("accepted=%s\n", fast.write(s));
```

```
accepted=true
```

`write()` reports acceptance, not output. Reading a compressing stream immediately after writing yields only the two bytes zlib has released so far — the header — and a second read returns `null`:

The exported constants are `Z_NO_COMPRESSION` (0), `Z_BEST_SPEED` (1), `Z_BEST_COMPRESSION` (9) and `Z_DEFAULT_COMPRESSION` (-1), plus the flush levels `Z_NO_FLUSH`, `Z_PARTIAL_FLUSH`, `Z_SYNC_FLUSH`, `Z_FULL_FLUSH` and `Z_FINISH`. They are the values from `zlib.h`, exported for convenience; the flush constants have no effect through this module's interface, since the stream methods do not take a flush argument.

Stream handles

`deflater()` and `inflater()` return `resource` values with three methods: `write`, `read` and `error`.

```
import * as z from "zlib";

let inf = z.inflater();
inf.write(z.deflate("streamed content"));

printf("[%s]\n", inf.read());
```

```
[streamed content]
```

`write()` returns `true` when the data was accepted. `read()` returns the bytes available so far, or `null` when there is nothing left to read. The third method, `error()`, returns a message string describing the state of the last operation; the strings come from the system's error table rather than from zlib's own text, so observed values include `"No data available"` for an empty output buffer and `"Operation not permitted"` after a successful read. They are useful for debugging and should not be tested against:

```
import * as z from "zlib";

let inf = z.inflater();
inf.write("this is not deflate data");

printf("[%s] %s\n", inf.read(), inf.error());
```

```
[] unknown error
```

A decompressing stream consumes a complete compressed member and yields its contents; a stream handed garbage yields an empty string and a status message rather than raising.

A compressing stream accumulates input on `write()` and releases compressed output on `read()`. zlib holds back both the bulk of the data and the final block until the stream is flushed, and this module exposes no `flush()` and takes no flush flag, so the bytes a compressing stream releases are not a complete, decodable member:

```
import * as z from "zlib";

let def = z.deflater();
def.write("hello hello hello");

printf("%s\n", z.inflate(def.read()));
```

A complete compressed member comes from the one-shot `deflate()`. The stream interface suited to scripts is the decompressing one, which consumes members produced elsewhere; for producing output, the one-shot function is the reliable path.

base64 and hex

Four conversion functions are part of the core language rather than of any module, so they need no `import`: `b64enc`, `b64dec`, `hexenc` and `hex`.

```
printf("%s %s\n", b64enc("hi"), b64dec("aGk="));
```

```
aGk= hi
```

`b64dec()` returns `null` when its argument is not valid base64:

```
printf("%s\n", b64dec("###not base64###"));
```

```
(null)
```

`hexenc()` converts bytes to a hexadecimal string. `hex()` does not do the reverse: it converts a string of hex digits into a **number**, as if parsing a C hexadecimal literal, and the `0x` prefix is accepted:

```
printf("%s %s %s\n", hex("ff"), hex("0xff"), hex("12"));
```

```
255 255 18
```

Unparsable and non-string arguments produce `NaN`, and a value too long for a 64-bit integer is clamped rather than wrapped or rejected:

```
printf("%s %s %s\n", hex("zz"), hex(300),
hex("0123456789abcdef0123456789abcdef"));
```

```
NaN NaN 9223372036854775807
```

That clamping matters when converting a digest: `hex()` cannot decode a 32-character MD5 hex string into bytes, because the result does not fit in an integer. There is no built-in hex-to-bytes function, so the conversion is done a pair

of characters at a time — which also shows the two idioms that ucode requires here: `substr()` for slicing strings, since a string does not support indexing, and `chr()` for turning a number into a one-byte string:

```
let h = "414243";
let s = "";

for (let i = 0; i < length(h); i += 2) {
        s += chr(hex(substr(h, i, 2)));
}

printf("[%s]\n", s);
```

```
[ABC]
```

Combinations

The functions compose in the order the data needs, which is usually read from the inside out:

```
import * as dg from "digest";

printf("%s\n", b64enc(dg.md5("hello")));
```

```
NWQ0MTQwMmFiYzRiMmE3NmI5NzE5ZDkxMTAxN2M1OTI=
```

Note what that produced: base64 of the *hexadecimal text* of the digest, because `md5()` returns text. It is 44 characters where the raw digest is 16. To base64 the digest's bytes, convert first — and since `digest` returns hex, the conversion is the pair-at-a-time loop from the previous section:

```
import * as dg from "digest";

function hexdec(h) {
        let s = "";

        for (let i = 0; i < length(h); i += 2) {
                s += chr(hex(substr(h, i, 2)));
        }

        return s;
}

let raw = hexdec(dg.md5("hello"));

printf("%d %s\n", length(raw), b64enc(raw));
```

```
16 XUFAKrxLKna5cZ2REBfFkg==
```

Sixteen bytes, as a 128-bit digest should be, and 24 characters of base64 rather than 44. This is the one place in the chapter where a helper function earns its keep, and the pattern recurs in any script that has to put a hash into a binary packet or a URL.

ffi: calling C without writing C

The `ffi` module calls into shared libraries at runtime. It parses C declarations written as strings, builds call frames with libffi, and converts between ucode values and C types. It exists so that a script on a device can reach a vendor library, an unusual syscall wrapper, or a function no ucode module exposes, without a C toolchain and without a compiled extension.

The full reference — every function, constant and resource method — is generated from the module source and published at ucode-lang.org: *the ffi module*.

```
import * as ffi from "ffi";
```

Declaring and calling

Two functions load libraries; both take the C declarations the script wants to use.

```
import * as ffi from "ffi";

let c = ffi.import("libc.so.6", `
        int getpid(void);
        size_t strlen(const char *s);
        int atoi(const char *s);
`);

printf("pid>0=%s len=%d atoi=%d\n", c.getpid() > 0, c.strlen("hello world"),
c.atoi("42abc"));
```

```
pid>0=true len=11 atoi=42
```

`ffi.import(library, declarations)` resolves every declared symbol in the library at call time and returns an object holding them. The library name is the file name the dynamic loader expects, and the declarations are ordinary C prototypes in a string — a template literal keeps multi-line declarations readable.

The lower-level entry point gives more control: `ffi.dlopen(name[, global[, declarations]])` opens a library and returns a handle, with declarations optionally attached as its third argument:

```
import * as ffi from "ffi";

let lib = ffi.dlopen("libc.so.6", false, "int getpid(void);");

printf("pid>0=%s\n", lib.getpid() > 0);
```

```
pid>0=true
```

Declarations belong to a library object. The module also exports `ffi.C` and `ffi.cdef()` for the familiar pattern of declaring first and calling through a global namespace, but cdef'd symbols are not reachable that way: `cdef("int getpid(void);")` succeeds and `ffi.C.getpid()` then fails to call.

```
import * as ffi from "ffi";

ffi.cdef("int getpid(void);");

printf("%d\n", ffi.C.getpid());
```

Bind declarations to the library that provides them and use `ffi.import()` or `ffi.dlopen()` with a `cdefs` argument.

When a declaration or symbol is wrong

Both kinds of mistake are reported when the declarations are processed, with messages that identify the problem:

```
import * as ffi from "ffi";

ffi.import("libc.so.6", "int getpid(void");
```

```
import * as ffi from "ffi";

ffi.import("libc.so.6", "int uc_no_such_symbol(void);");
```

Both failures are exceptions raised while `import()` runs, and both are catchable like any other error, which means a script can wrap its declarations in `try` / `catch` and recover rather than dying on the first bad prototype. A malformed declaration carries the C parser's own message — `invalid C type: ')' expected near '<eof>'` — and an unresolved symbol carries `unable to resolve symbol 'uc_no_such_symbol' in library`. A missing symbol is fatal to the whole `import` call rather than leaving that one entry unset, so a script that wants to use a symbol that may not exist imports it on its own and catches the failure.

Resolution is per-library, and the C library does not contain everything: `sqrt` lives in libm.

```
import * as ffi from "ffi";

let m = ffi.import("libm.so.6", "double sqrt(double x);");

printf("sqrt=%.6f\n", m.sqrt(2));
```

```
sqrt=1.414214
```

Type names and layout

Four functions answer questions about C types by name: `sizeof`, `alignof`, `offsetof` and `typeof`. Structures can be declared for the purpose.

```
import * as ffi from "ffi";

ffi.cdef("struct point { int x; int y; };");

printf("sizeof=%d offsetof_y=%d alignof=%d\n",
       ffi.sizeof("struct point"), ffi.offsetof("struct point", "y"),
       ffi.alignof("struct point"));
```

```
sizeof=8 offsetof_y=4 alignof=4
```

These are the values the C compiler would use on the same machine, which makes them the right tool for building a byte layout to hand to an `ioctl()` or for decoding a header the kernel writes. `typeof` and `ctype` return `ctype` objects; their printed form is an opaque identifier (`ctype: 9` for `int`), so they are useful for comparisons and for passing a resolved type back into `ffi.cast()`, not for display.

```
import * as ffi from "ffi";

printf("int=%d ptr=%d char=%d array=%d\n",
       ffi.sizeof("int"), ffi.sizeof("void *"), ffi.sizeof("char"),
ffi.sizeof("int[4]"));
```

```
int=4 ptr=8 char=1 array=16
```

Sizes are those of the machine running the script. A script that hard-codes a size copied from another platform's header will disagree with `sizeof` here, and a struct declared in a `cdef` that does not match the C definition the other end expects produces silently wrong data. The declarations are the contract; nothing checks them against the library.

Passing and returning strings

A ucode string is passed directly where a `const char *` is expected, as the `strlen` and `atoi` examples above show. A function returning `char *` yields a CData value, which is a pointer resource rather than a ucode string, so reading it means a conversion:

```
import * as ffi from "ffi";

let c = ffi.import("libc.so.6", "char *getenv(const char *name);");
let p = c.getenv("HOME");

printf("type=%s value=%s\n", type(p), ffi.string(p));
```

```
type=resource value=/home/jow
```

`ffi.string(arg[, len])` copies bytes out of C memory into a ucode string, stopping at the NUL terminator unless a length is given. The pointer stays owned by C: `getenv` returns a pointer into the process environment, and the copy `ffi.string` makes is what the script keeps. Passing the pointer on to other C functions is fine; assuming it stays valid after the C side frees or reallocates is not, and nothing in the module will warn about it.

Buffers the C side writes to

Functions that write into caller-supplied memory need a buffer. The module exports no allocator, so the way to get one is to import `malloc` and `free` and manage the memory by hand. `ffi.fill(dest, len[, value])` sets the bytes, `ffi.copy(dest, src[, len])` moves bytes between C memory and ucode strings or other C memory, and `ffi.string()` reads back what was written.

The full sequence — allocate, clear, call, read, release — is the pattern to copy:

```
import * as ffi from "ffi";

let c = ffi.import("libc.so.6", `
        void *malloc(size_t size);
        void free(void *p);
        int gethostname(char *name, size_t len);
`);

let buf = c.malloc(64);
ffi.fill(buf, 64, 0);

let rc = c.gethostname(buf, 64);
let host = ffi.string(buf);
c.free(buf);

printf("rc=%d host_empty=%s\n", rc, host == "");
```

```
rc=0 host_empty=false
```

Three things are worth noting. The `64` passed to `gethostname` has to match the allocation, because nothing checks it. The string is copied into `host` before `free`, so the order of those two statements matters and reversing them reads freed memory without complaint. And `host_empty` reports only whether the result is empty, since the hostname itself belongs to whatever machine runs the script.

`ffi.cast(type, value)` converts between types — an integer to a pointer, a pointer to an integer, one pointer type to another — and raises a type error when the conversion is not available:

```
import * as ffi from "ffi";

let c = ffi.import("libc.so.6", "char *getenv(const char *name);");

ffi.copy(1, c.getenv("HOME"));
```

That call raises `cannot convert argument #1 from 'number' to 'const void *'`. `ffi.copy` takes a destination C pointer, a source, and an optional length; there is no conversion from a bare integer to a pointer. Where an integer address really is the value at hand, the explicit step is `ffi.cast("void *", addr)`.

errno

`ffi.errno()` is a function, not a variable, and returns the C library's current `errno` value.

```
import * as ffi from "ffi";

let c = ffi.import("libc.so.6", "int access(const char *path, int mode);");

let rc = c.access("/nonexistent-xyz", 0);
printf("rc=%d errno=%d\n", rc, ffi.errno());
```

```
rc=-1 errno=2
```

The value is the thread's `errno` at the moment `ffi.errno()` is called, which is why the call has to follow immediately after the failing library call: any other libc call in between — including those the interpreter itself makes — may overwrite it. It is not reset by reading it, so a stale value looks exactly like a fresh one; check the return value of the C function first, as the `rc == -1` test above does.

The limits of the approach

Callbacks

A ucode function can be passed to C where a function pointer is expected. When an argument's declared type is a function pointer, the module builds a libffi closure around the ucode function and passes that address, so the C code calls an ordinary C function which marshals the arguments into ucode values and the return value back to the declared C type. No helper and no declaration beyond the prototype itself are needed:

```
import * as ffi from "ffi";

let c = ffi.import("libc.so.6", `
        void *malloc(size_t size);
        void qsort(void *base, size_t nmemb, size_t size,
                   int (*compar)(const void *, const void *));
`);

let p = ffi.cast("int *", c.malloc(6 * ffi.sizeof("int")));
let vals = [9, 7, 5, 3, 1, 8];

for (let i = 0; i < 6; i++) {
        p[i] = vals[i];
}

c.qsort(p, 6, ffi.sizeof("int"), function (x, y) {
        return ffi.cast("int *", x)[0] - ffi.cast("int *", y)[0];
});

printf("sorted=%d %d %d %d %d %d\n", p[0], p[1], p[2], p[3], p[4], p[5]);
```

```
sorted=1 3 5 7 8 9
```

The comparator receives the declared types, not convenient ones, which is why it casts `const void *` to `int *` before indexing. Everything else about a callback is the same as about any other bound function: declare the signature exactly as the header has it, treat the parameters as the C types they are named as, and return what the declaration promises.

When the C side has to build a function pointer out of a ucode function — a vtable or an ops struct filled in from ucode — use `ffi.closure(type, func)`, the counterpart of `wrap()`: it creates an `ffi.closure` resource, a function-pointer cdata bound to the ucode function, that the C code can hold onto between calls. The resource keeps the bound function alive for as long as it is reachable; dropping all of its references releases the closure. It supports `ptr()`, `tostring()` and `cast()`, and can be stored into a struct field of function-pointer type with `set()`:

```
import * as ffi from "ffi";

ffi.cdef(`
    struct sorter { int (*compare)(const void *, const void *); };
`);

let cmp = ffi.closure('int (*)(const void *, const void *)',
    function (a, b) {
        return a.deref('int') - b.deref('int');
    });

let s = ffi.ctype('struct sorter');
s.set('compare', cmp);

printf("bound=%s\n", cmp.tostring());
```

```
bound=int (*)() 0x<address>
```

The closure is a resource, so the same limits the chapter's list of them applies: a function type with variable arguments cannot be bound to a closure, and the calling convention is fixed at the point `ffi.closure()` is called, so the type argument must be the pointer type the C side will actually call.

What is *not* accepted is a closure being converted to a function pointer via `cast()` — casting a plain ucode function value is refused with the same message:

```
import * as ffi from "ffi";

let cb = ffi.cast("int (*)(int)", function (x) {
        return x * 2;
});
```

```
Type error: cannot convert from 'closure' to 'int (*)()'
```

So a callback works when it is handed over at the call site, which covers `qsort()` and the whole family of register-a-callback entry points. When the program has to build a function pointer itself — a vtable or an ops struct filled in from ucode, say — `ffi.closure()` is the tool for it: the closure resource *is* the function pointer, and `set()` / `cast()` hand it over in the shape the C struct expects.

The remaining limits are the ones that come with runtime binding rather than compiled extensions:

- Declarations are written by hand and are never validated. A prototype that disagrees with the real signature does not fail; it produces wrong values, and for pointer types it can corrupt memory. Keeping the declaration

text close to the vendor header it was copied from, and quoting the header name in a comment, is the only defence.

- `varargs` declarations are not usable; a `printf`-style function cannot be declared and called. `vsnprintf`-style wrappers with a fixed signature can be.
- Memory allocated through `malloc` is not tracked. If the script exits without calling `free` the leak lasts as long as the process, which for a long-running daemon under `uwsd` is the lifetime of the service.
- Names are resolved against the library the script names, so the script now depends on `libc.so.6` and `libm.so.6` by name. Those file names are not portable across libc implementations; a musl-based device has `libc.so` instead, and the string in the script has to match the target.
- Each declaration is parsed when the script runs, and resolution is a `dlsym()` per symbol, so a script with many declarations pays for them at startup — the cost is microseconds each, and irrelevant unless the declaration list is long.

What the module does bring is reach. Reading a hardware register through `ioctl()`, calling a vendor SDK entry point, or reaching a libc function no ucode module wraps, becomes a matter of writing a prototype. The declarations are the whole of the interface between the script and the machine, and the discipline the chapter's examples are meant to instil — allocate, clear, check the return code, copy before freeing, read `errno` immediately — is what keeps that interface from failing somewhere it cannot be debugged.

Processes and signals

Running another program, and reacting to being told to stop, are the two things a script on a router does constantly. ucode offers `system()` for one-line command execution, `fs.popen()` for feeding or capturing a child's I/O, `signal()` for installing handlers, and the `uloop` module's process watcher for full child lifecycle management.

The references are generated from the module sources and published at ucode-lang.org: *the core functions* for `system()` and `signal()`, *the filesystem module* for `popen()`, and *the uloop module* for the process watcher.

Running a command

`system()` passes its argument to `/bin/sh` and returns the command's exit status.

```
printf("true=%d false=%d code=%d\n", system("true"), system("false"),
system("exit 3"));
```

```
true=0 false=1 code=3
```

The value is the exit code itself, not the raw status word a C `wait()` would produce, so `0` means success and any other number is whatever the command exited with. A command that cannot be run at all — a shell built into no shell, a syntax error in the command line — is reported the same way the shell reports it, usually `127`, so the return value distinguishes "the command said no" from "the command does not exist" only by convention.

The argument is a shell command line, which brings the whole shell with it: pipes, redirection, globbing, variables, and the shell's search of `PATH`.

```
printf("rc=%d\n", system("grep -c . /etc/passwd > /dev/null"));
```

```
rc=0
```

The child inherits the script's standard input, output and error. `system()` captures nothing; output goes where the script's output goes. That is what makes it right for commands whose effect is what matters and wrong for collecting results — for those, use `fs.popen()`.

The array form

`system()` and `fs.popen()` also accept an **array** instead of a string. An array is an argument vector: the first element names the program, the rest are its arguments, and the program is executed directly, without `sh` being involved at all.

```
printf("rc=%d\n", system(["/bin/echo", "hello; there"]));
```

```
hello; there
rc=0
```

The semicolon reached `echo` as text because there was no shell to read it. That is the reason to prefer the array form: **a string argument is shell code**, so a script that builds a command line out of a device name, a filename, a value from UCI or a payload received over ubus is handing that data to the shell for evaluation. With an array, no element is ever reinterpreted — each one arrives as exactly one `argv` entry, whatever it contains.

The array form is also how `process.spawn()` already works, so a script that uses the module does not have to think about this; the two built-ins are where the choice appears.

The trade-off is the shell's own features. Because no shell runs, an array command has no redirection, no globbing, no `~` expansion, no `;` sequencing and no pipelines; `>` is an argument, not a redirection. The program name is still resolved through `PATH`, as a shell would resolve it:

```
let missing = system(["/nonexistent-xyz-prog"]);
let found = system(["false"]);

printf("not found: rc=%d\nfound via PATH: rc=%d\n", missing, found);
```

```
not found: rc=255
found via PATH: rc=1
```

The calls come before the printing for a reason. Output sits in a buffer that a forked child inherits, and a child that cannot exec its program exits normally — flushing the copy of the buffer it inherited, which duplicates whatever the parent had printed but not yet flushed. With nothing pending at the time of the call there is nothing to duplicate. `fork()` below describes the same hazard from the other side.

A program that cannot be executed is reported as status `255`, since there is no shell around to answer `127`. An empty array is a type error, `Passed command array is empty`, rather than a silently successful no-op.

Reading a command's output works the same way in either form; the array form only changes how the program is started:

```
import * as fs from "fs";

let p = fs.popen(["/bin/echo", "one; two"], "r");

print(p.read("line"));
p.close();
```

```
one; two
```

Use the array form whenever any part of the invocation comes from outside the script. Use the string form when the command genuinely needs a shell — a pipeline, a redirection, a glob — and every interpolated value is under the script's control. In that case quote what you interpolate by hand, since the language has no quoting function:

```
let name = "wlan0";
let quoted = "'" + replace(name, "'", "'\\''") + "'";

printf("quoted=[%s] rc=%d\n", quoted, system("echo " + quoted + " > /dev/
null"));
```

```
quoted=['wlan0'] rc=0
```

Wrapping in single quotes and escaping any embedded single quote is the quoting rule, and `printf` has no `%q` conversion: it is not one of ucode's conversions, so the format string is emitted verbatim and no argument is consumed (chapter 21 describes what happens to unknown conversions). Quoting by hand is workable but easy to get wrong in code that grew into a command line over several edits; the array form does not have that failure mode.

Reading a command's output

`fs.popen()` starts a command and returns a file handle connected to it, using the same modes as `fs.open()`: `"r"` reads the command's standard output, `"w"` writes to its standard input.

```
import * as fs from "fs";

let p = fs.popen("printf abc; printf def", "r");
let first = p.read(64);
let second = p.read(64);

printf("[%s] [%s] rc=%d\n", first, second, p.close());
```

```
[abcdef] [] rc=0
```

The handle is an ordinary I/O handle, which means the read rules of that module apply: a length must be given, `""` is the end-of-file marker, and `null` indicates an error rather than exhaustion. `read()` with no length returns `null`, not the command's output — the most common mistake with `popen`, because a pipeline that "returned nothing" is usually a missing length rather than a command that produced nothing.

The exit status of the child comes from `close()`. It is not available any other way, so a script that cares whether the command succeeded has to close the handle and look at the return value:

```
import * as fs from "fs";

function run_capture(cmd) {
        let p = fs.popen(cmd, "r");
        let out = "";
        let chunk;

        while (chunk = p.read(4096)) {
                out += chunk;
        }

        return { code: p.close(), out: trim(out) };
}

let r = run_capture("echo hello; echo world >&2");

printf("code=%d out=[%s]\n", r.code, r.out);
```

```
code=0 out=[hello]
```

The `world` went to the script's own standard error, because `popen` with `"r"` connects only stdout. The loop reads until `""`; since the `while` condition stops on any falsy value, an error reading the pipe (which would give `null`) exits the loop too, with whatever was collected so far.

Writing to a child is symmetrical: the handle accepts writes to the command's standard input, and the command's output is not captured.

```
import * as fs from "fs";

let p = fs.popen("wc -c", "w");
p.write("12345");
p.close();
```

```
5
```

`wc -c` counted five bytes and printed the result to the inherited standard output, which is why the number appears in this example's output. To read that result back, the command has to be started with `"r"` and fed some other way.

Signals

`signal()` queries or installs a handler for one signal. Signal names are accepted with or without the `SIG` prefix and in any case, and signal numbers work too.

```
printf("term=%s int=%s hup=%s\n", signal("TERM", "ignore"), signal("INT",
"default"),
       signal("sigusr1", "ignore"));
```

```
term=ignore int=default hup=ignore
```

The return value is the handler now in force for that signal. Called with one argument, `signal()` reports the current handler without changing it:

```
printf("before=%s ", signal("TERM"));
signal("TERM", "ignore");
printf("after=%s\n", signal("TERM"));
```

```
before=default after=ignore
```

An unrecognised name returns `null` and installs nothing; there is no exception to catch.

```
printf("bogus=%s\n", signal("NOTASIGNAL", "ignore"));
```

```
bogus=(null)
```

A ucode function can be installed as a handler. Handlers are invoked by the interpreter itself, not by an event loop, so a plain script with no `uloop` still receives them. The handler runs at the next safe point after the signal arrives, not in the middle of the statement that was executing:

```
signal("USR1", function (sig) {
        printf("caught %J\n", sig);
});

system("(sleep 0.05; kill -USR1 $PPID) &");
system("sleep 0.2");
```

```
caught 10
```

The argument is the **signal number**, not the name — `10` for `SIGUSR1` on this platform. A handler installed through `uloop.signal()` receives no argument at all (`null`), which is one reason to install one handler per signal rather than a shared dispatcher.

The handler is a callback, and its own errors are reported the way errors are anywhere in the script — an exception raised inside a handler that is not caught terminates the script. What a handler does is therefore best kept to setting a flag or calling `uloop.end()` , and the actual work left to the main flow:

```
let stopping = false;

signal("TERM", function () {
        stopping = true;
});

printf("stopping=%s\n", stopping);
```

```
stopping=false
```

That snippet prints `false` because nothing raised `SIGTERM` ; it is the shape of a daemon's main loop condition — `while (!stopping) { ... }` — where the handler's only job is to make the next iteration test fail.

Signals that arrive while the interpreter is between safe points are coalesced by the operating system: two `SIGUSR1` deliveries in quick succession may run the handler once. For an event-driven program the `uloop` module registers signals through a signal file descriptor instead, which integrates with the loop and does not run arbitrary code in an asynchronous context:

```
import * as uloop from "uloop";

uloop.signal("USR1", function (sig) {
        printf("got %J\n", sig);
        uloop.end();
});

system("(sleep 0.05; kill -USR1 $PPID) &");
uloop.run();
```

```
got null
```

The loop's own termination function is `uloop.end()`; there is no `uloop.stop()`. Inside a signal callback, `uloop.end()` is the safe way to finish, since it only sets a flag the loop checks.

Choosing between them

The four mechanisms differ in what they do with the child's I/O and its exit status, and that is usually the whole decision.

Mechanism	Output goes to	Exit status	Use for
`system(cmd)`	inherited	return value	commands whose effect is the point
`fs.popen(cmd, "r")`	captured via handle	`close()`	collecting a command's stdout
`fs.popen(cmd, "w")`	inherited	`close()`	feeding a command's stdin
`uloop.process(...)`	callbacks	callback	daemons, long-lived children

A script that shells out in a loop, with `system()` or `popen`, starts a shell each time and pays for the fork, the exec and the shell's own startup. On a slow device that cost is visible after a few hundred calls; where the loop is tight, the alternative is usually to read the file or sysfs entry the command was going to read, or to keep the child open across iterations with `popen`. `uloop.process` is covered with the rest of the event loop in the `uloop` chapter, including how to reap children without blocking the loop.

log

The `log` module writes to the system log. It contains two APIs side by side, because ucode grew up alongside two logging conventions: the `ulog_*` interface from OpenWrt's `procd`, which can send to several destinations at once, and the classic C `openlog()` / `syslog()` / `closelog()` trio. Both are in one module, and the module's exported names make clear which is which.

The full reference — every function, constant and resource method — is generated from the module source and published at ucode-lang.org: *the log module*.

```
import * as log from "log";
```

The ulog interface

`ulog_open()` selects the destination channels, the facility and the identity string; `ulog()` writes one message.

```
import * as log from "log";

log.ulog_open(log.ULOG_STDIO, log.LOG_USER, "ucode-demo");
log.ulog(log.LOG_INFO, "interface %s is up\n", "wlan0");
```

```
ucode-demo: interface wlan0 is up
```

Three channels are available, and they are bit values so they can be combined:

Constant	Value	Destination
`ULOG_KMSG`	1	`/dev/kmsg` — the kernel log buffer
`ULOG_SYSLOG`	2	the system log daemon via `/dev/log`
`ULOG_STDIO`	4	the process's standard error

The default is the system log, which is why a script that logs without calling `ulog_open()` produces nothing on the terminal. During development, opening `ULOG_STDIO` makes the output visible; the array form selects several destinations at once, so a daemon can write to the system log and to stderr while running under `uwsd` in the foreground:

```
import * as log from "log";

printf("opened=%s\n", log.ulog_open(log.ULOG_STDIO | log.ULOG_SYSLOG,
log.LOG_DAEMON,
                                    "multi"));
```

```
opened=true
```

`ulog_open()` returns `true` on success and `false` for an invalid argument — an unrecognised channel or facility name, or a channel given as an array. Combine the bit values with `|`; the array form mentioned in the module's own documentation is rejected. The identity string is prefixed to every message, followed by `": "`, which is the only place it appears: the module does not expose it again.

Newlines are your business

`ulog()` does not terminate the message. Writing to stderr without a trailing newline interleaves messages into one line, which is confusing enough to be worth seeing:

```
import * as log from "log";

log.ulog_open(log.ULOG_STDIO, log.LOG_USER, "nl");
log.ulog(log.LOG_INFO, "first");
log.ulog(log.LOG_INFO, "second\n");
```

```
nl: firstnl: second
```

The system log strips trailing newlines when it stores a message, so a script that logs to both places through the array form needs the newline and tolerates its removal there. Every message is one log entry regardless: there is no multi-line record.

The format string is processed by ucode's own formatter, with the conversions described in the formatting chapter.

```
import * as log from "log";

log.ulog_open(log.ULOG_STDIO, log.LOG_USER, "fmt");
log.ulog(log.LOG_INFO, "stats %d %.2f %J\n", 42, 3.14159, { rx: 1, tx: 2 });
```

```
fmt: stats 42 3.14 { "rx": 1, "tx": 2 }
```

Priorities and filtering

The priority argument uses the `LOG_*` values, which have the standard numeric ordering from most to least severe: `LOG_EMERG` (0), `LOG_ALERT`, `LOG_CRIT`, `LOG_ERR` (3), `LOG_WARNING`, `LOG_NOTICE`, `LOG_INFO` (6), `LOG_DEBUG` (7).

`ulog_threshold()` sets the most verbose priority that is still written:

```
import * as log from "log";

log.ulog_open(log.ULOG_STDIO, log.LOG_USER, "thr");
log.ulog_threshold(log.LOG_ERR);
log.ulog(log.LOG_INFO, "this is filtered out\n");
log.ulog(log.LOG_ERR, "this survives\n");
```

```
thr: this survives
```

Filtering happens inside the module, before the message reaches any channel, so a script that logs verbosely and sets a threshold once at startup pays nothing for the suppressed messages beyond the call itself. `ulog_close()` detaches the channels again.

Convenience functions

Four shorthand functions log at a fixed priority, taking the same format string and arguments as `ulog()` without the priority argument. Their names are the uppercase priority abbreviations `INFO`, `NOTE`, `WARN` and `ERR`:

```
import * as log from "log";

log.ulog_open(log.ULOG_STDIO, log.LOG_DAEMON, "conv");
log.INFO("scanned %d networks\n", 12);
log.WARN("no router advertisement from %s\n", "fe80::1");
log.NOTE("reloading\n");
log.ERR("ioctl failed: %s\n", "Operation not permitted");
```

```
conv: scanned 12 networks
conv: no router advertisement from fe80::1
conv: reloading
conv: ioctl failed: Operation not permitted
```

There is no `DEBUG()` counterpart, and no `EMERG()` either: the shorthand covers the four priorities that ordinary scripts use. Like `ulog()`, these functions add no newline.

The classic syslog interface

`openlog()`, `syslog()` and `closelog()` behave as they do in C.

```
import * as log from "log";

log.openlog("ucode-classic", log.LOG_PID, log.LOG_DAEMON);
log.syslog(log.LOG_WARNING, "config reload took %d ms\n", 125);
log.closelog();
```

Nothing is printed to the terminal by this interface: the messages go to the system log daemon through `/dev/log`. The options are the `LOG_*` option bits — `LOG_PID` appends the process id to each message, `LOG_CONS` writes to the console if the daemon is unreachable, `LOG_NDELAY` opens the socket immediately rather than on first use, `LOG_ODELAY` is the default deferred behaviour, and `LOG_NOWAIT` applies to `vsyslog()` in C and has no effect here.

The exported facility names are `LOG_AUTH`, `LOG_AUTHPRIV`, `LOG_CRON`, `LOG_DAEMON`, `LOG_FTP`, `LOG_KERN`, `LOG_LPR`, `LOG_MAIL`, `LOG_NEWS`, `LOG_SYSLOG`, `LOG_USER`, `LOG_UUCP` and `LOG_LOCAL0` through `LOG_LOCAL7`. On a Linux system, the kernel facilities other than `LOG_KERN` are ignored for messages written from user space, and `LOG_KERN` itself is only meaningful through `ULOG_KMSG`.

Choosing an interface

`ulog` is the better choice for anything that runs on a device. It can write to stderr, which matters when a service is being supervised and its output is being collected by `uwsd` or a container runtime; it filters locally through `ulog_threshold()`; and it can write to `/dev/kmsg`, which is the only way to reach the kernel log buffer from a script — useful during boot, before a syslog daemon exists.

```
import * as log from "log";

log.ulog_open(log.ULOG_SYSLOG | log.ULOG_STDIO, log.LOG_DAEMON, "wifi-mgr");
log.ulog_threshold(log.LOG_INFO);

function channel_report(chan, dbm) {
        log.INFO("chan %d rssi %.1f dBm\n", chan, dbm);

        if (dbm < -80) {
                log.WARN("chan %d below sensitivity floor\n", chan);
        }
}

channel_report(36, -62.5);
channel_report(149, -88.25);
```

```
wifi-mgr: chan 36 rssi -62.5 dBm
wifi-mgr: chan 149 rssi -88.2 dBm
wifi-mgr: chan 149 below sensitivity floor
```

The classic interface remains useful for one thing: matching the logging behaviour of an existing C program, including `LOG_PID` and the option bits, without thinking about channels. Both write to the same system log, so mixing them in one script is harmless — the identity string set by `openlog()` applies only to `syslog()` calls, and the one set by `ulog_open()` only to `ulog()` and its shorthands.

socket

The `socket` module is a thin, direct layer over the operating system's socket interface. It is deliberately not an abstraction: there are no streams, no buffering, no `read line` helpers and no protocol implementations. What it exposes is the C API's vocabulary — `create`, `bind`, `listen`, `accept`, `send`, `recv`, `getopt`, `setopt`, `poll` — with addresses converted into ucode objects and errors converted into `null` values with a message behind them.

The full reference — every function, constant and resource method — is generated from the module source and published at ucode-lang.org: *the socket module*.

```
import * as socket from "socket";
```

The module exports eleven functions and two hundred and thirty-three constants. The constants are the C names, spelled as in the system headers: address families (`AF_INET`, `AF_UNIX`, `AF_PACKET`, `AF_CAN`), socket types (`SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_RAW`, plus the `SOCK_NONBLOCK` and `SOCK_CLOEXEC` modifier bits), message flags (`MSG_*`), socket levels and options (`SOL_SOCKET`, `SO_*`, `IP_*`, `IPV6_*`, `TCP_*`, `PACKET_*`, `CAN_*`), shutdown modes (`SHUT_RD`, `SHUT_WR`, `SHUT_RDWR`), resolver flags (`AI_*`, `NI_*`) and poll events (`POLLIN`, `POLLOUT`, `POLLERR`, `POLLHUP`, `POLLNVAL`, `POLLRDHUP`). Which of them exist depends on the platform headers the module was built against; `AF_PACKET` and `AF_CAN`, for instance, are Linux-only.

Addresses

Everything that names a destination or a bound endpoint accepts the same handful of forms, and `sockaddr()` is the function that shows what any of them means.

```
import * as socket from "socket";

printf("%J\n", socket.sockaddr("192.168.0.1:8080"));
printf("%J\n", socket.sockaddr([192, 168, 0, 1]));
printf("%J\n", socket.sockaddr("[fe80::1%lo]:8080"));
printf("%J\n", socket.sockaddr("/var/run/daemon.sock"));
```

```
{ "family": 2, "address": "192.168.0.1", "port": 8080 }
{ "family": 2, "address": "192.168.0.1", "port": 0 }
{ "family": 10, "address": "fe80::1", "port": 8080, "flowinfo": 0,
"interface": "lo" }
{ "family": 1, "path": "/var/run/daemon.sock" }
```

The four accepted input forms are:

- **A string** — `"host:port"` for IPv4, `"[addr]:port"` for IPv6 (with an optional `%interface` suffix for link-local addresses), or a filesystem path for a Unix socket.
- **An array of address bytes** — four numbers for IPv4, sixteen for IPv6, most significant byte first. This form carries no port, so one comes out as `0`; any other length is rejected the way a malformed string is.
- **An address object** — `{ address: "192.168.0.1", port: 8080 }`, optionally with `family`, `flowinfo` and `interface`. A Unix address is `{ path: "..." }`.
- **An object with `address` omitted but `path` present**, which is a Unix address.

`sockaddr()` is a converter rather than a validator with side effects; it is useful for normalising user input before handing it to another call.

```
import * as socket from "socket";

let bad = socket.sockaddr("not an address");

printf("bad=%J error=%J\n", bad, socket.error());
```

```
bad=null error="Unable to parse IP address: Invalid argument"
```

Addresses come back out of the module as objects of the same shape, which is what `sockname()`, `peername()`, `addrinfo()` and the receive-address out-parameter all produce. The `family` field is the numeric constant (`2` for `AF_INET`, `10` for `AF_INET6`, `1` for `AF_UNIX`), never a string.

Creating sockets

`create(domain, type[, protocol])` makes an unconnected socket. The type argument can carry the modifier bits, which the module applies with `fcntl()` after the `socket()` call.

```
import * as socket from "socket";

let raw = socket.create(socket.AF_INET, socket.SOCK_STREAM);
let nonblocking = socket.create(socket.AF_INET, socket.SOCK_DGRAM |
                                                socket.SOCK_NONBLOCK |
socket.SOCK_CLOEXEC);

printf("raw=%s nb=%s valid descriptor=%s\n", type(raw), type(nonblocking),
       nonblocking.fileno() > 2);
```

```
raw=resource nb=resource valid descriptor=true
```

A socket value is a resource that owns a file descriptor. It is closed when the resource is garbage-collected, but a script that should release a descriptor at a known point calls `close()`.

`open(fd)` goes the other way, wrapping a descriptor that came from somewhere else — an inherited descriptor, one received through `SCM_RIGHTS`, one opened by an `ffi` call — into a socket object with all the methods attached.

```
import * as socket from "socket";

let pair = socket.pair();
let wrapped = socket.open(pair[0].fileno());

printf("wrapped=%s same fd=%s\n", type(wrapped),
       wrapped.fileno() == pair[0].fileno());
```

```
wrapped=resource same fd=true
```

Wrapping does not duplicate the descriptor, so the wrapper and the original both refer to the same open file; closing either one affects both.

`pair([type])` creates two connected Unix-domain sockets, which is the shortest way to get a bidirectional channel inside one process, or between a process and a child it is about to `fork()`-and-exec.

Connecting and listening

The module-level `connect()` and `listen()` functions are the short path: they create, and if necessary bind, and then connect or listen, all in one call. Host and service are separate arguments, or a single address object.

```
import * as socket from "socket";

let server = socket.listen("127.0.0.1", 0, null, 128, true);
let port = server.sockname().port;

printf("kernel assigned an ephemeral port: %s\n", port > 1024);

let client = socket.connect("127.0.0.1", port);
let conn = server.accept();

client.send("status");
printf("server received %J\n", conn.recv(6));

server.close();
client.close();
conn.close();
```

```
kernel assigned an ephemeral port: true
server received "status"
```

The port above was chosen by the kernel, since `0` was requested; a script that listens on an ephemeral port and tells someone else about it reads the port back with `sockname()`.

The arguments of `listen(host, service, hints, backlog, reuseaddr)` are all optional except the host. `hints` is a table of resolver hints in the style of `addrinfo()`, `backlog` defaults to `128`, and `reuseaddr` sets `SO_REUSEADDR` before binding, which is what makes a restarted daemon able to rebind its port immediately.

The socket methods are the same operations in the explicit order:

```
import * as socket from "socket";

let srv = socket.create(socket.AF_INET, socket.SOCK_STREAM);

srv.setopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, true);
srv.bind("127.0.0.1:0");
srv.listen(16);

let cli = socket.create(socket.AF_INET, socket.SOCK_STREAM);
cli.connect(srv.sockname());

let incoming = srv.accept();
cli.send("abc");

printf("got %J from the same port the client bound: %s\n", incoming.recv(3),
       incoming.peername().port == cli.sockname().port);

srv.close();
cli.close();
incoming.close();
```

```
got "abc" from the same port the client bound: true
```

`bind()`, `connect()` and `send()` all take any of the address forms, so `connect(srv.sockname())` works directly on the object `sockname()` returned.

Sending and receiving

```
sk.send(data[, flags[, address]])   -> number of bytes written, or null
sk.recv([length=4096][, flags[, peer]])  -> string, "" or null
```

`send()` returns the number of bytes handed to the kernel, which for a datagram socket is the whole message or nothing, and for a stream socket may be a partial write. The `address` argument is only meaningful on connectionless sockets, and because it comes after `flags`, sending a datagram to an explicit destination must pass the flags argument even when there are none:

```
import * as socket from "socket";

let sender = socket.create(socket.AF_INET, socket.SOCK_DGRAM);
let receiver = socket.create(socket.AF_INET, socket.SOCK_DGRAM);

receiver.bind({ address: "127.0.0.1", port: 0 });
sender.bind({ address: "127.0.0.1", port: 0 });

let to = { address: "127.0.0.1", port: receiver.sockname().port };
let peer = {};

printf("sent %d bytes\n", sender.send("query", 0, to));
printf("received %J, and the recorded sender is the socket that sent it:
%s\n",
       receiver.recv(64, 0, peer), peer.port == sender.sockname().port);

sender.close();
receiver.close();
```

```
sent 5 bytes
received "query", and the recorded sender is the socket that sent it: true
```

`peer` is an out-parameter: an empty object is passed in and filled in with the sender's address while the data is returned as the call's value. This replaces `recvfrom()`; there is no separate method by that name.

`recv()` defaults to a 4096-byte buffer, which is large enough for a path-MTU IPv6 datagram. Its return convention is the one used throughout ucode's I/O layer:

Return	Meaning
non-empty string	data received
`""`	end of stream — the peer closed its write side
`null`	an error; ask `error()`

```
import * as socket from "socket";

let pair = socket.pair();
let a = pair[0], b = pair[1];

a.send("hello");
printf("read %J\n", b.recv());

a.shutdown(socket.SHUT_WR);
printf("after shutdown: %J error=%J\n", b.recv(), b.error());

a.close();
b.close();
```

```
read "hello"
after shutdown: "" error=null
```

A stream read that returns `""` will keep returning `""`; it is the loop-termination signal, not a transient condition. `null` is the failure case, and `error()` distinguishes it:

```
import * as socket from "socket";

let s = socket.create(socket.AF_INET, socket.SOCK_DGRAM);

printf("recv with nothing queued=%J\n", s.recv(16, socket.MSG_DONTWAIT));
printf("error=%J\n", s.error());
printf("strerror(2)=%J\n", socket.strerror(2));
```

```
recv with nothing queued=null
error="recv(): Resource temporarily unavailable"
strerror(2)="No such file or directory"
```

Receiving from a socket with nothing queued and `MSG_DONTWAIT` set gives `EAGAIN`, reported as `null`. The text from `error()` names the operation that failed — `"recv(): Resource temporarily unavailable"` — since the module prefixes the name of the failing call to the C library's message. `error()` returns a **string**, not a number, and takes no argument — it reports the condition left by the last failed operation on that socket, or on the module as a whole when called as `socket.error()`. `strerror(n)` is the separate, general-purpose lookup: it converts a numeric `errno` value, which the module does not itself hand out, into the same kind of message.

`sendmsg()` and `recvmsg()` are the structured forms. They take and return message objects that carry a scatter/gather list of buffers and, for Unix-domain sockets, ancillary data — file descriptors via `SCM_RIGHTS` and peer credentials via `SCM_CREDENTIALS`. Credentials can also be read directly with `peercred()`:

```
import * as socket from "socket";

let pair = socket.pair();

let cred = pair[0].peercred();

printf("fields: %s %s %s\n", type(cred.uid), type(cred.gid), type(cred.pid));
printf("uid is this process's: %s\n", cred.uid >= 0);

pair[0].close();
pair[1].close();
```

```
fields: int int int
uid is this process's: true
```

`peercred()` returns `{ uid, gid, pid }` for a socket connected to a peer on the same machine. It is the Unix-domain equivalent of asking the kernel who is on the other end, and it is the reason a local control socket can authenticate its client without a password.

Options

`setopt(level, option, value)` and `getopt(level, option)` map onto `setsockopt()` and `getsockopt()`. Each option has its own value type, taken from the module's option table rather than inferred from the bytes it stores.

```
import * as socket from "socket";

let s = socket.create(socket.AF_INET, socket.SOCK_STREAM);

printf("type=%d recvbuf>0=%s reuseaddr=%s\n", s.getopt(socket.SOL_SOCKET,
socket.SO_TYPE),
       s.getopt(socket.SOL_SOCKET, socket.SO_RCVBUF) > 0,
       s.getopt(socket.SOL_SOCKET, socket.SO_REUSEADDR));
printf("set=%s now=%s\n", s.setopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,
true),
       s.getopt(socket.SOL_SOCKET, socket.SO_REUSEADDR));

s.close();
```

```
type=1 recvbuf>0=true reuseaddr=false
set=true now=true
```

Boolean options come back as `false` and `true`, integer options as numbers, and string options such as `SO_BINDTODEVICE` or `TCP_CONGESTION` as strings. A failed `getopt()` returns `null` — asking a `SOCK_DGRAM` socket for a `TCP_*` option, for example, fails with `error()` reporting "Protocol not available" (`ENOPROTOOPT`).

Polling

`poll()` takes the timeout **first**, then any number of sockets, and returns one entry per socket in the order given, each an array of the socket and its ready event mask.

```
import * as socket from "socket";

let a = socket.create(socket.AF_INET, socket.SOCK_DGRAM);
let b = socket.create(socket.AF_INET, socket.SOCK_DGRAM);

b.bind({ address: "127.0.0.1", port: 0 });

printf("idle entries: %d\n", length(socket.poll(0, [b, socket.POLLIN])));

let events = socket.poll(0, [b, socket.POLLIN]);

printf("mask before: %d\n", events[0][1]);

a.send("x", 0, { address: "127.0.0.1", port: b.sockname().port });

events = socket.poll(0, [b, socket.POLLIN], [a, socket.POLLIN]);
printf("b readable=%s a readable=%s\n", events[0][1] & socket.POLLIN ?
"yes" : "no",
       events[1][1] & socket.POLLIN ? "yes" : "no");

a.close();
b.close();
```

```
idle entries: 1
mask before: 0
b readable=yes a readable=no
```

A timeout of `-1` blocks until at least one descriptor is ready, `0` returns immediately, and any positive value is milliseconds. A bare socket instead of a `[socket, events]` pair is accepted, in which case `POLLIN` is assumed. The returned mask is tested with `&`, since a socket can report `POLLIN | POLLHUP` together.

`poll()` covers the single-socket wait cases. A daemon with timers, signals and descriptors to watch is better served by the `uloop` module, whose `handle` watcher registers a socket descriptor with an event loop and avoids a poll per iteration.

Names

`addrinfo(node[, service[, hints]])` resolves a host and service into the list of addresses that `connect()` could use, and `nameinfo(address[, flags])` goes the other way.

```
import * as socket from "socket";

let list = socket.addrinfo("localhost", "domain");

printf("results: %s, port 53: %s, family known: %s\n", length(list) > 0,
       list[0].addr.port == 53,
       list[0].family == socket.AF_INET || list[0].family ==
socket.AF_INET6);
printf("reverse: %J\n", socket.nameinfo({ address: "127.0.0.1", port: 53 }));
```

```
results: true, port 53: true, family known: true
reverse: { "hostname": "localhost", "service": "domain" }
```

Each result is an object with `flags`, `family`, `socktype`, `protocol`, `addr` (a socket address object) and `canonname`. `nameinfo()` returns a two-field object, `hostname` and `service`, where the service is rendered as a name when one is known — `domain` for port 53. Numeric output is available with the `NI_NUMERICHOST` and `NI_NUMERICSERV` flags, which have the usual meanings.

The resolver used is the system resolver, so `/etc/hosts`, `nsswitch.conf` and — on musl — the behaviour of `resolv.conf` options are all in play. Resolution failures return `null` with the C library's message:

```
import * as socket from "socket";

printf("result=%J\n", socket.addrinfo("no-such-host.invalid"));
printf("error=%J\n", socket.error());
```

```
result=null
error="getaddrinfo(): Name or service not known"
```

A daemon that must not block on DNS should call `addrinfo()` before entering its event loop, or run it in a child process; there is no asynchronous form in the module.

A small exchange

The pieces fit together into the shape almost every socket program has: bind, accept, loop over poll, read until the peer stops, answer, close.

```
import * as socket from "socket";

function serve(srv, limit) {
        let conn = srv.accept();
        let buf = "";

        while (length(buf) < limit) {
                let chunk = conn.recv(64);

                if (!chunk) {
                        break;
                }

                buf += chunk;
        }

        conn.send("echo:" + buf);
        conn.close();
}

let srv = socket.listen("127.0.0.1", 0, null, 4, true);
let cli = socket.connect("127.0.0.1", srv.sockname().port);

cli.send("hello");
serve(srv, 5);

printf("reply=%J\n", cli.recv(64));

srv.close();
cli.close();
```

```
reply="echo:hello"
```

The example is deliberately synchronous — the server and the client are the same script, so it can run to completion without a real event loop. Replacing `poll()` with `uloop.handle()` and giving each connection its own callback is the step that turns it into a daemon; the buffering, the `""`-means-done test and the `peername()` bookkeeping carry over unchanged.

resolv and netaddr

Two modules cover the network's naming and addressing. `resolv` answers DNS questions directly, without help from libc's resolver routines and without involving the event loop: `query()` sends its packets, waits, and returns a finished answer structure. `netaddr` works on the addresses themselves — IPv4, IPv6 and MAC — turning them into first-class values you can validate, inspect and do CIDR arithmetic on.

The full references are generated from the module sources and published at ucode-lang.org: *the DNS resolve module* and *the netaddr module*.

```
import * as resolv from "resolv";
```

`query()` is synchronous. A call blocks the script for as long as the resolver keeps retrying, which makes it right for a script that resolves a name once before doing its work, and wrong for a long-lived service that must stay responsive — a service either resolves in a `uloop` task or accepts the delay.

Looking up a name

```
import * as resolv from "resolv";

let res = resolv.query("localhost");

printf("%J\n", res);
```

```
{ "localhost": { "A": [ "127.0.0.1" ], "AAAA": [ "::1" ] } }
```

The result is an object keyed by the name that was queried, holding an object whose keys are record type names and whose values are arrays of records. With no options, a domain name is queried for `A` and `AAAA`, and an address is queried for `PTR`:

```
import * as resolv from "resolv";

let res = resolv.query("127.0.0.1");
let ptr = res["1.0.0.127.in-addr.arpa"].PTR;

printf("reverse lookup returned records: %s\n", length(ptr) > 0);
```

```
reverse lookup returned records: true
```

Keying by the name actually queried matters for reverse lookups: the entry appears under the generated `in-addr.arpa` or `ip6.arpa` name, not under the address that was passed in. Asking for `PTR` alongside a forward type makes that visible, since the address then also gets its own entry with the response code for the nonsensical forward query:

```
import * as resolv from "resolv";

let res = resolv.query(["localhost", "127.0.0.1"], { type: ["A", "PTR"] });

printf("%J\n", sort(keys(res)));
```

```
[ "1.0.0.127.in-addr.arpa", "127.0.0.1", "localhost" ]
```

Three entries for two inputs: the address `127.0.0.1` is queried as itself for `A`, which no zone answers, and again after `in-addr.arpa` conversion for `PTR`. Whatever a server makes of the address itself is environment-dependent; the shape of the keys is not.

Record types

`type` selects the record types to query, from `A`, `AAAA`, `CNAME`, `MX`, `NS`, `PTR`, `SOA`, `SRV`, `TXT` and `ANY`. An unrecognised name rejects the whole call rather than being skipped.

```
import * as resolv from "resolv";

let res = resolv.query("openwrt.org", { type: ["MX"] });
let mx = res["openwrt.org"].MX;

printf("records=%d, entry is [number, string]: %s\n", length(mx),
       type(mx[0]) == "array" && type(mx[0][0]) == "int" && type(mx[0][1])
== "string");
```

```
records=1, entry is [number, string]: true
```

The record itself looks like `[[ 10, "util-01.infra.openwrt.org" ]]`; the example tests its shape instead of its contents so that the result does not depend on how the zone is configured today.

Records are returned in one of three shapes. Addresses, host names and text are plain strings; `MX` records are `[ preference, exchange ]` and `SRV` records are `[ priority, weight, port, target ]`; an `SOA` record is the seven-element array `[ mname, rname, serial, refresh, retry, expire, minimum ]`. `NS`, `CNAME`, `PTR` and `SOA` responses were shown above; a `SOA` answer looks like this:

```
import * as resolv from "resolv";

let soa = resolv.query("openwrt.org", { type: ["SOA"] })
["openwrt.org"].SOA[0];

printf("fields=%d, names are strings: %s, timers are numbers: %s\n",
length(soa),
       type(soa[0]) == "string" && type(soa[1]) == "string",
       type(soa[3]) == "int" && type(soa[6]) == "int");
```

```
fields=7, names are strings: true, timers are numbers: true
```

A live answer expands as `[ "ns1.digitalocean.com", "hostmaster.openwrt.org", 0, 10800, 3600, 604800, 1800 ]` — primary nameserver, contact mailbox, serial, refresh, retry, expire and minimum TTL. Names are fully qualified strings; the five numbers are integers, and the contact is `hostmaster.openwrt.org` rather than `hostmaster@openwrt.org`, following the usual wire encoding of an `SOA` mailbox. Serial arithmetic is `SERIAL ± n mod 2^32`, which a script has to perform itself — and note that a serial of `0` here is what the zone serves, not a parse failure; `dig` reports the same value.

Text records

TXT records are character-strings, and a single DNS TXT record may contain several of them. By default the module joins them into one string per record, separated by a space — and it puts that space in front of the first string as well, so the value carries a leading blank:

```
import * as resolv from "resolv";

let txt = resolv.query("openwrt.org", { type: ["TXT"] })
["openwrt.org"].TXT[0];

printf("leading blank: %s, same after trim: %s\n", substr(txt, 0, 1) == " ",
       substr(txt, 1) == trim(txt));
```

```
leading blank: true, same after trim: true
```

For a single-string record the raw value is `" v=spf1 ip4:46.101.232.90 -all"`, with one space in front of the text.

`txt_as_array` keeps the structure instead, giving one array of strings per record, with no joined value and no leading blank:

```
import * as resolv from "resolv";

let res = resolv.query("openwrt.org", { type: ["TXT"], txt_as_array: true });
let txt = res["openwrt.org"].TXT;

printf("at least one record: %s, first is an array: %s\n", length(txt) > 0,
       type(txt[0]) == "array");
```

```
at least one record: true, first is an array: true
```

For a record shorter than 255 bytes the two forms differ only by that leading space, which is easy to miss when a value is compared or fed to another program — `trim()` the default form if the exact bytes matter.

Nameservers, timeout and retries

Without a `nameserver` option, `query()` reads `/etc/resolv.conf` and falls back to `127.0.0.1` if that file names nothing. The option takes an array of server addresses; a port is appended with `#`, not a colon, and IPv6 addresses may carry an interface scope:

```
import * as resolv from "resolv";

let res = resolv.query("example.org", {
        nameserver: ["8.8.8.8#53", "2001:4860:4860::8888"],
        timeout: 1000,
        retries: 1,
});

printf("one entry per queried name: %s\n", length(keys(res)) == 1);
```

```
one entry per queried name: true
```

Whether that entry carries answers or a `TIMEOUT` status depends on whether the script can reach those servers; the option syntax is the point here. IPv6 addresses are written bare — `"2001:4860:4860::8888"` — and not in the bracketed form that URLs and socket addresses use; `"[2001:4860:4860::8888]"` is rejected with *"Unable to*

resolve nameserver address". Because `#` separates the port and `%` the interface scope, an address needs no brackets to be unambiguous.

`timeout` is the total budget for the query in milliseconds, defaulting to 5000, and `retries` (default 2) is the number of attempts spread across that budget — the interval between sends is the timeout divided by the number of attempts. Both have to be usable as unsigned integers within that arithmetic: `retries` must be at least 1 and `timeout` must not be negative, and either is rejected with `Invalid argument` otherwise.

```
import * as resolv from "resolv";

let res = resolv.query("example.org", { retries: 0 });

printf("res=%J\nerror=%J\n", res, resolv.error());
```

```
res=null
error="Invalid argument: Retries must be a positive integer"
```

A server that never answers is reported, not thrown: every outstanding name gets a `TIMEOUT` status.

```
import * as resolv from "resolv";

let res = resolv.query("example.org", {
        nameserver: ["192.0.2.1"],
        timeout: 200,
        retries: 1,
});

printf("%J\nerror=%J\n", res, resolv.error());
```

```
{ "example.org": { "rcode": "TIMEOUT" } }
error="Connection timed out: Server did not respond"
```

`edns_maxsize` caps the UDP payload size advertised with EDNS0 and defaults to 4096; setting it to 0 leaves the option out of the query entirely, which is what a path with a small MTU needs.

Response codes and missing answers

A name that does not exist is a normal result, keyed by the name with a `rcode` in place of record data:

```
import * as resolv from "resolv";

let res = resolv.query("no-such-host.invalid");

printf("%J\n", res);
```

```
{ "no-such-host.invalid": { "rcode": "NXDOMAIN" } }
```

The status string is one of the DNS codes — `NOERROR`, `FORMERR`, `SERVFAIL`, `NXDOMAIN`, `NOTIMP`, `REFUSED`, `YXDOMAIN`, `YXRRSET`, `NXRRSET`, `NOTAUTH`, `NOTZONE` — plus `TIMEOUT` for a query that ran out of budget. `NOERROR` with no records for the requested type produces an entry without type keys, so a test for `rcode` alone is not enough to tell success from emptiness:

```
import * as resolv from "resolv";

let res = resolv.query("example.org", { type: ["NS"] });
let entry = res["example.org"];

printf("rcode=%s, has ns records: %s\n", entry.rcode ? entry.rcode :
"NOERROR",
       entry.NS ? length(entry.NS) > 0 : false);
```

```
rcode=NOERROR, has ns records: true
```

The result carries record data only. There is no TTL, no class, and no distinction between answers and additional-section records; a script that needs to cache according to TTL has to pick its own interval.

Errors

`query()` returns `null` when the call itself is invalid — an unrecognised record type, a malformed nameserver, a bad option value — and `error()` then explains it. A name that is not a string is coerced to one, so `query(42)` looks up `"42"` rather than failing. The message is `strerror()` text for the underlying errno, followed by the module's own message:

```
import * as resolv from "resolv";

printf("%J %J\n", resolv.query("openwrt.org", { type: ["nope"] }),
resolv.error());
printf("%J %J\n", resolv.query("openwrt.org", { nameserver: [{ address:
"127.0.0.1" }] }),
       resolv.error());
printf("%J %J\n", resolv.query("openwrt.org", { retries: 0 }),
resolv.error());
```

```
null "Invalid argument: Unrecognized query type 'nope'"
null "Invalid argument: Unable to resolve nameserver address '{ \"address\":
\"127.0.0.1\" }'"
null "Invalid argument: Retries must be a positive integer"
```

The message is consumed by reading it: a second call returns `null`, so ask for it once and keep it.

A reusable lookup helper

Pulling the pieces together — query, distinguish names that answered from names that reported a status, and flatten what is left:

```
import * as resolv from "resolv";

function resolve(names, types) {
        let res = resolv.query(names, { type: types, timeout: 3000, retries:
2 });
        let out = {};
        let answers = 0;

        if (res == null) {
                die("resolv: " + resolv.error());
        }

        for (let name in res) {
                for (let type in res[name]) {
                        if (type == "rcode") {
                                continue;
                        }

                        out[name] = { type: type, records: res[name]
[type] };
                        answers++;
                }
        }

        return { answers: answers };
}

let r = resolve(["localhost", "no-such-host.invalid"], ["A"]);

printf("answers=%d\n", r.answers);
```

```
answers=1
```

`out` is indexed by the names the resolver used, which is where a reverse lookup keyed by an address would appear under its `arpa` name. For a script that only needs to know whether a host resolves, the count is enough — and printing the count rather than the address keeps the result independent of the network it runs on.

The netaddr module

`resolv` hands you addresses as strings; `netaddr` turns them into values. It parses and validates IPv4, IPv6 and MAC addresses and does the CIDR arithmetic that scripts constantly need — network and broadcast addresses, masks, host ranges, containment — without any help from libc. Where the core's `iptoarr()` and `arrtoip()` (chapter 20) convert a dotted quad to a four-byte array and back, `netaddr` is the full tool.

```
import * as na from "netaddr";
```

Every constructor returns a `netaddr.range` instance — a resource, not a string — and the family is fixed by which constructor you call: `v4()` for IPv4, `v6()` for IPv6, `mac()` for an ethernet address, and `new()` to let the module detect the family from the value.

Constructing addresses

```
import * as na from "netaddr";

let a = na.new("192.168.1.1");

printf("%s family=%d bits=%d\n", a, a.family, a.bits);
```

```
192.168.1.1 family=4 bits=32
```

`family` is `4` for IPv4, `6` for IPv6 and `1` for a MAC address; `bits` is the prefix size, defaulting to the family's full width (`32` , `128` and `48`). `new()` detects the family from the value, so the same call handles all three:

```
import * as na from "netaddr";

printf("%s\n", na.new("2001:db8::1"));
printf("%s\n", na.new("de:ad:be:ef:00:01"));
```

```
2001:db8::1
DE:AD:BE:EF:00:01
```

Note the MAC is rendered in canonical uppercase. A string may carry the prefix or a netmask after a slash, and a second argument overrides the prefix; a byte array or a number (host byte order) is accepted too:

```
import * as na from "netaddr";

printf("%s\n", na.v4("192.168.1.0/24"));
printf("%s\n", na.v4("192.168.1.0/255.255.255.0"));
printf("%s\n", na.v4("192.168.1.0/24", 16));
printf("%s\n", na.v4([192, 168, 1, 1]));
printf("%s\n", na.v4(0x0100007f));
```

```
192.168.1.0/24
192.168.1.0/24
192.168.1.0/16
192.168.1.1
1.0.0.127
```

Validating without exceptions

The constructors raise on bad input. To validate untrusted input without try/catch, the `check*` functions return the canonical address as a plain string, or `null` if the value is not a valid address of that family:

```
import * as na from "netaddr";

printf("%s\n", na.checkv4("192.168.1.1"));
printf("%s\n", na.checkv4("999.1.1.1"));
printf("%s\n", na.checkv6("0:0:0:0:0:0:0:1"));
printf("%s\n", na.checkmac("00:11:22:cc:dd:ee"));
```

```
192.168.1.1
(null)
::1
00:11:22:CC:DD:EE
```

The return value is canonicalised — `checkv6` compresses to `::` form and `checkmac` uppercases — so it is safe to compare or store. The constructors, by contrast, throw:

```
import * as na from "netaddr";

printf("check: %s\n", na.checkv4("999.1.1.1"));

try {
        na.v4("999.1.1.1");
} catch (e) {
        printf("v4 raises: %s\n", e.message);
}
```

```
check: (null)
v4 raises: Invalid IPv4 address
```

CIDR arithmetic

A range carries a prefix, and the methods derive the addresses that prefix implies. Each returns a new range; `size` is a property giving the number of addresses in the range:

```
import * as na from "netaddr";

let a = na.v4("192.168.1.0/24");

printf("network   %s\n", a.network());
printf("broadcast %s\n", a.broadcast());
printf("mask      %s\n", a.mask());
printf("minhost   %s\n", a.minhost());
printf("maxhost   %s\n", a.maxhost());
printf("size      %d\n", a.size);
```

```
network   192.168.1.0
broadcast 192.168.1.255
mask      255.255.255.0
minhost   192.168.1.1
maxhost   192.168.1.254
size      256
```

`minhost` and `maxhost` are the first and last usable host addresses, so for a `/24` they exclude the network and broadcast addresses. `size` is `2` to the power of the host bits, and is `null` when the count does not fit in a 64-bit integer (a `/64` and wider).

The comparison methods take a string or a range:

```
import * as na from "netaddr";

let a = na.v4("192.168.1.0/24");
let sub = na.v4("192.168.1.64/26");

printf("a contains sub: %s\n", a.contains(sub));
printf("sub contains a: %s\n", sub.contains(a));
printf("a equals /24:   %s\n", a.equal("192.168.1.0/24"));
printf("a higher than 192.168.0.0/24: %s\n", a.higher("192.168.0.0/24"));
```

```
a contains sub: true
sub contains a: false
a equals /24:   true
a higher than 192.168.0.0/24: true
```

`add()` and `sub()` shift the address by a number of addresses, clamping at the top and bottom of the range:

```
import * as na from "netaddr";

let h = na.v4("192.168.1.10");

printf("%s + 5 = %s\n", h, h.add(5));
printf("%s - 3 = %s\n", h, h.sub(3));
```

```
192.168.1.10 + 5 = 192.168.1.15
192.168.1.10 - 3 = 192.168.1.7
```

Address properties and predicates

A range supports property access. Numeric keys read and write the individual address bytes (negative indices count from the end), and `bits` reads or writes the prefix:

```
import * as na from "netaddr";

let b = na.v4("192.168.1.1");

printf("first byte %d, last byte %d\n", b[0], b[-1]);
b[3] = 42;
printf("after b[3] = 42: %s\n", b);
```

```
first byte 192, last byte 1
after b[3] = 42: 192.168.1.42
```

`family`, `size`, `host` and `netmask` are read-only properties. The predicate methods answer questions about which part of the address space a value falls in:

```
import * as na from "netaddr";

let a = na.v4("192.168.1.0/24");

printf("is4 %s, rfc1918 %s, linklocal %s\n", a.is4(), a.is4rfc1918(),
a.is4linklocal());
```

```
is4 true, rfc1918 true, linklocal false
```

Alongside `is4()`, `is6()` and `ismac()` there are `is4rfc1918()` (private `10/8`, `172.16/12` and `192.168/16`), `is4linklocal()` (`169.254/16`), `is6linklocal()` (`fe80::/10`), `is6mapped4()` (`::ffff:0:0/96`), and for MAC addresses `ismaclocal()` (locally administered) and `ismacmcast()` (multicast).

MAC addresses and IPv6

A MAC address can be turned into its EUI-64 link-local form, and a link-local address back into a MAC:

```
import * as na from "netaddr";

let m = na.mac("de:ad:be:ef:00:01");

printf("%s local=%s mcast=%s\n", m, m.ismaclocal(), m.ismacmcast());
printf("link-local %s\n", m.tolinklocal());
```

```
DE:AD:BE:EF:00:01 local=true mcast=false
link-local fe80::dcad:beff:feef:1
```

`tomac()` does the inverse on a link-local address derived from a MAC, and returns `null` for one that was not. IPv6 addresses may carry a scope — an interface index or name — for link-local use, readable through `scope` and `scopeid`; the `mapped4` property recovers the embedded IPv4 address from a `::ffff:a.b.c.d` value:

```
import * as na from "netaddr";

let mapped = na.v6("::ffff:192.168.1.1");
printf("mapped4 %s\n", mapped.mapped4);

let s = na.v6("fe80::1", 64, 2);
printf("scope %d\n", s.scope);
```

```
mapped4 192.168.1.1
scope 2
```

rtnl: routing and interfaces via netlink

The `rtnl` module talks to the kernel's `NETLINK_ROUTE` socket: it reads and changes links, addresses, routes, neighbours, rules and the rest of the routing state, and it subscribes to the kernel's multicast notifications when that state changes. It is the interface a user-space network manager uses instead of shelling out to `ip`.

The full reference — every function, constant and resource method — is generated from the module source and published at ucode-lang.org: *the rtnl module*.

```
import * as rtnl from "rtnl";
```

The module exports `request`, `listener`, `error` and a `const` object holding 331 constants taken from the Linux routing headers. Those constants are the vocabulary of the module: commands such as `RTM_GETLINK`, request flags such as `NLM_F_DUMP`, address families, route types, neighbour states, and the multicast group numbers used by `listener`.

```
import * as rtnl from "rtnl";

printf("RTM_GETLINK=%d RTM_GETADDR=%d RTM_GETROUTE=%d\n",
       rtnl.const.RTM_GETLINK, rtnl.const.RTM_GETADDR,
rtnl.const.RTM_GETROUTE);
printf("NLM_F_REQUEST=%d NLM_F_DUMP=%d NLM_F_ACK=%d\n",
       rtnl.const.NLM_F_REQUEST, rtnl.const.NLM_F_DUMP,
rtnl.const.NLM_F_ACK);
```

```
RTM_GETLINK=18 RTM_GETADDR=22 RTM_GETROUTE=26
NLM_F_REQUEST=1 NLM_F_DUMP=768 NLM_F_ACK=4
```

Requests

`request(command, flags, payload)` builds a netlink message, sends it, waits for the answer and returns it as ucode data. The command is a number — one of the `RTM_*` constants — and the payload is an object whose fields are encoded into the message.

```
import * as rtnl from "rtnl";

let links = rtnl.request(rtnl.const.RTM_GETLINK,
                         rtnl.const.NLM_F_REQUEST | rtnl.const.NLM_F_DUMP,
                         {});

printf("array of entries: %s, has interfaces: %s\n", type(links) == "array",
       length(links) > 0);
```

```
array of entries: true, has interfaces: true
```

The number of entries is whatever the machine has; the shape is the same everywhere.

`NLM_F_DUMP` asks for the complete list, and the answer is an array with one object per entry. Without it, the same command is a single get and the answer is a lone object:

```
import * as rtnl from "rtnl";

let lo = rtnl.request(rtnl.const.RTM_GETLINK, rtnl.const.NLM_F_REQUEST, {
dev: "lo" });

printf("type=%s dev=%s mtu over 8000: %s\n", type(lo), lo.dev, lo.mtu >
8000);
```

```
type=object dev=lo mtu over 8000: true
```

A dump of the interface list returns a great deal per interface: the link header fields (`family` , `type` , `dev` , `flags` , `mtu` , `address` , `broadcast` , `txqlen` , `carrier` , `operstate`), the queue counts, `group` , `proto_down` , a `stats64` object with the full transmit and receive counters, and an `af_spec` object holding the per-address-family data — under `inet` and `inet6` , each with a `conf` sub-object mirroring the `ipv4.conf` / `ipv6.conf` tunables, `forwarding` , `accept_ra` , `rp_filter` and the rest.

Names are resolved in both directions. The `dev` field of a payload is a name that the module looks up before sending, and interface indexes coming back from the kernel are turned into names: `oif` on a route is an interface name, not a number.

```
import * as rtnl from "rtnl";

let routes = rtnl.request(rtnl.const.RTM_GETROUTE,
                          rtnl.const.NLM_F_REQUEST | rtnl.const.NLM_F_DUMP,
                          { family: rtnl.const.AF_INET });

let withdev = filter(routes, (r) => r.oif ? type(r.oif) == "string" : false);

printf("dump is not empty: %s, entries naming an interface: %s\n",
       length(routes) > 0, length(withdev) > 0);
```

```
dump is not empty: true, entries naming an interface: true
```

The route count belongs to the machine; what is consistent is that an outgoing interface is reported by name.

Addresses, routes and neighbours

The four dumps that a network script uses most are `RTM_GETLINK`, `RTM_GETADDR`, `RTM_GETROUTE` and `RTM_GETNEIGH`, each with the same shape of answer. An address entry carries `dev`, `family`, `label`, `scope`, `flags`, `address`, `local` and `broadcast`, plus a `cacheinfo` object with the lifetimes:

```
import * as rtnl from "rtnl";

let addrs = rtnl.request(rtnl.const.RTM_GETADDR,
                         rtnl.const.NLM_F_REQUEST | rtnl.const.NLM_F_DUMP,
                         { family: rtnl.const.AF_INET });

let v4 = filter(addrs, (a) => a.family == rtnl.const.AF_INET);

printf("has ipv4 addresses: %s, first names a device: %s\n", length(v4) > 0,
       length(v4) > 0 ? type(v4[0].dev) == "string" : false);
```

```
has ipv4 addresses: true, first names a device: true
```

A route entry carries `family`, `dst`, `gateway`, `prefsrc`, `oif`, `table`, `priority`, `type`, `scope`, `protocol`, `tos` and `flags` — with the fields that do not apply to a given route simply absent. A neighbour entry carries `dev`, `dst`, `lladdr`, `state`, `type`, `flags`, `probes` and `cacheinfo`, and its `state` is a bitmask of the `NUD_*` constants:

```
import * as rtnl from "rtnl";

let neigh = rtnl.request(rtnl.const.RTM_GETNEIGH,
                         rtnl.const.NLM_F_REQUEST | rtnl.const.NLM_F_DUMP,
                         { family: rtnl.const.AF_BRIDGE });

let reachable = filter(neigh, (n) => n.state & rtnl.const.NUD_REACHABLE);

printf("state is a number: %s, all reachable: %s\n", length(neigh) == 0 ||
       type(neigh[0].state) == "int", length(reachable) <= length(neigh));
```

```
state is a number: true, all reachable: true
```

Counting entries by state is the normal way to read this table. Which neighbours are present, and how many are reachable at the moment of the call, is a property of the machine and of the last few minutes of traffic on it.

`RTM_GETRULE`, `RTM_GETNEIGHTBL`, `RTM_GETNETCONF` and `RTM_GETADDRLABEL` complete the set of readable families, and `RTM_NEW*` and `RTM_DEL*` write. A write request needs `NLM_F_ACK` to produce an answer at all, and needs the privilege to make the change.

Errors

`request()` returns `null` when the request cannot be made or the kernel refuses it, and `error()` describes the failure. Validation happens in the module before anything is sent, so a mistake in the payload is reported in terms of the field that caused it:

```
import * as rtnl from "rtnl";

let res = rtnl.request(rtnl.const.RTM_GETLINK, rtnl.const.NLM_F_REQUEST,
                       { dev: "nosuchif0" });

printf("res=%J\nerror=%J\n", res, rtnl.error());
```

```
res=null
error="Invalid input data or parameter: field `dev` has invalid value
`nosuchif0`: interface not found"
```

A command passed as a name rather than as one of the constants is rejected the same way, with the generic message — the `RTM_*` constants are looked up in `rtnl.const`, not parsed from strings:

```
import * as rtnl from "rtnl";

printf("res=%J error=%J\n", rtnl.request("RTM_GETLINK", 0, {}),
rtnl.error());
```

```
res=null error="Invalid input data or parameter"
```

A request that the kernel rejects arrives prefixed with `RTNETLINK answers:`, which is how a refusal is told apart from a module-level problem such as an unknown field or an unresolvable name. Like the other modules, `error()` reports the last failure, is cleared by a successful request and is consumed by reading it, so a script that wants the message should read it where the failure happens.

Listening for changes

`listener(callback, [commands], [groups])` subscribes to the kernel's multicast groups and invokes the callback for each notification. Both optional arguments are arrays of numbers: the `RTM_*` commands to pay attention to, and the `RTNLGRP_*` groups to join.

```
import * as uloop from "uloop";
import * as rtnl from "rtnl";

let events = 0;

let l = rtnl.listener(function (msg) {
        events++;
}, [rtnl.const.RTM_NEWLINK, rtnl.const.RTM_DELLINK],
[rtnl.const.RTNLGRP_LINK]);

printf("listener=%s\n", type(l));

uloop.timer(20, function () {
        printf("events seen=%d\n", events);
        uloop.end();
});

uloop.run();
l.close();
```

```
listener=resource
events seen=0
```

The listener is driven by `uloop`; notifications are delivered as ordinary callbacks while the loop runs, and nothing arrives before `run()` is called. `close()` unsubscribes and releases the socket, `set_commands()` changes which commands reach the callback without re-creating the listener, and the same script can hold several listeners on different groups.

A change-detecting daemon is this plus a diff: keep a copy of the previous dump, subscribe to the groups that cover what you care about, re-read on notification, and act on the difference. Reading the current state through `request()` rather than from the notification body keeps the logic the same for start-up and for changes, since a notification only describes what changed.

nl80211: wireless

The `nl80211` module talks to the `NL80211` netlink family — the interface `iw` uses. It reads and changes the wireless hardware and its virtual interfaces: phy capabilities, channels and transmit powers, interface types, and the events the kernel emits when any of that changes.

The full reference — every function, constant and resource method — is generated from the module source and published at ucode-lang.org: *the nl80211 module*.

```
import * as nl from "nl80211";
```

The module exports `request`, `waitfor`, `listener`, `error` and a `const` object with 188 constants. As in `rtnl`, commands and attributes are numbers, and the constants are the only way to name them.

```
import * as nl from "nl80211";

printf("GET_WIPHY=%d GET_INTERFACE=%d NEW_INTERFACE=%d\n",
       nl.const.NL80211_CMD_GET_WIPHY, nl.const.NL80211_CMD_GET_INTERFACE,
       nl.const.NL80211_CMD_NEW_INTERFACE);
printf("IFTYPE_STATION=%d IFTYPE_AP=%d IFTYPE_MONITOR=%d\n",
       nl.const.NL80211_IFTYPE_STATION, nl.const.NL80211_IFTYPE_AP,
       nl.const.NL80211_IFTYPE_MONITOR);
```

```
GET_WIPHY=1 GET_INTERFACE=5 NEW_INTERFACE=7
IFTYPE_STATION=2 IFTYPE_AP=3 IFTYPE_MONITOR=6
```

Those numbers are part of the kernel's wireless ABI, so they are the same on every platform — which is more than can be said for anything the answers contain.

Requests

`request(command, flags, payload)` sends a request and returns the decoded answer: an array for a dump, an object for a single get, or `null` on failure. A dump is requested with `NLM_F_DUMP`, exactly as in `rtnl`.

```
import * as nl from "nl80211";

const REQ = nl.const.NLM_F_REQUEST;
const DUMP = nl.const.NLM_F_DUMP;

let phys = nl.request(nl.const.NL80211_CMD_GET_WIPHY, REQ | DUMP, {});

printf("at least one phy: %s\n", length(phys) > 0);
```

```
at least one phy: true
```

A single get names the object it wants. Here is the first radio by index, which is how the kernel addresses phys:

```
import * as nl from "nl80211";

let w = nl.request(nl.const.NL80211_CMD_GET_WIPHY, nl.const.NLM_F_REQUEST, {
wiphy: 0 });

printf("single get: %s, name is a phy name: %s\n", type(w),
index(w.wiphy_name, "phy") == 0);
```

```
single get: object, name is a phy name: true
```

The wiphy entry is large: `wiphy_name` and `wiphy` (its index), `wiphy_bands`, the antenna masks, `wiphy_coverage_class`, `wiphy_frag_threshold`, `wiphy_retry_long` and `wiphy_retry_short`, the `supported_iftypes`, `software_iftypes`, `supported_commands`, `interface_combinations` and `cipher_suites` tables that describe what the hardware can do, and the HT capability mask.

The bands are where the channels live. Each band entry holds a `freqs` array, a `rates` array and the HT capability data; each frequency entry is a channel:

```
import * as nl from "nl80211";

const REQ = nl.const.NLM_F_REQUEST;

let w = nl.request(nl.const.NL80211_CMD_GET_WIPHY, REQ |
nl.const.NLM_F_DUMP, {});
let band = w[0].wiphy_bands[0];

printf("band keys: %s\n", join(",", sort(keys(band))));
printf("first channel: %s\n", join(",", sort(keys(band.freqs[0]))));
```

```
band keys: freqs,ht_ampdu_density,ht_ampdu_factor,ht_capa,ht_mcs_set,rates
first channel: freq,max_tx_power,offset
```

A channel is therefore `{freq, offset, max_tx_power}` — a frequency in MHz, an offset for channels that are not on the 5 MHz grid, and the regulatory transmit limit. How many channels there are, and which bands appear in which order, depends on the radio and on the regulatory domain currently in force.

Interfaces

`NL80211_CMD_GET_INTERFACE` dumps the wireless virtual interfaces. Note what an entry does not contain:

```
import * as nl from "nl80211";

const REQ = nl.const.NLM_F_REQUEST;

let wdevs = nl.request(nl.const.NL80211_CMD_GET_INTERFACE, REQ |
nl.const.NLM_F_DUMP, {});

printf("keys of one entry: %s\n", join(",", sort(keys(wdevs[0]))));
printf("has a name: %s\n", wdevs[0].ifname != null);
```

```
keys of one entry:
4addr,iftype,mac,vif_radio_mask,wdev,wiphy,wiphy_tx_power_level
has a name: false
```

The nl80211 protocol identifies an interface by its wdev index and its MAC address; the name is a property of the network stack, not of the wireless layer. To work with names, join the two dumps on the hardware address — `rtnl` provides the interface list with names and MAC addresses, `nl80211` the wireless side of the same devices:

```
import * as nl from "nl80211";
import * as rtnl from "rtnl";

const REQ = nl.const.NLM_F_REQUEST;

let wdevs = nl.request(nl.const.NL80211_CMD_GET_INTERFACE, REQ |
nl.const.NLM_F_DUMP, {});
let links = rtnl.request(rtnl.const.RTM_GETLINK, REQ | nl.const.NLM_F_DUMP,
{});

let named = filter(wdevs, (w) => length(filter(links, (l) => l.address ==
w.mac)) > 0);

printf("some wireless interface matches a netdev: %s\n", length(named) > 0);
```

```
some wireless interface matches a netdev: true
```

Not every wdev matches. A `P2P_DEVICE` or a monitor interface has no netdev, and a netdev can exist while its wdev is unassigned — which is the normal state while a WiFi configuration is being applied. Matching on the MAC is also not one-to-one on multi-interface radios, so scripts that care about which one they mean should keep track of the wdev index they created.

Events

Changes are announced on the nl80211 multicast groups. `listener(callback, [commands], [groups])` registers a callback driven by `uloop`; `waitfor(commands, timeout)` is the synchronous form for scripts that are not running a loop.

```
import * as nl from "nl80211";

let res = nl.waitfor([nl.const.NL80211_CMD_NEW_INTERFACE], 10);

printf("result=%J error=%J\n", res, nl.error());
```

```
result=null error="No event received"
```

`waitfor()` returns nothing in the ordinary case — the information arrives in the event, and a script that wants state after an event re-reads it with `request()`. That is the pattern worth adopting: treat a notification as a reason to look again rather than as the data itself, so the start-up path and the change path run the same code.

Errors

`error()` describes the last failure and is consumed by reading it. An unknown command is reported as a missing object, and a request that is malformed in a way the module can see — a single get with no object to get — is rejected before it is sent:

```
import * as nl from "nl80211";

let res = nl.request(9999, nl.const.NLM_F_REQUEST, {});

printf("res=%J error=%J\n", res, nl.error());
```

```
res=null error="Object not found"
```

Like its sibling modules, `nl80211` carries no doc comments of its own: the constants, the attribute names in the decoded answers and the accepted payload fields are what the module exports, and the authoritative description of each is the kernel's `linux/nl80211.h`. When a payload field is rejected, the message names it in the same style as `rtnl` does.

uloop: the event loop

`uloop` is the loop that ucode's asynchronous work runs on. It is not an optional extra sitting beside the language: the `uloop` module owns the single event loop that the interpreter drives, and everything asynchronous is dispatched through it — timers, file descriptor readiness, child process termination, signals, `ubus` subscriptions and calls, and the interaction between a script and the debugger. A script that waits on a socket, subscribes to a multicast group, or serves a `ubus` object has to run the loop, because nothing else will dispatch its callbacks.

The full reference — every function, constant and resource method — is generated from the module source and published at ucode-lang.org: *the uloop module*.

```
import * as uloop from "uloop";
```

The module exports `init`, `run`, `end`, `running`, `cancelling`, `done`, `error`, the four watcher constructors `timer`, `interval`, `handle`, `process`, plus `task` and `signal`, and the descriptor-event flags `ULOOP_READ`, `ULOOP_WRITE`, `ULOOP_EDGE_TRIGGER` and `ULOOP_BLOCKING`.

The loop and its lifecycle

A watcher is inert until the loop runs. `run()` executes the loop until there is nothing left to wait for, or until somebody asks it to stop. A callback is entered with the watcher that fired as `this`, which is how a callback reaches the methods of the thing that woke it up.

```
import * as uloop from "uloop";

uloop.timer(10, function () {
        printf("fired while running=%s\n", uloop.running());
        uloop.end();
});

printf("before run: running=%s\n", uloop.running());
uloop.run();
printf("after run: running=%s\n", uloop.running());
```

```
before run: running=false
fired while running=true
after run: running=false
```

`running()` reports whether the loop is currently executing, which is why a timer callback sees `true` and code outside the loop sees `false`. A plain script has nothing to do but call `run()`; a `ubus` object or a socket server registers its watchers and then calls `run()` once, and that call is the program's main body.

`run([timeout])` takes an optional timeout in milliseconds. Without an argument it waits for events indefinitely; with one, it returns after that much time even if events remain pending. The return value is an internal status code from the loop implementation and carries no guaranteed meaning; the state of the loop is what `running()`, `cancelling()` and `end()` are for.

```
import * as uloop from "uloop";

uloop.timer(50, function () {
        print("later\n");
        uloop.end();
});

uloop.run(5);
print("returned before the timer fired\n");

uloop.timer(10, function () {
        print("soon\n");
});
uloop.run();
```

```
returned before the timer fired
soon
later
```

The first `run()` gave up after five milliseconds, leaving the 50 ms timer pending; the second call to `run()` dispatched it, together with the timer created in between, in order of expiry. Pending watchers survive across calls, so a bounded `run(…)` is a way to do work in slices — the pattern behind a script that services events but still has to reach a final statement.

`end()` requests that the loop stop at the earliest safe point. It is the correct exit from inside a callback, where returning normally would simply hand control back to the loop.

```
import * as uloop from "uloop";

let count = 0;

uloop.interval(1, function () {
        count++;

        if (count == 3) {
                uloop.end();
        }
});

uloop.run();
printf("stopped after %d ticks, cancelling=%s\n", count, uloop.cancelling());
```

```
stopped after 3 ticks, cancelling=false
```

`cancelling()` reports whether the loop is in the process of being stopped, which is `true` only while `run()` is still unwinding after `end()`; once `run()` has returned it reads `false` again. It is therefore useful inside teardown callbacks and not as a record of how the loop ended. `done()` performs the loop's post-run cleanup and returns `null`; ordinary scripts do not need to call it, but a long-lived script that restarts the loop after shutting down should.

How the loop stops matters. A script whose loop finishes because its last watcher was dispatched and nothing remained pending can fail to terminate at process exit, whereas a loop stopped by `uloop.end()` exits cleanly:

```
import * as uloop from "uloop";

uloop.timer(10, function () {
        print("last event\n");
        uloop.end();
});

uloop.run();
print("script reaches its final statement\n");
```

```
last event
script reaches its final statement
```

The practical rule is for the last callback of a script's work to call `uloop.end()`, rather than leaving the loop to drain by itself.

`init()` prepares the loop for use. The interpreter initialises it before the script runs, so `init()` matters only after `done()` has torn the loop down. `uloop.error()` reports the error left by a failed watcher operation, in the same message-string style as the `socket` module.

Timers

`timer(timeout, callback)` creates a one-shot timer in milliseconds and starts it immediately. The time left before expiry can be read back with `remaining()`, which answers a number of milliseconds lying between zero and the timeout it was created with:

```
import * as uloop from "uloop";

let limit = 50;
let t = uloop.timer(limit, function () {
        print("one shot\n");
        uloop.end();
});

let rem = t.remaining();
printf("within-bounds=%s\n", rem >= 0 && rem <= limit);
uloop.run();
```

```
within-bounds=true
one shot
```

`remaining()` reports the milliseconds left before expiry, which is why the value printed here is one less than the timeout: the loop had already begun its first pass. `set(ms)` re-arms a timer — from inside its own callback, that is how a repeating timer with jitter is built — and `cancel()` disarms it.

`interval(timeout, callback)` fires repeatedly. It carries one extra piece of state worth knowing about: `expirations()` counts how many times the timer fired since the last check, so a callback that was starved by a long-running operation sees the backlog rather than silently missing ticks.

```
import * as uloop from "uloop";

let iv;

iv = uloop.interval(2, function () {
        printf("tick, expirations=%d\n", iv.expirations());
        iv.cancel();
        uloop.end();
});

uloop.run();
```

```
tick, expirations=1
```

Note the two-step declaration of `iv`. A callback cannot refer to a `let` binding that is still being initialised — writing `let iv = uloop.interval(2, function () { iv.cancel(); })` is refused at compile time with *"Can't access lexical declaration 'iv' before initialization"*. Declaring the name first and assigning afterwards is one way to write a watcher whose callback disarms it.

The simpler way is not to name the watcher at all: a callback is entered with **its own watcher as `this`**, so it can cancel itself through that.

```
import * as uloop from "uloop";

let count = 0;

uloop.interval(1, function () {
        count++;

        if (count == 3) {
                this.cancel();
                printf("stopped after %d ticks, remaining=%d\n", count,
this.remaining());
                uloop.end();
        }
});

uloop.run();
uloop.done();
```

```
stopped after 3 ticks, remaining=-1
```

`remaining()` reports `-1` once the timer is disarmed. Every watcher callback in this module receives its watcher as `this` — timers, intervals and fd handles alike — which is also how a callback deletes a handle it was never given a name for.

File descriptors

`handle(fd, callback, events)` registers a descriptor with the loop. The descriptor comes from `fileno()` on a `socket` resource, an `io` handle, or any other file descriptor.

```
import * as uloop from "uloop";
import * as socket from "socket";

let pair = socket.pair();
let w;

w = uloop.handle(pair[1].fileno(), function (src, events) {
        print("descriptor ready\n");
        w.delete();
        pair[1].close();
        uloop.end();
}, uloop.ULOOP_READ);

printf("watching a real descriptor: %s\n", w.fileno() > 2);
pair[0].send("wakeup");
uloop.run();
```

```
watching a real descriptor: true
descriptor ready
```

As with timers, the callback can reach its watcher through `this`, so `this.fileno()` and `this.delete()` work and the script does not have to keep `w` around just to close it. The first callback argument is not the watcher.

The event mask is built from `ULOOP_READ` and `ULOOP_WRITE`; `ULOOP_EDGE_TRIGGER` requests edge-triggered notification and `ULOOP_BLOCKING` clears the non-blocking flag on the descriptor. The callback receives the watcher and the event mask, and for a normal readable notification the mask is not a reliable indicator of which condition fired — the code inside the callback should read the descriptor and let the read result say what happened, exactly as it would after a `poll()`.

The watcher keeps the descriptor registered until it is removed. `delete()` unregisters it; `handle()` returns the descriptor it was given. A socket closed underneath a live watcher is a use-after-free hazard in any event-driven program, so the ordering that keeps a script safe is to delete the watcher before closing the descriptor.

Child processes

`process(executable, [args], [env], callback)` starts a child and reaps it in the background — the loop's `SIGCHLD` handling means the script never sees a zombie, and never blocks waiting for the child.

```
import * as uloop from "uloop";

let p;

p = uloop.process("/bin/sh", ["-c", "exit 7"], null, function (code) {
        printf("callback sees the same watcher: %s\n", this.pid() ==
p.pid());
        printf("child exited with %d\n", code);
        uloop.end();
});

printf("started pid %s\n", p.pid() > 0);
uloop.run();
```

```
started pid true
callback sees the same watcher: true
child exited with 7
```

The pid itself is whatever the kernel hands out, so compare it rather than print it — and note that `p` is declared before it is assigned, because the callback reads it (the same initialisation rule as the interval above).

The callback's single argument is the child's exit code, and as everywhere in this module the callback is entered with its watcher as `this`, so the child can be inspected or detached from inside: `this.pid()` is the child's pid and `this.delete()` releases the watcher.

The arguments are positional and all four are expected — the callback is the **fourth**, after `env`. Passing `uloop.process(exe, args, callback)` makes the callback the child's environment: no completion handler is registered, the watcher stays registered, and a loop that is waiting for it never finishes. Pass `null` for `env` when there is nothing to change.

A script that launched several children still distinguishes them by closing each callback over its own continuation, since `this` identifies the watcher, not the purpose it was created for.

This is the asynchronous counterpart to `system()`. Where `system()` blocks for the duration, `uloop.process()` returns immediately and reports the result through the loop, which is what a daemon that must keep answering `ubus` calls while it runs `sysctl` or `fw4 reload` needs. The `env` argument, when given, replaces the environment of the child; pass `null` to inherit.

`task` is the lower-level primitive behind it: `uloop.task(callback)` forks, runs the callback in the child and delivers its result to a completion callback in the parent. Its methods are `pid()`, `kill(signo)` and `finished()`. Because it forks the whole interpreter state, it is used for isolating work that must not block the loop, and it needs more care than `process()` about what the child is allowed to touch.

Signals

`uloop.signal(signal, callback)` watches a signal through the loop rather than through an asynchronous signal handler, so callbacks run at a safe point with the interpreter in a consistent state. The callback receives no argument; the signal number is available from the watcher's `signo()` method, and `delete()` stops watching.

```
import * as uloop from "uloop";

let w;

w = uloop.signal("USR1", function () {
        printf("received signal %d\n", w.signo());
        w.delete();
        uloop.end();
});

uloop.signal("TERM", function () {
        print("terminated\n");
        uloop.end();
});

system("(sleep 0.05; kill -USR1 $PPID) &");
uloop.run();
```

```
received signal 10
```

Compare this with the core `signal()` function described in the processes chapter, which installs an interpreter-level handler that runs without a loop. `uloop.signal()` is the better choice in a program that already runs a loop: signal delivery is serialised with all other callbacks, and a handler that touches interpreter state cannot interrupt another callback in mid-flight.

Putting it together

A small service is the sum of these pieces: it watches a `ubus` object or a socket for input, arms timers for periodic work, spawns children without blocking, and shuts down on signal.

```
import * as uloop from "uloop";

let state = { polls: 0, stopping: false };

uloop.signal("TERM", function () {
        state.stopping = true;
        uloop.end();
});

let poll;

poll = uloop.interval(5, function () {
        state.polls++;

        if (state.polls >= 3) {
                poll.cancel();

                uloop.process("/bin/sh", ["-c", "exit 0"], null, function
(code) {
                        printf("polled %d times, cleanup rc=%d\n",
state.polls, code);
                        uloop.end();
                });
        }
});

uloop.run();
printf("stopping=%s\n", state.stopping);
```

```
polled 3 times, cleanup rc=0
stopping=false
```

The shape is worth holding on to, because it recurs wherever ucode runs for a long time: watchers created at the top level, `run()` as the last statement, and shutdown driven by a signal watcher that either cancels the periodic watchers or simply ends the loop. Under `uwsd` or an init script, the same structure answers `SIGTERM` and exits cleanly without a zombie child or a pending timer keeping the process alive.

ubus

`ubus` is OpenWrt's inter-process bus: a small broker daemon, `ubusd`, over a Unix socket, on which processes publish named objects with typed methods, call each other's methods, and exchange notifications and events. The module is a binding to `libubus`, and through it to the event loop — an ubus connection is registered with `uloop` the moment it is opened, so every reply, notification and event is delivered while `uloop.run()` runs. That coupling is the first thing to understand: an ubus program is an event-loop program.

The full reference — every function, constant and resource method — is generated from the module source and published at ucode-lang.org: *the ubus module*.

```
import * as ubus from "ubus";
```

The examples in this chapter need a bus. They are written against a `ubusd` listening on `/tmp/ub.sock`, started as `ubusd -s /tmp/ub.sock`; an OpenWrt system has its bus on a socket of its own, which the examples reach by leaving the path out. Only the few that need no daemon — the module's shape, the status codes, the error reporting — are self-contained.

The objects of the module

Connecting produces a resource, and almost everything else is a method on it:

Resource	Created by	Carries
`ubus.connection`	`connect()`,	a connection to the bus
`ubus.object`	`connection.publish()`	an object published on the bus
`ubus.request`	a method call arriving at a published object	the incoming call: data, identity, reply
`ubus.deferred`	`connection.defer()`	a call awaiting its reply
`ubus.notify`	`object.notify()`	a notification in flight
`ubus.subscriber`	`connection.subscriber()`	an interest in object notifications
`ubus.listener`	`connection.listener()`	an interest in events
`ubus.channel`	`open_channel()`, `request.new_channel()`	a stream of data alongside calls

```
import * as ubus from "ubus";

printf("connect is a function: %s\n", type(ubus.connect) == "function");
printf("open_channel is a function: %s\n", type(ubus.open_channel) ==
"function");
printf("status codes are integers: %s\n", type(ubus.STATUS_OK) == "int");
```

```
connect is a function: true
open_channel is a function: true
status codes are integers: true
```

The names of the connection's methods also appear at module scope. Those module-level copies use the default socket — `/var/run/ubus/ubus.sock` — and not the connection you opened, so a program that connects anywhere else must call the methods on the connection:

```
let c = ubus.connect("/tmp/ub.sock");

printf("connection methods: %s\n", type(c.list) == "function");
printf("module-level list(): %J\n", ubus.list());
printf("error after that: %J\n", ubus.error());
```

```
connection methods: true
module-level list(): null
error after that: "Unable to connect to ubus socket"
```

Connecting

```
import * as ubus from "ubus";

let c = ubus.connect("/tmp/ub.sock");

printf("connected: %s, type=%s\n", c != null, type(c));
printf("disconnect: %J\n", c.disconnect());
```

```
connected: true, type=resource
disconnect: true
```

`connect([socket], [timeout])` takes the socket path and a timeout in seconds for subsequent operations, which defaults to 30. With the path omitted the default socket is used. A failure to connect returns `null` and leaves the reason in `error()`, which is consumed by reading it:

```
import * as ubus from "ubus";

let c = ubus.connect("/tmp/ucode-manual-absent.sock");

printf("connection: %J\n", c);
printf("error: %J\n", ubus.error());
printf("error again: %J\n", ubus.error());
```

```
connection: null
error: "Unable to connect to ubus socket"
error again: null
```

Listing and calling

`list()` returns the names of the objects on the bus; `list(name)` returns the signatures of one object's methods, with each argument's type given as a blob type code — `3` is a string:

```
let c = ubus.connect("/tmp/ub.sock");

printf("%J\n", sort(c.list()));
printf("%J\n", c.list("manual.demo"));
```

```
[ "manual.demo" ]
[ { "hello": { "name": 3 }, "notes": { } } ]
```

`call(object, method[, data[, return[, fd[, fd_cb]]]])` invokes a method and waits for the reply. The object may be named or given by numeric id, and `data` is an object whose fields become the method's arguments:

```
let c = ubus.connect("/tmp/ub.sock");

printf("%J\n", c.call("manual.demo", "hello", { name: "world" }));
```

```
{ "message": "Hello, world" }
```

An object may answer a single call more than once. The fourth argument says what to do with the several replies: `"single"` (the default) keeps the first, `"multiple"` collects them into an array, `"ignore"` discards them and returns `null`:

```
let c = ubus.connect("/tmp/ub.sock");

printf("single:   %J\n", c.call("manual.demo", "multi", {}));
printf("multiple: %J\n", c.call("manual.demo", "multi", {}, "multiple"));
printf("ignore:   %J\n", c.call("manual.demo", "multi", {}, "ignore"));
```

```
single:   { "n": 1 }
multiple: [ { "n": 1 }, { "n": 2 }, { "n": 3 } ]
ignore:   null
```

Failure is reported by a `null` return plus a message from `error()`, which combines the status of the bus with the operation that failed:

```
let c = ubus.connect("/tmp/ub.sock");

printf("%J %J\n", c.call("absent.object", "get", {}), ubus.error());
printf("%J %J\n", c.call("manual.demo", "nosuchmethod", {}), ubus.error());
printf("%J %J\n", c.call("manual.demo", "hello", { name: 42 }),
ubus.error());
printf("%J %J\n", c.call("manual.demo", "refuse", {}), ubus.error());
```

```
null "Not found: Failed to resolve object name 'absent.object'"
null "Method not found: Failed to invoke function 'nosuchmethod' on object
'manual.demo'"
null "Invalid argument: Failed to invoke function 'hello' on object
'manual.demo'"
null "Permission denied: Failed to invoke function 'refuse' on object
'manual.demo'"
```

The third line is worth separating from the others: the callee declared `args: { name: "string" }`, so the mismatched call was rejected by the bus layer and the handler never ran. A method can signal any of these statuses itself, with `error()` on the request object.

Publishing an object

`publish(name[, methods[, subscribe_callback]])` registers an object. Each method is an object with a `call` function and an optional `args` object declaring the arguments the method accepts. The declaration works by example: the **value** given for each name fixes its type, and the value itself is discarded. A number exemplar declares a number, a boolean a boolean, an array an array, an object a table, a string a string — whatever the string says:

```
import * as ubus from "ubus";
import * as uloop from "uloop";

let c = ubus.connect("/tmp/ub.sock");

let obj = c.publish("manual.demo", {
        hello: {
                call: function (req) {
                        req.reply({ message: "Hello, " + req.args.name });
                },
                args: { name: "string" }
        },
        notes: {
                call: function (req) {
                        req.reply({ list: [1, 2, 3], seen: true });
                }
        },
        refuse: {
                call: function (req) {
                        req.error(ubus.STATUS_PERMISSION_DENIED);
                }
        }
});

printf("published: %s, type=%s\n", obj != null, type(obj));
uloop.run();
```

```
published: true, type=resource
```

The declaration is checked on the bus, in the caller's process and before the handler is entered, so the types involved are the bus's own — blob message types — and `list(name)` reports them by number:

Exemplar	Declares	Code
`"text"`, any string	string	3
`0`, any integer but 8, 16 or 64	32-bit integer	5
`8`	8-bit integer	7
`16`	16-bit integer	6
`64`	64-bit integer	4
`false`	boolean, carried as an 8-bit integer	7
`1.0`	double	8
`[]`	array	1
`{}`	nested table	2

Two consequences follow, and both are easy to get wrong.

First, an entry is a value, not a type name. The idiom `{ name: "string", count: "integer" }` reads as a list of types and is in fact two string declarations, because both exemplars are strings. An integer argument is declared by an integer — `count: 0`.

Second, the check is exact, in both directions. A number sent by a ucode caller is encoded as a 32-bit integer if it fits in one, and as a 64-bit integer otherwise, so an argument declared with the exemplar `0` takes a value like a million but refuses one that needs 32 bits unsigned, and an argument declared `64` is the opposite case:

```
// args: { small: 0, big: 64, b: false, d: 1.0 }
printf("%J\n", c.call("manual.demo", "sized", { small: 1000000 }) != null);
printf("%J\n", c.call("manual.demo", "sized", { small: 2 ** 31 }) != null);
printf("%J\n", c.call("manual.demo", "sized", { big: 1 }) != null);
printf("%J\n", c.call("manual.demo", "sized", { big: 2 ** 40 }) != null);
```

```
true
false
false
true
```

A double is not satisfied by an integer — `1` is not read as `1.0` — and a boolean exemplar accepts only booleans, since booleans travel as 8-bit integers and the number `1` does not. The practical rule is to declare integers with an exemplar of `0`, to reserve `8`, `16` and `64` for objects that must interoperate with a C service using those exact widths, and to keep doubles doubles:

Undeclared arguments are refused too: a payload naming a field the method did not declare fails the call rather than slipping the extra field through unnoticed. A call rejected by the declaration never reaches the handler. The caller sees the failure described in the previous section — `Invalid argument: Failed to invoke function 'name' on object 'name'` — and `error(true)` gives the status as `2`, `UBUS_STATUS_INVALID_ARGUMENT`.

The handler is called with the request object as its only argument. The incoming data is in `req.args`; `req.info` describes the call, with the caller's credentials under `info.acl`:

```
call: function (req) {
        print(sprintf("%.J", req.args) + "\n");
        print(sprintf("%.J", req.info) + "\n");
        req.reply({ message: "Hello, " + req.args.name });
}
```

```
{
        "name": "world"
}
{
        "acl": {
                "user": "jow",
                "group": "jow"
        }
}
```

Inside the handler, `this` is the published object, which lets the methods of one object share state through closures over the object's own declarations. The module's own documentation shows the handler taking the request as a second argument after the message; the implementation passes the request alone, and the message arrives as `req.args`.

The request object answers the call:

Method	Effect
`reply([data[, rcode]])`	reply with data and status `STATUS_OK`; a negative `rcode` means further replies follow
`error([rcode])`	finish the call with an error status and no data
`defer()`	keep the call open so that `reply()` can be called later, from a timer or a callback
`get_fd()`, `set_fd(fd)`	read or hand on a file descriptor accompanying the call
`new_channel(...)`	open a channel to the caller

A method that neither replies nor defers leaves the caller waiting until it times out, and a method that throws is handled by the exception handler described below.

An object published with a third argument is told when its subscriber set changes. The callback is invoked with no arguments — `subscribed()` answers the question — and it can fire while `publish()` is still executing, so a callback that refers to the object by name reads a variable that has not been assigned yet. Declaring the variable before the call and assigning it after avoids the question entirely:

```
let obj = null;

obj = c.publish("manual.demo", methods, function () {
        print("subscriber set changed: " + obj.subscribed() + "\n");
});
```

```
subscriber set changed: true
```

Writing `let obj = c.publish(...)` with the callback referring to `obj` raises `Can't access lexical declaration 'obj' before initialization` for each early invocation.

`remove()` on the object takes it off the bus, which the remaining clients see as an empty `list()`:

```
printf("removed: %J\n", obj.remove());
printf("bus now: %J\n", c.list());
```

```
removed: true
bus now: [ ]
```

Notifications

A published object can notify its subscribers, which is a different transaction from a call: there is no reply, and delivery is to a set of subscribers rather than to one caller.

```
let n = obj.notify("reload", { source: "manual" });

printf("notify returns a %s\n", type(n));
```

```
notify returns a resource
```

`notify(type[, data][, data_cb][, status_cb][, cb])` returns a `ubus.notify` resource tracking the delivery; the optional callbacks observe per-subscriber data, per-subscriber status, and overall completion, and the resource has `completed()` and `abort()`.

On the receiving side, `subscriber(notify_callback, remove_callback[, patterns])` registers an interest. With `patterns` — an array of globs — the bus subscribes the caller to matching objects as they appear. The notification callback is called with one argument, a request-shaped object carrying `type`, `data` and `info`; the removal callback is called with the id of the object that went away:

```
import * as ubus from "ubus";
import * as uloop from "uloop";

let c = ubus.connect("/tmp/ub.sock");

let s = c.subscriber(
        function (req) {
                print("notification " + req.type + " " + sprintf("%.J",
req.data) + "\n");
        },
        function (id) {
                print("object " + id + " is gone\n");
        },
        ["manual.demo"]
);

uloop.timer(3000, () => uloop.end());
uloop.run();
```

```
notification reload {
        "source": "manual"
}
```

The subscriber resource has `subscribe(object)`, `unsubscribe(object)` and `remove()`.

Events

Events need no publisher object and no registration on the sending side: a program sends a typed message to the bus, and whoever is listening receives it. The pattern matching `*` and `?` is available in the listener's pattern.

```
let l = c.listener("manual.*", function (type, data) {
        print("event: " + type + " " + sprintf("%.J", data) + "\n");
});

uloop.timer(2000, () => uloop.end());
uloop.run();
```

```
event: manual.ping {
        "seq": 7
}
```

Sending one is `event(type[, data])`, which returns `true` once the message has been handed to the bus — delivery is the listeners' concern:

```
printf("sent: %J\n", c.event("manual.ping", { seq: 7 }));
```

```
sent: true
```

A listener is registered against a connection, and `remove()` unregisters it. Note that the event callback receives the type and the data as two separate arguments, where a subscriber's callback receives them as properties of one object.

Calls without waiting

`defer(object, method[, data[, cb[, data_cb[, fd[, fd_cb]]]]])` sends a call and returns immediately with a `ubus.deferred`. The reply arrives through `cb` when the event loop gets to it; the resource itself is for status and cancellation:

```
let d = c.defer("manual.demo", "slow", {}, function (rc, data) {
        print("rc=" + rc + " reply " + sprintf("%.J", data) + "\n");
});

printf("type=%s completed=%J\n", type(d), d.completed());

uloop.timer(500, () => {
        printf("completed=%J\n", d.completed());
        uloop.end();
});
uloop.run();
```

```
rc=0 reply {
        "elapsed": 240
}
type=resource completed=false
completed=true
```

The callback is called with the return code first and the reply data second, and is entered with the deferred resource as `this`. `completed()` says whether the reply has arrived — `false` until the loop has run — `abort()` gives up on it, and `await()` waits for it. Since the reply is delivered to the callback, the deferred resource itself carries no result value. Deferring several calls and running the loop is how a program asks the same question of many services at once.

File descriptors and channels

A call can carry a file descriptor, and so can a reply. Pass `fileno` as the fifth argument to `call()`, or `set_fd()` on a request object while handling a call, and read the arriving one with `get_fd()`. For a longer exchange over one descriptor, `new_channel()` on the request and `open_channel(fd, cb[, disconnect_cb[, timeout]])` on the receiving side turn a descriptor into a `ubus.channel`, over which `request()` and `defer()` work as they do on a connection, with the callback seeing each message as it arrives.

Ownership of the descriptor follows its origin: given a plain integer file descriptor, `open_channel()` takes it over and closes it on `disconnect()`; given a resource that has a `fileno()` method — a file from `fs.open()`, a socket — the resource keeps ownership and the channel merely detaches.

Status codes

The status codes are the ones defined by the ubus protocol, exported as `STATUS_*`:

```
import * as ubus from "ubus";

printf("ok=%d continue=%d invalid argument=%d method not found=%d\n",
       ubus.STATUS_OK, ubus.STATUS_CONTINUE, ubus.STATUS_INVALID_ARGUMENT,
       ubus.STATUS_METHOD_NOT_FOUND);
printf("not found=%d permission denied=%d timeout=%d unknown=%d\n",
       ubus.STATUS_NOT_FOUND, ubus.STATUS_PERMISSION_DENIED,
ubus.STATUS_TIMEOUT,
       ubus.STATUS_UNKNOWN_ERROR);
```

```
ok=0 continue=-1 invalid argument=2 method not found=3
not found=4 permission denied=6 timeout=7 unknown=9
```

`STATUS_CONTINUE` is the value a handler uses to say "more replies follow"; the rest describe failures. `error()` renders the last one in words, and `error(true)` returns its number instead:

```
let c = ubus.connect("/tmp/ub.sock");

c.call("manual.demo", "nosuchmethod", {});
printf("method not found: %J %J\n", ubus.error(), ubus.error());
```

```
method not found: 3 null
```

Reading it in either form consumes it, which is why the second read above is `null`; ask for the text and the number in one expression if both are wanted.

Errors, and what ends the loop

Two rules cover error handling in this module. `error()` reports the last failure and is consumed by reading it, so read it in the same breath as the failed result:

```
import * as ubus from "ubus";

ubus.connect("/tmp/ucode-manual-absent.sock");
printf("%J %J\n", ubus.error(), ubus.error());
```

```
"Unable to connect to ubus socket" null
```

Second, an exception raised inside a callback — a method handler, a notification callback, a listener — does not propagate to the code that ran the loop. It is passed to the handler registered with `guard(fn)`, and where there is none, it ends the event loop. Since the loop is what keeps a service alive, a callback that throws reliably stops the program, which is a reason to keep handlers short and to install a guard:

```
import * as ubus from "ubus";

printf("no guard installed: %J\n", ubus.guard());

let handler = function (ex) { print("handled: " + ex + "\n"); };

ubus.guard(handler);
printf("guard installed: %s\n", ubus.guard() == handler);
```

```
no guard installed: null
guard installed: true
```

A service in full

The shape of a typical ubus service — publish, serve, and stop when the bus goes away:

```
import * as ubus from "ubus";
import * as uloop from "uloop";

let c = ubus.connect();

if (!c) {
        fprintf(stderr, "cannot connect: %s\n", ubus.error());
        exit(1);
}

let count = 0;
let obj = null;

obj = c.publish("manual.counter", {
        increment: {
                call: function (req) {
                        count += req.args.by;
                        req.reply({ count: count });
                },
                args: { by: 0 }
        },
        get: {
                call: function (req) {
                        req.reply({ count: count });
                }
        }
}, function () {
        obj.notify("subscribers", { subscribed: obj.subscribed() });
});

uloop.signal("TERM", () => {
        obj.remove();
        c.disconnect();
        uloop.end();
});

uloop.run();
```

```
import * as ubus from "ubus";

let c = ubus.connect();
let r = c.call("manual.counter", "increment", { by: 3 });

if (r) {
        printf("count is now %d\n", r.count);
}
```

```
count is now 3
```

uci

`libuci` is OpenWrt's configuration library: the `/etc/config` files that describe the network, the firewall and everything else, plus the staged-change model that lets a program modify them without rewriting them until it decides the result is good. The `uci` module binds it.

The full reference — every function, constant and resource method — is generated from the module source and published at ucode-lang.org: *the uci module*.

```
import * as uci from "uci";
```

The module exports exactly two names: `cursor` and `error`.

```
import * as uci from "uci";

printf("exports: %s\n", join(",", sort(keys(uci))));
```

```
exports: cursor,error
```

All the work happens on a cursor, which is a context plus the state of the packages it has read. `uci.cursor([confdir], [savedir], [conf2dir], [flags])` creates one; every path is optional and defaults to the compiled-in `/etc/config` and `/tmp/state` areas, and the directories need not exist yet. Pointing a cursor somewhere else is what makes it possible to experiment with real UCI files, and the examples in this chapter do exactly that — they write a configuration into a temporary directory and operate on it, so nothing depends on the machine running them.

```
import * as uci from "uci";
import * as fs from "fs";

const DIR = "/tmp/ucode-manual-uci";
const SAVE = "/tmp/ucode-manual-uci-save";

fs.mkdir(DIR, 0o755);
fs.writefile(DIR + "/demo", `config demo 'main'
        option name 'router'
        list ports 'lan'
        list ports 'wan'
        option mtu '1500'

config demo 'spare'
        option name 'spare'
`);

let s = uci.cursor(DIR, SAVE);

printf("packages: %s\n", join(",", s.configs()));
printf("main is of type: %s\n", s.get("demo", "main"));
printf("name=%s mtu=%s ports=%J\n", s.get("demo", "main", "name"),
       s.get("demo", "main", "mtu"), s.get("demo", "main", "ports"));
printf("missing option=%J error=%J\n", s.get("demo", "main", "nope"),
s.error());
```

```
packages: demo
main is of type: demo
name=router mtu=1500 ports=[ "lan", "wan" ]
missing option=null error="Entry not found"
```

Reading

`get()` is one function with three shapes, and the arity decides what comes back: with a package and a section it returns the section's **type**, with a package, a section and an option it returns the option's value, and a package alone is rejected.

```
import * as uci from "uci";
import * as fs from "fs";

const DIR = "/tmp/ucode-manual-uci";
const SAVE = "/tmp/ucode-manual-uci-save";

fs.mkdir(DIR, 0o755);
fs.writefile(DIR + "/demo", "config demo 'main'\n\toption name 'router'\n");

let s = uci.cursor(DIR, SAVE);

printf("pkg only=%J error=%J\n", s.get("demo"), s.error());
printf("section=%J\n", s.get("demo", "main"));
printf("option=%J\n", s.get("demo", "main", "name"));
```

```
pkg only=null error="Invalid argument"
section="demo"
option="router"
```

Values are always strings, or arrays of strings for `list` options. UCI stores no types — a number in a configuration file is a sequence of digits in a file, and `s.get(..., "mtu")` returns `"1500"`, not `1500`. Anything numeric has to be converted explicitly, and a value that is not a number at all converts to `null`:

```
import * as uci from "uci";
import * as fs from "fs";

const DIR = "/tmp/ucode-manual-uci";
const SAVE = "/tmp/ucode-manual-uci-save";

fs.mkdir(DIR, 0o755);
fs.writefile(DIR + "/demo", "config demo 'main'\n\toption mtu
'1500'\n\toption weird 'not a number'\n");

let s = uci.cursor(DIR, SAVE);

printf("mtu type=%s\n", type(s.get("demo", "main", "mtu")));
printf("as number=%d weird as number=%s\n", s.get("demo", "main", "mtu") * 1,
       s.get("demo", "main", "weird") * 1);
```

```
mtu type=string
as number=1500 weird as number=NaN
```

`get_all(package)` returns the whole package as a table keyed by section name. Each section carries its own metadata under dotted keys, which is how a script tells an anonymous section from a named one and learns the order the file used:

```
import * as uci from "uci";
import * as fs from "fs";

const DIR = "/tmp/ucode-manual-uci";
const SAVE = "/tmp/ucode-manual-uci-save";

fs.mkdir(DIR, 0o755);
fs.writefile(DIR + "/demo", "config demo 'main'\n\toption name 'router'\n");

let s = uci.cursor(DIR, SAVE);
let all = s.get_all("demo");

printf("sections: %s\n", join(",", keys(all)));
printf("section metadata: %s\n", join(",", sort(keys(all.main))));
printf(".type=%J .index=%J .anonymous=%J\n", all.main[".type"],
all.main[".index"],
       all.main[".anonymous"]);
```

```
sections: main
section metadata: .anonymous,.index,.name,.type,name
.type="demo" .index=0 .anonymous=false
```

`foreach(package, type, callback)` walks the sections of a package, calling the callback with a section object of the same shape. The second argument filters by section type, and `null` in that position means no filter at all: the walk then covers every section in the package, whatever its type.

```
import * as uci from "uci";
import * as fs from "fs";

const DIR = "/tmp/ucode-manual-uci";
const SAVE = "/tmp/ucode-manual-uci-save";

fs.mkdir(DIR, 0o755);
fs.writefile(DIR + "/demo", "config demo 'main'\n\toption name
'router'\n\nconfig other 'spare'\n\toption name 'spare'\n");

let s = uci.cursor(DIR, SAVE);

let all = [];

s.foreach("demo", null, (section) => {
        all[length(all)] = section[".type"];
});

printf("unfiltered: %s\n", join(",", sort(all)));

let names = [];

s.foreach("demo", "demo", (section) => {
        names[length(names)] = section[".name"];
});

printf("filtered to demo: %s\n", join(",", sort(names)));
```

```
unfiltered: demo,other
filtered to demo: main
```

The arguments are positional, so dropping the filter is not the same as writing `null` in its place. Given two arguments, the callback is read as the type filter and no callback is left, so the walk is abandoned at the argument check:

```
import * as uci from "uci";
import * as fs from "fs";

const DIR = "/tmp/ucode-manual-uci";
const SAVE = "/tmp/ucode-manual-uci-save";

fs.mkdir(DIR, 0o755);
fs.writefile(DIR + "/demo", "config demo 'main'\n\toption name
'router'\n\nconfig other 'spare'\n\toption name 'spare'\n");

let s = uci.cursor(DIR, SAVE);

let seen = 0;
let rv = s.foreach("demo", (section) => {
        seen++;
});

printf("callback ran: %d\n", seen);
printf("return=%J error=%J\n", rv, s.error());

let n = 0;

s.foreach("demo", "nosuchtype", (section) => {
        n++;
});

printf("a filter matching nothing: ran=%d error=%J\n", n, s.error());
```

```
callback ran: 0
return=null error="Invalid argument"
a filter matching nothing: ran=0 error=null
```

The two cases differ in kind. A type that matches nothing is an ordinary empty walk: it returns `false`, and `error()` has nothing to report. A misplaced argument is rejected before the walk begins — and since `error()` is the only channel it uses, a short call looks like a package with nothing in it.

`get_first(package, type, option)` is the same lookup in single-section form, except that here the type is not optional: `null` in that position is refused rather than read as "any type".

```
import * as uci from "uci";
import * as fs from "fs";

const DIR = "/tmp/ucode-manual-uci";
const SAVE = "/tmp/ucode-manual-uci-save";

fs.mkdir(DIR, 0o755);
fs.writefile(DIR + "/demo", "config demo 'main'\n\toption name
'router'\n\nconfig other 'spare'\n\toption name 'spare'\n");

let s = uci.cursor(DIR, SAVE);

printf("unfiltered get_first: %J %J\n", s.get_first("demo", null, "name"),
s.error());
printf("filtered get_first: %J\n", s.get_first("demo", "other", "name"));
```

```
unfiltered get_first: null "Invalid argument"
filtered get_first: "spare"
```

Changing

Nothing a cursor does takes effect on disk immediately. `set()`, `delete()`, `add()`, `rename()`, `list_append()` and `list_remove()` modify the in-memory copy of the package, and `changes()` reports what has diverged:

```
import * as uci from "uci";
import * as fs from "fs";

const DIR = "/tmp/ucode-manual-uci";
const SAVE = "/tmp/ucode-manual-uci-save";

fs.mkdir(DIR, 0o755);
fs.writefile(DIR + "/demo", "config demo 'main'\n\toption name 'router'\n");

let s = uci.cursor(DIR, SAVE);

s.set("demo", "main", "name", "gateway");

printf("in memory: %J\n", s.get("demo", "main", "name"));
printf("changes: %J\n", s.changes("demo"));
printf("file still: %J\n", index(fs.readfile(DIR + "/demo"), "router") !=
null);
```

```
in memory: "gateway"
changes: { "demo": [ [ "set", "main", "name", "gateway" ] ] }
file still: true
```

Each change is a command tuple — the operation, the section, the option and the new value where there is one — which is why `changes()` output reads like the `uci` command line. Renaming a section and deleting an option produce `rename` and `remove` tuples the same way.

The staged model exists so that a set of edits can be abandoned as a unit. `revert(package)` throws away the differences:

```
import * as uci from "uci";
import * as fs from "fs";

const DIR = "/tmp/ucode-manual-uci";
const SAVE = "/tmp/ucode-manual-uci-save";

fs.mkdir(DIR, 0o755);
fs.writefile(DIR + "/demo", "config demo 'main'\n\toption name 'router'\n");

let s = uci.cursor(DIR, SAVE);

s.set("demo", "main", "name", "gateway");
s.delete("demo", "main", "missing");
s.revert("demo");

printf("after revert: %J changes: %J\n", s.get("demo", "main", "name"),
s.changes("demo"));
```

```
after revert: "router" changes: { }
```

Lists are edited by value rather than by position, since the on-disk form has no indices to offer:

```
import * as uci from "uci";
import * as fs from "fs";

const DIR = "/tmp/ucode-manual-uci";
const SAVE = "/tmp/ucode-manual-uci-save";

fs.mkdir(DIR, 0o755);
fs.writefile(DIR + "/demo", "config demo 'main'\n\tlist ports 'lan'\n\tlist
ports 'wan'\n");

let s = uci.cursor(DIR, SAVE);

s.list_append("demo", "main", "ports", "wlan");
printf("appended: %J\n", s.get("demo", "main", "ports"));

s.list_remove("demo", "main", "ports", "wan");
printf("removed by value: %J\n", s.get("demo", "main", "ports"));
```

```
appended: [ "lan", "wan", "wlan" ]
removed by value: [ "lan", "wlan" ]
```

`add(package, type)` creates a section of the given type with a generated name, and returns that name. Generated names are unique but not predictable, so compare them, never print them. Note that `add()` is the one entry point that does not load the package on demand — it reports `Entry not found` against a package the cursor has not read yet, where `get()` and `set()` would have loaded it:

```
import * as uci from "uci";
import * as fs from "fs";

const DIR = "/tmp/ucode-manual-uci";
const SAVE = "/tmp/ucode-manual-uci-save";

fs.mkdir(DIR, 0o755);
fs.writefile(DIR + "/demo", "config demo 'main'\n\toption name 'router'\n");

let s = uci.cursor(DIR, SAVE);

printf("add before load=%J error=%J\n", s.add("demo", "wifi"), s.error());
s.load("demo");

let name = s.add("demo", "wifi");

printf("add after load: generated name=%s\n", index(name, "cfg") == 0 &&
length(name) == 9);
printf("section exists=%s of type %J\n", name in s.get_all("demo"),
s.get("demo", name));

s.set("demo", name, "ssid", "net");

printf("options can be added to it: %J\n", s.get("demo", name, "ssid"));
```

```
add before load=null error="Entry not found"
add after load: generated name=true
section exists=true of type "wifi"
options can be added to it: "net"
```

Committing

There are two ways to make changes durable, and they are not interchangeable. `save(package)` writes the pending delta to the save directory and clears the change set, leaving the configuration file alone. `commit(package)` rewrites the configuration file itself:

```
import * as uci from "uci";
import * as fs from "fs";

const DIR = "/tmp/ucode-manual-uci";
const SAVE = "/tmp/ucode-manual-uci-save";

fs.mkdir(DIR, 0o755);
fs.writefile(DIR + "/demo", "config demo 'main'\n\toption name 'router'\n");

let s = uci.cursor(DIR, SAVE);

s.set("demo", "main", "name", "gateway");
s.save("demo");

printf("after save: changes=%J file untouched=%s\n", s.changes("demo"),
       index(fs.readfile(DIR + "/demo"), "router") != null);

s.set("demo", "main", "name", "border");
s.commit("demo");

let fresh = uci.cursor(DIR, SAVE);

printf("after commit: rc=%J file now has it=%s\n", true,
       index(fs.readfile(DIR + "/demo"), "border") != null);
printf("a new cursor reads: %J\n", fresh.get("demo", "main", "name"));
```

```
after save: changes={ } file untouched=true
after commit: rc=true file now has it=true
a new cursor reads: "border"
```

The delta in the save directory is what makes a staged configuration survive a program exiting: another process opening the same directories sees the saved delta applied on top of the file. That is the mechanism behind `uci show` displaying changes that have not been committed.

The delta is a text file, one operation per line, with the package name qualifying every entry and a leading character naming the operation — nothing for setting an option, `+` for adding a section, `-` for removing something, `|` for appending to a list. A `save()` of a few edits therefore leaves something like this behind:

```
+demo.cfg021cd4='wifi'
demo.main.name='gw'
|demo.main.ports='wan'
-demo.main.ports
```

`changes()` reports the same information in ucode data structures rather than as text, which is usually what a script wants; the file form matters when reading a leftover delta by hand. Committing empties the delta file — it stays in place, at zero length, rather than being removed.

The name a generated section gets is `cfg` followed by eight hexadecimal digits, built by `libuci` from a per-package section counter and a hash of the section type. Nothing about it is meaningful, and the counter means the same edit performed in a different order, or a different run, can yield a different name — the `cfg` prefix and the length are all that can be relied on. No interface lets a script ask for a particular name, so code that needs to refer to the new section keeps the string `add()` returned.

Errors

`error()` describes the last failure and is consumed by reading it, so a second call reports nothing. A missing entry and a rejected argument are different messages, and both are worth distinguishing when a script is driven by a name it got from somewhere else:

```
import * as uci from "uci";
import * as fs from "fs";

const DIR = "/tmp/ucode-manual-uci";
const SAVE = "/tmp/ucode-manual-uci-save";

fs.mkdir(DIR, 0o755);
fs.writefile(DIR + "/demo", "config demo 'main'\n\toption name 'router'\n");

let s = uci.cursor(DIR, SAVE);

s.get("demo", "nosuch", "nope");
printf("missing=%J error=%J\n", null, s.error());
printf("read again=%J\n", s.error());
```

```
missing=null error="Entry not found"
read again=null
```

A configuration file that does not parse is listed by `configs()` like any other, but every lookup against it fails, and the failure is reported as a parse error rather than as a missing entry. Since `configs()` succeeds, the only sign of the problem is the error text — check it rather than concluding the section is simply absent:

```
import * as uci from "uci";
import * as fs from "fs";

const BAD = "/tmp/ucode-manual-uci-bad";
const SAVE = "/tmp/ucode-manual-uci-save";

fs.mkdir(BAD, 0o755);
fs.writefile(BAD + "/demo", "config demo ' bad name here\n");

let s = uci.cursor(BAD, SAVE);

printf("listed=%J get=%J error=%J\n", "demo" in s.configs(), s.get("demo",
"main", "name"),
       s.error());
```

```
listed=true get=null error="Parse error"
```

The cursor API

The cursor's methods, with the return values observed on a working package:

Method	Purpose
`load(pkg)` / `unload(pkg)`	read a package into the cursor / drop it again
`configs()`	the package names in the configuration directory
`get(pkg, sec[, opt])`	section type, or an option's value
`get_all(pkg)`	the whole package, with `.name` / `.type` / `.index` / `.anonymous`
`get_first(pkg, type, opt)`	the first section of `type` that has `opt`
`set(pkg, sec, opt, value)`	stage a value
`delete(pkg, sec[, opt])`	stage a removal
`add(pkg, type)`	stage a new section, returns its generated name
`rename(pkg, sec, name)`	stage a section rename
`list_append(pkg, sec, opt, value)`	add one list entry
`list_remove(pkg, sec, opt, value)`	remove one list entry by value
`changes([pkg])`	the pending delta as command tuples
`revert([pkg])`	discard the pending delta
`save([pkg])`	write the delta to the save directory
`commit([pkg])`	rewrite the configuration file
`foreach(pkg, type, cb)`	call `cb(section)` for each section of `type`, or for every section when `type` is `null`
`reorder(...)`	section ordering; its arguments are not exercised here
`error()`	the last error, consumed on read

Three of these take more arguments than their name suggests — `get_first`, `foreach` and the three-argument `get` — and none of them raises when given the wrong ones: they return `null` and leave a message such as `Invalid argument` for `error()` to report later, at most once. That is the shape of the failure to watch for in this module, since a rejected call and an empty configuration look the same until the error is read.

serial

The `serial` module configures and drives a serial line: baud rate, framing, flow control, the modem control lines, and the read-behaviour knobs that decide when a `read()` returns. It is a thin layer over `termios` and a handful of `ioctl`s, and that shape shows in its one unusual design decision — the module has no function for opening a port.

The full reference — every function, constant and resource method — is generated from the module source and published at ucode-lang.org: *the serial module*.

```
import * as serial from "serial";
```

The module exports twenty-one functions and a couple of hundred constants, and `open` is not among the functions. You open the device node with `fs.open()` and pass the resulting handle to whatever `serial` function you need:

```
import * as serial from "serial";
import * as fs from "fs";

printf("the module has no open(): %s\n", "open" in serial);
printf("it has the configuration functions: %s\n",
       "setspeed" in serial && "setraw" in serial && "attr" in serial);
```

```
the module has no open(): false
it has the configuration functions: true
```

Because a port is an ordinary file handle, everything that works on one keeps working on the other: `f.read()` and `f.write()` move the data, `f.fileno()` gives the descriptor for `socket.poll`-style waiting, and `f.close()` releases it. `serial` only supplies the parts that files do not have.

```
import * as serial from "serial";
import * as fs from "fs";

let f = fs.open("/dev/ttyS99999", "r+");

printf("open failure=%J error=%J\n", f, fs.error());
```

```
open failure=null error="No such file or directory"
```

The handle, or the bare descriptor, is accepted everywhere:

```
let f = fs.open("/dev/ttyUSB0", "r+");

let a = serial.attr(f);
let b = serial.attr(f.fileno());
```

Checking what you have

`isatty(handle)` distinguishes a terminal-like device from a pipe or a regular file, which is worth doing before configuring anything, because the `termios` calls will fail on the wrong kind of descriptor:

```
import * as serial from "serial";
import * as fs from "fs";

fs.writefile("/tmp/ucode-manual-serial-file", "");

let f = fs.open("/tmp/ucode-manual-serial-file", "rw+");

printf("isatty=%J\n", serial.isatty(f));
printf("attr=%J error=%J\n", serial.attr(f), serial.error());
printf("error read again=%J\n", serial.error());
```

```
isatty=false
attr=null error="Inappropriate ioctl for device"
error read again=null
```

That error text is what any `serial` function returns on a descriptor that is not a terminal — the `termios` and `ioctl` calls report `ENOTTY` and the module passes the operating system's own wording through.

Reading and changing the settings

`attr(handle)` returns the current line settings as a plain object, with the flag words as integers and the control characters as an array indexed by the `V` constants:

```
import * as serial from "serial";
import * as fs from "fs";

let f = fs.open("/dev/ttyUSB0", "r+");
let a = serial.attr(f);

printf("%s\n", join(",", sort(keys(a))));
```

```
cc,cflag,iflag,lflag,oflag,ispeed,ospeed
```

`ispeed` and `ospeed` are the input and output speeds in the same numeric space as the `B` constants, `cc` is an array of `NCCS` entries, and the four flag words carry the bits named by the `I`, `O`, `C` and `L` constants.

The four ways to change the settings differ in how much they ask you to know:

Function	Effect
`setspeed(handle, speed[, when])`	set input and output baud rate to one of the `B` constants
`setraw(handle)`	switch the line to raw mode, clearing canonical-mode and echo processing
`setblocking(handle, vmin, vtime[, when])`	set the `VMIN` and `VTIME` read-behaviour parameters
`setattr(handle, attrs[, when])`	write flag words and control characters directly

`when` is `TCSANOW`, `TCSADRAIN` or `TCSAFLUSH` — change immediately, after the output buffer has drained, or after the input buffer has been discarded — and defaults to `TCSANOW`.

```
import * as serial from "serial";
import * as fs from "fs";

let f = fs.open("/dev/ttyUSB0", "r+");

serial.setraw(f);
serial.setspeed(f, serial.B115200);

let a = serial.attr(f);

printf("canonical=%s echo=%s speed=%d\n", (a.lflag & serial.ICANON) != 0,
       (a.lflag & serial.ECHO) != 0, a.ispeed);
```

```
canonical=false echo=false speed=115200
```

`setattr` takes an object whose keys are `iflag`, `oflag`, `cflag`, `lflag`, `ispeed`, `ospeed` and `cc`, and applies only the ones present; the rest are read from the line first. So a typical edit is read, modify, write:

```
import * as serial from "serial";
import * as fs from "fs";

let f = fs.open("/dev/ttyUSB0", "r+");
let a = serial.attr(f);

serial.setattr(f, { lflag: a.lflag & ~serial.ECHO }, serial.TCSAFLUSH);

printf("echo now off: %s\n", (serial.attr(f).lflag & serial.ECHO) == 0);
```

```
echo now off: true
```

When a read returns

`setblocking` is the function that decides how `read()` behaves, and its arguments are not a boolean — it takes `VMIN` and `VTIME`, the two `termios` parameters that govern it. The name describes the question it answers rather than the form of its arguments.

`VMIN`	`VTIME`	Behaviour of `read()`
`n`	`0`	return as soon as `n` bytes are available, blocking indefinitely
`0`	`t`	return as soon as one byte is available, or after `t` tenths of a second, whichever comes first
`n`	`t`	return after `n` bytes, restarting the timer on each byte; `0` means no timer
`0`	`0`	return immediately with whatever is available, possibly nothing

```
import * as serial from "serial";
import * as fs from "fs";

let f = fs.open("/dev/ttyUSB0", "r+");

serial.setraw(f);
serial.setblocking(f, 0, 10);

let a = serial.attr(f);

printf("vmin=%d vtime=%d\n", a.cc[serial.VMIN], a.cc[serial.VTIME]);
```

```
vmin=0 vtime=10
```

A raw port with `VMIN` 0 and a timeout is the usual arrangement for a device that answers unpredictably: the script cannot stall forever waiting for a reply that is not coming.

Moving data

Data moves through the handle's own `read()` and `write()`, with `serial` providing the buffer-state queries and the queue control around them:

```
import * as serial from "serial";
import * as fs from "fs";

let f = fs.open("/dev/ttyUSB0", "r+");

serial.setraw(f);
serial.setspeed(f, serial.B115200);
serial.setblocking(f, 0, 20);

f.write("AT\r\n");

if (serial.input_waiting(f) > 0) {
        printf("reply starts %s\n", slice(f.read(32), 0, 4));
}

printf("output cleared: %s\n", serial.drain(f) && serial.flush(f,
serial.TCIOFLUSH));
```

```
reply starts OK
output cleared: true
```

`input_waiting()` and `output_waiting()` report the bytes queued in the driver in each direction, `drain()` waits for the output buffer to empty, and `flush([queue])` discards pending data — `TCIFLUSH` for input, `TCOFLUSH` for output, `TCIOFLUSH` for both.

The modem lines and the UART itself

These are the reasons to use a serial port rather than a socket, and also the parts that need real hardware. `mget(handle)` reads the modem status bits, `mbis(handle, bits)` and `mbic(handle, bits)` set and clear them, `mset(handle, bits)` writes the set directly, and `dtr(handle, on)` and `rts(handle, on)` are shorthands for the two lines most often toggled. `mbis(handle, serial.TIOCM_DTR)` asserts `DTR`, and `mbis(handle, serial.TIOCM_RTS)` asserts `RTS`; the readable lines are `TIOCM_CTS`, `TIOCM_DSR` and `TIOCM_CAR` / `TIOCM_RI`.

`sendbreak(handle)` transmits a break condition, `getinfo(handle)` reads the driver's own report — port type, irq, baud base, the hardware names carried by the `PORT_` constants — and `lowlatency(handle, on)` asks the driver for low-latency receive handling at the cost of some CPU.

A pseudo-terminal has none of this. On a pty, `mget()`, `getinfo()` and `lowlatency()` all fail with the same `Inappropriate ioctl for device`, which is a useful fact when a script is being tested without a console cable: the data path works, the control path does not.

Constants

The constant set comes straight from the system headers — the `TCS*` and `TC*` request codes, the `I` / `O` / `C` / `L` flag bits, the `V*` control-character indices, the `B*` baud-rate codes, the `TIOCM*` modem-line bits, and the `ASYNC_*` and `PORT_*` driver values — and it is filtered by what the build host's headers define. The `#ifdef` guards around every single one mean a constant that exists on Linux with glibc can be absent on another libc, so the `B` and flag constants are the only correct way to name a speed or a bit. The numeric encoding of speeds in particular is not portable: one system's `B9600` is another system's `15`.

```
import * as serial from "serial";

printf("two distinct speeds: %s\n", serial.B9600 != serial.B115200);
printf("VMIN and VTIME are indices: %s\n",
       type(serial.VMIN) == "int" && type(serial.VTIME) == "int" &&
serial.VMIN != serial.VTIME);
printf("TCSANOW is the default action: %s\n", type(serial.TCSANOW) == "int");
```

```
two distinct speeds: true
VMIN and VTIME are indices: true
TCSANOW is the default action: true
```

Errors

Every function reports failure the same way: `null` (or `false` for `isatty`) plus an error message taken from the operating system's error table, and `error()` is consumed by reading it. What it does *not* do is clear itself when a later call succeeds — so a script that checks `error()` instead of the return value can be told about a failure that happened several calls ago:

```
import * as serial from "serial";
import * as fs from "fs";

fs.writefile("/tmp/ucode-manual-serial-file", "");

let f = fs.open("/tmp/ucode-manual-serial-file", "rw+");

serial.attr(f);
serial.isatty(f);

printf("after a failure then a success: %J\n", serial.error());
printf("and after reading it: %J\n", serial.error());
```

```
after a failure then a success: "Inappropriate ioctl for device"
and after reading it: null
```

Test the return value, then read the error while it is still the error from that call. This is the same rule as in `fs` and `io`, and the reason all three put the diagnostic in a single slot rather than returning it.

An embedding overview

ucode was written to be embedded. The interpreter is a shared library, `libucode`, with a C API of about a hundred and thirty functions, and the command line interpreter is built on that same API — it is a forty-line wrapper around a compiler call, a VM initialisation and an execute call, which means everything you can do from the shell you can do from your own program, and everything you can do from your own program is what `uhttpd`, `uwsd`, `fw4` and `rpcd` do.

There are two distinct jobs, and this part of the book is about the first one.

Embedding means owning the process, creating a VM, deciding which scripts run, what they can see and which functions they may call. That is Part IV, chapters 40 to 50.

Extending means writing a `.so` that a VM loads on `import`, adding a handful of functions to somebody else's interpreter. That is chapter 48, and it needs a fraction of this API.

Getting the library

The build produces three things of interest:

Artifact	Installed to
`libucode.so`	`${libdir}`
the standard library modules (`fs.so`, `socket.so`, `uloop.so`, …)	`${libdir}/ucode`
the headers (`ucode.h`, `types.h`, `vm.h`, …)	`include/ucode`

A host program needs the headers and one link flag:

```
$ cc -O2 -I/usr/include program.c -o program -lucode
```

Inside the build tree, without installing anything:

```
$ cc -std=gnu11 -I include program.c -o program -L build -lucode -Wl,-rpath,
$PWD/build
```

Include one umbrella header, or pick the pieces you need:

```c
#include <ucode/ucode.h>      /* platform, types, vm, compiler, source,
program, module, lib */
```

```c
#include <ucode/compiler.h>   /* uc_parse_config_t, uc_compile() */
#include <ucode/lib.h>        /* uc_stdlib_load(), uc_search_path_*(),
uc_fn_arg() */
#include <ucode/vm.h>         /* uc_vm_t and everything about running code */
```

The shape of the API

The names group by prefix, and the groups correspond to the phases of running a script:

Prefix	Area	Representative calls
`ucv_`	values: create, read, compare, refcount	`ucv_string_new`, `ucv_object_get`, `ucv_array_push`, `ucv_get`, `ucv_put`
`uc_source_`	source text, from buffer or file	`uc_source_new_buffer`, `uc_source_new_file`, `uc_source_put`
`uc_compile`, `uc_program_`	turning source into bytecode, and bytecode in and out of files	`uc_compile`, `uc_program_write`, `uc_program_load`
`uc_vm_`	a running interpreter: scope, stack, calls, execution, signals	`uc_vm_init`, `uc_vm_execute`, `uc_vm_invoke`, `uc_vm_call`
`uc_stdlib_`, `uc_search_path_`	the standard library and module lookup	`uc_stdlib_load`, `uc_search_path_init`
`uc_module_`	the ABI of a loadable module	`uc_module_init` (chapter 48)

The whole interaction is a pipeline:

```
config → source → program → vm (scope + stdlib + natives) → execute → value
or status
```

A host program, line by line

This is a complete host. It compiles a script held in a string, gives the script a global variable and a native function, runs it, and prints what came back:

```
#include <stdio.h>

#include <ucode/compiler.h>
#include <ucode/lib.h>
#include <ucode/vm.h>


static const char script[] =
        "function label(name) {\n"
        "    return sprintf('%s on %s', name, iface);\n"
        "}\n"
        "\n"
        "printf('%s\\n', label('bridge'));\n"
        "printf('twice(21) = %s\\n', twice(21));\n"
        "\n"
        "return [1, 2, 3];\n";

static uc_parse_config_t config = {
        .raw_mode = true,
        .strict_declarations = false,
        .lstrip_blocks = true,
        .trim_blocks = true
};

static uc_value_t *
uc_twice(uc_vm_t *vm, size_t nargs)
{
        return ucv_double_new(2 * ucv_to_double(uc_fn_arg(0)));
}

int main(void)
{
        uc_value_t *scope, *retval = NULL;
        char *error = NULL, *s;

        uc_search_path_init(&config.module_search_path);

        uc_source_t *src = uc_source_new_buffer("example", strdup(script),
strlen(script));
        uc_program_t *program = uc_compile(&config, src, &error);

        uc_source_put(src);

        if (!program) {
                fprintf(stderr, "Compile failed: %s\n", error);
```

```
                free(error);

                return 1;
        }

        uc_vm_t vm = { 0 };

        uc_vm_init(&vm, &config);

        scope = uc_vm_scope_get(&vm);

        uc_stdlib_load(scope);
        ucv_object_add(scope, "iface", ucv_string_new("br0"));
        ucv_object_add(scope, "twice", ucv_cfunction_new("twice", uc_twice));

        switch (uc_vm_execute(&vm, program, &retval)) {
        case STATUS_OK:
                s = ucv_to_string(&vm, retval);
                printf("returned: %s\n", s);
                free(s);
                break;

        default:
                printf("the script did not finish normally\n");
                break;
        }

        ucv_put(retval);
        uc_program_put(program);
        uc_vm_free(&vm);
        uc_search_path_free(&config.module_search_path);

        return 0;
}
```

```
bridge on br0
twice(21) = 42
returned: [ 1, 2, 3 ]
```

Each part of it:

Configuration. `uc_parse_config_t` holds the switches that decide how source is turned into code. The fields are `lstrip_blocks`, `trim_blocks`, `strict_declarations`, `raw_mode`, `module_search_path`, `force_dynlink_list`, `setup_signal_handlers` and `compile_module`.

> **Set `raw_mode` explicitly.** A zero-initialised config compiles sources in *template* mode, because that is what `false` means here: `raw_mode == false` is the template mode of chapter 16, where text outside `{{ }}` and `{% %}` is output. A host that never intends to render templates and leaves the field unset will watch its script print itself and return `null`. The command line interpreter sets `.raw_mode = true` in its default config and clears it for `-T`.

`strict_declarations` rejects use of undeclared names. `module_search_path` is a vector of `char *` templates used to resolve module names, and `uc_search_path_init()` fills it with the defaults compiled into the library:

```
${prefix}/${libdir}/ucode/*.so:${prefix}/share/ucode/*.uc:./*.so:./*.uc
```

Each entry must contain a `*`; an entry without one is skipped. A name is resolved by substituting it for the star, and a `.` in the name becomes a directory separator, so `/usr/lib/ucode/*.so` resolves `fs` to `/usr/lib/ucode/fs.so` and `net.http` to `/usr/lib/ucode/net/http.so`. A name that already contains a `/` bypasses the search path and is taken as a path relative to the including source, extension included. The list is a CMake cache variable (`-DLIB_SEARCH_PATH=...`) fixed at build time, and `uc_search_path_add()` extends it at run time. Leave the vector empty and any script that imports a module fails to compile.

`force_dynlink_list` is the programmatic equivalent of `-c dynlink=name` from chapter 17: imports of the listed names are compiled as runtime loads of `<name>.so` instead of being resolved at compile time. `setup_signal_handlers` controls whether `uc_vm_init` installs handlers for the signals the script registers.

Source. `uc_source_new_buffer(name, data, len)` wraps a block of text; the source takes ownership of `data`, which is why the example passes `strdup(script)`. `uc_source_new_file(path)` reads a file and is what the CLI uses; the name you give a source or file appears in every error message and stack trace, so make it useful. Sources are reference counted — `uc_source_get`, `uc_source_put` — and a compiled program keeps the sources it came from alive, which is how an error at run time can still quote the offending line.

Compilation. `uc_compile(&config, src, &error)` returns a `uc_program_t *`, or `NULL` with a heap-allocated message in `error` that the caller must `free()`. Compilation is a separate step from execution, and that is the whole reason `ucc` and precompiled `.uc.o` files exist: a device can compile at build time and load bytecode with `uc_program_load()` at run time, needing no compiler at all in the hot path (chapter 47).

The VM. `uc_vm_t vm = { 0 };` plus `uc_vm_init(&vm, &config)`; the struct may live on the stack. The VM owns its global scope, a value registry, the module cache, the signal handler table, the exception state, the operand stack and the call frames. `uc_vm_free(&vm)` releases them; a VM that runs a long-lived service is initialised once and reused (chapter 42 shows what the registry is good for).

Scope and standard library. `uc_vm_scope_get(&vm)` returns the global scope as an ordinary object, and `uc_stdlib_load(scope)` fills it with the core functions — `print`, `sprintf`, `keys`, `json`, `gc`, the whole table of chapter 20. An embedder that wants a smaller world can skip this call and add exactly the functions it means to expose; a script that gets an empty scope has `printf` only if the host put it there.

Giving the script things. `ucv_object_add(scope, name, value)` returns a `bool` and transfers the reference on success: after a successful call the scope owns `value`, so do not `ucv_put()` it yourself; after a failed one (the target is not an object) the caller still owns it and must release it. The example adds data (`iface`) and behaviour (`twice`, a native function created with `ucv_cfunction_new`). Native functions are covered in chapter 44; note the shape here — a `uc_function_t`-shaped C function taking `(vm, nargs)` and returning a `uc_value_t *`, reading its arguments with the `uc_fn_arg(n)` macro, which needs `vm` and `nargs` in scope under exactly those names.

Running. `uc_vm_execute(&vm, program, &retval)` runs the program's top-level function and stores its return value, transferring a reference to the caller. The return code says how the run ended:

Status	Meaning	`retval`
`STATUS_OK`	the script returned normally	its return value
`STATUS_EXIT`	the script called `exit(n)`	the exit code as a number
`STATUS_BREAK`	execution was interrupted by `uc_vm_break_request()`	`NULL`
`ERROR_RUNTIME`	an uncaught exception	`NULL`
`ERROR_COMPILE`	a runtime compile failure (`loadstring`, `import`)	`NULL`

For the two error statuses the returned value is `NULL`; the failure itself is in the VM's exception state, readable with `uc_vm_exception_object(&vm)` as an ordinary ucode value carrying `type`, `message` and `stacktrace`. An exception handler installed with `uc_vm_exception_handler_set()` is called before `uc_vm_execute` returns, and the default behaviour of the CLI — printing the message with source context and a stack trace — is that handler doing its job. Chapter 46 covers all of it.

Reading results. `ucv_to_string(&vm, value)` renders any value the way `print` would, returning a `malloc`ed string you free yourself. To take values apart rather than render them, use the accessors of chapter 41: `ucv_type`, `ucv_string_get`, `ucv_int64_get`, `ucv_array_get`, `ucv_object_get` and so on. Note that `ucv_object_get(obj, key, &present)` takes a third argument — a pointer to a `bool` which tells you whether the key existed, since a stored `null` and a missing key look identical otherwise.

Cleanup. Release in the reverse order of acquisition: the returned value, the program, the VM, the search path. The source was released right after compilation in the example above; the program kept its own reference for as long as it needed the text, which is why that is safe.

Calling back into a script

A host usually wants to invoke a function defined by the script, not just run the script once. There are two supported shapes.

The convenient one is `uc_vm_invoke(&vm, "name", nargs, arg1, arg2, …)`, which looks the name up in the global scope, pushes the arguments, calls, and returns the result. It borrows each argument — it takes its own reference — so the caller keeps ownership of what it passed:

```c
#include <stdio.h>

#include <ucode/compiler.h>
#include <ucode/lib.h>
#include <ucode/vm.h>


static const char script[] =
        "greet = function (name, loud) {\n"
        "    let text = sprintf('Hello, %s!', name);\n"
        "\n"
        "    return loud ? uc(text) : text;\n"
        "};\n";

static uc_parse_config_t config = { .raw_mode = true };

int main(void)
{
        uc_value_t *name = ucv_string_new("world");
        uc_value_t *result;
        char *s;

        uc_source_t *src = uc_source_new_buffer("greetings", strdup(script),
strlen(script));
        uc_program_t *program = uc_compile(&config, src, NULL);

        uc_source_put(src);

        uc_vm_t vm = { 0 };

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));
        uc_vm_execute(&vm, program, NULL);

        result = uc_vm_invoke(&vm, "greet", 2, name, ucv_boolean_new(true));

        ucv_put(name);

        s = ucv_to_string(&vm, result);
        printf("%s\n", s);

        free(s);
        ucv_put(result);
        uc_program_put(program);
        uc_vm_free(&vm);
```

```
        return 0;
}
```

```
HELLO, WORLD!
```

The lookup is in the **global scope**, and a top-level `function greet() {}` declaration does not create a global — declarations live in the file scope of the program that made them, as chapter 5 explains, and that scope is gone once the program returns. A script intended to be driven from C therefore publishes its entry points by assignment, as above, or hands them back in a table:

```c
#include <stdio.h>

#include <ucode/compiler.h>
#include <ucode/lib.h>
#include <ucode/vm.h>


static const char script[] =
        "return {\n"
        "    add: function (a, b) {\n"
        "        return a + b;\n"
        "    }\n"
        "};\n";

static uc_parse_config_t config = { .raw_mode = true };

int main(void)
{
        uc_source_t *src = uc_source_new_buffer("ns", strdup(script),
strlen(script));
        uc_program_t *program = uc_compile(&config, src, NULL);
        uc_vm_t vm = { 0 };
        uc_value_t *ns, *result;
        char *s;

        uc_source_put(src);
        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));

        if (uc_vm_execute(&vm, program, &ns) != STATUS_OK)
                return 1;

        uc_vm_stack_push(&vm, ucv_get(ucv_object_get(ns, "add", NULL)));
        uc_vm_stack_push(&vm, ucv_int64_new(20));
        uc_vm_stack_push(&vm, ucv_int64_new(22));

        if (uc_vm_call(&vm, false, 2) != EXCEPTION_NONE)
                return 1;

        result = uc_vm_stack_pop(&vm);
        s = ucv_to_string(&vm, result);
        printf("add(20, 22) = %s\n", s);

        free(s);
        ucv_put(result);
        ucv_put(ns);
```

```
        uc_program_put(program);
        uc_vm_free(&vm);

        return 0;
}
```

```
add(20, 22) = 42
```

Pushing the callee and its arguments on the VM stack and calling `uc_vm_call(&vm, false, nargs)` is the general mechanism — it takes any value, including a method reached through an object, and it is what `uc_vm_invoke` does underneath. `uc_vm_stack_push` takes ownership of the value pushed; `uc_vm_stack_pop` returns a value the caller owns. The details, including the `mcall` argument and how `this` works, are in chapter 44.

Keeping state between runs

`uc_vm_registry_set(&vm, "key", value)`, `uc_vm_registry_get`, `uc_vm_registry_exists` and `uc_vm_registry_delete` manage values under string keys on the VM itself. The registry is not visible to scripts, is not cleared between executions, and is a root for the garbage collector, so it is the natural place for the host's own state: the socket a server is listening on, the event loop, a per-connection context, a configuration handle. Handlers installed on the VM (exception handler, signal handlers) live alongside it for the same reason.

A few things worth knowing before you start

- **One VM per thread.** Nothing inside `uc_vm_t` is locked, and some helper state is per-thread (`uc_thread_context_get()`). Give each thread its own VM and do not pass values between VMs on different threads.
- **Reference counting is the whole memory story for you as a host.** Values handed to you (`uc_fn_arg`, `uc_vm_stack_pop`, `ucv_object_get`) are borrowed unless documented otherwise; hold them past the end of the current call with `ucv_get()` and give them back with `ucv_put()`. Chapter 41 has the table.
- **A script can end your process.** A top-level `exit(3)` in the script surfaces as `STATUS_EXIT`; it is a status, not a longjmp, but a host that ignores the status of `uc_vm_execute` will keep going with a value that is an exit code. Scripts run in an interpreter that has `system()`, `fs`, `socket` and friends — sandboxing means choosing what to load into the scope, not trusting the collector.
- **Compile once, run many.** A program compiled from source may be executed repeatedly against the same VM; the compiler state and the run state are separate. Services that run a script per request should keep the `uc_program_t` and re-execute it, not re-parse it per request — this is exactly the difference between the `state-reuse` and `state-reset` examples.

- **Errors quote your source.** Because a program holds its sources, a runtime error can print the offending line — which is why `uc_source_new_buffer` takes a name and why passing something like `"config.uc"` instead of `"buf"` pays for itself the first time a script fails.

The shipped examples

The `examples/` directory contains six small hosts, built with the rest of the tree and runnable from `build/examples/`. Their header comments give the standalone build line of the form `gcc -o execute-string -lucode execute-string.c`.

Example	What it shows
`execute-string`	compile a C string literal (a template-wrapped script) into a program, inject globals, run it, dispatch on all four status codes, read the return value
`execute-file`	the same, from a file named on the command line, using `uc_source_new_file`
`native-function`	registering C functions and calling them from the script
`exception-handler`	installing `uc_vm_exception_handler_set()` and printing the exception object, its type, message and stack trace
`state-reuse`	one VM executed repeatedly, with globals carrying over between runs
`state-reset`	a VM initialised and freed inside the loop, so nothing carries over

Their observed output is worth a look before writing your own:

```
$ build/examples/execute-string
123 + 456 is 579
Program finished successfully.
Function return value is 579
$ build/examples/native-function
add() = 10.1
multiply() = 36.5
$ build/examples/state-reuse | head -3
Iteration 1: Current value is 1
Iteration 2: Current value is 2
Iteration 3: Current value is 4
$ build/examples/state-reset | head -2
Iteration 1: Global variable is null? true
Iteration 2: Global variable is null? true
```

Where to go next

Chapter 41 is the reference for `uc_value_t` — every type, every constructor and accessor, and the ownership rules. Chapter 42 covers `uc_vm_t`: scope, registry, stack, status. Chapter 43 is the compiler and source layer, chapter 44 native functions, chapter 45 resource types (how a C handle gets a sane lifetime inside a garbage-collected world), chapter 46 exceptions and interrupts, chapter 47 bytecode and precompilation, chapter 48 loadable modules, chapter 49 a guided reading of the examples, and chapter 50 a worked embedding: a small daemon with an event loop, resources, and untrusted scripts.

Values in C

Every value a script manipulates, and every value a host passes to or receives from a script, is a `uc_value_t *`. This chapter is the reference for that pointer: what it can point at, how to create and read each kind of value, and — the part that takes the most getting used to — who is responsible for freeing it.

The representation

The public header defines `uc_value_t` as a bitfield struct:

```
typedef struct uc_value {
        uint32_t type:4;
        uint32_t mark:1;
        uint32_t ext_flag:1;
        uint32_t refcount:26;
} uc_value_t;
```

That is the common header of heap-allocated values, but not every value is a pointer to one. ucode stores small values **in the pointer itself**: the low two bits of the pointer are a type tag, and for some types the remaining bits carry the value.

Kind	Where it lives
`null`	the null pointer
`boolean`	tag `UC_BOOLEAN`, the value in bit 2 — there are exactly two such "pointers", `true` and `false`
small integers	tag `UC_INTEGER`, the number in the upper bits
strings up to `sizeof(void *) - 2` bytes	tag `UC_STRING`, the length in the upper bits of the first byte and the characters in the bytes after it
everything else	a heap allocation whose first word is the header above

The consequences are practical rather than academic:

- `ucv_get()` and `ucv_put()` return immediately for null and for any tagged value — there is no reference count to touch, and calling them is free.
- Comparing two values by pointer equality is meaningful for booleans (there is only one `true`) but not for numbers in general, since the same number can be represented as an immediate integer, a heap integer, or a double.

- The `refcount` field is 26 bits wide. `ucv_get()` asserts it has not overflowed, which means a value may not be referenced more than 2^26 − 1 times — a limit that in practice only an assertion-free release build can sail past, and then silently.
- `ucv_is_marked()`, `ucv_set_mark()` and `ucv_clear_mark()` are the collector's bits; hosts have no reason to touch them.

Two helpers report the type, and they take the pointer, not a struct:

```
uc_type_t  ucv_type(uc_value_t *uv);       /* UC_NULL, UC_INTEGER, ... */
const char *ucv_typename(uc_value_t *uv);  /* "null", "integer", ... */
```

`ucv_typename()` takes the value itself (not a `uc_type_t`) and returns the same name the script's `type()` gives. Note that the enumeration in `include/ucode/types.h` contains types a script never sees directly — `UC_UPVALUE`, `UC_PROGRAM`, `UC_SOURCE` — and that both signed and unsigned 64-bit integers report as `UC_INTEGER`; `ucv_is_u64()` is what distinguishes them.

```c
#include <stdio.h>

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static void
show(uc_vm_t *vm, const char *label, uc_value_t *v)
{
        char *s = ucv_to_string(vm, v);

        printf("%-14s type=%-2d name=%-8s value=%s\n", label, ucv_type(v),
ucv_typename(v), s);

        free(s);
}

int main(void)
{
        uc_vm_t vm = { 0 };

        uc_vm_init(&vm, &config);

        show(&vm, "null", NULL);
        show(&vm, "true", ucv_boolean_new(true));
        show(&vm, "integer", ucv_int64_new(42));
        show(&vm, "big integer", ucv_int64_new(1LL << 53));
        show(&vm, "unsigned", ucv_uint64_new(1ULL << 63));
        show(&vm, "double", ucv_double_new(1.5));
        show(&vm, "string", ucv_string_new("hello"));
        show(&vm, "array", ucv_array_new(&vm));
        show(&vm, "object", ucv_object_new(&vm));

        printf("\nthe two booleans are one value: %d\n",
               ucv_boolean_new(true) == ucv_boolean_new(false));
        printf("small integers are immediates:  %d\n",
               ucv_int64_new(42) == ucv_int64_new(42));
        printf("the unsigned flag: %d, %d\n",
               ucv_is_u64(ucv_uint64_new(1ULL << 63)),
ucv_is_u64(ucv_int64_new(42)));
        printf("ucv_type(NULL) = %d, ucv_get(NULL) = %d\n", ucv_type(NULL),
ucv_get(NULL) == NULL);

        uc_vm_free(&vm);
```

```
        return 0;
}
```

```
null           type=0  name=null     value=null
true           type=2  name=boolean  value=true
integer        type=1  name=integer  value=42
big integer    type=1  name=integer  value=9007199254740992
unsigned       type=1  name=integer  value=9223372036854775808
double         type=4  name=double   value=1.5
string         type=3  name=string   value=hello
array          type=5  name=array    value=[ ]
object         type=6  name=object   value={ }

the two booleans are one value: 0
small integers are immediates:  1
the unsigned flag: 1, 0
ucv_type(NULL) = 0, ucv_get(NULL) = 1
```

Ownership

Reference counting is manual, and the API is consistent about one thing: **a function that creates a value gives you a reference you own; a function that fetches a value out of somewhere else gives you a borrowed pointer you must not release.** The two escaping rules are that anything named `ucv_get` / `ucv_put`, and the handful of functions documented below as returning a new reference, behave the other way.

You receive a new reference (you must eventually `ucv_put`) from	You receive a borrowed reference (do not `ucv_put`) from
`ucv_boolean_new`, `ucv_int64_new`, `ucv_uint64_new`, `ucv_double_new`, `ucv_string_new`, `ucv_string_new_length`, `ucv_stringbuf_finish`	`ucv_array_get`, `ucv_object_get`, `ucv_property_get`
`ucv_array_new`, `ucv_array_new_length`, `ucv_object_new`, `ucv_regexp_new`	`uc_fn_arg`, `uc_fn_this`, `uc_vm_stack_peek`
`ucv_cfunction_new`, `ucv_resource_new`	`ucv_prototype_get`, `ucv_resource_data`
`ucv_array_pop`, `ucv_array_shift`	`ucv_object_foreach`'s `val`, and `ucv_array_sort` comparator arguments
`uc_vm_stack_pop`, `uc_vm_invoke`, `uc_vm_execute`'s `retval`	the `scope` returned by `uc_vm_scope_get`
`ucv_to_string`, `ucv_to_jsonstring`, `ucv_to_number`, `ucv_key_set`'s return	`uc_vm_exception_object`'s return — see chapter 46
`ucv_from_json`	

The container mutators **take ownership of the value you hand in**, on success:

Call	Returns	On failure
`ucv_array_push(array, v)`	`v`	`NULL` for a non-array or a constant array — `v` is still yours
`ucv_array_set(array, idx, v)` / `ucv_array_unshift(array, v)`	`true`	`false` — `v` is still yours
`ucv_object_add(object, key, v)`	`true`	`false` — `v` is still yours
`ucv_prototype_set(value, proto)`	`true`	`false` if `proto` is not an object — `proto` is still yours

```
ucv_object_add(scope, "iface", ucv_string_new("br0"));   /* the scope owns the string now */
```

`ucv_key_set()` is the one function that runs the other way, and the header says so: it *retains* the value it stores and *returns a reference to it*. So the value you pass is not consumed, and what comes back must be released:

```
uc_value_t *rv = ucv_key_set(&vm, obj, key, val);   /* borrows key and val */

if (rv)
        ucv_put(rv);                                    /* the returned
reference is yours */
```

Replacing what is already stored

Every mutator that overwrites a slot releases the value that was in it, so the replaced value is freed if the container held its last reference:

Call	Consumes the new value	Releases the previous value	Safe when the new value is the one being replaced
`ucv_array_set(array, idx, v)`	yes	yes, when `idx` is inside the array	no
`ucv_object_add(object, key, v)`	yes	yes, when the key existed	no
`ucv_resource_value_set(resource, idx, v)`	yes	yes	no
`ucv_prototype_set(value, proto)`	yes	yes, the previous prototype	no
`ucv_key_set(vm, scope, key, v)`	no, it borrows	yes, through the paths above	yes
`ucv_array_push`, `ucv_array_unshift`	yes	nothing to release	—
`ucv_array_delete`, `ucv_object_delete`	—	yes, the removed values	—

The two columns that matter together are the last two. The direct mutators release the old value *before* they store the new one and they take no reference of their own on the way in, so a value that the container itself is the last holder of is freed during the call and then stored anyway:

```
uc_value_t *v = ucv_array_get(a,
0);      /* borrowed; the array may hold the only reference */

ucv_array_set(a, 0, v);                  /* releases v, then stores the
released pointer */
```

AddressSanitizer, built against `build-asan`, reports the resulting fault as `attempting double-free on 0x...` inside `ucv_free()`, on the release that follows. The same shape reaches `ucv_object_add()` through `ucv_object_get()`. `ucv_key_set()` is not exposed to it, because it takes its own reference to the value before the overwrite happens — which is the pattern to follow by hand when a store might replace the very value being stored:

```
ucv_object_add(o, "same", ucv_get(v));    /* the extra reference keeps it
alive across the put */
```

A script cannot reach this: `a[0] = a[0]` goes through `ucv_key_set()`, which holds the reference described above.

`ucv_put(NULL)` is a no-op, so a destructor chain can release unconditionally.

Scalars

Type	Constructor	Accessor	Notes
boolean	`ucv_boolean_new(bool)`	`ucv_boolean_get(v)`	returns one of two constant values
signed integer	`ucv_int64_new(int64_t)`	`ucv_int64_get(v)`	`errno` is cleared, then set on out-of-range conversions
unsigned integer	`ucv_uint64_new(uint64_t)`	`ucv_uint64_get(v)`	`ucv_is_u64(v)` tells the two apart
double	`ucv_double_new(double)`	`ucv_double_get(v)`	
string	`ucv_string_new(const char *)`, `ucv_string_new_length(const char *, size_t)`	`ucv_string_get(v)`, `ucv_string_length(v)`	the length form is binary safe
regexp	`ucv_regexp_new(src, icase, newline, global, &err)`	—	`err` receives a `malloc`ed `regcomp()` message on failure

An accessor called on the wrong type does not raise. It clears `errno`, then reports the failure its own way, and the answer differs per accessor:

Accessor	Wrong type	Out of range
`ucv_int64_get`, `ucv_uint64_get`	`errno = EINVAL`, returns `0`	`errno = ERANGE`, returns the clamped `INT64_MIN` / `INT64_MAX` / `0`
`ucv_double_get`	`errno = EINVAL`, returns NaN	`errno = ERANGE` for an integer beyond 2^53, value returned unchanged
`ucv_string_get`	returns `NULL`, `errno` untouched	—
`ucv_boolean_get`	returns `false`, `errno` untouched	—
`ucv_string_length`	returns `0`	—

A host that needs to complain should check `ucv_type()` first — which is what the standard library does before raising its own type errors — and a host that reads `errno` must read it *after* the call, not pass it as a companion argument to `printf()`, whose argument order is unspecified.

`ucv_string_get` is a macro that takes the **address** of its argument (`(uc_value_t **)&uv`), because short strings live in the pointer. The practical effect is that its argument must be a `uc_value_t *` variable, not an expression:

```
uc_value_t *v = ucv_object_get(o, "name", NULL);

printf("%s\n", ucv_string_get(v));        /* fine */
printf("%s\n", ucv_string_get(ucv_object_get(o, "name", NULL)));   /* does
not compile */
```

> **A ucode string is a length, not a C string.** `ucv_string_length()` is authoritative; the bytes may contain NULs, and `ucv_string_new_length()` copies exactly `length` bytes without appending a terminator. Handing the characters to a C string function is still safe: heap strings are allocated with `xcalloc()` for `length + 1` bytes, and the short strings packed inside the pointer word have zeros in the remaining bytes, so a zero byte always follows the characters. What truncates such a read is an embedded NUL, which is why the length is the field to trust. When calling into C, copy:
>
> ```
> char *c = strndup(ucv_string_get(v), ucv_string_length(v));
> ```
>
> Strings are immutable by convention. Although `ucv_string_get()` hands out a writable pointer, changing the bytes changes every holder of that value at once; build a new string instead.

To build a string incrementally, use the string buffer, which is a `uc_stringbuf_t` (json-c's `printbuf`) that begins life pre-filled with an empty string header so that finishing it is cheap:

```
uc_stringbuf_t *ucv_stringbuf_new(void);
void ucv_stringbuf_append(uc_stringbuf_t *, const char
*literal);   /* string literal only */
void ucv_stringbuf_addstr(uc_stringbuf_t *, const char *str, size_t len);
void ucv_stringbuf_printf(uc_stringbuf_t *, const char *fmt, ...);
uc_value_t *ucv_stringbuf_finish(uc_stringbuf_t
*);                  /* frees the buffer, owns the result */
```

`ucv_stringbuf_append()` takes a string literal (the macro measures it with `sizeof`), `ucv_stringbuf_addstr()` takes a pointer and a length, and `ucv_stringbuf_printf()` is a macro onto json-c's `sprintbuf()` — so a host using the buffer must link json-c as well as ucode: `-lucode -ljson-c` .

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

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_value_t *s;
        uc_stringbuf_t *buf;
        double d;
        int64_t n;
        const char *data = "a\0b\0c";

        uc_vm_init(&vm, &config);

        s = ucv_string_new_length(data, 4);
        printf("binary string: length=%zu, bytes equal=%d\n",
               ucv_string_length(s), memcmp(ucv_string_get(s), data, 4) ==
0);
        ucv_put(s);

        buf = ucv_stringbuf_new();
        ucv_stringbuf_append(buf, "ifname=");
        ucv_stringbuf_printf(buf, "%s-%d", "lan", 1);
        ucv_stringbuf_addstr(buf, "!", 1);

        s = ucv_stringbuf_finish(buf);
        printf("buffer: %s\n", ucv_to_string(&vm, s));
        ucv_put(s);

        s = ucv_string_new("42");
        n = ucv_int64_get(s);
        printf("int64 of a string: %ld, errno: %d\n", (long)n, errno);

        d = ucv_double_get(s);
        printf("double of a string: %f, errno: %d\n", d, errno);

        uc_vm_free(&vm);

        return 0;
}
```

```
binary string: length=4, bytes equal=1
buffer: ifname=lan-1!
int64 of a string: 0, errno: 22
double of a string: nan, errno: 22
```

Turning values into numbers

`ucv_to_number(v)` returns a **new** numeric value: integers stay integers, doubles stay doubles, `null` becomes `0`, booleans become `0` or `1`, and a string is parsed as a number — decimal, or hexadecimal with an `0x` prefix, or `null` when it does not parse at all. Non-scalars such as arrays have no numeric value and also yield `null`. Three inline helpers wrap it for the common cases, giving `0` where the conversion failed:

```
uc_value_t *ucv_to_number(uc_value_t *v);
double   ucv_to_double(uc_value_t *v);
int64_t  ucv_to_integer(uc_value_t *v);
uint64_t ucv_to_unsigned(uc_value_t *v);
```

```c
#include <stdio.h>

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

int main(void)
{
        uc_vm_t vm = { 0 };
        const char *inputs[] = { "42", "3.9", "  7abc", "abc", "", "0x10" };
        size_t i;

        uc_vm_init(&vm, &config);

        printf("ucv_to_number:");

        for (i = 0; i < sizeof(inputs) / sizeof(*inputs); i++) {
                uc_value_t *n = ucv_to_number(ucv_string_new(inputs[i]));
                char *s = ucv_to_string(&vm, n);

                printf("  [%s]=%s/%s", inputs[i], ucv_typename(n), s);

                free(s);
                ucv_put(n);
        }

        printf("\nnull=%d true=%d array=%d\n",
               (int)ucv_to_integer(NULL),
(int)ucv_to_integer(ucv_boolean_new(true)),
               (int)(ucv_to_number(ucv_array_new(&vm)) == NULL));

        uc_vm_free(&vm);

        return 0;
}
```

```
ucv_to_number:  [42]=integer/42  [3.9]=double/3.9  [  7abc]=null/null
[abc]=null/null  []=integer/0  [0x10]=integer/16
null=0 true=1 array=1
```

Arrays

```
uc_value_t *ucv_array_new(uc_vm_t *vm);
uc_value_t *ucv_array_new_length(uc_vm_t *vm, size_t len);
size_t      ucv_array_length(uc_value_t *array);
uc_value_t *ucv_array_get(uc_value_t *array, size_t index);         /*
borrowed, NULL if out of range */
uc_value_t *ucv_array_push(uc_value_t *array, uc_value_t *value);   /* owns
value, returns it or NULL */
bool        ucv_array_set(uc_value_t *array, size_t index, uc_value_t
*value);
uc_value_t *ucv_array_pop(uc_value_t *array);
uc_value_t *ucv_array_shift(uc_value_t *array);
bool        ucv_array_unshift(uc_value_t *array, uc_value_t *value);
bool        ucv_array_delete(uc_value_t *array, size_t index, size_t count);
void        ucv_array_sort(uc_value_t *array, int (*cmp)(const void *, const
void *));
void        ucv_array_sort_r(uc_value_t *array, int (*cmp)(uc_value_t *,
uc_value_t *, void *), void *ud);
```

Arrays and objects created with a `vm` argument are linked into that VM's list of live containers, which is also the inventory the cycle collector walks (chapter 18) and the counter behind `vm->alloc_refs` and `gc_interval`. Two consequences:

- the VM must already have been passed to `uc_vm_init()`. A merely declared `uc_vm_t vm = { 0 };` has a null list head, and creating an array or an object through it faults immediately inside `ucv_ref()`;
- passing `NULL` instead of a VM is supported and creates a container that belongs to no VM, invisible to the collector's cycle scan. That is right for a temporary you release yourself, and wrong for anything that may take part in a reference cycle.

Setting an index past the end grows the array and leaves **holes**, which are `NULL` entries indistinguishable from stored nulls when read back with `ucv_array_get()`. Out-of-range reads return `NULL` rather than failing.

`ucv_array_sort()` takes a `qsort()`-style comparator — the elements it hands your comparison function are pointers *to* the array's `uc_value_t *` slots, so a comparator starts with a double indirection. The `_r` variant is friendlier: it passes the values directly plus a user pointer, and it is the one to use if the comparison needs a VM (for calling a script-supplied comparator, say). Note that neither variant runs the `__lt__` metamethod; the script-level `sort()` function does.

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

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static int
by_length(const void *pa, const void *pb)
{
	uc_value_t *a = *(uc_value_t **)pa, *b = *(uc_value_t **)pb;

	return (int)ucv_string_length(a) - (int)ucv_string_length(b);
}

static int
by_name(uc_value_t *a, uc_value_t *b, void *ud)
{
	bool descending = *(bool *)ud;

	return descending ? strcmp(ucv_string_get(b), ucv_string_get(a))
	                  : strcmp(ucv_string_get(a), ucv_string_get(b));
}

int main(void)
{
	uc_vm_t vm = { 0 };
	uc_value_t *a, *tmp;
	bool descending = true;

	uc_vm_init(&vm, &config);

	a = ucv_array_new(&vm);
	ucv_array_push(a, ucv_string_new("ccc"));
	ucv_array_push(a, ucv_string_new("a"));
	ucv_array_push(a, ucv_string_new("bb"));

	ucv_array_sort(a, by_length);
	printf("by length: %s\n", ucv_to_string(&vm, a));

	ucv_array_sort_r(a, by_name, &descending);
	printf("by name:   %s\n", ucv_to_string(&vm, a));

	ucv_array_set(a, 5, ucv_boolean_new(true));
	printf("after set(5): length=%zu get(3)=%s get(5)=%s get(99)=%s\n",
```

```
                ucv_array_length(a), ucv_typename(ucv_array_get(a, 3)),
                ucv_typename(ucv_array_get(a, 5)),
ucv_typename(ucv_array_get(a, 99)));

        tmp = ucv_array_pop(a);
        printf("pop: %s, length=%zu\n", ucv_typename(tmp),
ucv_array_length(a));
        ucv_put(tmp);

        tmp = ucv_array_shift(a);
        printf("shift: %s, length=%zu\n", ucv_typename(tmp),
ucv_array_length(a));
        ucv_put(tmp);

        free(ucv_to_string(&vm, a));
        ucv_put(a);
        uc_vm_free(&vm);

        return 0;
}
```

```
by length: [ "a", "bb", "ccc" ]
by name:   [ "ccc", "bb", "a" ]
after set(5): length=6 get(3)=null get(5)=boolean get(99)=null
pop: boolean, length=5
shift: string, length=4
```

(The `printf` calls above leak the string `ucv_to_string()` returns; that is the one shortcut a throwaway example takes.)

Objects

```
uc_value_t *ucv_object_new(uc_vm_t *vm);
size_t      ucv_object_length(uc_value_t *object);
uc_value_t *ucv_object_get(uc_value_t *object, const char *key, bool
*present);
bool        ucv_object_add(uc_value_t *object, const char *key, uc_value_t
*value);
bool        ucv_object_delete(uc_value_t *object, const char *key);
void        ucv_object_sort(uc_value_t *object, int (*cmp)(const void *,
const void *));
void        ucv_object_sort_r(uc_value_t *object,
                              int (*cmp)(const char *, uc_value_t *, const
char *, uc_value_t *, void *),
                              void *ud);
```

`ucv_object_get()` needs the `present` pointer. A stored `null` and a missing key both come back as `NULL`, and only `*present` tells them apart; pass `NULL` for the third argument if you do not care. Object keys are C strings; to use an arbitrary ucode value as a key, go through `ucv_key_get()` below.

Iteration is the `ucv_object_foreach(object, key, val)` macro. It declares `key` and `val` itself, so it cannot be used twice in the same scope, and it gives you a borrowed `val`:

```
ucv_object_foreach(scope, name, value) {
        printf("%s = %s\n", name, ucv_typename(value));
}
```

`ucv_object_get()` looks at **own keys only**. `ucv_property_get(object, key)` walks the prototype chain instead, and is the function behind "read this property the way a script would" for a plain object:

```c
#include <stdio.h>

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_value_t *o, *proto, *tmp;
        bool present;

        uc_vm_init(&vm, &config);

        o = ucv_object_new(&vm);
        proto = ucv_object_new(&vm);

        ucv_object_add(o, "name", ucv_string_new("lan"));
        ucv_object_add(o, "up", NULL);
        ucv_object_add(proto, "type", ucv_string_new("bridge"));

        tmp = ucv_object_get(o, "name", &present);
        printf("own key:    %s (present=%d)\n", ucv_typename(tmp), present);

        tmp = ucv_object_get(o, "up", &present);
        printf("stored null:%s (present=%d)\n", ucv_typename(tmp), present);

        tmp = ucv_object_get(o, "down", &present);
        printf("missing:    %s (present=%d)\n", ucv_typename(tmp), present);

        ucv_prototype_set(o, proto);   /* the object owns `proto` from here
on */

        printf("object_get through prototype: present=%d\n",
               ucv_object_get(o, "type", &present) != NULL || present);
        printf("property_get through prototype: %s\n",
ucv_typename(ucv_property_get(o, "type")));

        printf("iterating: ");
        ucv_object_foreach(o, key, val) {
                printf("%s=%s ", key, ucv_typename(val));
        }
        printf("\n");

        printf("rendered: %s\n", ucv_to_string(&vm, o));
```

```
        free(ucv_to_string(&vm, o));
        ucv_put(o);
        uc_vm_free(&vm);

        return 0;
}
```

```
own key:    string (present=1)
stored null:null (present=1)
missing:    null (present=0)
object_get through prototype: present=0
property_get through prototype: string
iterating: name=string up=null
rendered: { "name": "lan", "up": null }
```

Note that `ucv_prototype_set()` consumes the prototype reference, so the example does not release `proto` separately; releasing it as well frees it while the object still refers to it, and the damage surfaces much later as an unrelated crash in `malloc()`.

Prototypes may be attached to objects and arrays (`ucv_prototype_set` , `ucv_prototype_get`) and the value stored must be an object. Script-visible behaviour, including the `__get__` -style metamethods and the dispatching accessors, is chapter 12 for the language side and the next section for the C side.

Generic key access

The `ucv_key_*` family is what the VM itself uses for `a[b]` , and it is the right choice for a host that has a `uc_value_t *` key rather than a C string, or that wants script semantics — including the `__get__` , `__set__` and `__delete__` metamethods — to apply:

```
uc_value_t *ucv_key_get(uc_vm_t *vm, uc_value_t *scope, uc_value_t *key);
uc_value_t *ucv_key_set(uc_vm_t *vm, uc_value_t *scope, uc_value_t *key,
uc_value_t *value);
bool        ucv_key_delete(uc_vm_t *vm, uc_value_t *scope, uc_value_t *key);

uc_value_t *ucv_key_rawget(uc_vm_t *vm, uc_value_t *scope, uc_value_t *key);
uc_value_t *ucv_key_rawset(uc_vm_t *vm, uc_value_t *scope, uc_value_t *key,
uc_value_t *value);
bool        ucv_key_rawdelete(uc_vm_t *vm, uc_value_t *scope, uc_value_t
*key);
```

Own keys and numeric array indices win over metamethods, and the metamethods are the fallback; with a `NULL` `vm` nothing is dispatched. The `raw` variants are the escape hatch a `__set__` implementation needs to reach the storage beneath itself instead of recursing. `ucv_key_set()` retains the value it stores and returns a new reference to it, or `NULL` on failure; the others return borrowed references.

`ucv_key_delete()` returns whether the key was removed. An object removes an own key directly; an array or a resource has no own-key storage, so for those a `__delete__` is the only way a key gets handled, and a value with neither the storage nor the metamethod raises a reference error instead of returning false. An array index is refused on the delete path without consulting a metamethod, since elements are positional and `ucv_array_delete()` removes them.

Truth, equality and ordering

```
bool ucv_is_truish(uc_value_t *v);
bool ucv_is_equal(uc_value_t *a, uc_value_t *b);
```

`ucv_is_truish()` implements the script's notion of truth: numbers are false at zero and — unlike most languages — also false at NaN, strings are false when empty, and arrays and objects are **always true**, empty or not.

`ucv_is_equal()` is a strict, type-exact comparison: different types are never equal, so `ucv_is_equal()` says no for the integer `1` and the double `1.0` even though the script's `==` says yes. Use it when you want identity in the sense of chapter 6's `===`. To compare two values the way the script's `<` and `>` do, convert both with `ucv_to_number()` first — that is the coercion the interpreter itself applies — and compare the doubles; there is no public three-way comparator.

Rendering and conversion

```
char *ucv_to_string(uc_vm_t *vm, uc_value_t
*value);            /* malloc'ed, caller frees */
char *ucv_to_jsonstring(uc_vm_t *vm, uc_value_t
*value);        /* malloc'ed, caller frees */
char *ucv_to_jsonstring_formatted(uc_vm_t *vm, uc_value_t *value, char
indent, size_t level);
void  ucv_to_stringbuf_formatted(uc_vm_t *vm, uc_stringbuf_t *buf,
uc_value_t *value,
                                 size_t level, char indent, size_t width);
```

`ucv_to_string()` renders a value the way `print()` would, and it is the quickest way to get something loggable out of the interpreter; it is the function behind the output of every example in this chapter. The JSON variants mirror

the `%J` conversion of chapter 15, including the same indentation conventions: pass `'\t'` for tab indentation, `' '` for spaces with the width in `level`, or `'\1'` for the compact single-line form the macros use.

Because `libucode` itself is built against json-c, hosts that already speak json-c can convert without an intermediate string. `ucv_to_json()` returns a **new** `json_object` you own, and `ucv_from_json()` reads a `json_object` you keep owning:

```
json_object *ucv_to_json(uc_value_t *value);
uc_value_t  *ucv_from_json(uc_vm_t *vm, json_object *jso);
```

```c
#include <stdio.h>

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_value_t *v, *back;
        json_object *jso;
        char *s;

        uc_vm_init(&vm, &config);

        v = ucv_object_new(&vm);
        ucv_object_add(v, "ifname", ucv_string_new("lan"));
        ucv_object_add(v, "mtu", ucv_int64_new(1500));
        ucv_object_add(v, "trunk", ucv_array_new(&vm));

        jso = ucv_to_json(v);
        printf("as json: %s\n", json_object_to_json_string(jso));
        json_object_put(jso);

        back = ucv_from_json(&vm, json_tokener_parse("{\"ports\":[1,2],
\"up\":true}"));
        s = ucv_to_string(&vm, back);
        printf("from json: %s\n", s);

        free(s);
        ucv_put(back);
        ucv_put(v);
        uc_vm_free(&vm);

        return 0;
}
```

```
as json: { "ifname": "lan", "mtu": 1500, "trunk": [ ] }
from json: { "ports": [ 1, 2 ], "up": true }
```

A program that names these two functions must link json-c as well as ucode (`-lucode -ljson-c`); for anything else the library's own json-c dependency stays hidden.

Functions, and the rest

```
uc_value_t *ucv_cfunction_new(const char *name, uc_cfn_ptr_t fn);   /* a
native function */
bool        ucv_is_callable(uc_value_t *v);
bool        ucv_is_arrowfn(uc_value_t *v);
bool        ucv_is_scalar(uc_value_t *v);
uc_value_t *ucv_metamethod_lookup(uc_value_t *v, const char *name);
```

`ucv_is_callable()` is true for closures and native functions, and also for any object, array or resource whose prototype chain provides a `__call__` *function* — the check inspects the type of that method, so a non-callable value stored under `__call__` does not make the value callable. `ucv_metamethod_lookup()` walks that chain and returns the first non-null value found under the name: a function, for invocation, or an object, for delegation, so a caller that means to invoke checks `ucv_is_callable()` on the result first. It looks at the prototype chain only, not at the value's own keys, and it does not honour `__call__` itself; it is how the runtime finds `__get__`, `__set__` and friends (chapter 12), and `__len__` and `__tostring__`. A closure over an already-compiled script function has no public constructor; the public route to a script-defined function is a lookup in the global scope, and the route to a program's entry function is `uc_program_main()` (chapter 43). Chapter 44 deals with writing the functions; chapter 45 with `ucv_resource_*`, which have a lifetime model of their own.

Two remaining groups are worth knowing by name:

- `ucv_is_constant()` / `ucv_set_constant()` flag an array or object as a compile-time constant. Literals in a compiled program carry the flag, and mutation attempts on them fail — `ucv_array_push()` returns `NULL` and `ucv_object_add()` returns `false` — rather than corrupting a value shared between runs. Untagging a value you built yourself is legitimate; untagging a program's literal is not.
- `ucv_gc(uc_vm_t *vm)` runs the cycle collector now instead of at its own threshold, the same call behind the script-level `gc()` of chapter 18.

The virtual machine state

A `uc_vm_t` is the interpreter: its operand stack, its call frames, its global scope, the host-side registry, the exception state, the signal and break machinery, and the settings that decide how it behaves. Chapter 40 showed how one comes into existence and goes away; this chapter is the reference for the state it holds and for the entry points that run code in it.

The struct is defined in the installed `include/ucode/types.h`, so it may live in a host's own allocation — `uc_vm_t vm = { 0 };` on the stack is the normal pattern — but nothing in it is yours to touch. The one exception is noted where it matters.

Lifecycle

```
void uc_vm_init(uc_vm_t *vm, uc_parse_config_t *config);
void uc_vm_free(uc_vm_t *vm);
```

`uc_vm_init` expects either a zero-initialised VM or one that was previously released with `uc_vm_free`. It allocates the global scope, sets the output stream to `stdout`, installs the default exception handler (`uc_vm_output_exception`, which prints the report), wires the per-thread context, and prepares the signal and break state. Passing `NULL` for `config` selects the library's own default, `uc_default_parse_config`, which is fine for the module search path but has `raw_mode` clear — so `include()` renders its argument as a template instead of running it. A host that has a config of its own should pass it:

```
uc_vm_init(&vm, &config);          /* right: the config you compiled with */
uc_vm_init(&vm,
NULL);             /* works, but include() becomes a template render */
```

`uc_vm_free` releases the scope, the registry, the live-value list, the registered resource types, the installed breakpoints and the signal state, and drops the thread-context reference.

One VM per thread. Nothing inside the struct is locked, and the per-thread context the collector and the resource layer use is reference counted at init and free. Give each thread its own VM; do not move values between VMs on different threads.

The state it holds

The fields, in the order the header lists them:

Field	Contents	Released by
`stack`	the operand stack, a vector of `uc_value_t *`	`uc_vm_free`, and each run
`exception`	the pending exception: `type`, `message`, `stacktrace`	the run that raised it, if caught
`callframes`	the call frames, one per active function	each run unwinds its own
`open_upvals`	the chain of open upvalue references	each run
`config`	the `uc_parse_config_t *` passed to `uc_vm_init`	not owned
`globals`	the global scope, an ordinary object	`uc_vm_free`, or `uc_vm_scope_set`
`registry`	the host-side object, created on first use	`uc_vm_free`
`sources`	a table of sources by name, used for error reports	`uc_vm_free`
`values`	the head of the live arrays/objects/ closures list	`uc_vm_free`
`restypes`	resource types registered in this VM (chapter 45)	`uc_vm_free`
`breakpoints`	installed breakpoints (chapter 60)	`uc_vm_free`
`arg`	a union carrying the payload of the last status, e.g. the exit code	not owned
`alloc_refs`, `gc_flags`, `gc_interval`	the collector's counter, switch and threshold	—
`strbuf`	a shared string buffer used while formatting	`uc_vm_free`
`exhandler`	the exception handler	—
`output`	the `FILE *` that script output goes to	not owned
`signal`	the raised-signal bitmask, the handler array, the self-pipe	`uc_vm_free`
	the interrupt flag and its wake-up pipe	`uc_vm_free`

Field	Contents	Released by
`break_requested`, `break_notifyfd`		

Two of these are the reason a VM can be reused: `globals` holds what a script learned between runs, and `registry` holds what the host wants to remember. The rest is execution state, and a completed run leaves it empty — the operand stack and the frame stack both come back at zero after a run that ended in an uncaught exception as cleanly as after one that returned.

Scope

```
uc_value_t *uc_vm_scope_get(uc_vm_t *vm);
void uc_vm_scope_set(uc_vm_t *vm, uc_value_t *ctx);
```

`uc_vm_scope_get` returns the global scope, borrowed; that is the object `uc_stdlib_load()` fills and the one you add host values to. `uc_vm_scope_set` **releases the current scope and takes ownership of the one you pass**, so the previous globals — everything the script declared — are gone:

```
uc_vm_stack_push(&vm, ucv_int64_new(1));      /* a value the VM must release
again */
```

```
uc_value_t *fresh = ucv_object_new(&vm);

ucv_object_add(fresh, "only", ucv_string_new("here"));
uc_vm_scope_set(&vm, fresh);                  /* the old scope is released
here */
```

A fresh scope has no standard library, so the next `uc_stdlib_load(uc_vm_scope_get(&vm))` is normally part of the swap. Swapping is how a host gives an untrusted script a minimal world: a scope carrying the three names it may use and an empty prototype, as `include()` itself demonstrates in chapter 17.

The registry

```
void         uc_vm_registry_set(uc_vm_t *vm, const char *key, uc_value_t
*value);
uc_value_t  *uc_vm_registry_get(uc_vm_t *vm, const char *key);
bool         uc_vm_registry_exists(uc_vm_t *vm, const char *key);
bool         uc_vm_registry_delete(uc_vm_t *vm, const char *key);
```

The registry is an object the VM holds for the host, created on first use. Three properties make it the right place for host state: it is **not** part of the global scope, so a script cannot read or shadow its keys; it is a root for the cycle collector, so anything stored in it is kept alive; and it survives every run, including a swap of the global scope. `uc_vm_registry_set` takes ownership of the value; `get` returns a borrowed reference and `NULL` for a key that is not there.

```c
#include <stdio.h>

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_value_t *other;
        char *s;

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));

        uc_vm_registry_set(&vm, "session", ucv_int64_new(42));
        printf("exists: %s\n", uc_vm_registry_exists(&vm, "session") ?
"true" : "false");
        printf("value: %ld\n", (long)ucv_to_integer(uc_vm_registry_get(&vm,
"session")));
        printf("delete: %s\n", uc_vm_registry_delete(&vm, "session") ?
"true" : "false");
        printf("exists now: %s\n", uc_vm_registry_exists(&vm, "session") ?
"true" : "false");

        /* registry keys are not visible as globals */
        uc_vm_registry_set(&vm, "secret", ucv_string_new("host only"));
        s = ucv_to_string(&vm, ucv_property_get(uc_vm_scope_get(&vm),
"secret"));
        printf("script view of that name: %s\n", s);
        free(s);
        uc_vm_registry_delete(&vm, "secret");

        /* a new scope means a new world */
        other = ucv_object_new(&vm);
        ucv_object_add(other, "only", ucv_string_new("here"));
        uc_vm_scope_set(&vm, other);

        s = ucv_to_string(&vm, ucv_property_get(uc_vm_scope_get(&vm),
"only"));
        printf("after scope_set: only=%s, printf=%s\n", s,
               ucv_typename(ucv_property_get(uc_vm_scope_get(&vm),
"printf")));
        free(s);
```

```
        uc_vm_free(&vm);

        return 0;
}
```

```
exists: true
value: 42
delete: true
exists now: false
script view of that name: null
after scope_set: only=here, printf=null
```

Note the last line: replacing the scope also drops the standard library, because it lived in that scope.

Running a program

```
uc_vm_status_t uc_vm_execute(uc_vm_t *vm, uc_program_t *program, uc_value_t
**retval);
uc_vm_status_t uc_vm_resume(uc_vm_t *vm);
```

`uc_vm_execute` wraps the program's entry function in a closure, pushes a frame for it, runs it to completion and reports how it ended. The program may be executed against the same VM as often as you like; compilation and execution are separate, and holding a `uc_program_t` is what makes a per-request host cheap (chapter 47).

Status	How the run ended	What `*retval` receives
`STATUS_OK`	the program returned	the return value
`STATUS_EXIT`	the program called `exit(n)`	the exit code `n` as a number
`STATUS_BREAK`	`uc_vm_break_request()` interrupted it	`NULL`
`ERROR_RUNTIME`	an uncaught exception	`NULL`
`ERROR_COMPILE`	a run-time compile failed (`loadstring` , `import`)	`NULL`

The return value is a new reference the caller owns, and it is `NULL` for every status but `STATUS_OK` and `STATUS_EXIT` . Two things to know about it: a program with no `return` statement yields whatever the last executed instruction left on the operand stack, which is not a meaningful value (`null` after a run of declarations, the function object after a trailing function declaration), so only an explicit `return` makes it worth reading. And when a run ends with `STATUS_BREAK` the value is not lost but left on the stack for `uc_vm_resume` to deliver.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static const char *
statusname(uc_vm_status_t s)
{
	switch (s) {
	case STATUS_OK: return "STATUS_OK";
	case STATUS_EXIT: return "STATUS_EXIT";
	case STATUS_BREAK: return "STATUS_BREAK";
	case ERROR_COMPILE: return "ERROR_COMPILE";
	default: return "ERROR_RUNTIME";
	}
}

static void
run(uc_vm_t *vm, const char *label, const char *text)
{
	uc_source_t *source = uc_source_new_buffer(label, strndup(text,
strlen(text)), strlen(text));
	uc_program_t *program = uc_compile(&config, source, NULL);
	uc_value_t *retval = NULL;
	char *s;

	uc_source_put(source);

	printf("%-12s %-12s", label, statusname(uc_vm_execute(vm, program,
&retval)));

	s = ucv_to_string(vm, retval);
	printf(" value=%s\n", s);

	free(s);
	ucv_put(retval);
	uc_program_put(program);
}

int main(void)
{
	uc_vm_t vm = { 0 };
```

```
        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));

        run(&vm, "returning", "return 'text';\n");
        run(&vm, "no return", "let a = 1;\nlet b = 2;\n");
        run(&vm, "exit call", "exit(3);\n");

        uc_vm_free(&vm);

        return 0;
}
```

```
returning    STATUS_OK    value=text
no return    STATUS_OK    value=null
exit call    STATUS_EXIT  value=3
```

What a run leaves behind

The exception record is the one piece of execution state that outlives a run. If a program dies on an uncaught exception, `vm->exception` keeps its type, message and stack trace for a host to inspect, and both entry points into the interpreter, `uc_vm_execute()` and `uc_vm_call()` (`uc_vm_invoke()` goes through the latter), clear the record when they are entered, so a failed run does not poison the next one: the VM is always runnable again, and a host that wants the record reads it before the next entry point clears it.

```
        run(&vm, "exit call", "exit(3);\n");
        run(&vm, "after exit", "return 'text';\n");
```

```
exit call    STATUS_EXIT  value=3
after exit   STATUS_OK    value=text
```

The clear on entry also releases the record's message and stack trace, so a long-lived VM that runs many failing programs retains at most the record of the last failure. There is no public reset for the record: the internal `uc_vm_clear_exception()` is static and not exported, and a host that wants to mark a read record as consumed without running anything writes the type field itself:

```
vm.exception.type = EXCEPTION_NONE;
```

The assignment alone does not release the message and stack trace; the next entry point's clear does. A related detail: the exit code a `STATUS_EXIT` reports is read from `vm->arg`, which is the last payload the interpreter stored there.

Calling a function of the script

```
uc_exception_type_t uc_vm_call(uc_vm_t *vm, bool mcall, size_t nargs);
uc_value_t *uc_vm_invoke(uc_vm_t *vm, const char *fname, size_t nargs, ...);
```

The general protocol is stack based, and it is what every other call goes through. Push the callee, then the arguments, oldest last, then call. On return the result is on the stack and must be popped:

```
uc_vm_stack_push(&vm, ucv_get(fn));                  /* the VM takes a
reference */
uc_vm_stack_push(&vm, ucv_int64_new(6));
uc_vm_stack_push(&vm, ucv_int64_new(7));

if (uc_vm_call(&vm, false, 2) == EXCEPTION_NONE)
        result = uc_vm_stack_pop(&vm);                  /* a reference of
yours */
```

`uc_vm_call` reports the exception type, not a status, and `EXCEPTION_NONE` means it got through. Its second argument selects a method call: with `mcall` true the value below the callee — `stack[nargs + 1]` — is used as `this`, which is how `obj.method(...)` reaches the object. The argument count is masked to sixteen bits, so the practical ceiling is 65535 arguments.

The stack API:

```
void         uc_vm_stack_push(uc_vm_t *vm, uc_value_t *value);   /* the VM
owns value now */
uc_value_t  *uc_vm_stack_pop(uc_vm_t *vm);                        /* the
caller owns the result */
uc_value_t  *uc_vm_stack_peek(uc_vm_t *vm, size_t offset);        /*
borrowed, 0 is the top */
```

`uc_vm_stack_peek` does not bounds check; `offset` must be within the current depth.

`uc_vm_invoke` is the convenience form: it looks the name up in the global scope with `ucv_property_get` — so a prototype attached to the scope is honoured — takes the arguments as varargs of type `uc_value_t *`, calls, and returns the result. It borrows each argument, so you keep the references you passed; it returns `NULL` when the name is not callable and when the call raised, and it clears the exception record on entry.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_source_t *source;
        uc_program_t *program;
        uc_exception_type_t ex;
        uc_value_t *fn, *result;
        const char *code = "mul = function (a, b) { return a * b; };\n";
        char *s;

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));

        source = uc_source_new_buffer("lib", strndup(code, strlen(code)),
strlen(code));
        program = uc_compile(&config, source, NULL);
        uc_source_put(source);
        uc_vm_execute(&vm, program, NULL);

        fn = ucv_property_get(uc_vm_scope_get(&vm), "mul");

        uc_vm_stack_push(&vm, ucv_get(fn));
        uc_vm_stack_push(&vm, ucv_int64_new(6));
        uc_vm_stack_push(&vm, ucv_int64_new(7));

        ex = uc_vm_call(&vm, false, 2);
        result = uc_vm_stack_pop(&vm);
        printf("uc_vm_call: exception=%d, result=%ld\n", (int)ex,
(long)ucv_to_integer(result));
        ucv_put(result);

        result = uc_vm_invoke(&vm, "mul", 2, ucv_int64_new(3),
ucv_int64_new(9));
        s = ucv_to_string(&vm, result);
        printf("uc_vm_invoke: %s\n", s);

        free(s);
```

```
        ucv_put(result);
        ucv_put(fn);
        uc_program_put(program);
        uc_vm_free(&vm);

        return 0;
}
```

```
uc_vm_call: exception=0, result=42
uc_vm_invoke: 27
```

`uc_vm_invoke` finds functions that the script **published**, which a top-level `function name() {}` declaration does not provide: a declaration binds in the file scope of the program that made it, and that scope disappears with the run (chapter 5). The forms a host can reach are an assignment (`greet = function (name) { … };`), a module import bound into the scope, or the program's return value — a table of entry points is the pattern that keeps the script's own scope intact:

```
/* driven from C: the host keeps the returned object and calls its members */
return {
        handle: function (request) { … },
        cleanup: function () { … }
};
```

```
uc_vm_stack_push(&vm, ucv_get(ucv_object_get(handlers, "handle", NULL)));
uc_vm_stack_push(&vm, ucv_get(request));

if (uc_vm_call(&vm, false, 1) == EXCEPTION_NONE)
        response = uc_vm_stack_pop(&vm);
```

Interrupting and resuming

```
bool uc_vm_break_requested(uc_vm_t *vm);
void uc_vm_break_request(uc_vm_t *vm);
int  uc_vm_break_notifyfd(uc_vm_t *vm);
```

A break request stops a running program at the next instruction boundary, which is the same point at which pending signals are dispatched. `uc_vm_break_request` sets the flag and writes one byte to the pipe `uc_vm_break_notifyfd` reports, so a thread other than the one running the VM can ask for it to stop and be able to

wake a `select()` that is waiting on the VM. The flag is cleared when it is honoured, so `uc_vm_break_requested` reads false after a `STATUS_BREAK`.

The frames and the stack are left in place, which is what makes `uc_vm_resume` possible: it continues the interrupted run from where it stopped and reports its final status. Together they give a host a way to time slice a script — run it, stop it if it has taken too long, carry on later — without killing the interpreter:

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

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

/* a native function the script calls to ask to be interrupted */
static uc_value_t *
uc_stop(uc_vm_t *vm, size_t nargs)
{
        uc_vm_break_request(vm);

        return NULL;
}

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_source_t *source;
        uc_program_t *program;
        uc_value_t *retval = NULL;
        uc_vm_status_t status;
        const char *code = "steps = 0;\n"
                           "for (let i = 0; i < 500000; i++) {\n"
                           "    steps = steps + 1;\n"
                           "    if (steps == 5) stop();\n"
                           "}\n"
                           "return steps;\n";
        char *s;

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));
        ucv_object_add(uc_vm_scope_get(&vm), "stop",
ucv_cfunction_new("stop", uc_stop));

        source = uc_source_new_buffer("loop", strndup(code, strlen(code)),
strlen(code));
        program = uc_compile(&config, source, NULL);
        uc_source_put(source);

        status = uc_vm_execute(&vm, program, &retval);
        printf("long loop: %s, break pending: %s\n",
               status == STATUS_BREAK ? "STATUS_BREAK" : "other",
```

```
                uc_vm_break_requested(&vm) ? "true" : "false");

        status = uc_vm_resume(&vm);
        s = ucv_to_string(&vm, uc_vm_stack_peek(&vm, 0));
        printf("resume: %s, value on stack: %s\n", status == STATUS_OK ?
"STATUS_OK" : "other", s);

        free(s);
        uc_vm_stack_pop(&vm);
        ucv_put(retval);
        uc_program_put(program);
        uc_vm_free(&vm);

        return 0;
}
```

```
long loop: STATUS_BREAK, break pending: false
resume: STATUS_OK, value on stack: 500000
```

Read the second line carefully: the run stopped after five iterations and, after resume, reached all five hundred thousand, because resume continues rather than aborts. The value `return steps` produced is on the stack after the resume — `uc_vm_execute` had already returned and did not take it — so the host pops it itself.

Signals

The VM carries a signal layer that keeps signal handling inside the cooperative model: the operating system never runs script code. A script registers a handler with the `signal()` builtin (chapter 20), which stores it in a per-VM array indexed by signal number. Signal numbers reach that array from two places: the dispositions `uc_vm_init` installs when `config->setup_signal_handlers` is set, and `uc_vm_signal_raise()`, which a host calls from wherever it learns about the signal — its own handler, a `sigaction`, a control socket, a signal fd it watches:

```
void          uc_vm_signal_raise(uc_vm_t *vm, int signo);
uc_exception_type_t uc_vm_signal_dispatch(uc_vm_t *vm);
int           uc_vm_signal_notifyfd(uc_vm_t *vm);
void          uc_vm_signal_handlers_ensure(uc_vm_t *vm);
```

`uc_vm_signal_raise` records the signal and writes it to a self-pipe; `uc_vm_signal_dispatch` drains the pipe and runs the handler of each recorded signal, returning the first exception the handler raised. The interpreter calls dispatch itself after each instruction, so a running program picks signals up by itself. A host that is not executing script code at the time — a daemon waiting in `select()` — calls it from its own loop, and `uc_vm_signal_notifyfd` is the descriptor to wait on:

```
FD_SET(uc_vm_signal_notifyfd(&vm), &readfds);

/* ... after select() returns ... */

uc_vm_signal_dispatch(&vm);
```

The pipe exists only if the VM was set up for it, so a host that did not set `setup_signal_handlers` says so explicitly before relying on any of the above, which is what `uc_vm_signal_handlers_ensure()` is for:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_source_t *source;
        uc_program_t *program;
        const char *code = "count = 0;\n"
                           "signal('USR1', function (sig) { count = count +
sig; });\n";
        char *s;

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));
        uc_vm_signal_handlers_ensure(&vm);

        printf("notify fd is valid: %s\n", uc_vm_signal_notifyfd(&vm) >= 0 ?
"true" : "false");

        source = uc_source_new_buffer("sig", strndup(code, strlen(code)),
strlen(code));
        program = uc_compile(&config, source, NULL);
        uc_source_put(source);
        uc_vm_execute(&vm, program, NULL);

        uc_vm_signal_raise(&vm, SIGUSR1);

        printf("dispatch: %d, ", (int)uc_vm_signal_dispatch(&vm));

        s = ucv_to_string(&vm, ucv_property_get(uc_vm_scope_get(&vm),
"count"));
        printf("handler saw signal %s\n", s);

        free(s);
        uc_program_put(program);
        uc_vm_free(&vm);
```

```
        return 0;
}
```

```
notify fd is valid: true
dispatch: 0, handler saw signal 10
```

Without the `uc_vm_signal_handlers_ensure()` call, `uc_vm_signal_dispatch` finds no pipe and returns immediately, and the raise writes to descriptor `-1`: the script's handler simply never runs, silently. The function is safe to call more than once.

Collecting, tracing, output

```
bool       uc_vm_gc_start(uc_vm_t *vm, uint16_t interval);
bool       uc_vm_gc_stop(uc_vm_t *vm);
uint32_t   uc_vm_trace_get(uc_vm_t *vm);
void       uc_vm_trace_set(uc_vm_t *vm, uint32_t level);
```

The cycle collector is off by default (chapter 18). `uc_vm_gc_start` turns it on with a given interval, the number of allocations after which a pass runs, `UC_GC_DEFAULT_INTERVAL` being 1000. Both functions report whether **the state changed**, not whether they succeeded, so calling `uc_vm_gc_start` twice with the same interval returns true and then false:

```
printf("%s %s %s\n", uc_vm_gc_start(&vm, 50) ? "on" : "same",
                     uc_vm_gc_start(&vm, 50) ? "on" : "same",
                     uc_vm_gc_stop(&vm) ? "off" : "was off");
/* on same off */
```

Collection is also driven by `vm->alloc_refs`, the running count of containers created against this VM, and `ucv_gc()` (chapter 41) runs one pass directly. A host that creates many short-lived containers and has not started the collector is holding cycles until it does.

`uc_vm_trace_set` at level 1 prints each stack operation, each frame and each source context to `stderr` while code runs — the same facility as the interpreter's `-t` flag, and the quickest way to see what a call protocol of yours is actually pushing.

Script output goes to `vm->output`, which `uc_vm_init` sets to `stdout`. `print`, `printf` and the interpreter's own output path all write there, so pointing it elsewhere captures a script without a pipe:

```
FILE *capture = tmpfile();

vm.output = capture;
uc_vm_invoke(&vm, "report", 0);
fflush(NULL);
rewind(capture);
/* read the script's output back from `capture` */

vm.output = stdout;
```

The stream stays the host's property: do not close it while the VM still points at it, and restore it before releasing the VM. A host that captures output this way still sees exceptions on `stderr`, since the exception handler writes there rather than to `output`.

Errors

```
uc_exception_handler_t *uc_vm_exception_handler_get(uc_vm_t *vm);
void  uc_vm_exception_handler_set(uc_vm_t *vm, uc_exception_handler_t
*handler);
void  uc_vm_raise_exception(uc_vm_t *vm, uc_exception_type_t type, const
char *fmt, ...);
uc_value_t *uc_vm_exception_object(uc_vm_t *vm);
```

The handler is called when an exception reaches the top of a run, before `uc_vm_execute` returns with its status, with the VM's `uc_exception_t` attached. The default prints the message, the offending source line and a stack trace; a host replaces it to route reports into a log instead, or to stay silent while it inspects the failure itself:

```
static void
log_exception(uc_vm_t *vm, uc_exception_t *ex)
{
        fprintf(stderr, "script failed: %s\n", ex->message ? ex->message :
"?");
}

uc_vm_exception_handler_set(&vm, log_exception);
```

`uc_vm_raise_exception` is how a native function fails: it sets the exception state with a formatted message, and the interpreter unwinds from it as from any other failure. Chapter 44 covers the native side and chapter 46 the exception objects, whose fields (`type`, `message`, `stacktrace`) `uc_vm_exception_object` assembles into an ordinary ucode value. Note that this getter assumes there is something to report: calling it when no exception is pending walks into `strlen(NULL)` and faults, so ask the status first.

The debugger's system breakpoint, `UC_BREAKPOINT_UNCAUGHT_EXCEPTION`, can be installed as a breakpoint whose `ip` is that sentinel; its callback then runs with the call frames still intact just before an exception that nothing will catch starts unwinding. That is chapter 60's subject.

A host's checklist for reuse

For the pattern the shipped `state-reuse` example shows — one VM, many programs — these are the properties worth knowing:

- the operand stack and the frame stack come back empty after a run, including after one that failed;
- the global scope and the registry keep everything they held, by design; a scope swap clears the first and leaves the second;
- an uncaught exception leaves its record behind and blocks the next run until the type is cleared;
- sources keep their `FILE *` open while the program that read them is alive, so the descriptor footprint is the number of live sources plus whatever the script itself holds open — steady per run, growing with retained programs and modules;
- the exit code in `vm->arg` is the last one the interpreter stored, and `alloc_refs` only grows until the collector runs a pass.

Compiling sources

Compilation is the step that turns text into something a VM can run. Three objects take part in it: a *source* holds the text or the precompiled bytes, a *program* holds the compiled code, and the VM executes a program. This chapter is about the first two and about what the compiler tells you when the text is not a program; the byte format they serialise to is chapter 47.

The public surface is small:

```
uc_program_t *uc_compile(uc_parse_config_t *config, uc_source_t *source,
char **errp);

uc_source_t  *uc_source_new_file(const char *path);
uc_source_t  *uc_source_new_buffer(const char *name, char *buf, size_t len);
uc_source_t  *uc_source_get(uc_source_t *source);
void          uc_source_put(uc_source_t *source);
size_t        uc_source_get_line(uc_source_t *source, size_t *offset);

uc_program_t *uc_program_new(void);
uc_program_t *uc_program_get(uc_program_t *program);
void          uc_program_put(uc_program_t *program);
uc_value_t *uc_program_main(uc_vm_t *vm, uc_program_t *program);
void          uc_program_write(uc_program_t *program, FILE *fp, bool debug);
uc_program_t *uc_program_load(uc_source_t *source, char **errp);
```

One entry point does the compiling. Everything else is ownership, naming and the two conversion functions that move programs between memory and files.

Ownership

The headers are explicit, and the rules are the same ones chapter 41 lays out for values:

Call	Ownership
`uc_source_new_buffer(name, buf, len)`	the source takes ownership of `buf`, which is freed when the source is released
`uc_source_new_file(path)`	opens the file and keeps the `FILE *` for the source's life
`uc_source_get` / `uc_source_put`	the only ways to acquire and release a source
`uc_compile`	returns a new reference to the program, or `NULL` with a `malloc`ed message in `*errp`
`uc_program_load(source, &err)`	takes ownership of the source, including on failure
`uc_program_main(vm, program)`	returns a new reference to a closure over the entry function, owned by the caller
`uc_program_get` / `uc_program_put`	the only ways to acquire and release a program

That is why chapter 40's host releases its source right after compiling and still gets source context in its error reports: the program took its own reference to the sources it was compiled from.

```
uc_source_t *source = uc_source_new_buffer("config.uc",
                                           strndup(text, strlen(text)),
strlen(text));
uc_program_t *program = uc_compile(&config, source, &error);

uc_source_put(source);                          /* the program keeps what it
needs */
```

What the configuration decides

Chapter 40 covered `uc_parse_config_t` as a whole; four of its fields are specifically compile-time:

Field	Effect on compilation
`raw_mode`	false compiles the source as a template, so its text is emitted rather than executed
`strict_declarations`	requires names to have been declared: an assignment no longer creates a global, and redeclaring a local in the same scope is a syntax error
`module_search_path`	the `*` templates that resolve `import` and module names
`force_dynlink_list`	names to compile as a run-time load of `<name>.so` rather than to resolve now
`compile_module`	compiles the source as a module, so its `export` statements are legal

The last is the reason a file written as a module can be compiled on its own: the interpreter exposes it as `-cmodule`, and without it the same file is rejected with "Exports may only appear at top level of a module".

When compilation fails

`uc_compile` returns `NULL` and, if you passed an address, a `malloc`ed message in `*errp` that you release with `free()`. The message is already a complete report — the class of failure, the line and byte, and the offending line quoted with a marker under the position — because it is built by the same code that formats run-time errors. Pass `NULL` instead of an address if you only need to know that it failed.

A compile-time report names the position (`In line 1, byte 5:`) while a run-time report names the source (`In config.uc, line 3, byte 7:`), so it is the run-time reports that a log line has to identify for whoever reads it.

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

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static void
try(const char *label, const char *text)
{
        uc_source_t *source = uc_source_new_buffer(label, strndup(text,
strlen(text)), strlen(text));
        uc_program_t *program;
        uc_vm_t vm = { 0 };
        char *error = NULL;

        program = uc_compile(&config, source, &error);
        uc_source_put(source);

        if (!program) {
                printf("%s: %s\n", label, error);
                free(error);

                return;
        }

        printf("%s: compiled\n", label);

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));
        printf("   status=%d\n", (int)uc_vm_execute(&vm, program, NULL));
        uc_vm_free(&vm);
        uc_program_put(program);
}

int main(void)
{
        /* reports arrive on standard error, so line-buffer standard output
to keep a
           merged transcript in the order the events happened */
        setvbuf(stdout, NULL, _IOLBF, 0);

        try("ok", "return 1;\n");
        try("syntax", "let = ;\n");
```

```
        try("text", "1 + 2\n");

        config.strict_declarations = true;
        try("strict", "counter = 1;\nreturn counter;\n");
        config.strict_declarations = false;

        config.raw_mode = false;
        try("as template", "1 + 2\n");

        return 0;
}
```

```
ok: compiled
   status=0
syntax: Syntax error: Expecting variable name
In line 1, byte 5:

 `let = ;`
      ^-- Near here



text: compiled
   status=0
strict: compiled
Reference error: access to undeclared variable counter
In strict, line 1, byte 11:

 `counter = 1;`
            ^-- Near here


   status=4
as template: compiled
1 + 2
   status=0
```

Three things in that transcript are worth a host's attention. The syntax message ends with a blank line, so prefixing it is prettier than suffixing it. The strict run compiles and only fails when it runs, because `strict_declarations` does not make an undeclared name a compile-time error: what it does is stop an assignment from silently creating a global, and stop a local name from being redeclared in the same scope. The failure it produces is a `Reference error` at the point of use, which is the behaviour a configuration author wants and a host has to be ready to see at run time rather than at load time.

The last two lines are the `raw_mode` switch at work on one and the same text. As a program, `1 + 2` is an expression statement that evaluates and discards its value and prints nothing; as a template, the same text is literal text and comes out on the other side. Both report "compiled", because a template is not an error case and a host cannot tell the two modes apart from the return value. Set the mode deliberately.

Sources and their names

A source carries the name that appears in every report, and for a file source it is also the base for resolving relative paths — an `include` or a module name given without a slash resolves against the including source's path, so a file source is named with the path it was opened from and a buffer source should be named with the path it stands for:

```
/* a script fetched over the network, compiled as if it lived in /etc/ucode
*/
uc_source_t *s = uc_source_new_buffer("/etc/ucode/hooks/upgrade.uc", buf,
len);
```

`uc_source_new_file` keeps the file open in the source until it is released, which is why a long-lived VM that keeps many sources compiled holds open descriptors for all of them. Read the text yourself and use `uc_source_new_buffer` if you would rather not hold the descriptor.

The one accessor in the public interface maps a byte offset back to a line:

```
size_t uc_source_get_line(uc_source_t *source, size_t *offset);
```

It reads a per-line byte index the source may carry, `source->lineinfo`, in which one byte records one line's length with its high bit marking a line break. The in-out argument is what makes it usable at both ends of a report: pass the byte offset, and what comes back in it is the one-based position of that byte inside its line. Rendering the line itself with a marker under the position is what the interpreter's own reports do, and that formatting lives in `uc_source_context_format()`, declared in `ucode/internal/lib.h` and therefore not reachable from a host outside this tree. Chapter 46 shows the portable route to the same text: read the exception object, whose `message` is the formatted report already.

The index is not built for a source that has only been compiled from text — the compiler carries line and byte information with the code it emits rather than with the source — so on a plain text source the call has nothing to walk and answers with line 1 and the offset returned unchanged apart from the one-based adjustment:

```
uc_source_t *s = uc_source_new_buffer("lines.uc",
                                      strndup("one\ntwo\nthree four\n", 17),
17);
size_t offset = 6;
size_t line = uc_source_get_line(s, &offset);

printf("byte 6 reported as line %zu, position %zu\n", line, offset);

uc_source_put(s);
```

```
byte 6 reported as line 1, position 7
```

Sources that do carry the index are the ones loaded from a precompiled file written with source information and the ones the `debug` module has been asked about: it walks `lineinfo` to turn a breakpoint's line and column into a byte offset. So treat `uc_source_get_line()` as a helper for those, and get position information out of the exception object or the debug module's location functions in every other case.

Programs

`uc_program_main` returns a closure over the program's top-level function — the one `uc_vm_execute` runs — or `NULL` for a program with nothing in it. It is the handle by which a host can tell an empty load from a usable one, and the way to get at a compiled program as a callable value. The returned value is a new reference, so a host that does not keep it releases it with `ucv_put()`:

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

#include <ucode/ucode.h>

static uc_program_t *
compile(const char *tag, const char *text)
{
        uc_parse_config_t config = { 0 };
        uc_program_t *program;
        uc_source_t *source;
        char *error = NULL;

        config.raw_mode = true;

        source = uc_source_new_buffer(tag, strdup(text), strlen(text));
        program = uc_compile(&config, source, &error);
        uc_source_put(source);

        if (!program) {
                printf("%s: %s\n", tag, error);
                free(error);
                exit(1);
        }

        return program;
}

int
main(void)
{
        uc_parse_config_t config = { 0 };
        uc_value_t *entry;
        uc_program_t *empty, *program;
        uc_vm_t vm = { 0 };

        config.raw_mode = true;
        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));

        empty = uc_program_new();
        entry = uc_program_main(&vm, empty);
        printf("empty program entry: %s\n", entry ? "present" : "absent");

        program = compile("demo", "\"entry closure text\";");
```

```
        entry = uc_program_main(&vm, program);
        printf("compiled entry: %s, callable: %d\n",
               ucv_typename(entry), ucv_is_callable(entry));
        ucv_put(entry);

        uc_program_put(empty);
        uc_program_put(program);
        uc_vm_free(&vm);

        return 0;
}
```

```
empty program entry: absent
compiled entry: closure, callable: 1
```

The entry function itself — `uc_function_t *`, owned by the program — is reached only by the internal `uc_program_entry()`, declared in `include/ucode/internal/program.h` and hidden; the closure returned by `uc_program_main()` is the public form of the same thing. Because `uc_program_main()` needs a VM to build the closure in, the emptiness test is a VM-taking call. A program also holds no other named function: the rest are reached by name through the globals, as chapter 42 describes.

A program is a reference counted object, and executing it does not consume it: the pattern for a service is to compile once at start-up, keep the one reference, and call `uc_vm_execute` per request (chapter 42). A program also holds its sources, so a report from the tenth run still quotes the text.

Writing and loading

```
void          uc_program_write(uc_program_t *program, FILE *fp, bool debug);
uc_program_t *uc_program_load(uc_source_t *source, char **errp);
```

`uc_program_write` serialises a program to a stream. Its third argument is **not** a compression flag, in spite of the header; it selects debug information, and the source text travels with the file only when it is set. That single bit is the difference between a deployed program that can say where it failed and one that cannot, and it is roughly four times the size:

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

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static const char failing[] = "function boom() {\n    let x = 1;\n    x();\n}
\n\nboom();\n";

static void
write_out(uc_program_t *program, const char *path, bool debug)
{
        FILE *fp = fopen(path, "wb");

        uc_program_write(program, fp, debug);
        fclose(fp);
}

static void
run_file(const char *label, const char *path)
{
        uc_vm_t vm = { 0 };
        uc_source_t *source = uc_source_new_file(path);
        uc_program_t *program;
        uc_value_t *rv = NULL;
        char *error = NULL;

        program = uc_program_load(source, &error);

        if (!program) {
                printf("%s: %s", label, error);
                free(error);

                return;
        }

        printf("== %s\n", label);
        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));
        printf("   status=%d\n", (int)uc_vm_execute(&vm, program, &rv));
        ucv_put(rv);
        uc_vm_free(&vm);
        uc_program_put(program);
```

```
}

int main(void)
{
        /* reports arrive on standard error, so line-buffer standard output
to keep a
           merged transcript in the order the events happened */
        setvbuf(stdout, NULL, _IOLBF, 0);

        uc_source_t *source = uc_source_new_buffer("boom.uc",
                strndup(failing, strlen(failing)), strlen(failing));
        uc_program_t *program = uc_compile(&config, source, NULL);

        uc_source_put(source);

        write_out(program, "/tmp/ucode-ch43-debug.uc.o", true);
        write_out(program, "/tmp/ucode-ch43-bare.uc.o", false);

        run_file("written with debug set", "/tmp/ucode-ch43-debug.uc.o");
        run_file("written with debug clear", "/tmp/ucode-ch43-bare.uc.o");

        uc_program_put(program);

        return 0;
}
```

```
== written with debug set
Type error: left-hand side is not a function
In boom.uc, line 3, byte 7:
  (1 tail call frames omitted)

 `    x();`
        ^-- Near here


   status=4
== written with debug clear
Type error: left-hand side is not a function
In [no source], line 1, byte 10:
  (1 tail call frames omitted)


   status=4
```

The program still runs identically either way, and both runs reach the same type error; only the report differs. A bare file reports the source as `[no source]`, and its line and byte are meaningless because the line index is gone. The debug form recovers the name given at compile time, the line, the byte and the quoted text. Deploy with debug set unless the size matters on the device, in which case keep the source files around so a byte offset can at least be looked up by hand.

`uc_program_load` accepts either form of source — text or precompiled bytes — and takes ownership of it in both the success and the failure case, since the source is what a precompiled program quotes its lines from. The file form is recognised by its magic word, `\033ucb`, and the four-byte version number packed into the flags that follow it.

Versions

Bytecode is checked against the library that reads it, and a mismatch is reported through the same error string rather than by running nonsense. The version is `UCODE_BYTECODE_VERSION`, currently `0x02`. A file begins with the magic word and a big-endian flags word whose first byte is the version, so the check is three lines of C to demonstrate:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

int main(void)
{
        uc_source_t *source;
        uc_program_t *program;
        unsigned char header[8], byte = 0x7f;
        FILE *fp;
        char *error = NULL;

        source = uc_source_new_buffer("tiny.uc", strndup("return 1;\n", 10),
10);
        program = uc_compile(&config, source, NULL);
        uc_source_put(source);

        fp = fopen("/tmp/ucode-ch43-version.uc.o", "wb");
        uc_program_write(program, fp, false);
        fclose(fp);
        uc_program_put(program);

        fp = fopen("/tmp/ucode-ch43-version.uc.o", "r+b");
        if (fread(header, 1, 8, fp) == 8)
                printf("magic=0x%02x%02x%02x%02x
flags=0x%02x%02x%02x%02x\n",
                       header[0], header[1], header[2], header[3],
                       header[4], header[5], header[6], header[7]);

        fseek(fp, 4, SEEK_SET);
        fwrite(&byte, 1, 1, fp);
        fclose(fp);

        program = uc_program_load(uc_source_new_file("/tmp/ucode-ch43-
version.uc.o"), &error);

        if (!program) {
                printf("%s", error);
                free(error);

                return 0;
```

```
        }

        printf("loaded after all\n");
        uc_program_put(program);

        return 0;
}
```

```
magic=0x1b756362 flags=0x02000000
Bytecode version mismatch, got 0x7f, expected 0x02
```

Which is the practical form of a rule worth stating plainly: a deployed `.uc.o` belongs to the library version that produced it, and an update of the library means recompiling the deployed programs. There is no compatibility promise across versions, which is why the number is checked rather than negotiated. The other flag bits are what chapter 47 itemises: debug information, source information, embedded source text, and whether the program exports names.

Doing it from the build system

The interpreter is also its own compiler driver, chosen by the name it is invoked under. The build creates two symbolic links to it, and the mode decides the defaults rather than the behaviour:

Name	Mode
`ucode`	run the program
`utpl`	compile in template mode (`raw_mode` cleared), for rendering
`ucc`	compile and write a program, defaulting the output to `./uc.out`

```
$ ucode -cmodule -o upgrade.uc.tmp upgrade.uc && mv upgrade.uc.tmp upgrade.uc
$ ls -l upgrade.uc.tmp
-rwxrwxr-x 1 jow jow 317 Sep 18 21:32 upgrade.uc.tmp
```

`-c` is what selects the compile-mode flags, and its words are `module` , `dynlink=<name>` , `interp=<path>` and `no-interp` — the same four that appear in `uc_parse_config_t` , with `-o` naming the output and `-s` stripping debug information. The flag words are attached to the letter (`-cmodule` , `-cdynlink=uci`) for the reason chapter 40 gives, and `-o` has to come after the `-c` whose output it names, because selecting compile mode resets the output path (chapter 47). The `interp` words are about the shebang line the output gets ahead of its bytecode, which is what lets a precompiled file be made executable and run by itself. A `Makefile` or `CMakeLists.txt` rule that precompiles a device's scripts is then one line, and the device needs no compiler at run time.

Native functions

A native function is a C function a script can call. It is what every standard-library function is: `length`, `printf`, `fs.readfile`, `uci.get` — none of them is special, all of them are a C function wrapped in a value and stored in a scope. This chapter is about writing one, about the contract between it and the VM, and about the handful of helpers `ucode/lib.h` provides so you write the wrapper once rather than forty times.

```
typedef uc_value_t *(*uc_cfn_ptr_t)(uc_vm_t *vm, size_t nargs);

uc_value_t *ucv_cfunction_new(const char *name, uc_cfn_ptr_t fptr);

/* ucode/lib.h */
typedef struct {
        const char *name;
        uc_cfn_ptr_t func;
} uc_function_list_t;

void          uc_stdlib_load(uc_value_t *scope);
uc_cfn_ptr_t  uc_stdlib_function(const char *name);

bool uc_function_register(uc_value_t *object, const char *name, uc_cfn_ptr_t
fn);
bool uc_function_list_register(uc_value_t *object, const uc_function_list_t
*list);

uc_value_t *uc_fn_arg(n);                       /* n-th argument, or NULL
when absent */
uc_value_t *_uc_fn_this_res(vm);                /* the receiver of a method
call */
void       *uc_fn_this(expectedtype);           /* the receiver's data
pointer */
void       *uc_fn_thisval(expectedtype);        /* the receiver resource
itself */

uc_exception_type_t uc_call(size_t nargs);      /* uc_vm_call(vm, false,
nargs) */
```

The function and the value

A native is a `uc_cfunction_t`: a value of type `UC_CFUNCTION` holding one function pointer and a name.

```
static uc_value_t *
uc_myagent_now(uc_vm_t *vm, size_t nargs)
{
        return ucv_double_new(ucv_to_double(uc_fn_arg(0)) + 1.0);
}

/* ... */
uc_function_register(uc_vm_scope_get(&vm), "advance", uc_myagent_now);
```

The wrapper is created by `ucv_cfunction_new`, which is what `uc_function_register` calls; the name is stored in the value and is what a trace and a stack report print, so give it the name the script calls it by, or `NULL` for a function that exists only as a value. In a script the result is an ordinary function value: it can be stored, passed, and called. There is nothing to declare, and no type table to tell the VM about — a native accepts any number of any values, and the argument count is the only arity information there is.

`ucv_cfunction_new` takes no VM, so a native can be created before `uc_vm_init`, unlike the containers of chapter 41 which must not be.

Registering

`uc_function_register` is a macro over `ucv_object_add` plus `ucv_cfunction_new`:

```
#define uc_function_register(object, name, function) \
        ucv_object_add(object, name, ucv_cfunction_new(name, function))
```

So "registering" a native is putting it in an object, and the object you put it in is the namespace it is reached through. Into the global scope it is a bare name; into an object you hand to the script it is `thatobject.name`. Registering a name that the standard library also uses replaces it for the rest of the VM's life, which is how you wrap or shadow a built-in.

For more than one function, describe them in a table:

```
static const uc_function_list_t handlers[] = {
        { "advance", uc_myagent_advance },
        { "reset",   uc_myagent_reset   }
};

uc_function_list_register(scope, handlers);
```

The list is walked in order and each entry is registered the same way, so the return value is true when all of them went in. `uc_stdlib_load(scope)` is the same mechanism applied to the standard library's own tables into a scope

(chapter 40), and `uc_stdlib_function("length")` gives you the function pointer behind a name already in the library, so a native of your own can call the built-in implementation rather than duplicate it. To take a name away again:

```
ucv_object_delete(uc_vm_scope_get(&vm), "advance");
```

The VM keeps no separate table of natives. A name looked up in a scope finds a native exactly as it finds a script function, so the script has no way to tell the two apart and no reason to.

The call

`uc_vm_call_native` pushes a call frame for the call, calls the function, and pops the frame again:

```
        frame = uc_vector_push(&vm->callframes, {
                .stackframe = vm->stack.count - nargs - 1,
                .cfunction = fptr,
                .ctx = ctx,
                .mcall = mcall
        });

        res = fptr->cfn(vm, nargs);
```

Three consequences follow from those few lines.

The function always gets its own frame, even when the script wrote the call where a tail call would have been allowed; a native cannot be a bottomless tail call, so recursion through one consumes a frame the way a script call does and hits the same limit.

The arguments sit above the callee on the operand stack, which is why `uc_fn_arg` reads them by counting back from the top:

```
static inline uc_value_t *
_uc_fn_arg(uc_vm_t *vm, size_t nargs, size_t n)
{
        if (n >= nargs)
                return NULL;

        return uc_vm_stack_peek(vm, nargs - n - 1);
}
```

Use the macro rather than reach into the stack yourself, because it is the one accessor that answers `NULL` for a position that was not passed. It is written against a function whose parameters are the conventional `vm` and `nargs`,

which it takes from the enclosing scope: There is no separate count query for a native; `nargs` is the parameter the VM handed you.

The frame carries the call context, which is what makes a native usable as a method (see *Receivers* below).

An argument is **borrowed**, as everywhere else: the value belongs to the caller's stack slot and is released when the frame is popped. Take a reference with `ucv_get` if you keep it, and release it when you are done with it. The same rule in the other direction: a value you received as an argument and store into a longer-lived container needs the reference, exactly as in a script (chapter 18).

An absent argument is `NULL`, not `null`, and the conversion helpers accept it without complaint:

```
double x = ucv_to_double(uc_fn_arg(0));      /* 0.0 when nothing was passed
*/
```

That is convenient up to a point and silent after it, so a function that needs its argument should look for the pointer itself. A complete example of both patterns:

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

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static uc_value_t *
report(uc_vm_t *vm, size_t nargs)
{
        uc_stringbuf_t *buf = ucv_stringbuf_new();
        size_t i;

        ucv_stringbuf_printf(buf, "%zu given:", nargs);

        for (i = 0; i < nargs + 1; i++)
                ucv_stringbuf_printf(buf, " %s", uc_fn_arg(i) ?
ucv_typename(uc_fn_arg(i)) : "absent");

        return ucv_stringbuf_finish(buf);
}

static uc_value_t *
plusone(uc_vm_t *vm, size_t nargs)
{
        if (uc_fn_arg(0) == NULL)
                return NULL;                            /* null, see Returning
below */

        return ucv_double_new(ucv_to_double(uc_fn_arg(0)) + 1.0);
}

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_value_t *scope;
        uc_program_t *program;
        uc_source_t *source;
        const char *code =
                "print(report(1, 'two'), '\\n');\n"
                "print(report(), '\\n');\n"
                "print('plusone(41) = ', plusone(41), '\\n');\n"
                "print('plusone() is null: ', plusone() == null, '\\n');\n";
```

```
        uc_vm_init(&vm, &config);
        scope = uc_vm_scope_get(&vm);
        uc_stdlib_load(scope);

        uc_function_register(scope, "report", report);
        uc_function_register(scope, "plusone", plusone);

        source = uc_source_new_buffer("args.uc", strdup(code), strlen(code));
        program = uc_compile(&config, source, NULL);
        uc_source_put(source);

        printf("status=%d\n", (int)uc_vm_execute(&vm, program, NULL));

        uc_program_put(program);
        uc_vm_free(&vm);

        return 0;
}
```

```
2 given: integer string absent
0 given: absent
plusone(41) = 42
plusone() is null: true
status=0
```

Note that `report` walks to `nargs + 1` deliberately, so the last column shows what an over-run reads. It is `NULL` — which is `null` to the script side as well — and the standard library answers the same way, which is why built-ins so often behave as though extra arguments were ignored: they were never seen.

The type names in that transcript are what `ucv_typename` answers and they are not the words a script sees from `type()` (chapter 20): `ucv_typename` says `integer` and `double` where the script sees `int` and `double`, and both a script function and a native are `function` to the script and `closure` or `cfunction` here. Use the script's words when a message is for a person reading script output.

Returning

Return a value you own, exactly as chapter 41 has you: a fresh reference from one of the constructors, or an existing value with `ucv_get` on it. The VM pushes what you return without looking at it, so there is no declared return type to satisfy, no conversion, and no complaint about returning an array where the script expected a number.

Returning `NULL` *is* how a native returns null. In the value representation of chapter 41 the null value is the null pointer, so `ucv_typename(NULL)` answers `"null"`, the script's `x == null` holds, and there is no constructor to

call — a function of the shape `ucv_null_new()` does not exist and nothing in the tree has a name for the null value beyond `NULL` itself. Around ninety of the standard library's own functions return null by returning `NULL`.

For text, build the string with a string buffer, which is the pattern the standard library follows for anything longer than a literal:

```
        uc_stringbuf_t *buf = ucv_stringbuf_new();

        ucv_stringbuf_printf(buf, "%s/%zu", name, count);

        return ucv_stringbuf_finish(buf);        /* the buffer becomes the
value */
```

`ucv_stringbuf_printf` reaches json-c's formatter, so a host that uses it links `-ljson-c` (chapter 41). `ucv_stringbuf_finish` releases the buffer and hands back the string value it held, so the one reference you must now release is the value's.

Raising

A native reports a failure by raising an exception, and the script's `try` / `catch` catches it like any other:

```
void uc_vm_raise_exception(uc_vm_t *vm, uc_exception_type_t type, const char
*fmt, ...);
```

The message is a `printf` format, which is what makes the call worth its characters:

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

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static uc_value_t *
needint(uc_vm_t *vm, size_t nargs)
{
        uc_value_t *arg = uc_fn_arg(0);

        if (arg == NULL) {
                uc_vm_raise_exception(vm, EXCEPTION_TYPE, "expecting a
number, got nothing");

                return NULL;                            /* nothing is pushed
while an exception is pending */
        }

        if (ucv_type(arg) != UC_INTEGER && ucv_type(arg) != UC_DOUBLE) {
                uc_vm_raise_exception(vm, EXCEPTION_TYPE, "expecting a
number, got %s", ucv_typename(arg));

                return NULL;
        }

        return ucv_int64_new(ucv_to_integer(arg) * 2);
}

static uc_value_t *
fail(uc_vm_t *vm, size_t nargs)
{
        (void) nargs;

        uc_vm_raise_exception(vm, EXCEPTION_USER, "the radio is not
calibrated");

        return NULL;
}

int main(void)
{
        uc_vm_t vm = { 0 };
```

```
        uc_program_t *program;
        uc_source_t *source;
        const char *code =
                "print('doubled: ', needint(21), '\\n');\n"
                "try { needint('x'); } catch (e) { print('caught: ', e, '\
\n'); }\n"
                "try { needint(); } catch (e) { print('caught: ', e, '\
\n'); }\n"
                "try { fail(); } catch (e) { print('caught: ', e, '\\n'); }
\n"
                "print('still running\\n');\n";

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));
        uc_function_register(uc_vm_scope_get(&vm), "needint", needint);
        uc_function_register(uc_vm_scope_get(&vm), "fail", fail);

        source = uc_source_new_buffer("raise.uc", strdup(code),
strlen(code));
        program = uc_compile(&config, source, NULL);
        uc_source_put(source);

        printf("status=%d\n", (int)uc_vm_execute(&vm, program, NULL));

        uc_program_put(program);
        uc_vm_free(&vm);

        return 0;
}
```

```
doubled: 42
caught: expecting a number, got string
caught: expecting a number, got nothing
caught: the radio is not calibrated
still running
status=0
```

Which type to name is a question about how the message will read rather than about machinery: `EXCEPTION_TYPE` reports as "Type error: …", `EXCEPTION_REFERENCE` as "Reference error: …", `EXCEPTION_RUNTIME` as "Runtime error: …", and `EXCEPTION_USER` as the message with no prefix at all — it is what the script's own `raise` uses. `EXCEPTION_EXIT` is not for failures; it is what `exit` raises and it ends the program with a status (chapter 42).

Two properties of the call matter. The declaration carries `format(printf, 3, 0)`, and with a zero in the last position that attribute does not check the arguments against the format, so a mismatched `%s` is not a compile

warning: check the arguments by eye. And the raised value is *not* returned: the VM's own check is on the pending exception, so the value a function returns while one is pending is released rather than pushed:

```
        /* push return value */
        if (!vm->exception.type)
                uc_vm_stack_push(vm, res);
        else
                ucv_put(res);
```

Which is why every raising path above returns `NULL`, and why returning something else after a raise is harmless but pointless. If you call back into script code and *that* raises, the same rule covers you; if you need to know what happened, the pending type is `vm.exception.type`, as chapter 46 covers.

Calling back into the script

Passing a function to a native and calling it is how a native gets to be a control structure. The protocol is the one chapter 42 set out, and `uc_call(nargs)` is `uc_vm_call(vm, false, nargs)` spelled shorter:

```
static uc_value_t *
uc_myagent_apply(uc_vm_t *vm, size_t nargs)
{
        uc_value_t *fn = uc_fn_arg(0);
        uc_value_t *a = uc_fn_arg(1);
        uc_value_t *b = uc_fn_arg(2);
        uc_value_t *res;

        if (fn == NULL || ucv_type(fn) != UC_CLOSURE)
                return NULL;

        uc_vm_stack_push(vm, ucv_get(fn));       /* callee first, then the
arguments */
        uc_vm_stack_push(vm, ucv_get(a));
        uc_vm_stack_push(vm, ucv_get(b));

        if (uc_call(2) != EXCEPTION_NONE)
                return NULL;                          /* let the pending
exception be the answer */

        res = uc_vm_stack_pop(vm);               /* the result the script
function returned */

        return res;
}
```

Push the callee and each argument with a reference of your own, because the call consumes one reference per slot; then the arguments and the callee are gone and the result sits on top. Check the returned exception type rather than assuming the call succeeded — a script function that fails leaves an exception pending, and returning a value then would be a value no one asked for.

There is one trap in this sequence and it is worth knowing before it costs an afternoon. `uc_fn_arg(n)` addresses the stack from its *top* — `_uc_fn_arg` is `uc_vm_stack_peek(vm, nargs - n - 1)` — so the arguments are where they are only as long as the stack is the way the call left it. The first push for an outbound call moves the top and with it everything `uc_fn_arg` answers. Read the arguments you need into locals before you push anything, as above:

```
uc_value_t *fn = uc_fn_arg(0);            /* read first */
uc_value_t *a  = uc_fn_arg(1);

uc_vm_stack_push(vm,
ucv_get(fn));        /* pushing first would move both of these */
uc_vm_stack_push(vm, ucv_get(a));
```

The standard library adds one step to the same sequence. Its callback-calling natives push a call context slot before the callee and pass `mcall` true, so the script function sees a `this` — the context of the code that called the native — rather than none:

```
        uc_vm_stack_push(vm, ucv_get(ctx));             /* what `this` will
be, or NULL */
        uc_vm_stack_push(vm, ucv_get(func));
        /* ... the arguments ... */

        if (uc_vm_call(vm, true, nargs) == EXCEPTION_NONE)
                rv = uc_vm_stack_pop(vm);
```

The helper the library uses for this, `uc_vm_ctx_push()`, is static to `lib.c` and takes the context from the frame two down; the three lines above are all a host needs of it. Whether to pass a context is a design question about the function you are calling: pass one when the value you were handed is a method of something, and leave it out when it is a plain callback.

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

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static uc_value_t *
apply(uc_vm_t *vm, size_t nargs)
{
        uc_value_t *fn = uc_fn_arg(0);
        uc_value_t *res;

        uc_vm_stack_push(vm, ucv_get(fn));
        uc_vm_stack_push(vm, ucv_int64_new(20));
        uc_vm_stack_push(vm, ucv_int64_new(22));

        if (uc_call(2) != EXCEPTION_NONE)
                return NULL;

        res = uc_vm_stack_pop(vm);

        return res;
}

static uc_value_t *
sum(uc_vm_t *vm, size_t nargs)
{
        double total = 0.0;
        size_t i;

        for (i = 0; i < nargs; i++)
                total += ucv_to_double(uc_fn_arg(i));

        return ucv_double_new(total);
}

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_value_t *scope;
        uc_program_t *program;
        uc_source_t *source;
        const char *code =
```

```
                "print('callback: ', apply(function(a, b) { return a + b; },
0, 0), '\\n');\n"
                "print('a native as the callback: ', apply(sum, 3, 4), '\
\n');\n"
                "try { apply('not a function', 0, 0); } catch (e)
{ print('caught: ', e, '\\n'); }\n"
                "print('stack after: ', stackdepth(), '\\n');\n";

        uc_vm_init(&vm, &config);
        scope = uc_vm_scope_get(&vm);
        uc_stdlib_load(scope);

        uc_function_register(scope, "apply", apply);
        uc_function_register(scope, "sum", sum);

        source = uc_source_new_buffer("callback.uc", strdup(code),
strlen(code));
        program = uc_compile(&config, source, NULL);
        uc_source_put(source);

        printf("status=%d\n", (int)uc_vm_execute(&vm, program, NULL));

        uc_program_put(program);
        uc_vm_free(&vm);

        return 0;
}
```

There is a balance to keep here that the VM does not police: every push you make and do not hand to a call has to be popped, or the operand stack grows once per call. `apply` above is balanced — three pushes in, one pop out, the call taking the other three. When the callee raises, its own frame pop unwinds to the frame `uc_vm_call_native` installed, which is why the native's code can simply return and leave the exception pending; the frame pop in `uc_vm_call_native` guards against the case where the managed code it called already reset the frame stack.

Receivers and `this`

A native can be called as a method, and then the object it was called on is the call context. The accessors are in `ucode/lib.h`:

```
uc_value_t *uc_fn_this_res(uc_vm_t
*vm);                    /* the context as a value */
void       *uc_fn_this(uc_vm_t *vm, const char *expectedtype);     /* its
data pointer */
void       *uc_fn_thisval(uc_vm_t *vm, const char *expectedtype);  /* the
resource itself */
```

The first is the raw one and works for any receiver:

```
static uc_value_t *
uc_myagent_name(uc_vm_t *vm, size_t nargs)
{
        uc_value_t *self = _uc_fn_this_res(vm);

        if (self == NULL)
                uc_vm_raise_exception(vm, EXCEPTION_TYPE, "name() must be
called as a method");

        return ucv_get(ucv_property_get(self, ucv_string_new("name")));
}
```

`uc_fn_this` and `uc_fn_thisval` add the resource-type check of chapter 45 and are what a resource's methods use. A plain call has no receiver at all, so the context is not "the global scope" — it is nothing, and a method-shaped native has to be ready to say so:

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

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static uc_value_t *
describe(uc_vm_t *vm, size_t nargs)
{
        uc_value_t *self = _uc_fn_this_res(vm);

        if (self == NULL)
                return ucv_string_new("no receiver");

        {
                uc_stringbuf_t *buf = ucv_stringbuf_new();

                ucv_stringbuf_printf(buf, "a %s holding %zu member(s)",
ucv_typename(self),
                                     ucv_object_length(self));

                return ucv_stringbuf_finish(buf);
        }
}

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_value_t *scope;
        uc_program_t *program;
        uc_source_t *source;
        const char *code =
                "print('as a method: ', device.describe(), '\\n');\n"
                "let f = device.describe;\n"
                "print('through a name: ', f(), '\\n');\n"
                "print('the plain name: ', describe(), '\\n');\n";

        uc_vm_init(&vm, &config);
        scope = uc_vm_scope_get(&vm);
        uc_stdlib_load(scope);

        {
                uc_value_t *o = ucv_object_new(&vm);
```

```
                uc_function_register(o, "describe", describe);
                uc_function_register(scope, "describe", describe);
                ucv_object_add(scope, "device", o);
        }

        source = uc_source_new_buffer("this.uc", strdup(code), strlen(code));
        program = uc_compile(&config, source, NULL);
        uc_source_put(source);

        printf("status=%d\n", (int)uc_vm_execute(&vm, program, NULL));

        uc_program_put(program);
        uc_vm_free(&vm);

        return 0;
}
```

```
as a method: a object holding 1 member(s)
through a name: no receiver
the plain name: no receiver
status=0
```

`o.describe` sees the object, which is why the member count is two: `name` from the script and `describe` from the host. The same function lifted out of the object and called on its own has no receiver. The difference is visible to the native and to nothing else, which is how the standard library's resource methods work without checking a type name for every call.

Reusing what the library already has

`uc_stdlib_function` is a lookup into the loaded standard library, giving the function pointer under a name. Calling it is the stack protocol, which is what makes a wrapper of your own about a built-in a few lines:

```c
#include <stdio.h>

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_cfn_ptr_t length = uc_stdlib_function("length");
        uc_value_t *arr, *res;

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));

        arr = ucv_array_new(&vm);
        ucv_array_push(arr, ucv_int64_new(1));
        ucv_array_push(arr, ucv_int64_new(2));
        ucv_array_push(arr, ucv_int64_new(3));

        uc_vm_stack_push(&vm, ucv_cfunction_new("length", length));
        uc_vm_stack_push(&vm, arr);

        printf("call status=%d\n", (int)uc_vm_call(&vm, false, 1));

        res = uc_vm_stack_pop(&vm);

        printf("length of the array: %lld\n", (long
long)ucv_to_integer(res));

        ucv_put(res);
        uc_vm_free(&vm);

        return 0;
}
```

```
call status=0
length of the array: 3
```

`uc_stdlib_function` answers `NULL` for a name that is not there, which happens when the standard library was not loaded into this VM, so check it before pushing it as a callee. The built-in is looked up by name at the point you ask, so the value you push is the implementation of that moment; if the script replaces the name in the scope, the pointer

you are holding is unaffected — call it through the scope with `ucv_property_get` if following the script's choice is what you want.

What the VM does not check

The native interface is thin on purpose, and the checks that a script gets for free are the native's own responsibility:

- **Argument count.** A call passing three values to a two-argument native gives it three; `uc_fn_arg(2)` answers `NULL` if it asks, and nothing says otherwise.
- **Argument type.** Nothing converts or rejects; `ucv_to_double` on a string is the conversion the script would get, and `ucv_to_double(NULL)` is `0`.
- **Return type.** Whatever you return is what the script receives.
- **Stack balance.** Extra values left on the operand stack stay there for the rest of the frame's life, and the first thing the enclosing code does with the top of the stack will be wrong.
- **Reference counts.** Every rule of chapter 41 applies with nobody watching; a missing `ucv_get` on a value you keep is a released value you later touch, and a missing `ucv_put` is a value the collector cannot see.
- **Unwinding.** Do not jump out of a native past the VM's bookkeeping — no `longjmp`, no C++ exception; set an exception with `uc_vm_raise_exception` and return; that is the supported unwinding path.

Trace output is worth having while a native is being written, because a native's frame appears in it with its name: `uc_vm_trace_set(&vm, 1)` prints the frames and every stack push, which is the quickest way to see that a call left one value too many behind (chapter 42).

Resource types

A resource is a value that owns something on the host's side. The script sees an opaque object it can hold, pass, and call methods on; the host sees a pointer it can rely on being valid for exactly as long as the value is, and a callback the moment it stops being. That is the whole idea, and it is the reason file handles, sockets, netlink sockets, UCI contexts and FFI handles are resources rather than integers in an object member: an integer is forgotten when the script loses it, and a resource is not.

```
uc_resource_type_t *ucv_resource_type_add(uc_vm_t *vm, const char *name,
                                          uc_value_t *proto, void (*free)
(void *));
uc_resource_type_t *ucv_resource_type_lookup(uc_vm_t *vm, const char *name);

uc_value_t *ucv_resource_new(uc_resource_type_t *type, void *data);
uc_value_t *ucv_resource_new_ex(uc_vm_t *vm, uc_resource_type_t *type, void
**data,
                                size_t uvcount, size_t datasize);
uc_value_t *ucv_resource_new_with_proto(uc_vm_t *vm, uc_resource_type_t
*type, void **data,
                                        size_t uvcount, size_t datasize,
uc_value_t *proto);

void *ucv_resource_data(uc_value_t *uv, const char *name);
void **ucv_resource_dataptr(uc_value_t *uv, const char *name);

uc_value_t *ucv_resource_value_get(uc_value_t *uv, size_t idx);
bool ucv_resource_value_set(uc_value_t *uv, size_t idx, uc_value_t *val);

/* ucode/lib.h */
uc_resource_type_t *uc_type_declare(uc_vm_t *vm, const char *name,
                                    const uc_function_list_t *methods, void
(*free)(void *));
void *uc_fn_this(const char *name);              /* plain resources: the
address of the data */
void *uc_fn_thisval(const char *name);           /* both shapes: the data
itself */
```

The two shapes

A resource is either *plain* or *extended*, and the difference is where its data lives.

```
typedef struct {
        uc_value_t header;
        uc_resource_type_t *type;
        void *data;                                     /* the host allocated
this */
} uc_resource_t;

typedef struct {
        uc_value_t header;
        uc_weakref_t ref;
        uc_resource_type_t *type;

        uint32_t reserved:2;
        uint32_t hasproto:1;
        uint32_t persistent:1;
        uint32_t uvcount:8;
        uint32_t datasize:20;

        uint32_t _pad;
} uc_resource_ext_t;
```

The plain form holds a pointer to memory the host allocated, and `ucv_resource_new` makes one. The extended form carries the data *inside the value*: `ucv_resource_new_ex` takes a byte count, rounds it up to eight, and writes the address of that block through its `data` out-parameter, so a host that needs a fixed-shape struct does not allocate it separately. The same allocation also carries the value slots — `uvcount` of them, plus one more when the instance has its own prototype — laid out after the data block, which is why the extended form is the one to use when a resource has to hold script values. `ucv_resource_new_with_proto` is the extended constructor that also asks for the instance prototype, and passing it `NULL` leaves you with a plain `ucv_resource_new_ex` result without the prototype slot.

The two counters are bitfields: fewer than 256 value slots and a data block under 8 MiB, both checked by `assert` rather than by a return value. An extended resource is registered in the VM's value list only when it actually has slots, which is the reason a plain handle does not appear in the count of live containers of chapter 41.

Declaring a type

A type is a name, a prototype, and a release callback:

```
typedef struct {
        const char *name;
        uc_value_t *proto;
        void (*free)(void *);
} uc_resource_type_t;
```

`ucv_resource_type_add` registers one with the VM, and `uc_type_declare` from `ucode/lib.h` builds the prototype for you out of a method table, the same `uc_function_list_t` shape chapter 44 used for functions:

```
static const uc_function_list_t chanmethods[] = {
        { "write", chan_write },
        { "name",  chan_name  }
};

type = uc_type_declare(&vm, "chan", chanmethods, chan_free);
```

Types belong to the VM, so a VM that is freed and initialised again has to declare them again, and a loadable module that adds a type adds it to whichever VM loads it (chapter 48). The name is both the key for `ucv_resource_type_lookup` and the string the data accessors check against, so it is the thing that stops one kind of handle being passed where another is expected.

The same name twice

Registration is keyed on the name and the first one wins. Passing a name that is already registered returns the type that is already there and releases the prototype handed over with it, so the second declaration is dropped rather than merged or honoured:

```
        first = ucv_resource_type_add(&vm, "demo.thing", protofirst,
thing_free);
        again = ucv_resource_type_add(&vm, "demo.thing", protosecond,
thing_free);

        printf("the second registration returned the same type: %d\n", first
== again);
```

What a value made through that name then has is the first prototype's members, including the release function the first call named:

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

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static uc_value_t *
which_first(uc_vm_t *vm, size_t nargs)
{
        (void)vm;
        (void)nargs;

        return ucv_string_new("first");
}

static uc_value_t *
which_second(uc_vm_t *vm, size_t nargs)
{
        (void)vm;
        (void)nargs;

        return ucv_string_new("second");
}

static uc_value_t *
make_thing(uc_vm_t *vm, size_t nargs)
{
        (void)nargs;

        return ucv_resource_new(ucv_resource_type_lookup(vm, "demo.thing"),
                                calloc(1, sizeof(int)));
}

static void
thing_free(void *data)
{
        free(data);
}

static const char script[] =
        "let o = demo.make();\n"
        "print(\"the method that ran: \", o.which(), \"\\n\");\n"
        "print(\"a member only the second prototype had: \", exists(o,
```

```
\"addedlate\"), \"\\n\");\n";

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_resource_type_t *first, *again;
        uc_value_t *protofirst, *protosecond, *scope;

        setvbuf(stdout, NULL, _IOLBF, 0);

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));

        protofirst = ucv_object_new(&vm);
        ucv_object_add(protofirst, "which", ucv_cfunction_new("which",
which_first));

        protosecond = ucv_object_new(&vm);
        ucv_object_add(protosecond, "which", ucv_cfunction_new("which",
which_second));
        ucv_object_add(protosecond, "addedlate", ucv_int64_new(9));

        first = ucv_resource_type_add(&vm, "demo.thing", protofirst,
thing_free);
        again = ucv_resource_type_add(&vm, "demo.thing", protosecond,
thing_free);

        printf("the second registration returned the same type: %d\n", first
== again);

        /* make one through the registered name and see which methods it has
*/
        {
                uc_source_t *source;
                uc_program_t *program;
                uc_value_t *rv = NULL;

                scope = ucv_object_new(&vm);
                uc_function_register(scope, "make", make_thing);
                ucv_object_add(uc_vm_scope_get(&vm), "demo",
ucv_get(scope));

                source = uc_source_new_buffer("t.uc", strndup(script,
strlen(script)), strlen(script));
                program = uc_compile(&config, source, NULL);
                uc_source_put(source);
```

```
                if (program != NULL) {
                        printf("status=%d\n", (int)uc_vm_execute(&vm,
program, &rv));
                        ucv_put(rv);
                        uc_program_put(program);
                }
                else {
                        printf("the script did not compile\n");
                }

                ucv_put(scope);
        }

        uc_vm_free(&vm);

        return 0;
}
```

```
the second registration returned the same type: 1
the method that ran: first
a member only the second prototype had: false
status=0
```

Two things follow for the cases where a name does get registered twice. A module reloaded into a VM that still has its first copy is not refreshed by its own entry point: the type keeps the members it was first given, and only a VM with no trace of the name installs the new shape (chapter 48). And a program that takes the return value of a second registration and installs members on it is adding them to the first type, which is legal and worth knowing — `ucv_resource_type_add` returning the existing type is the only way a caller learns the name was taken.

The prototype is an ordinary object, reachable afterwards as `type->proto` if you want to add something to it from C, and from the script side as `proto(handle)`.

Reading the data back

Call	Plain resource	Extended resource
`ucv_resource_data(uv, "chan")`	the `data` pointer	the inline block
`ucv_resource_dataptr(uv, "chan")`	the address of the `data` slot	`NULL`
`uc_fn_this("chan")`	the address of the `data` slot	`NULL`
`uc_fn_thisval("chan")`	the `data` pointer	the inline block
`ucv_resource_data(uv, "other")`	`NULL`	`NULL`

Both data accessors answer `NULL` when the value is not a resource of the named type, which is the type check, and `ucv_resource_dataptr` answers `NULL` for the extended shape because there is no slot there to point at — so `uc_fn_this` belongs to plain resources and `uc_fn_thisval` to both, despite the names suggesting a choice. Both return `void *`, so a method casts:

```
static uc_value_t *
chan_write(uc_vm_t *vm, size_t nargs)
{
        chan_t **slot = (chan_t **)uc_fn_this("chan");
        chan_t *ch = slot ? *slot : NULL;
        ..
}
```

The address-of-data form exists because the host may replace the pointer inside the slot. That is how the file handles of chapter 25 work: `close` puts a fresh `FILE *` into the slot, or `NULL` in it, and every method that reads the slot through the address sees the change without the value having to be replaced.

A plain resource, end to end

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

typedef struct {
        FILE *fp;
        const char *name;
} chan_t;

static uc_value_t *
chan_write(uc_vm_t *vm, size_t nargs)
{
        chan_t **slot = (chan_t **)uc_fn_this("chan");
        chan_t *ch = slot ? *slot : NULL;
        uc_value_t *text = uc_fn_arg(0);

        if (ch == NULL || ch->fp == NULL || text == NULL)
                return ucv_int64_new(-1);

        fputs(ucv_to_string(vm, text), ch->fp);

        return ucv_int64_new(ftell(ch->fp));
}

static uc_value_t *
chan_name(uc_vm_t *vm, size_t nargs)
{
        chan_t **slot = (chan_t **)uc_fn_this("chan");

        return ucv_string_new((*slot) ? (*slot)->name : "(closed)");
}

static const uc_function_list_t chanmethods[] = {
        { "write", chan_write },
        { "name",  chan_name  }
};

static void
chan_free(void *data)
{
        chan_t *ch = data;
```

```
        printf("[released %s]\n", ch->name);

        if (ch->fp != NULL)
                fclose(ch->fp);

        free(ch);
}

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_resource_type_t *type;
        uc_value_t *handle;
        chan_t *ch;
        uc_program_t *program;
        const char *code =
                "print('type: ', type(ch), '\\n');\n"
                "print('name: ', ch.name(), '\\n');\n"
                "print('bytes: ', ch.write('first line\\n'), '\\n');\n"
                "print('same handle: ', ch == ch, '\\n');\n"
                "print('a method is not an own key: ', exists(ch, 'write'),
'\\n');\n";

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));

        type = uc_type_declare(&vm, "chan", chanmethods, chan_free);
        printf("type declared: %s, lookup %s\n", type->name,
               ucv_resource_type_lookup(&vm, "chan") == type ? "agrees" :
"differs");

        ch = calloc(1, sizeof(*ch));
        ch->fp = fopen("/tmp/ucode-ch45-chan", "w");
        ch->name = "log0";

        handle = ucv_resource_new(type, ch);

        printf("rendered as: %s\n", strncmp(ucv_to_string(&vm, handle),
"<chan ", 6) == 0 ?
               "'<chan ' then the pointer" : "something else");
        printf("asked for the wrong type: %s\n",
               ucv_resource_data(handle, "socket") == NULL ? "NULL" : "a
pointer");
        printf("asked for the right type: %s\n",
               ucv_resource_data(handle, "chan") == ch ? "the host's
pointer" : "another");
```

```
        printf("the slot it is kept in: %s\n",
               *(chan_t **)ucv_resource_dataptr(handle, "chan") == ch ?
"where the pointer lives" : "another");

        ucv_object_add(uc_vm_scope_get(&vm), "ch", handle);

        {
                uc_source_t *src = uc_source_new_buffer("chan.uc",
strdup(code), strlen(code));

                program = uc_compile(&config, src, NULL);
                uc_source_put(src);

                if (program == NULL)
                        return 1;

                printf("status=%d\n", (int)uc_vm_execute(&vm, program,
NULL));
                uc_program_put(program);
        }

        printf("the handle goes away with its name\n");
        ucv_object_delete(uc_vm_scope_get(&vm), "ch");

        uc_vm_free(&vm);

        return 0;
}
```

```
type declared: chan, lookup agrees
rendered as: '<chan ' then the pointer
asked for the wrong type: NULL
asked for the right type: the host's pointer
the slot it is kept in: where the pointer lives
type: resource
name: log0
bytes: 11
same handle: true
a method is not an own key: false
status=0
the handle goes away with its name
[released log0]
```

Read the last line and the one before it together: releasing the last reference to the value is what calls `chan_free`, and it happens while the object that held it is being modified, not at some later sweep. A resource is released the

moment its reference count reaches zero, which is what makes a file handle written by a script not outlive the name that held it, and it is also why a host that wants a handle to survive must keep a reference of its own — the registry of chapter 42 being the place made for keeping one.

The extended shape

The extended form puts the data and the value slots inside the value. This one keeps its counter in the inline block, exposes a slot to the script through a native, and adds a method through its own instance prototype:

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

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

typedef struct {
        char label[8];
        size_t count;
} stat_t;

/* a slot reader, since a script has no accessor for slots of its own */
static uc_value_t *
statvalue(uc_vm_t *vm, size_t nargs)
{
        uc_value_t *res = uc_fn_arg(0);
        uc_value_t *idx = uc_fn_arg(1);

        if (res == NULL || idx == NULL)
                return NULL;

        return ucv_get(ucv_resource_value_get(res,
(size_t)ucv_to_integer(idx)));
}

static uc_value_t *
stat_note(uc_vm_t *vm, size_t nargs)
{
        stat_t *st = (stat_t *)uc_fn_thisval("stat");

        if (st == NULL)
                return NULL;

        return ucv_int64_new((int64_t)(++st->count));
}

static uc_value_t *
stat_label(uc_vm_t *vm, size_t nargs)
{
        stat_t *st = (stat_t *)uc_fn_thisval("stat");

        return ucv_string_new(st ? st->label : "(gone)");
}
```

```
/* reachable only through the instance prototype */
static uc_value_t *
stat_shout(uc_vm_t *vm, size_t nargs)
{
        (void) vm; (void) nargs;

        return ucv_string_new("COUNTED");
}

static const uc_function_list_t statmethods[] = {
        { "note",  stat_note  },
        { "label", stat_label }
};

static void
stat_free(void *data)
{
        stat_t *st = data;

        printf("[released %s after %zu notes]\n", st->label, st->count);
}

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_resource_type_t *type;
        uc_value_t *handle, *instanceproto;
        stat_t *data = NULL;
        uc_program_t *program;
        const char *code =
                "print('label: ', s.label(), '\\n');\n"
                "print('notes: ', s.note(), s.note(), s.note(), '\\n');\n"
                "print('slot 0: ', statvalue(s, 0), '\\n');\n"
                "print('instance prototype: ', s.shout(), '\\n');\n";

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));
        type = uc_type_declare(&vm, "stat", statmethods, stat_free);

        handle = ucv_resource_new_with_proto(&vm, type, (void **)&data, 1,
sizeof(stat_t),
                                             ucv_object_new(&vm));
        strcpy(data->label, "rx");

        /* one value slot, set from the host side */
        ucv_resource_value_set(handle, 0, ucv_string_new("a slot"));
```

```
        instanceproto = ucv_resource_proto_get(handle);
        ucv_object_add(instanceproto, "shout", ucv_cfunction_new("shout",
stat_shout));
        printf("instance prototype: %s\n", ucv_typename(instanceproto));

        ucv_object_add(uc_vm_scope_get(&vm), "s", handle);
        uc_function_register(uc_vm_scope_get(&vm), "statvalue", statvalue);

        {
                uc_source_t *src = uc_source_new_buffer("stat.uc",
strdup(code), strlen(code));

                program = uc_compile(&config, src, NULL);
                uc_source_put(src);

                if (program == NULL)
                        return 1;

                printf("status=%d\n", (int)uc_vm_execute(&vm, program,
NULL));
                uc_program_put(program);
        }

        printf("the slot from C: %s\n", ucv_to_string(&vm,
ucv_resource_value_get(handle, 0)));
        printf("the name goes, the handle stays\n");
        ucv_object_delete(uc_vm_scope_get(&vm), "s");
        printf("and now it goes\n");
        ucv_put(handle);

        uc_vm_free(&vm);

        return 0;
}
```

```
instance prototype: object
label: rx
notes: 123
slot 0: a slot
instance prototype: COUNTED
status=0
the slot from C: a slot
the name goes, the handle stays
[released rx after 3 notes]
and now it goes
```

`uc_fn_thisval` is what a method of an extended resource uses, since the data is the inline block and there is no slot to take the address of. The instance prototype is an ordinary object the value carries in addition to the type's, and the two chain: `note` and `label` come from the type and `shout` from the instance. Only the constructors that are given a prototype set the flag that makes the slot exist, so `ucv_resource_proto_set` on a value created by `ucv_resource_new_ex` answers false rather than adding the slot. Releasing the name left the handle alive because the host still held a reference, and the release callback ran when the host's own `ucv_put` finished the count.

Value slots

Slots are ordinary `uc_value_t *` slots inside the value, numbered from zero, with the instance prototype in a slot of its own at `UCV_RESOURCE_PROTO_IDX`. They are roots: the collector marks them, so a value reachable only from a slot is not collected. `ucv_resource_value_set` consumes the value and releases whatever was in the slot, the same rule as every other container mutator (chapter 41).

There is no script-visible accessor for them. `s[0]` does not reach slot zero; it is a member lookup on the value, which finds nothing. The standard library exposes its own state through methods written in C for the purpose, which is the shape of `statvalue` above.

Because the value accessors neither check the type of the value they were handed nor check the shape, a `ucv_resource_value_set` on something that is not an extended resource reads a count from unrelated memory and indexes by it. The discipline is on the host: reach for a slot only from a value you obtained as a receiver or checked with `ucv_resource_data`.

Slots earn their keep when the resource has to hold on to script values. A watch object that runs a callback is the smallest case, and it needs no data block at all:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static uc_resource_type_t *watchtype;

static uc_value_t *
watch_new(uc_vm_t *vm, size_t nargs)
{
        (void) nargs;

        /* no data block, one value slot */
        return ucv_resource_new_ex(vm, watchtype, NULL, 1, 0);
}

static uc_value_t *
watch_when(uc_vm_t *vm, size_t nargs)
{
        uc_value_t *self = _uc_fn_this_res(vm);
        uc_value_t *cb = uc_fn_arg(0);

        return ucv_boolean_new(ucv_resource_value_set(self, 0, ucv_get(cb)));
}

static uc_value_t *
watch_fire(uc_vm_t *vm, size_t nargs)
{
        uc_value_t *self = _uc_fn_this_res(vm);
        uc_value_t *cb = ucv_resource_value_get(self, 0);
        uc_value_t *arg = uc_fn_arg(0);

        if (cb == NULL)
                return ucv_int64_new(-1);

        /* the argument is read into a local before anything is pushed:
uc_fn_arg
           addresses from the top of the stack, so a push moves it (chapter
44) */
        uc_vm_stack_push(vm, ucv_get(cb));
        uc_vm_stack_push(vm, ucv_get(arg));
```

```
        if (uc_call(1) != EXCEPTION_NONE)
                return NULL;

        return uc_vm_stack_pop(vm);
}

static const uc_function_list_t watchmethods[] = {
        { "when", watch_when },
        { "fire", watch_fire }
};

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_program_t *program;
        const char *code =
                "let w = watch();\n"
                "print('armed: ', w.when(function(v) { return 'seen: ' +
v; }), '\\n');\n"
                "print('fire: ', w.fire('boot'), '\\n');\n"
                "print('fire again: ', w.fire(42), '\\n');\n"
                "print('no slots of its own: ', keys(w), '\\n');\n";

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));

        watchtype = uc_type_declare(&vm, "watch", watchmethods, NULL);
        uc_function_register(uc_vm_scope_get(&vm), "watch", watch_new);

        {
                uc_source_t *src = uc_source_new_buffer("watch.uc",
strdup(code), strlen(code));

                program = uc_compile(&config, src, NULL);
                uc_source_put(src);

                if (program == NULL)
                        return 1;

                printf("status=%d\n", (int)uc_vm_execute(&vm, program,
NULL));
                uc_program_put(program);
        }

        uc_vm_free(&vm);
```

```
        return 0;
}
```

```
armed: true
fire: seen: boot
fire again: seen: 42
no slots of its own:
status=0
```

The callback is a value the script owns and the resource keeps alive, and the resource is the only path to it, so the pair lives and dies together. The receiver is `_uc_fn_this_res(vm)` in both methods, since a method's first argument is its first *argument* rather than the object it was called on — getting that wrong hands a closure to `ucv_resource_value_set`, which is the case the accessors do not check.

The release callback

`type->free` is called with the data — the host's pointer for a plain resource, the inline block for an extended one — after the slots have been released, and it is the only teardown hook there is. There is no `__gc` metamethod and no finaliser a script can install (chapter 18), and `__delete__` is something else entirely: the key-deletion hook of chapter 12. So whatever a resource must not leak goes in that callback, and nothing else does.

For a plain resource the callback is where the host frees what it allocated, since the value never owned the allocation's memory, only the pointer. For an extended resource the block is inside the value and goes with it, so the callback is for what lives *through* the block: descriptors, `FILE *`, a library handle.

`persistent` is a bit of the extended header, set through `ucv_resource_persist_*` and consulted when the VM is torn down: a persistent resource outlives the VM it belongs to rather than being released with it, which is how a loadable module keeps a handle alive across the VMs its host cycles through. Chapter 48 has the details, and nothing a script can see is affected by the bit.

What a script can see

- `type(handle)` is `"resource"`; the type's *name* is not visible to the script, so a message that should name the kind has to come from a method.
- `==` compares handles by identity: two names for one handle are equal, and two handles for one host object are not, which is a reason to hand out one handle per object and let the script copy the name.
- Rendering is `<chan 0x55d9…>` — the type name and the address of the value, or `<chan (nil)>` when the value carries no data. `__tostring` is not consulted for a resource, nor anywhere else (chapter 9), so a handle that should print usefully gets a method that says what it is.

- `exists(handle, "write")` is false and `keys(handle)` is empty for a resource whose methods all come from the type's prototype, because those two look at own keys. Method calls resolve through the prototype chain either way.
- Assigning a member is permitted and is per-value: `handle.name = "x"` adds an own key that shadows a prototype member for that one handle, which is occasionally the quickest way to attach a label.

Resources are the right tool when the thing behind the value has a lifetime the script should not be able to get wrong. For configuration and results, an object is cheaper, printable, and serialisable; for a descriptor or a handle, an object is a leak with a name on it.

Exceptions and signals in an embedder

A host needs three things from the failure machinery: to be told that a run failed and in what way, to be able to find out what went wrong, and to be able to stop a run that has gone wrong. Those are three separate mechanisms in ucode, and mixing them up is the usual source of confusion, so this chapter keeps them apart: the *status* a run comes back with, the *exception record* the VM holds, and the *break request*, plus the signal plumbing a program uses to react to the system around it.

```
uc_vm_status_t uc_vm_execute(uc_vm_t *vm, uc_program_t *program, uc_value_t
**retval);
uc_vm_status_t uc_vm_resume(uc_vm_t *vm);

uc_exception_type_t uc_vm_call(uc_vm_t *vm, bool mcall, size_t nargs);
uc_value_t *uc_vm_exception_object(uc_vm_t *vm);
uc_exception_handler_t *uc_vm_exception_handler_get(uc_vm_t *vm);
void uc_vm_exception_handler_set(uc_vm_t *vm, uc_exception_handler_t
*handler);
void uc_vm_raise_exception(uc_vm_t *vm, uc_exception_type_t type, const char
*fmt, ...);

void uc_vm_break_request(uc_vm_t *vm);
bool uc_vm_break_requested(uc_vm_t *vm);
int  uc_vm_break_notifyfd(uc_vm_t *vm);

uc_exception_type_t uc_vm_signal_dispatch(uc_vm_t *vm);
void uc_vm_signal_raise(uc_vm_t *vm, int signo);
int  uc_vm_signal_notifyfd(uc_vm_t *vm);
void uc_vm_signal_handlers_ensure(uc_vm_t *vm);

extern const char *exception_type_strings[];
extern const char *uc_system_signal_names[];
```

The status a run comes back with

`uc_vm_execute` hands back one of five codes, and it also writes the run's value through `retval` — but not in every case, which is the half of the contract that is usually missed.

Status	Value in	What `*retval` receives
`STATUS_OK`	the run finished	the value the program returned
`STATUS_EXIT`	`exit` was reached	the exit code, as an integer
`STATUS_BREAK`	a break request stopped it	nothing at all: the null value
`ERROR_COMPILE`	a syntax error is pending	nothing
`ERROR_RUNTIME`	any other exception is pending	nothing

The exception type decides the status rather than the other way round: `EXCEPTION_NONE` is `STATUS_OK`, `EXCEPTION_EXIT` is `STATUS_EXIT`, `EXCEPTION_SYNTAX` is `ERROR_COMPILE`, and every other type is `ERROR_RUNTIME`. The two error statuses therefore differ only in which kind of failure was recorded.

A program that ends without returning anything leaves the null value, so `STATUS_OK` with a null in `*retval` covers both "returned null" and "fell off the end"; the two are not distinguishable, which is worth knowing before a host builds a protocol on the return value (chapter 40's published-by-assignment pattern sidesteps it).

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

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static uc_value_t *
boom(uc_vm_t *vm, size_t nargs)
{
        (void) nargs;

        uc_vm_raise_exception(vm, EXCEPTION_USER, "the radio is not
calibrated");

        return NULL;
}

static void
describe(uc_vm_t *vm, const char *label, uc_vm_status_t status, uc_value_t
*rv)
{
        printf("%-14s status=%d return=%s", label, (int)status,
               rv ? ucv_to_string(vm, rv) : "the null value");

        /* the getter has to be asked only when something is pending: with
nothing
           raised it builds its object from a null message and faults */
        if (vm->exception.type != EXCEPTION_NONE) {
                uc_value_t *exo = uc_vm_exception_object(vm);

                printf(", %zu keys: type=%s message=%s stacktrace=%s",
                       ucv_object_length(exo),
                       ucv_to_string(vm, ucv_object_get(exo, "type", NULL)),
                       ucv_to_string(vm, ucv_object_get(exo, "message",
NULL)),
                       ucv_typename(ucv_object_get(exo, "stacktrace",
NULL)));

                ucv_put(exo);
        }

        printf("\n");
```

```
        ucv_put(rv);
}

static void
run(const char *label, const char *code, bool withnative)
{
        uc_vm_t vm = { 0 };
        uc_program_t *program;
        uc_source_t *src;
        uc_value_t *rv = NULL;
        uc_vm_status_t status;

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));

        if (withnative)
                uc_function_register(uc_vm_scope_get(&vm), "boom", boom);

        src = uc_source_new_buffer("probe.uc", strdup(code), strlen(code));
        program = uc_compile(&config, src, NULL);
        uc_source_put(src);

        if (program == NULL) {
                printf("%s: compile failed\n", label);
                uc_vm_free(&vm);

                return;
        }

        status = uc_vm_execute(&vm, program, &rv);
        describe(&vm, label, status, rv);

        uc_program_put(program);
        uc_vm_free(&vm);
}

int main(void)
{
        setvbuf(stdout, NULL, _IOLBF, 0);

        run("type error", "let x = 1;\nx();\n", false);
        run("undeclared", "return missing;\n", false);
        run("user raise", "boom();\n", true);
        run("explicit exit", "let a = 1;\nexit(3);\n", false);
        run("clean run", "return 7;\n", false);
```

```
        return 0;
}
```

```
Type error: left-hand side is not a function
In probe.uc, line 2, byte 3:

 `x();`
    ^-- Near here


type error     status=4 return=the null value, 3 keys: type=Type error
message=left-hand side is not a function stacktrace=array
undeclared     status=0 return=the null value
the radio is not calibrated
In probe.uc, line 1, byte 6:

 `boom();`
       ^-- Near here


user raise     status=4 return=the null value, 3 keys: type=Error message=the
radio is not calibrated stacktrace=array
explicit exit  status=1 return=3, 3 keys: type=Exit message=Terminated
stacktrace=array
clean run      status=0 return=7
```

Note that reading the record is worth doing before reading `*retval`, and that a run whose program *reads* an undeclared name does not fail at all: `return missing;` is `STATUS_OK` with the null value, because reading a global that is not there is defined to give null. A host that wants such a name to be an error compiles with `strict_declarations` (chapter 43), which turns the assignment case into a run-time `Reference error`.

The exception record

The record is three fields of the VM, set when something is raised and left in place after the run returns:

```
struct {
        uc_exception_type_t type;
        const char *message;
        uc_value_t *stacktrace;
} exception;
```

`type` is the enumeration, `message` is a plain C string owned by the VM, and `stacktrace` is a script value — an array — that describes the frames. `exception_type_strings[type]` gives the same wording the reports use, and `uc_system_signal_names` is the matching table for signal numbers.

There is no function that clears this record: `uc_vm_clear_exception()` exists but is static inside `vm.c`, and it is what both entry points, `uc_vm_execute()` and `uc_vm_call()`, run on entry (see *What a run leaves behind*), so a host that wants the record of a failed run reads it before the next run starts. A host that wants to mark a read record as consumed without running anything writes the type field itself:

```
vm.exception.type = EXCEPTION_NONE;
```

The assignment does not release the `message` or the `stacktrace`; the next entry point's clear does, as does the next raise or `uc_vm_free`.

The object for a script

`uc_vm_exception_object` assembles the same three fields into an object with the keys `type`, `message` and `stacktrace`, which is the shape the script's own `catch` hands over, so a host can hand a failure to a script function without formatting anything:

```
uc_value_t *reporter = uc_vm_invoke(&vm, "onerror", 1,
uc_vm_exception_object(&vm));
```

The object carries a prototype that supplies a `tostring`, so printing it gives the same multi-line report the default handler prints. That prototype is created on first use and then kept in the VM's registry under the name `vm.exception.proto`, which is one of the few places the registry's contents are visible from the outside (chapter 42).

There are two things to know about it. It returns a value the caller owns, so release it. It also cannot tell you whether anything is wrong: when `type` is `EXCEPTION_NONE`, it still builds the object, feeds the absent message to a string constructor, and faults in `strlen`. Ask the record first:

```
if (vm.exception.type != EXCEPTION_NONE) {
        uc_value_t *exo = uc_vm_exception_object(&vm);
        /* ... use it ... */
        ucv_put(exo);
}
```

The exception handler

The handler is what prints the report. It is a function of the shape `void (*)(uc_vm_t *vm, uc_exception_t *exc)` stored in the VM, and a VM comes with one already installed — `uc_vm_output_exception`, the function that writes the `Type error: …` block with the quoted source line to standard error. A host replaces it, keeps the one it displaced, and puts it back:

```
static uc_exception_handler_t *previous;

previous = uc_vm_exception_handler_get(&vm);
uc_vm_exception_handler_set(&vm, myhandler);
```

The VM calls it at the point a run reports a failure, which is inside `uc_vm_execute`, `uc_vm_resume` and `uc_vm_invoke` on the error path — not inside `uc_vm_call`, so a host driving the stack protocol of chapter 42 reports failures itself — and never for `STATUS_OK`, `STATUS_EXIT` or `STATUS_BREAK`. The handler receives the record by pointer, so `exc->message` is the VM's string: use it before returning, do not keep it.

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

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };
static uc_exception_handler_t *saved;

static void
quiet(uc_vm_t *vm, uc_exception_t *exc)
{
        (void) vm;

        printf("  [reported: type=%d message=%s]\n", (int)exc->type,
               exc->message ? exc->message : "(none)");
}

static uc_program_t *
compile(uc_vm_t *vm, const char *code)
{
        uc_source_t *src = uc_source_new_buffer("host.uc", strdup(code),
strlen(code));
        uc_program_t *program = uc_compile(&config, src, NULL);

        uc_source_put(src);

        return program;
}

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_program_t *healthy, *failing, *again;
        uc_value_t *rv = NULL;
        uc_vm_status_t st;

        setvbuf(stdout, NULL, _IOLBF, 0);

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));

        saved = uc_vm_exception_handler_get(&vm);
        uc_vm_exception_handler_set(&vm, quiet);
        printf("the default handler is kept aside: %s\n", saved ? "yes" :
```

```
"no");

        healthy = compile(&vm, "let a = 1;\nlet b = a + 1;\nreturn b;\n");
        failing = compile(&vm, "let x = 1;\nx();\n");
        again = compile(&vm, "let a = 2;\nlet b = a + 1;\nreturn b;\n");

        printf("a healthy program\n");
        st = uc_vm_execute(&vm, healthy, &rv);
        printf("  status=%d value=%s\n", (int)st, rv ? ucv_to_string(&vm,
rv) : "null");
        ucv_put(rv); rv = NULL;

        printf("a failing one\n");
        st = uc_vm_execute(&vm, failing, &rv);
        printf("  status=%d residue type=%d message=%s\n", (int)st,
               (int)vm.exception.type, vm.exception.message ?
vm.exception.message : "-");
        ucv_put(rv); rv = NULL;

        printf("then a healthy one again\n");
        st = uc_vm_execute(&vm, again, &rv);
        printf("  status=%d value=%s residue=%d\n", (int)st,
               rv ? ucv_to_string(&vm, rv) : "null", (int)vm.exception.type);
        ucv_put(rv); rv = NULL;

        printf("clearing the record only\n");
        vm.exception.type = EXCEPTION_NONE;
        st = uc_vm_execute(&vm, again, &rv);
        printf("  status=%d value=%s\n", (int)st, rv ? ucv_to_string(&vm,
rv) : "null");
        ucv_put(rv); rv = NULL;

        printf("and the default handler still reports properly\n");
        uc_vm_exception_handler_set(&vm, saved);
        st = uc_vm_execute(&vm, failing, &rv);
        printf("  status=%d\n", (int)st);
        ucv_put(rv);

        uc_program_put(healthy);
        uc_program_put(failing);
        uc_program_put(again);
        uc_vm_free(&vm);

        return 0;
}
```

```
the default handler is kept aside: yes
a healthy program
  status=0 value=2
a failing one
  [reported: type=3 message=left-hand side is not a function]
  status=4 residue type=3 message=left-hand side is not a function
then a healthy one again
  status=0 value=3 residue=0
clearing the record only
  status=0 value=3
and the default handler still reports properly
Type error: left-hand side is not a function
In host.uc, line 2, byte 3:

 `x();`
    ^-- Near here


  status=4
```

A handler that suppresses the report does not suppress the status: the run still comes back `ERROR_RUNTIME`, which is what makes the pair `quiet` + status a way to collect failures without printing them. And the middle of that transcript is the reason this section and the next belong together.

What a run leaves behind

A run resets its own working state as it returns, with one exception, and it does not touch the record:

After a run with	Operand stack	Call frames	Exception record	Break flag
`STATUS_OK`	empty	empty	clear	unchanged
`STATUS_EXIT`	empty	empty	`EXCEPTION_EXIT`, message and trace retained	unchanged
`ERROR_RUNTIME`	empty	empty	retained: type, message and trace	unchanged
`STATUS_BREAK`	**left as the break found it**	**one frame retained**	untouched	cleared by the check

That table has two operational consequences, both of which the examples above demonstrate.

The record is retained for a host to inspect, but it is not *sticky* across runs: both entry points into the interpreter, `uc_vm_execute` and `uc_vm_call`, clear it on entry. A run that fails therefore leaves its record in place for the host to read, and the next program that runs on that VM starts clean. The handler example above shows the pair: the failing run comes back `status=4` with the record set, and the healthy run that follows — with nothing done to the VM in between — comes back `status=0` with the record clear. A host that wants the record reads it before the next entry point clears it; no separate call is needed to keep the VM reusable.

The break residue is subtler and worth a look, because nothing reports it. A run stopped by a break leaves the operand stack and the frame it was working in, on the theory that a resume will continue that run. A resume does finish the run, and the finished run's value is left on the operand stack — `uc_vm_resume` answers with a status only. If nobody pops it, the next program's returned value is that leftover:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <poll.h>
#include <unistd.h>

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static uc_value_t *
stopnow(uc_vm_t *vm, size_t nargs)
{
        (void) nargs;

        uc_vm_break_request(vm);

        return NULL;
}

static uc_program_t *
compile(uc_vm_t *vm, const char *code)
{
        uc_source_t *src = uc_source_new_buffer("resid.uc", strdup(code),
strlen(code));
        uc_program_t *program = uc_compile(&config, src, NULL);

        uc_source_put(src);

        return program;
}

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_program_t *stopping, *healthy;
        uc_value_t *rv = NULL;
        struct pollfd pfd;
        uc_vm_status_t st;

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));
        uc_function_register(uc_vm_scope_get(&vm), "stopnow", stopnow);

        healthy = compile(&vm, "let a = 1;\nlet b = a + 1;\nreturn b;\n");
```

```
        stopping = compile(&vm, "let a = 1;\nstopnow();\nlet b = a + 1;
\nreturn b;\n");

        printf("a run broken in the middle\n");
        st = uc_vm_execute(&vm, stopping, &rv);
        printf("  status=%d value=%s stack depth=%zu frames=%zu\n", (int)st,
               rv ? ucv_to_string(&vm, rv) : "null", vm.stack.count,
vm.callframes.count);
        ucv_put(rv); rv = NULL;

        printf("resuming it to the end\n");
        st = uc_vm_resume(&vm);
        printf("  status=%d depth now=%zu\n", (int)st, vm.stack.count);

        printf("what a following program returns\n");
        st = uc_vm_execute(&vm, healthy, &rv);
        printf("  status=%d value=%s depth now=%zu\n", (int)st,
               rv ? ucv_to_string(&vm, rv) : "null", vm.stack.count);
        ucv_put(rv); rv = NULL;

        printf("draining the notification descriptor\n");
        pfd.fd = uc_vm_break_notifyfd(&vm);
        pfd.events = POLLIN;
        pfd.revents = 0;
        printf("  before: %d readable\n", poll(&pfd, 1, 0) == 1 &&
(pfd.revents & POLLIN) != 0);
        {
                char c;

                while (read(pfd.fd, &c, 1) == 1) {}
        }
        pfd.revents = 0;
        printf("  after: %d readable\n", poll(&pfd, 1, 0) == 1 &&
(pfd.revents & POLLIN) != 0);

        ucv_put(rv);
        uc_program_put(stopping);
        uc_program_put(healthy);
        uc_vm_free(&vm);

        return 0;
}
```

```
a run broken in the middle
  status=2 value=null stack depth=3 frames=1
resuming it to the end
  status=0 depth now=1
what a following program returns
  status=0 value=1 depth now=0
draining the notification descriptor
  before: 1 readable
  after: 0 readable
```

The healthy program returns `1`, which is a leftover rather than its own answer of `2`. A host that breaks runs therefore has to finish the job by hand: after a resume, pop the value the run ended with, and after a break that will not be resumed, drain the stack and the frames back to empty. `vm.stack.count` is the depth, and popping to zero is the whole of it:

```
while (vm.stack.count > 0)
        ucv_put(uc_vm_stack_pop(&vm));
```

Interrupting a run

A VM carries a request flag and a pipe whose readable end a host can watch:

```
void uc_vm_break_request(uc_vm_t *vm);
bool uc_vm_break_requested(uc_vm_t *vm);
int  uc_vm_break_notifyfd(uc_vm_t *vm);
```

The flag is tested in the instruction loop, after the signal check described below, so the granularity is one instruction and a run that is inside a long native is not stopped until that native returns. The test clears the flag as it trips, so a request is spent by the first run that reaches the check — including a request made while nothing at all was running, which stops the next program at its first check. A requested run answers `STATUS_BREAK` and leaves `*retval` set to `NULL`.

The pipe is created by `uc_vm_init` and closed by `uc_vm_free`, so `uc_vm_break_notifyfd` is usable at any point, and `uc_vm_break_request` writes a byte into it as well as setting the flag. That byte is what wakes a blocked event loop, and the VM never reads it back, so a host that watches the descriptor drains it itself.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static size_t ticks = 0;
static bool armed = true;

/* a native the loop calls, which asks the VM to stop after a few turns */
static uc_value_t *
tick(uc_vm_t *vm, size_t nargs)
{
        (void) nargs;

        ticks++;

        if (armed && ticks == 3) {
                armed = false;

                uc_vm_break_request(vm);
        }

        return ucv_int64_new((int64_t)ticks);
}

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_program_t *program;
        uc_source_t *src;
        uc_value_t *rv = NULL;
        uc_vm_status_t st;
        const char *code =
                "let n = 0;\n"
                "while (n < 6) {\n"
                "    n = tick();\n"
                "}\n"
                "return n;\n";

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));
        uc_function_register(uc_vm_scope_get(&vm), "tick", tick);
```

```
        printf("break requested before the run: %d\n",
(int)uc_vm_break_requested(&vm));
        printf("the descriptor an event loop watches: %s\n",
               uc_vm_break_notifyfd(&vm) >= 0 ? "a readable fd" : "not
available");

        src = uc_source_new_buffer("break.uc", strdup(code), strlen(code));
        program = uc_compile(&config, src, NULL);
        uc_source_put(src);

        st = uc_vm_execute(&vm, program, &rv);
        printf("run status=%d return=%s requested now=%d\n", (int)st,
               rv ? ucv_to_string(&vm, rv) : "null",
(int)uc_vm_break_requested(&vm));
        ucv_put(rv); rv = NULL;

        /* a request that has been served needs no value pushed: the loop
carries on
           from the instruction it stopped in front of */
        st = uc_vm_resume(&vm);
        printf("resume status=%d top of stack=%s\n", (int)st,
               vm.stack.count ? ucv_to_string(&vm, uc_vm_stack_peek(&vm,
0)) : "empty");

        /* the counter belongs to the run, so the next run behaves the same
*/
        ticks = 0;
        armed = true;
        st = uc_vm_execute(&vm, program, &rv);
        printf("a fresh run stops again: status=%d\n", (int)st);
        ucv_put(rv); rv = NULL;
        st = uc_vm_resume(&vm);
        printf("and finishes when it is left alone: status=%d value=%s\n",
(int)st,
               vm.stack.count ? ucv_to_string(&vm, uc_vm_stack_peek(&vm,
0)) : "empty");

        uc_program_put(program);
        uc_vm_free(&vm);

        return 0;
}
```

```
break requested before the run: 0
the descriptor an event loop watches: a readable fd
run status=2 return=null requested now=0
resume status=0 top of stack=6
a fresh run stops again: status=2
and finishes when it is left alone: status=0 value=6
```

The value a resumed run ends with is on the operand stack rather than in a return slot, which is how a debugger hands a value back into a broken run and how a host reads one: `uc_vm_stack_peek(&vm, 0)`. A break is not a failure, so it leaves no exception behind, which is why the two runs above are repeatable once the counter is reset.

For a watchdog the recipe is: watch `uc_vm_break_notifyfd` alongside whatever else the loop polls, and `uc_vm_break_request` from wherever the timeout is noticed; a request from another thread is a store to a flag plus one write to a pipe. Nothing else in the VM is safe to touch from a second thread while a run is in progress (chapter 42).

Signals

A script installs a signal handler with the `signal` built-in (chapter 20): `signal("USR1")` reports what is in place, `signal("USR1", "ignore")` and `signal("USR1", "default")` set the system disposition, and `signal("USR1", f)` registers `f` and directs the process signal disposition to the VM. Signal names work with or without the `SIG` prefix and in either case, and `uc_system_signal_names` is the same table from C.

What the last form does is *not* to run `f` in the signal context. The disposition the VM installs calls `uc_vm_signal_raise`, which sets a bit for the number and writes one byte to a self-pipe. The handler is invoked later, by `uc_vm_signal_dispatch`, from the instruction loop — which is the mechanism the loop uses between instructions, so a handler runs where a script could have been interrupted anyway and with the VM in a consistent state. The dispatch is also what the native code that a script called returns through: a signal that arrives while the VM is inside a long native call waits until that native returns.

The pieces a host uses:

```
void uc_vm_signal_raise(uc_vm_t *vm, int
signo);      /* the same path a real signal takes */
uc_exception_type_t uc_vm_signal_dispatch(uc_vm_t *vm);
int  uc_vm_signal_notifyfd(uc_vm_t *vm);              /* the read end of the
self-pipe */
void uc_vm_signal_handlers_ensure(uc_vm_t *vm);
```

The self-pipe, the handler table and the disposition template are not built by `uc_vm_init`. The config flag `setup_signal_handlers` asks for them; `uc_vm_signal_handlers_ensure` asks for them directly and is a no-op once they exist. Until they do, `uc_vm_signal_notifyfd` answers `-1`, and this is the thing to check, because a

script calling `signal("USR1", f)` before any of that has happened installs an unwritten disposition — which is to say, disposes of the process on the next such signal. `uc_vm_signal_raise` accepts numbers below `UC_SYSTEM_SIGNAL_COUNT` and ignores anything outside that.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static uc_value_t *
record(uc_vm_t *vm, size_t nargs)
{
        uc_value_t *n = uc_fn_arg(0);

        printf("  [script handler ran, argument=%s]\n", n ?
ucv_to_string(vm, n) : "none");

        return ucv_boolean_new(true);
}

static uc_value_t *
boom(uc_vm_t *vm, size_t nargs)
{
        (void) nargs;

        uc_vm_raise_exception(vm, EXCEPTION_USER, "the handler did not like
this");

        return NULL;
}

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_program_t *program;
        uc_source_t *src;
        uc_value_t *handler;

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));

        printf("the pipe is not set up by default: notifyfd=%d\n",
uc_vm_signal_notifyfd(&vm));
        uc_vm_signal_handlers_ensure(&vm);
        printf("after asking for it: %s\n",
```

```
                uc_vm_signal_notifyfd(&vm) >= 0 ? "there is a descriptor to
watch" : "still none");

        uc_function_register(uc_vm_scope_get(&vm), "record", record);
        uc_function_register(uc_vm_scope_get(&vm), "boom", boom);

        handler = ucv_cfunction_new("record", record);
        ucv_object_add(uc_vm_scope_get(&vm), "h", handler);

        {
                const char *code = "signal('USR1', h);\nprint('installed\
\n');\n";

                src = uc_source_new_buffer("sig.uc", strdup(code),
strlen(code));
                program = uc_compile(&config, src, NULL);
                uc_source_put(src);
                uc_vm_execute(&vm, program, NULL);
                uc_program_put(program);
        }

        printf("nothing raised yet: dispatch=%d\n",
(int)uc_vm_signal_dispatch(&vm));

        printf("raising USR1 from the host\n");
        uc_vm_signal_raise(&vm, SIGUSR1);
        printf("  raise alone runs nothing\n");

        {
                int d = (int)uc_vm_signal_dispatch(&vm);

                printf("  dispatch delivered it, returning %d\n", d);
        }

        printf("raising again, then running a program that reaches the
check\n");
        uc_vm_signal_raise(&vm, SIGUSR1);

        {
                const char *code = "let i = 0;\nwhile (i < 3) { i = i + 1; }
\nprint('looped\\n');\n";

                src = uc_source_new_buffer("run.uc", strdup(code),
strlen(code));
                program = uc_compile(&config, src, NULL);
                uc_source_put(src);
                uc_vm_execute(&vm, program, NULL);
```

```
                uc_program_put(program);
        }

        printf("a handler that fails: dispatch reports it\n");
        {
                const char *code = "signal('USR2', function(n) { boom(); });
\n";

                src = uc_source_new_buffer("fail.uc", strdup(code),
strlen(code));
                program = uc_compile(&config, src, NULL);
                uc_source_put(src);
                uc_vm_execute(&vm, program, NULL);
                uc_program_put(program);
        }

        uc_vm_signal_raise(&vm, SIGUSR2);
        printf("  dispatch=%d\n", (int)uc_vm_signal_dispatch(&vm));

        uc_vm_free(&vm);

        return 0;
}
```

```
the pipe is not set up by default: notifyfd=-1
after asking for it: there is a descriptor to watch
installed
nothing raised yet: dispatch=0
raising USR1 from the host
  raise alone runs nothing
  [script handler ran, argument=10]
  dispatch delivered it, returning 0
raising again, then running a program that reaches the check
  [script handler ran, argument=10]
looped
a handler that fails: dispatch reports it
  dispatch=5
```

Read that against the three claims it rests on. `uc_vm_signal_raise` records and returns; it runs nothing. `uc_vm_signal_dispatch` is what runs the handlers, and it hands each one the signal number as its single argument. And a run does the same dispatching on its own between instructions, which is why the handler ran with no host code asking for it in the middle case. A handler that fails is reported by dispatch as the exception type it raised; through the run path, that becomes a normal failure of the interrupted program, so a signal handler is script code subject to the same rules as any other and should be short.

The dispatch loop drains the self-pipe and walks a bitmap of pending numbers, so a signal that arrives between the drain and the walk is not lost, and repeated signals of one number collapse to one call — which is the right behaviour for a handler that means "state has changed, look again" and the wrong one for a handler that means "count this".

Two limits belong with the mechanism. The thread context remembers a single VM as its signal handler, so `uc_vm_signal_handlers_ensure` on a second VM in the same thread does nothing and that VM's `uc_vm_signal_notifyfd` stays `-1`: one VM per thread owns the system dispositions, and a second VM that installs a script handler installs the unwritten disposition of its own record. And a script handler runs inside the VM, so it must not block, must not wait on the host, and must not be expected to run while the VM is idle — for that, watch the descriptor and let the loop decide.

Breakpoints, and the one that is worth a host's attention

```
typedef struct uc_breakpoint {
        uint8_t *ip;
        void (*cb)(uc_vm_t *, struct uc_breakpoint *);
} uc_breakpoint_t;
```

A breakpoint is an instruction pointer and a callback, kept in `vm->breakpoints`. Addressed breakpoints are the debugger's material — the addresses are positions in a function's chunk, and the debugger interface is described in chapter 60 — but one entry is defined for hosts, by way of a sentinel value in the `ip` field:

```
extern uint8_t *const UC_BREAKPOINT_UNCAUGHT_EXCEPTION;
```

A breakpoint carrying that pointer is invoked at the one moment it can be useful: when an exception has been raised and nothing between the current frame and the run's own boundary would catch it, *before* the frames are unwound. The callback therefore sees the frames, the operand stack and the record as the failure left them, which is the state no log line reproduces.

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

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static size_t visits = 0;

/* runs while the frame that failed is still in place */
static void
atfailure(uc_vm_t *vm, uc_breakpoint_t *bp)
{
        (void) bp;

        visits++;

        /* the point of the hook is that none of this has been unwound yet */
        printf("  [%zu frames, %zu values on the stack, pending: %s]\n",
               vm->callframes.count, vm->stack.count,
               vm->exception.message ? vm->exception.message : "(no
message)");
}

int main(void)
{
        /* the hook writes to standard output, which is block buffered when
piped:
           line-buffer it so a merged transcript keeps the two channels in
order */
        setvbuf(stdout, NULL, _IOLBF, 0);

        uc_vm_t vm = { 0 };
        uc_breakpoint_t *bp;
        uc_program_t *program;
        uc_source_t *src;
        uc_value_t *rv = NULL;
        uc_vm_status_t st;
        const char *code =
                "function deep(n) {\n"
                "    if (n == 0) {\n"
                "        let x = 1;\n"
                "        x();\n"
                "    }\n"
```

```
                "    else {\n"
                "        return deep(n - 1);\n"
                "    }\n"
                "}\n"
                "deep(2);\n";

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));

        /* the VM releases the entries it holds, so this one is its to free
*/
        bp = malloc(sizeof(*bp));
        bp->ip = UC_BREAKPOINT_UNCAUGHT_EXCEPTION;
        bp->cb = atfailure;
        uc_vector_push(&vm.breakpoints, bp);

        src = uc_source_new_buffer("hook.uc", strdup(code), strlen(code));
        program = uc_compile(&config, src, NULL);
        uc_source_put(src);

        st = uc_vm_execute(&vm, program, &rv);
        printf("status=%d visits=%zu\n", (int)st, visits);
        ucv_put(rv);

        printf("a second failure reaches it again\n");
        st = uc_vm_execute(&vm, program, &rv);
        printf("status=%d visits=%zu\n", (int)st, visits);
        ucv_put(rv);

        uc_program_put(program);
        uc_vm_free(&vm);

        return 0;
}
```

```
  [1 frames, 4 values on the stack, pending: left-hand side is not a
function]
Type error: left-hand side is not a function
In hook.uc, line 4, byte 11:
  (3 tail call frames omitted)

 `        x();`
            ^-- Near here


status=4 visits=1
a second failure reaches it again
  [1 frames, 4 values on the stack, pending: left-hand side is not a
function]
Type error: left-hand side is not a function
In hook.uc, line 4, byte 11:
  (3 tail call frames omitted)

 `        x();`
            ^-- Near here


status=4 visits=2
```

The callback runs once per failure, before the handler, which is the ordering that makes it worth installing: it collects state while the handler is left to report. Three details of ownership and effect: push the structure with `uc_vector_push(&vm.breakpoints, ...)` and allocate it with `malloc`, because the VM frees the entries when the VM is freed; the callback may run script code, and if that code ends the program with `exit`, the run comes back as `STATUS_EXIT` with the frames reset; and the frames it inspects are the public `uc_callframe_t` entries, whose `closure` field is opaque to a host outside this tree, so the useful things to take from the moment are the counts, the record, the operand stack, and whatever the script's own `stacktrace` in the record carries.

What to install

Need	Use
Report a failure the way the interpreter does	nothing: the installed `uc_vm_output_exception` already does
Send failures to a log, a socket, `syslog`	`uc_vm_exception_handler_set`, keeping the displaced handler
The frames and the values as the failure found them	the `UC_BREAKPOINT_UNCAUGHT_EXCEPTION` hook
Hand the failure to a script function	`uc_vm_exception_object`, guarded on the type
Stop a run that has gone on too long	`uc_vm_break_request`, with `uc_vm_break_notifyfd` in the poll set
React to a signal with script code	`handlers_ensure` or the config flag, then `signal()` in the script
React to a signal with host code	`uc_vm_signal_notifyfd`, and let the loop call `uc_vm_signal_dispatch`
Run two programs in sequence on one VM	drain the stack after any break; the record clears itself on entry

The two rows at the bottom are not documented in any header, and they are the two that cost the most time to find by other means. The record half of the last row takes care of itself now that both entry points clear it on entry; the break residue is what still needs the host's hand.

Programs, bytecode and precompilation

A program is the compiled form of one or more sources: the functions, the constants they refer to, and the sources they name. Chapter 43 covered how to get one from text and how to write one back out; this chapter is about the file itself — what is in it, what the debug information really costs and buys, what the version check does and does not protect against, and where the shipped command-line tools fit.

```
uc_program_t *uc_program_new(void);
uc_program_t *uc_program_get(uc_program_t *program);
void          uc_program_put(uc_program_t *program);
uc_value_t *uc_program_main(uc_vm_t *vm, uc_program_t *program);

void          uc_program_write(uc_program_t *program, FILE *fp, bool debug);
uc_program_t *uc_program_load(uc_source_t *source, char **errp);

#define UC_PRECOMPILED_BYTECODE_MAGIC 0x1b756362
#define UCODE_BYTECODE_VERSION        0x02
```

Getting a program, either way

There are two ways to obtain a program, and one of them usually chooses itself. `uc_compile` looks at what the source begins with and goes down the appropriate path itself: text is compiled, and the four magic bytes of a precompiled file are handed to the loader.

```
        /* a path that may hold text or bytecode, either way it is a program
*/
        program = uc_compile(&config, uc_source_new_file(argv[1]), &error);
```

`uc_program_load` is the bytecode-only route below it. It is the one to use when you know what you have — reading your own cache, say — and it is what to avoid when you do not, because text handed to it is immediately refused:

```
        /* a path that must be bytecode */
        program = uc_program_load(uc_source_new_file(cachefile), &error);
```

Two details of that call are the opposite of what its header comment says, and both are the sort a host reads off the declaration rather than out of the code. The load takes no reference to the source, so the caller releases the source, and the error argument is not optional on the two header checks — a null there dies in the formatter rather than reporting the mismatch. The source it is given has to be one it can read: the loader reads from the source's stream, so a buffer source over an in-memory copy works as readily as a file.

The call that tells you what a file is, `uc_source_type_test()`, is declared in `ucode/internal/source.h` and marked hidden, so it is not a question a host outside this tree can ask; going through `uc_compile` is the way to avoid needing it.

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

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static const char script[] =
        "function greet(who) {\n"
        "    return 'hello, ' + who;\n"
        "}\n"
        "return greet('world');\n";

static long
writeout(const char *path, bool debug)
{
        uc_source_t *source = uc_source_new_buffer("greet.uc",
                                                   strndup(script,
strlen(script)), strlen(script));
        uc_program_t *program = uc_compile(&config, source, NULL);
        FILE *fp;
        long size;

        uc_source_put(source);

        if (program == NULL)
                return -1;

        fp = fopen(path, "wb");
        uc_program_write(program, fp, debug);
        fclose(fp);
        uc_program_put(program);

        {
                FILE *in = fopen(path, "rb");

                fseek(in, 0, SEEK_END);
                size = ftell(in);
                fclose(in);
        }

        return size;
}
```

```
static void
decode(const char *path)
{
        FILE *in = fopen(path, "rb");
        unsigned char header[8];
        uint32_t magic, flags;

        if (fread(header, 1, 8, in) != 8) {
                printf("short file\n");
                fclose(in);

                return;
        }
        fclose(in);

        magic = ((uint32_t)header[0] << 24) | ((uint32_t)header[1] << 16) |
                ((uint32_t)header[2] << 8) | header[3];
        flags = ((uint32_t)header[4] << 24) | ((uint32_t)header[5] << 16) |
                ((uint32_t)header[6] << 8) | header[7];

        printf("%s: magic %s, version 0x%02x, flags%s%s%s\n", path,
               magic == UC_PRECOMPILED_BYTECODE_MAGIC ? "as expected" : "not
as expected",
               (unsigned)(flags >> 24),
               (flags & 0x1) ? " debug" : "",
               (flags & 0x2) ? " sourceinfo" : "",
               (flags & 0x8) ? " exports" : "");
}

int main(void)
{
        long withdebug = writeout("/tmp/ucode-ch47-debug.uc.o", true);
        long bare = writeout("/tmp/ucode-ch47-bare.uc.o", false);

        printf("the same program, %ld bytes with debug and %ld without\n",
withdebug, bare);

        decode("/tmp/ucode-ch47-debug.uc.o");
        decode("/tmp/ucode-ch47-bare.uc.o");

        /* a program read back is a program like any other, and runs more
than once */
        {
                uc_source_t *source = uc_source_new_file("/tmp/ucode-ch47-
debug.uc.o");
                /* a null error pointer is safe on a file that loads, and
only there */
```

```
                uc_program_t *program = uc_program_load(source, NULL);
                uc_vm_t vm = { 0 };
                uc_value_t *rv = NULL, *entry;
                uc_vm_status_t st;

                uc_source_put(source);

                uc_vm_init(&vm, &config);
                uc_stdlib_load(uc_vm_scope_get(&vm));

                entry = uc_program_main(&vm, program);
                printf("entry closure present: %d\n", entry != NULL);
                ucv_put(entry);

                st = uc_vm_execute(&vm, program, &rv);
                printf("first run status=%d value=%s\n", (int)st,
                       rv ? ucv_to_string(&vm, rv) : "the null value");
                ucv_put(rv); rv = NULL;

                st = uc_vm_execute(&vm, program, &rv);
                printf("second run status=%d value=%s\n", (int)st,
                       rv ? ucv_to_string(&vm, rv) : "the null value");
                ucv_put(rv);
                uc_vm_free(&vm);
                uc_program_put(program);
        }

        /* the text of the script is not a program file */
        {
                char *error = NULL;
                uc_source_t *source;
                uc_program_t *program;
                uc_vm_t vm = { 0 };
                uc_value_t *rv = NULL;
                uc_vm_status_t st;

                source = uc_source_new_buffer("plain.uc", strndup(script,
strlen(script)), strlen(script));
                program = uc_program_load(source, &error);
                printf("loading text through uc_program_load: %s",
program ? "worked" : error);
                free(error);
                uc_source_put(source);

                /* through uc_compile the same source is fine */
                source = uc_source_new_buffer("plain.uc", strndup(script,
strlen(script)), strlen(script));
```

```
                program = uc_compile(&config, source, NULL);
                uc_source_put(source);
                uc_vm_init(&vm, &config);
                uc_stdlib_load(uc_vm_scope_get(&vm));
                st = uc_vm_execute(&vm, program, &rv);
                printf(" and through uc_compile: status=%d value=%s\n",
(int)st,
                       rv ? ucv_to_string(&vm, rv) : "the null value");

                ucv_put(rv);
                uc_vm_free(&vm);
                uc_program_put(program);
        }

        return 0;
}
```

```
the same program, 440 bytes with debug and 116 without
/tmp/ucode-ch47-debug.uc.o: magic as expected, version 0x02, flags debug
sourceinfo
/tmp/ucode-ch47-bare.uc.o: magic as expected, version 0x02, flags
entry closure present: 1
first run status=0 value=hello, world
second run status=0 value=hello, world
loading text through uc_program_load: Invalid file magic
 and through uc_compile: status=0 value=hello, world
```

What is in the file

Every multi-byte number is big-endian, and every variable-length item is a length followed by the bytes, so the file can be walked without knowing anything about the machine that wrote it. The command-line tools put a shebang line in front of all this, so a deployed file is directly executable and a reader has to skip that line before the magic; a file written by `uc_program_write` starts with the magic itself. The order after the optional shebang is:

Item	Present when	Contents
Magic	always	`0x1b756362` — an escape and `u` `c` `b`
Flags	always	the version in the top byte, then the program flags
Source information	`F_SOURCEINFO`	a count, then per source: its name, its text if it had one, and its line index
Constants	always	the constant pool, as a value list
Exports	`F_EXPORTS`	the names the first source exported
Function count	always	how many functions follow
Functions	that count	one record each

A function record is a flag word, then the name if it has one, then the two sizes and the two source positions, then the chunk:

```
        if (debug && func->name[0])
                flags |= UC_FUNCTION_F_HAS_NAME;

        if (debug && func->chunk.debuginfo.variables.count)
                flags |= UC_FUNCTION_F_HAS_VARDBG;

        if (debug && func->chunk.debuginfo.offsets.count)
                flags |= UC_FUNCTION_F_HAS_OFFSETDBG;
```

and the remaining flag bits record the things the VM has to know before it can run the code: whether the function is an arrow, whether it takes a variable number of arguments, whether it was compiled in strict mode, whether it is a module, and whether it has exception ranges — the table against which `try` / `catch` unwinds. Padding after a variable-length item brings the next one to a four-byte boundary.

The header carries no architecture field, no word size and no checksum. The two things that are checked are the magic and the version, and both are checked against this library. Everything else is taken on trust, which is the sentence to read before loading a bytecode file that arrived over a network: it is a program in exactly the sense that a shared object is, and it will run.

Debug information

`uc_program_write`'s third argument is the debug switch. Chapter 43 noted that it is not a compression flag, despite the header; what it does is set `UC_PROGRAM_F_DEBUG` on the file and, with it, `UC_PROGRAM_F_SOURCEINFO` —

function names, per-variable and per-offset debug tables, source names, and the line index — which for one small script is the difference between 116 bytes and 440.

The value of all that is diagnostic. A program loaded from a bare file reports its position as `In [no source], line 1, byte 10` with nothing to compare it against, because the loader has to invent a source to hang the positions on and calls it `[no source]`. A program loaded from a debug file reports the name, the line and the byte, which is the same report an interpreted run gives.

The one part that is not in the file is the source text, unless the program was compiled from a buffer. A program compiled from a *file* records the file's name, and on load the loader opens that file — which is how a deployed program gets its failing line quoted back, and what it means in practice is that a debug build and its sources travel together. The same example, with the source renamed away between the two runs:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <ucode/ucode.h>


static uc_parse_config_t config = { .raw_mode = true };

static const char script[] =
        "let x = 1;\n"
        "x();\n";

#define PATH "/tmp/ucode-ch47-lost.uc"

int main(void)
{
        uc_source_t *source;
        uc_program_t *program;
        FILE *fp;

        setvbuf(stdout, NULL, _IOLBF, 0);

        {
                FILE *out = fopen(PATH, "w");

                fputs(script, out);
                fclose(out);
        }

        source = uc_source_new_file(PATH);
        program = uc_compile(&config, source, NULL);
        uc_source_put(source);

        fp = fopen("/tmp/ucode-ch47-lost.uc.o", "wb");
        uc_program_write(program, fp, true);
        fclose(fp);
        uc_program_put(program);

        printf("the source is where it was\n");

        {
                uc_source_t *source = uc_source_new_file("/tmp/ucode-ch47-
lost.uc.o");
                uc_vm_t vm = { 0 };
```

```
                uc_program_t *loaded = uc_program_load(source, NULL);
                uc_value_t *rv = NULL;
                uc_vm_status_t st;

                uc_source_put(source);

                uc_vm_init(&vm, &config);
                uc_stdlib_load(uc_vm_scope_get(&vm));
                st = uc_vm_execute(&vm, loaded, &rv);
                printf("status=%d\n", (int)st);
                ucv_put(rv);
                uc_vm_free(&vm);
                uc_program_put(loaded);
        }

        printf("the same file after the source is renamed away\n");
        rename(PATH, "/tmp/ucode-ch47-lost.uc.moved");

        {
                uc_source_t *source = uc_source_new_file("/tmp/ucode-ch47-
lost.uc.o");
                uc_vm_t vm = { 0 };
                uc_program_t *loaded = uc_program_load(source, NULL);
                uc_value_t *rv = NULL;
                uc_vm_status_t st;

                uc_source_put(source);

                uc_vm_init(&vm, &config);
                uc_stdlib_load(uc_vm_scope_get(&vm));
                st = uc_vm_execute(&vm, loaded, &rv);
                printf("status=%d\n", (int)st);
                ucv_put(rv);
                uc_vm_free(&vm);
                uc_program_put(loaded);
        }

        rename("/tmp/ucode-ch47-lost.uc.moved", PATH);
        unlink(PATH);
        unlink("/tmp/ucode-ch47-lost.uc.o");

        return 0;
}
```

```
the source is where it was
Type error: left-hand side is not a function
In /tmp/ucode-ch47-lost.uc, line 2, byte 3:

 `x();`
    ^-- Near here


status=4
the same file after the source is renamed away
Unable to open source file /tmp/ucode-ch47-lost.uc: No such file or directory
Type error: left-hand side is not a function
In /tmp/ucode-ch47-lost.uc, line 2, byte 3:


status=4
```

The second run is what a device looks like when the bytecode was deployed without its sources. Loading the file prints a complaint; the position is still reported because it is in the file, and only the quoted line and its marker are missing. Three ways out, in order of how much they cost: keep the sources next to the deployed programs; compile from a buffer so the text is embedded, at the cost of having to read the text in yourself; or ship the bare file and accept that positions are numbers.

The third flag, `UC_PROGRAM_F_SOURCEBUF`, is defined and never written — the buffer branch above is what that bit was for — so nothing in a real file has it set.

Precompiled modules

A module can be shipped in compiled form as well, and the shape is not what one might guess: a precompiled module carries the ordinary module name and extension, because that is what the search path templates can reach. `uc_require_path()` expands a template into `<prefix><name><suffix>` and then accepts the result only when the suffix it expanded to is exactly `.so` or `.uc`, so the format is not distinguished by the name at all — a `.uc` file holding bytecode is a precompiled module, and a name outside those two suffixes is unreachable however the template is written:

```
$ ucode -L '*.uc.o' app.uc
Runtime error: No module named 'm' could be found
In app.uc, line 1, byte 19:

 `let m = import("m");`
  Near here --------^
```

Both halves of the import then work the ordinary way. Resolution happens at compile time as ever, so the module has to be present and findable while the importer is compiled; and since this tree cannot link a precompiled module into another program, the import is turned into a run-time load:

```
        /* We do not yet support linking precompiled modules at compile time,
           turn static import operation into dynamic load one. */
        if (uc_source_type_test(source) == UC_SOURCE_TYPE_PRECOMPILED)
                return uc_compiler_compile_dynload(compiler, modname,
imports);
```

The one consequence that matters when deploying is this: a program that imports a precompiled module still needs that module at run time, in the run-time search path, while a program that imports a *text* module has the module inside it. A whole application tree can be precompiled with this in mind:

```
# the module: compiled with the module flag, still named after the module
$ ucode -cmodule -o util.tmp util.uc && mv util.tmp mods/util.uc

# the application: compiled against it, and finding it again at run time
$ ucode -cmodule -L 'mods/*.uc' -o app.tmp app.uc && mv app.tmp app.uc
$ ucode -L 'mods/*.uc' app.uc
printed: mark-value
```

The `-L` entries go in front of the built-in ones, so a deployment path is an override rather than a fallback:

```
$ ucode -L 'mods/*.uc' -e 'for (let p in REQUIRE_SEARCH_PATH) print(p, "  |
");'
mods/*.uc  |  /usr/local/lib/ucode/*.so  |  /usr/local/share/ucode/*.uc
|  ./*.so  |  ./*.uc
```

The one diagnostic oddity of running a precompiled program is that its error reports quote the shebang, since the position in the file is a byte offset into a file whose first line is the interpreter line and nothing else:

```
$ ucode app.uc
Runtime error: No module named 'util' could be found
In app.uc, line 1, byte 1:

 `#!/usr/bin/env ucode`
  Near here --------^
```

A precompiled module found by the default search path needs no `-L` at all — the installed list ends in `./*.uc`, which is what makes `/usr/share/ucode/*.uc` and a program's own directory both work. A dotted module name keeps mapping to directories, so `example.test` resolves to `example/test.uc` whether that file holds text or

bytecode, and a precompiled module can import another precompiled module. The loaded module lands in `modules` as an object under the name it was imported by, so the same notes about refreshing that table apply (chapter 17).

Two combinations are worth knowing about. `require()` is not the module route (chapter 17) and the two failure shapes differ: a text module with exports cannot be required at all, while a precompiled one is loaded and yields the null value, because the module-ness is already in the file and nothing re-checks it:

```
$ ucode -e 'require("plain")'                    # a text module with exports
Runtime error: Unable to compile source file './plain.uc':

  | Syntax error: Exports may only appear at top level of a module

$ ucode -e 'let h = require("hello"); print("[", type(h), "]\n");'   # the
precompiled one
[]
```

And a debug build of a module remembers where its sources were, so deploying the compiled file without them produces a complaint per module at load — the module still works:

```
$ ucode -L 'libs/*.uc' app.uc
Unable to open source file /tmp/ch47h/libs/util.uc: No such file or directory
mark is the-unique-mark-string
```

That is the argument for `-s` on installed modules: nothing is lost that the deployment was going to lose anyway, since the sources are not on the device. The name in that complaint is the one recorded when the module was compiled, and it travels with the module — so precompiling an application that reads a precompiled module can produce it about a source the application never had:

The version and the magic

`UCODE_BYTECODE_VERSION` is a plain integer in the public header, currently 0x02, and it is the only thing about the file's shape that is checked besides the magic. The version byte is the top byte of the flags word, which is why the message about a mismatch speaks in two-digit hex:

```
Bytecode version mismatch, got 0x7f, expected 0x02
```

(That particular pair came from hand-poking a byte in a file's flags word; a real mismatch is between two builds of ucode, and there is no compatibility promise between versions, which in practice means a device's bytecode is rebuilt whenever its interpreter is.)

There is no other validation. The absence of a checksum and a signature is worth stating plainly, because a bytecode file is a program in the same sense a shared object is: nothing about it is checked before it runs, so a device that reads precompiled programs from somewhere it does not control is executing whatever that somewhere contains. Chapter 53's deployment notes and chapter 40's embedding rules both come back to this.

From the command line

`-c[flags]`	compile and write the program rather than run it
`-o file`	where to write it; `-` is standard output, the default is `./uc.out`
`-s`	leave out the debug information
`-cmodule`	compile in module mode, which a file containing `export` needs
`-cno-interp`	leave out the shebang line
`-cinterp=path`	use that shebang line, `/usr/bin/env ucode` by default
`-cdynlink=name`	leave imports of that name to run time

Everything the shebang is about is worth the detail, since it is what makes a precompiled file behave like a script. The tools write it first, the file is created executable, and giving it a shebang means it runs on its own:

```
$ ucode -cmodule -L 'libs/*.uc' -o app-embed.uc app.uc
$ ls -l app-embed.uc
-rwxrwxr-x 1 jow jow 609 Sep 18 21:22 app-embed.uc
$ ./app-embed.uc
mark is the-unique-mark-string
```

`-cno-interp` removes the line, which is what to use when the file is an input to something else rather than a program to run, and `head` shows the difference:

```
$ ucode -cno-interp -cmodule -L 'libs/*.uc' -o no-shebang.uc app.uc
$ head -c 8 no-shebang.uc | od -c | head -1
0000000 033   u   c   b 002  \0  \0 003
```

`-cmodule` is not optional once the source exports anything: compiling such a file in the ordinary mode is refused, which is the same rule `import` enforces (chapter 17):

```
$ ucode -c -o a3.uc upgrade.uc
Syntax error: Exports may only appear at top level of a module
In line 1, byte 1:

 `export function up() { return 1; };`
  ^-- Near here
```

Two ordering and cleanliness details of the tool itself are worth knowing, because both are silent. The output path is reset by the compile switch, so an `-o` that comes *before* the `-c` it belongs to is discarded and the output lands in `./uc.out` — the pair has to be written the other way round:

```
$ ucode -o wanted.uc -cmodule util.uc
$ ls -l wanted.uc uc.out
ls: cannot access 'wanted.uc': No such file or directory
-rwxrwxr-x 1 jow jow 357 Sep 18 21:30 uc.out
```

The same reset means the two switches have to be written the other way round: `-cmodule -o app.uc`, never `-o app.uc -cmodule`. The words `-c` accepts are checked, so an unknown one is at least announced:

```
$ ucode -cupgrade.uc.o upgrade.uc
Unrecognized -c flag "upgrade.uc.o", ignoring
```

And the output is opened before the source is compiled, so a run that fails on a compile error has still created its output, as an empty file — which matters to anything with a build rule keyed on that name:

```
$ ucode -c -o failed.uc util.uc
Syntax error: Exports may only appear at top level of a module
$ ls -l failed.uc
-rwxrwxr-x 1 jow jow 0 Sep 18 21:30 failed.uc
```

Programs in a host

A program is reference counted: `uc_program_new` returns one with a single owner, `uc_program_get` and `uc_program_put` add and drop a reference, and a host gives its reference back with `uc_program_put` (the `uc_vm_own_program` and `uc_vm_disown_program` pair of chapter 42 are the transfer forms of the same accounting). The one function of a program a host can name is the entry one, reached as a closure through `uc_program_main()`; it is the function a run starts at, and it is `NULL` for a program with no functions. The `uc_function_t` itself is internal. Everything else about a program — the function list, the constants, the exports table — is reached only through the internal header, so the ways to get named things out of a program are the two from chapter 40: let it return a namespace, or let it write into a scope.

What a loaded program is good for is being run, more than once: the example above ran the same loaded program twice and got the same value both times, because nothing about a program is consumed by a run (chapter 42's state notes cover what *is* consumed — the VM's own state, not the program's).

To	Do
Compile text	`uc_compile()` over a source
Read bytecode you wrote	`uc_program_load()`
Read a file that may be either	`uc_compile()` — it dispatches
Find where a run begins	`uc_program_main()`
Keep a program past a VM	`uc_program_get()` and `uc_program_put()`
Run it again	run it again

Summary

- A program is what a run executes; the compiled form is a file as well, with a magic word, a flags word carrying the version, and then the sources, constants, exports and functions.
- The tools write a shebang first and make the file executable, so precompiled programs run directly; the reader skips that line before looking for the magic.
- Debug information roughly quadruples a small program, carries the source *names* and line index rather than the text, and therefore needs the sources deployed beside it to quote a failing line; `-s` is the honest choice when they are not.
- A precompiled module keeps the ordinary `.uc` name, because nothing else is reachable through the search path; it is loaded at run time rather than linked in, so it has to be in the run-time path.
- Importing a *text* module embeds it, which is what makes a precompiled application self-contained; `-cdynlink=` and a precompiled module are the two ways out of that.
- Only the magic and the version are checked; there is no checksum, no signature, no compatibility promise between versions, and no reason to load bytecode from a source you do not control.
- Of a loaded program a host can reach the entry function and nothing else; it runs as often as you like.

Writing a native module

A module is a shared object the interpreter loads and asks, by one agreed function, to install itself. That is all the word means here: there is no manifest, no version record, no registration database, and the whole contract is a function name, an object to put things in, and the virtual machine to put types and state on. Chapter 17 covered the script side — `import`, `require()`, the search path — and chapter 40 the embedding API a module is written against; this chapter is the module itself, from the entry point to the file on the device.

The shipped set is written this way too. `lib/math.c` ends with exactly the function your module will end with, and the debugger the interpreter loads for `-D` is reached by the same code path as a third-party module, so nothing described here is a side door.

The entry point

```
void uc_module_init(uc_vm_t *vm, uc_value_t *scope);
```

Include `<ucode/module.h>` and define `uc_module_init`. The header declares it weak, so a module that does not define one is legal, and it also defines the symbol the loader actually looks for:

```
void uc_module_init(uc_vm_t *vm, uc_value_t *scope) __attribute__((weak));

void uc_module_entry(uc_vm_t *vm, uc_value_t *scope);
void uc_module_entry(uc_vm_t *vm, uc_value_t *scope)
{
        if (uc_module_init)
                uc_module_init(vm, scope);
}
```

`uc_module_entry` is what the loader resolves by name, and it is defined in the header rather than declared, which has one consequence worth knowing before a module is split into two files: including that header in two translation units of one module gives a duplicate definition at link time.

```
$ cc -fPIC -shared -I include tu1.c tu2.c -o tu.so
ld.bfd: /tmp/ccV1Gc84.o: in function `uc_module_entry':
tu2.c:(.text+0x0): multiple definition of `uc_module_entry'; /tmp/
ccFIx2Q7.o:tu1.c:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
```

Keep the one file that includes `<ucode/module.h>` as the file that defines the entry point, and give the others `<ucode/ucode.h>`. A module with nothing to install is accepted and yields an empty object:

```
$ ucode -L './*.so' -e
'let x = require("noinit"); print("type: ", type(x), " keys: ", length(x),
"\n");'
type: object keys: 0
```

What `scope` is

It is a fresh object, created by the loader immediately before the call:

```
        scope = ucv_object_new(vm);

        init(vm, scope);

        *res = scope;
```

Everything the module adds to it becomes a member of the module: the value `require()` returns, the names `import` binds out of, and what `import * as ns` hands over. It is not a scope in the sense chapters 5 and 40 use the word — there is no lexical chain above it, it is a plain object — so installing a member is `ucv_object_add` and nothing else. The natural shape of a module's body is therefore one table call per thing it offers, and this is the whole of what most modules do:

```
void
uc_module_init(uc_vm_t *vm, uc_value_t *scope)
{
        uc_function_list_register(scope, ch48_fns);

        ucv_object_add(scope, "version", ucv_int64_new(1));

        uc_type_declare(vm, "ch48.counter", counter_fns, counter_free);

        uc_vm_registry_set(vm, "ch48.state", ucv_int64_new(0));
}
```

The virtual machine is the second argument and is where the things that do not belong in an object go: resource types and the registry for state. It is the same VM the script is running on, so a module can reach the global scope with `uc_vm_scope_get(vm)` and install there instead. That is possible and it is a choice to make deliberately, because it puts a name in every script's environment rather than in the one place the script asked for:

```
$ ucode -L './*.so' -e 'let m = require("b"); print("prefixed: ", m.two(),
"\n"); print("bare global: ", installed(), "\n");'
prefixed: two
bare global: reached without a prefix
```

A module in full

This is the module the transcripts in this chapter were produced with. It offers two plain functions, one function returning a formatted string, and a resource type with a constructor and a method:

```c
#include <stdlib.h>
#include <string.h>

#include <ucode/module.h>


/* -- functions ---------------------------------------------------------- */

static uc_value_t *
ch48_greet(uc_vm_t *vm, size_t nargs)
{
        uc_value_t *who = uc_fn_arg(0);
        uc_stringbuf_t *buf = ucv_stringbuf_new();

        ucv_stringbuf_printf(buf, "hello, %s", who ? ucv_to_string(vm,
who) : "nobody");

        return ucv_stringbuf_finish(buf);
}

static uc_value_t *
ch48_answer(uc_vm_t *vm, size_t nargs)
{
        (void)vm;
        (void)nargs;

        return ucv_int64_new(42);
}

/* -- a resource type ---------------------------------------------------- */

typedef struct {
        int count;
} counter_t;

static void
counter_free(void *data)
{
        free(data);
}

static uc_value_t *
counter_next(uc_vm_t *vm, size_t nargs)
{
        counter_t *counter = uc_fn_thisval("ch48.counter");
```

```
        (void)nargs;

        counter->count++;

        return ucv_int64_new(counter->count);
}

static uc_value_t *
ch48_counter(uc_vm_t *vm, size_t nargs)
{
        counter_t *counter;

        (void)nargs;

        counter = calloc(1, sizeof(counter_t));

        return ucv_resource_new(ucv_resource_type_lookup(vm,
"ch48.counter"), counter);
}

/* -- the tables ------------------------------------------------------ */

static const uc_function_list_t counter_fns[] = {
        { "next",        counter_next },
};

static const uc_function_list_t ch48_fns[] = {
        { "greet",        ch48_greet },
        { "answer",        ch48_answer },
        { "counter",        ch48_counter },
};

/* -- the entry point ------------------------------------------------- */

void
uc_module_init(uc_vm_t *vm, uc_value_t *scope)
{
        uc_function_list_register(scope, ch48_fns);

        ucv_object_add(scope, "version", ucv_int64_new(1));

        uc_type_declare(vm, "ch48.counter", counter_fns, counter_free);

        uc_vm_registry_set(vm, "ch48.state", ucv_int64_new(0));
}
```

Three things in it are worth more than a glance, because each is a place where a first module goes wrong.

The tables are length-delimited, not null-terminated. `uc_function_list_register` and `uc_type_declare` get their count from `ARRAY_SIZE`, so a trailing `{ NULL, NULL }` row is not a terminator but an entry: it installs a member whose name is null. That is what every shipped module does — no list in `lib/` carries a sentinel — and the first time a module writer adds one out of habit, the load faults rather than complaining.

The type is on the VM, not in the object. `uc_type_declare` registers a resource type with the VM's type table, so a script cannot reach it as a member and cannot make one of its own. The module supplies a function that does, and that function looks the type up by name — `ucv_resource_type_lookup` — rather than keeping the pointer `uc_type_declare` returns. That is what `lib/socket.c` does, and for a module it is the sturdier choice for two reasons. The registry is per VM, so a pointer kept in a file-scope variable belongs to whichever VM loaded the module first and cannot be right for the others (the example at the end of this chapter measures the sharing). And registration is keyed on the name with the first one winning, so a module reloaded into a VM that still has its type gets that one back and a pointer it kept from an earlier load is not distinguishable from the live one by looking at it. Chapter 45 has the measurement.

`uc_fn_thisval` takes the type name and no virtual machine. The receiver macros fill in `vm` themselves, as chapter 44 set out; writing `uc_fn_thisval(vm, "ch48.counter")` is a type error the compiler catches.

Building it

```
$ cc -std=gnu11 -Wall -fPIC -shared -I /usr/include ch48.c -o ch48.so
$ ls -l ch48.so
-rwxrwxr-x 1 jow jow 16696 Sep 18 22:47 ch48.so
```

There is no `-lucode` in that command, and adding one is a mistake. A module's references into the library are left undefined and bound at load time from the program that is loading it:

```
$ nm -D ch48.so | grep " U u"
                 U ucv_cfunction_new
                 U ucv_int64_new
                 U uc_vm_registry_set
                 U uc_vm_stack_peek
                 U ucv_object_add
                 U ucv_object_new
                 U ucv_resource_data
                 U ucv_resource_type_add
                 U ucv_stringbuf_finish
                 U ucv_stringbuf_new
                 U ucv_to_string
$ ldd build/math.so
        linux-vdso.so.1
        libm.so.6
        libc.so.6
        /lib64/ld-linux-x86-64.so.2
```

The second command inspects the shipped `math.so`, and it names no `libucode` either. Two consequences follow that are easy to get backwards. A module cannot be opened on its own by a program that is not already the library — it needs a host with those symbols in it, which is what the interpreter is and what an embedder becomes (chapter 40). And the set of names a module may use is exactly the set the library exports: the default build gives every translation unit `visibility("hidden")`, so the entries in the internal headers are not there, and on this platform the build is the thing that says so:

```
$ cc -fPIC -shared -I include hidden.c -o hidden.so
ld.bfd: /tmp/ccX1eWIg.o: in function `hidden_grow':
hidden.c:(.text+0x3e): undefined reference to `uc_chunk_init'
ld.bfd: hidden.so: hidden symbol `uc_chunk_init' isn't defined
collect2: error: ld returned 1 exit status
```

On Apple, the build of a module carries an extra link option, `LINKER:-undefined,dynamic_lookup`, which defers exactly this question to load time instead of answering it at link time.

A module's other dependencies are ordinary library dependencies. The `ucv_stringbuf_printf` call in the example above is a macro that reaches the json-c library directly, and `sprintbuf` is left undefined in the module and found through the host's own dependency on that library, which is why `ldd` of the interpreter's `libucode` shows it:

```
$ nm -D ch48.so | grep " U s"
                 U sprintbuf
$ ldd build/libucode.so | grep json
        libjson-c.so.5 => /usr/lib/x86_64-linux-gnu/libjson-c.so.5
```

The installed location is `${prefix}/${libdir}/ucode/`, to which the module list in `CMakeLists.txt` installs the modules and which the built-in search path lists:

```
install(TARGETS ${LIBRARIES} LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}/
ucode)
```

```
$ ucode -e 'for (let p in REQUIRE_SEARCH_PATH) print(p, "\n");'
/usr/local/lib/ucode/*.so
/usr/local/share/ucode/*.uc
./*.so
./*.uc
```

Chapter 47's point about precompiled modules applies here in its plain form: a template has to end in `.so` for a native module to be found through it, and a dotted module name maps into directories — the file `pkgdir/a/b.so` is the module named `a.b` given a template of `pkgdir/*.so`:

```
$ ucode -L 'pkgdir/*.so' -e 'import * as m from "a.b"; print("dotted module
reached: ", m.two(), "\n");'
dotted module reached: two
```

Reaching it from a script

With the file in place, the three script routes behave as chapter 17 describes, and what they hand over is the object the module filled:

```
$ ucode -L './*.so' t1.uc
keys: answer counter greet version
answer: 42
greet: hello, world
a missing key has type:
$ ucode -L './*.so' i1.uc
greet: hello, named, answer: 42
$ ucode -L './*.so' i2.uc
as a namespace: object with greet: hello, ns
$ ucode -L './*.so' t3.uc
type of a counter: resource
next: 1 then: 2
```

The third transcript is the resource from the example module: a constructor member returning a resource, a method that keeps its own count, and `type()` answering `resource` as chapter 20 lists it. A missing member reads as the

null value rather than raising, so the members of a module are checked the way any object's are. Naming an export that is not there is a compile-time error with the name in it, and so is naming a module no template can reach:

```
$ ucode -L './*.so' -e 'import { nope } from "ch48";'
Reference error: Module does not export nope
$ ucode -L './*.so' -e 'import { greet } from "nosuchmodule";'
Syntax error: Unable to resolve path for module 'nosuchmodule'
```

`-l` is the preload form, documented as `-l [name=]library`, and it is not a separate mechanism: the option handler calls the same `require` the script would, and binds the result into the global scope.

```
$ ucode -L './*.so' -l ch48 -e
'print(type(ch48), " answer: ", ch48.answer(), "\n");'
object answer: 42
$ ucode -L './*.so' -l alias=ch48 -e 'print(type(alias), " greet: ",
alias.greet("alias"), "\n");'
object greet: hello, alias
```

That is how the interpreter gets its own debugger in front of the script for `-D`, and it is worth knowing that the preload happens through the script's own facilities, because it means a preload failure is a script exception with a script's message rather than something the loader reports.

What the loader checks, and what it cannot

The sequence in `uc_require_so` is a stat, a `dlopen`, one named lookup, an object, and the call, so a module can fail to load at the first three of those and then at whatever its own initialisation does:

	Message
The template reached no file	`No module named 'name' could be found`
The file is not a loadable object	`Unable to dlopen file '<path>': <dlerror>`
The object lacks the entry point	`Module '<path>' provides no 'uc_module_entry' function`
The init itself fails	whatever the module raises or faults

```
$ ucode -L './*.so' -e 'let x = require("notanobject");'
Runtime error: Unable to dlopen file './notanobject.so': ./notanobject.so:
invalid ELF header
$ ucode -L './*.so' -e 'let x = require("noentry");'
Runtime error: Module './noentry.so' provides no 'uc_module_entry' function
```

There is no version field in a module, and nothing compares one. Chapter 47's bytecode carries a version byte and refuses a mismatch; a module carries the signature of `uc_module_init` and that is the entire agreement. What that costs is easy to state and easy to skip over while it is still cheap: a module built against headers that disagree with the library it lands in links without complaint, because the linker is matching names and not types, and it fails at whatever point the disagreement is first executed. A module is therefore part of the build of the thing that loads it, and a device that carries modules is a device whose modules are rebuilt whenever its interpreter is.

Two details of `dlopen` are chosen rather than defaulted and are worth their two lines. `RTLD_LOCAL` keeps a module's own symbols out of the program's global namespace, so one module cannot accidentally satisfy another's references. `RTLD_LAZY` binds a module's undefined references as they are first used, so a missing reference surfaces at the call rather than at the load — on this platform the module build settles the question first, as above, and on a build that defers it the failure moves later.

Where a module's state lives

This is the part of a module that behaves differently from the way the same code would behave in an ordinary program, and the checked example at the end of the chapter is about nothing else.

A module is loaded into a process and never unloaded: the handle from the `dlopen` is not kept, so there is no close and no finalisation entry point. Its object, the one its members were installed into, belongs to the VM, and it is remembered in that VM's `modules` table by the loader's own caller — so a second `require()` in the same VM gives the same object, a `delete modules["ch48"]` gives a fresh load with the init run again on the next one, and the two are not the same thing:

```
$ ucode -L './*.so' t2.uc
the same object again: true
after dropping the cache entry: false
```

Below that, though, the module is one copy of one file in one process. Its own C variables are not per VM, and neither is the state of a library it calls. A host with two virtual machines in it — which chapter 42's state notes make a normal thing to want — therefore has two of everything the module installed and one of everything the module kept:

What	How many
The module's object, and its members	one per VM
The VM registry entries it sets	one per VM
Its resource types	one per VM
Its own C variables, and any library's	one per process

That has one consequence that bites before any of the others: a module cannot use a C variable to remember whether it has been initialised, because the second VM's load finds the flag set by the first and skips work that VM needs. `uc_module_init` runs once per load and does not know how many loads a process will see, so per-VM state belongs on the VM — in the registry, as `lib/math.c` does — and anything left in a module's data segment is shared by every script in the process whether it was meant to be or not.

Reaching it from a host

An embedder that wants a module rather than a script's `require()` does what the loader does, in the same order, and the three steps are the whole of it: an object, the entry point, and then calls. The checked example below compiles the module into the same file, standing in for a `dlopen` of the entry point:

```c
#include <stdio.h>
#include <dlfcn.h>
#include <ucode/ucode.h>

static uc_parse_config_t config = { .raw_mode = true };

int main(void)
{
        uc_vm_t vm = { 0 };
        uc_value_t *scope, *rv = NULL;
        void (*entry)(uc_vm_t *, uc_value_t *);
        void *handle;
        uc_exception_type_t exc;

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));

        handle = dlopen("/tmp/ucode-ch48.so", RTLD_LAZY | RTLD_LOCAL);
        printf("dlopen of the module: %s\n", handle ? "worked" : "failed");

        entry = (void (*)(uc_vm_t *, uc_value_t *))dlsym(handle,
"uc_module_entry");
        printf("the entry point: %s\n", entry ? "found" : "not found");

        scope = ucv_object_new(&vm);
        entry(&vm, scope);

        /* a plain call: the callee, then the arguments, and the callee was
borrowed
           out of the object so the push needs a reference of its own */
        uc_vm_stack_push(&vm, ucv_get(ucv_object_get(scope, "greet", NULL)));
        uc_vm_stack_push(&vm, ucv_string_new("from the host"));

        exc = uc_vm_call(&vm, false, 1);
        rv = uc_vm_stack_pop(&vm);

        printf("calling greet through the object: ");

        if (exc == EXCEPTION_NONE)
                printf("%s", ucv_to_string(&vm, rv));
        else
                printf("no value");

        printf("\n");

        ucv_put(rv);
```

```
        ucv_put(scope);
        uc_vm_free(&vm);

        return 0;
}
```

```
dlopen of the module: worked
the entry point: found
calling greet through the object: hello, from the host
```

The `.so` the block opens is built by the commands above rather than by the block itself, so it is worth running after them. Note the `ucv_get` on the line that pushes the callee: `uc_vm_stack_push` takes a reference with the value it is given, and `ucv_object_get` returns a borrowed one, so pushing it as it stands leaves the object's member one reference lighter than it should be. Nothing says so at the time. The member keeps working for a while, then the script's next call through it reports that the left-hand side is not a function, and the run after that is the one that corrupts the heap.

The whole thing, checked

One file, with the module and the host in it, and the state story told from both sides:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <ucode/ucode.h>
#include <ucode/module.h>


/* A module-level variable, living in the shared object's data segment. */
static int call_count;

static uc_value_t *
counter_tick(uc_vm_t *vm, size_t nargs)
{
        (void)vm;
        (void)nargs;

        call_count++;

        return ucv_int64_new(call_count);
}

/* A per-VM marker, kept in the VM's registry. */
static uc_value_t *
state_read(uc_vm_t *vm, size_t nargs)
{
        (void)nargs;

        return uc_vm_registry_get(vm, "demo.seen");
}

static uc_value_t *
state_write(uc_vm_t *vm, size_t nargs)
{
        uc_value_t *val = uc_fn_arg(0);

        uc_vm_registry_set(vm, "demo.seen", ucv_get(val));

        return NULL;
}

static const uc_function_list_t demo_fns[] = {
        { "tick",               counter_tick },
        { "seen",               state_read },
        { "remember",        state_write },
};
```

```
void uc_module_init(uc_vm_t *vm, uc_value_t *scope);

void
uc_module_init(uc_vm_t *vm, uc_value_t *scope)
{
        uc_function_list_register(scope, demo_fns);
}

static uc_parse_config_t config = { .raw_mode = true };

static uc_value_t *
call0(uc_vm_t *vm, uc_value_t *scope, const char *name)
{
        uc_value_t *rv = NULL;

        uc_vm_stack_push(vm, ucv_get(ucv_object_get(scope, name, NULL)));

        if (uc_vm_call(vm, false, 0) == EXCEPTION_NONE)
                rv = uc_vm_stack_pop(vm);

        return rv;
}

static uc_value_t *
call1(uc_vm_t *vm, uc_value_t *scope, const char *name, uc_value_t *arg)
{
        uc_value_t *rv = NULL;

        uc_vm_stack_push(vm, ucv_get(ucv_object_get(scope, name, NULL)));
        uc_vm_stack_push(vm, ucv_get(arg));

        if (uc_vm_call(vm, false, 1) == EXCEPTION_NONE)
                rv = uc_vm_stack_pop(vm);

        return rv;
}

int main(void)
{
        uc_value_t *scope1, *scope2, *rv;

        setvbuf(stdout, NULL, _IOLBF, 0);

        /* the first VM and its module object */
        uc_vm_t first = { 0 };
```

```
        uc_vm_init(&first, &config);
        uc_stdlib_load(uc_vm_scope_get(&first));
        scope1 = ucv_object_new(&first);
        uc_module_entry(&first, scope1);

        rv = call0(&first, scope1, "tick");
        printf("first VM, first tick: %s\n", ucv_to_string(&first, rv));
        ucv_put(rv);

        /* a second VM in the same process, with its own module object */
        uc_vm_t second = { 0 };

        uc_vm_init(&second, &config);
        uc_stdlib_load(uc_vm_scope_get(&second));
        scope2 = ucv_object_new(&second);
        uc_module_entry(&second, scope2);

        rv = call0(&second, scope2, "tick");
        printf("second VM, its first tick: %s\n", ucv_to_string(&second,
rv));
        ucv_put(rv);

        rv = call0(&first, scope1, "tick");
        printf("first VM, again: %s\n", ucv_to_string(&first, rv));
        ucv_put(rv);

        /* the registry half is per VM */
        call1(&first, scope1, "remember", ucv_string_new("in the first"));
        call1(&second, scope2, "remember", ucv_string_new("in the second"));

        rv = call0(&first, scope1, "seen");
        printf("what the first VM's registry holds: %s\n",
ucv_to_string(&first, rv));
        ucv_put(rv);

        rv = call0(&second, scope2, "seen");
        printf("what the second VM's registry holds: %s\n",
ucv_to_string(&second, rv));
        ucv_put(rv);

        ucv_put(scope1);
        ucv_put(scope2);
        uc_vm_free(&first);
        uc_vm_free(&second);

        return 0;
}
```

```
first VM, first tick: 1
second VM, its first tick: 2
first VM, again: 3
what the first VM's registry holds: in the first
what the second VM's registry holds: in the second
```

`tick` counts on one integer that both VMs share, while each VM's registry entry is the one that its own VM reads back. Put a `print` of a second `tick` call in the first VM and the count goes to four with nobody else having asked for it; that is the shape of the bug this table is here to prevent.

Summary

- A module is a shared object plus `void uc_module_init(uc_vm_t *vm, uc_value_t *scope)`; the loader looks for `uc_module_entry`, which is defined in `<ucode/module.h>` and so must appear in one translation unit of the module.
- The scope argument is a fresh object. Its members are what the script sees; the VM is where types and registry entries go.
- Build a module with `-fPIC -shared` and no `-lucode`: undefined references into the library bind at load, and the set of names available is the set the library exports. The build rejects a reference to a hidden internal entry on this platform.
- Tables passed to `uc_function_list_register` and `uc_type_declare` are length-delimited; a null row is an entry with a null name, not a terminator.
- Nothing version-checks a module. The signature of the entry point is the ABI, and modules belong to the build of the interpreter they are shipped with.
- The object is per VM and cached in `modules`; the module's own C data is per process. Per-VM state goes on the VM.
- A host reaches a module with `dlopen`, one named symbol, an object, and the call protocol of chapter 44, and `uc_vm_stack_push` needs a reference on every push.

The six example programs

Source files referenced in this chapter: `examples/execute-string.c`, `examples/execute-file.c`, `examples/native-function.c`, `examples/exception-handler.c`, `examples/state-reuse.c`, `examples/state-reset.c`, `examples/CMakeLists.txt`, `main.c`.

The tree carries six small host programs under `examples/`, and they are the shortest paths that exercise the parts of the library an embedder really uses. Between them they answer five questions — how to run a string, how to run a file, how to give scripts a C function, how to see a failure, and what happens to the variables of a program that runs more than once — and two more that are about what not to do. All six are under one generic build rule:

```
FILE(GLOB examples "*.c")
FOREACH(example ${examples})
  GET_FILENAME_COMPONENT(example ${example} NAME_WE)
  SET(CLI_SOURCES main.c)
  ADD_EXECUTABLE(${example} ${example}.c)
  TARGET_LINK_LIBRARIES(${example} libucode ${json})
ENDFOREACH(example)
```

Every `.c` file in the directory becomes a program linked against `libucode` and json-c, which is why none of them has a build file of its own, and why `examples/execute-string.c` can carry its recipe in a comment:

```
/* Build with  gcc -o execute-string -lucode execute-string.c */
```

That comment describes out-of-tree builds correctly. Building all six outside the tree against nothing but `-lucode` succeeds, which confirms two things: every name any example uses is visible to an ordinary link — none carries hidden linkage attributes making it unavailable across shared objects — and json-c does not have to be mentioned even though several of them call straight through into json-c code via `ucv_to_jsonstring_formatted()`, which resolves into a routine living inside `libucode` proper instead of needing the underlying json-c object file to sit on its own command line.

```
execute-string     links with -lucode only
execute-file       links with -lucode only
native-function    links with -lucode only
exception-handler  links with -lucode only
state-reuse        links with -lucode only
state-reset        links with -lucode only
```

What is behind it is recorded in the shared object itself. `readelf` lists the dependencies recorded in the library, and those entries name the versioned shared objects directly:

```
$ cc -std=gnu11 examples/exception-handler.c -o exception-handler -L build -
lucode \
      -Wl,-rpath,"$PWD"/build
$ readelf --dynamic build/libucode.so.0 | grep NEEDED
 0x0000000000000001 (NEEDED)             Shared library: [libjson-c.so.5]
 0x0000000000000001 (NEEDED)             Shared library: [libm.so.6]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x0000000000000001 (NEEDED)             Shared library: [ld-linux-x86-64.so.
2]
$ ldd exception-handler | awk '{print $1}' | sort | head -5
/lib64/ld-linux-x86-64.so.2
libc.so.6
libjson-c.so.5
libm.so.6
libucode.so.0
```

`ldd` resolves transitive requirements and prints what finally populates the address space; no separate mention of json-c or math is required. A system whose installed layout omits versioned `.so.N` files may need different specification (chapter 40), but these programs need nothing extra. If a host needs to name json-c explicitly anyway, chapter 40 explains how requesting that behaves.

Three decisions are common to all six, and each one of them is a choice a real host has to make as well:

```
static uc_parse_config_t config = {
        .strict_declarations = false,
        .lstrip_blocks = true,
        .trim_blocks = true
};
```

- The configuration is one static object used by every call. It holds no state between compilations, so one per program is enough.
- `strict_declarations` is off. Some example assigns to a name it has not declared, and with the flag on, the compile would have refused it.
- `raw_mode` is left at false, which is the part most likely to cost an afternoon. With a zeroed field the parser compiles **templates** (chapter 16), not scripts. Every program below is therefore wrapped in `{% ... %}`, which makes it template text whose only statement block happens to be the whole program. The alternative is `.raw_mode = true` and plain script source; you will want that whenever the host is a runner rather than a page generator, and the later section headed What the two modes do to a source measures the consequences directly.

The skeleton all six follow is the reference order of chapter 40, minus the module path when no module is loaded: create the source, compile it, release the source, check the program, initialise the VM, load the standard library, add anything of the host's own to the scope, run, branch on the status, release the program, release the VM.

Running a string: `execute-string`

The interesting part is the source text itself. A C string literal holding a program is miserable to read, so the example defines a stringify helper and writes the program as it would appear in a file:

```
#define MULTILINE_STRING(...) #__VA_ARGS__

static const char *program_code = MULTILINE_STRING(
        {%
                function add(a, b) {
                        c = a + b;

                        return c;
                }

                result = add(x, y);

                printf('%d + %d is %d\n', x, y, result);

                return result;
        %}
);
```

Two properties of that macro are worth carrying into your own host. Because arguments keep their commas, the wrapper is variadic (`...`) and the stringification is of `__VA_ARGS__` ; and because a stringifier also keeps whitespace, the program survives into the binary with its layout intact. What does not survive is any quote style the language needs but the C literal eats: comments written the C way inside such a wrapper would end the macro, which is one more reason the examples' embedded programs are brief.

From there the flow is the ordinary one. Note the sequence around the compile, since it is where ownership changes hands:

```
        /* create a source buffer containing the program code */
        uc_source_t *src = uc_source_new_buffer("my program",
strdup(program_code), strlen(program_code));

        /* compile source buffer into function */
        char *syntax_error = NULL;
        uc_program_t *program = uc_compile(&config, src, &syntax_error);

        /* release source buffer */
        uc_source_put(src);
```

The buffer is given away by copying it into the parse and released right after the compile, before the program is even tested; the syntax error message is a malloc'd string owned by whoever asks for it, hence the `free()` on the failure path. The name handed to `uc_source_new_buffer()`, here `"my program"`, is what any later report will print instead of a filename. It is also the `filename` value in the frames of any stack trace the program raises, as the failing run under *Watching a failure* shows.

Two values are then put into the scope before the run, which is the standard way to pass data into a program with no arguments of its own:

```
        ucv_object_add(uc_vm_scope_get(&vm), "x", ucv_int64_new(123));
        ucv_object_add(uc_vm_scope_get(&vm), "y", ucv_int64_new(456));
```

and the run's outcome is dispatched across the whole set of statuses, with the exit code extracted from the returned value:

```
        case STATUS_EXIT:
                exit_code = (int)ucv_int64_get(last_expression_result);
```

That is one of the two places in the six where the rule that "`STATUS_EXIT` gives the exit code back in the value slot" appears in working code. The way `case ERROR_COMPILE:` sets 1 while `ERROR_RUNTIME` sets 2 shows the other half of the same care: the two failures are distinct conditions a supervisor may treat differently. Running it gives the arithmetic and the return value:

```
$ ./build/examples/execute-string
123 + 456 is 579
Program finished successfully.
Function return value is 579
```

Running a file: `execute-file`

Eleven lines of difference separate this program from the last one, and those eleven are all the machinery a runner needs:

```
        if (argc != 2) {
                fprintf(stderr, "Usage: %s sourcefile.uc\n", argv[0]);

                return 1;
        }

        /* create a source buffer from the given input file */
        uc_source_t *src = uc_source_new_file(argv[1]);

        /* check if source file could be opened */
        if (!src) {
                fprintf(stderr, "Unable to open source file %s\n", argv[1]);

                return 1;
        }
```

`uc_source_new_file()` mmaps or reads the file and returns nothing when the path cannot be opened — the one failure mode the program tests for, and it reports it before touching a VM. After that point the program is word for word `execute-string` : the same configuration, the same `"x"` and `"y"` , the same switch over the four statuses. Which means the three cases beyond `STATUS_OK` can be seen in one program instead of six, and a missing path is enough to bring out the first of them:

```
$ ./build/examples/execute-file /tmp/no-such-file.uc
Unable to open source file /tmp/no-such-file.uc
$ echo $?
1
```

Feeding it a real file shows the wiring through to the value slot, using a source written with the configuration the program actually compiled under:

```
$ cat /tmp/handler-demo.uc
print("file says: ", x + y, "\n");
return x * y;
$ ./build/examples/execute-file /tmp/handler-demo.uc
print("file says: ", x + y, "\n");
return x * y;
Program finished successfully.
Function return value is null
```

The first line is the puzzle this section exists for. Under the default configuration the file's contents are template text, so the whole program is printed verbatim as text to emit and none of it is executed; the return value is absent for exactly that reason. With `raw_mode` set, the same binary treats the same file as a program:

```
$ ./rawmode /tmp/handler-demo.uc
file says: 579
Program finished successfully.
Function return value is 56088
```

The one change made to produce that second binary was inserting `.raw_mode = true,` into the configuration initialiser; the section headed *What the two modes do to a source* measures the same contrast directly instead of relying on the shipped binaries.

Giving scripts a C function: `native-function`

Here the program's reason for existing is the two native functions, and they are the two shapes nearly every native binding turns out to be — one fixed-arity and one variadic:

```
static uc_value_t *
multiply_two_numbers(uc_vm_t *vm, size_t nargs)
{
        uc_value_t *x = uc_fn_arg(0);
        uc_value_t *y = uc_fn_arg(1);

        return ucv_double_new(ucv_to_double(x) * ucv_to_double(y));
}

static uc_value_t *
add_all_numbers(uc_vm_t *vm, size_t nargs)
{
        double res = 0.0;

        for (size_t n = 0; n < nargs; n++)
                res += ucv_to_double(uc_fn_arg(n));

        return ucv_double_new(res);
}
```

`uc_fn_arg(n)` reaches into the current frame's argument slots and `nargs` is how many the call passed, so the loop is the entire variadic protocol (chapter 44). Notice what the bodies never do: they never convert to an integer first, because `ucv_to_double()` answers 0 for anything that is not numeric, which suits arithmetic built to tolerate loose input; and they never return NULL, so the caller always gets a number object back. Registration is one line each into the global scope:

```
        uc_function_register(uc_vm_scope_get(&vm), "add", add_all_numbers);
        uc_function_register(uc_vm_scope_get(&vm), "multiply",
multiply_two_numbers);
```

and the embedded program calls them the way it would call `sqrt()`:

```
$ ./build/examples/native-function
add() = 10.1
multiply() = 36.5
```

The one thing worth testing yourself is the arity mismatch: the loop reads past the supplied count when the caller sends fewer values than the fixed-arity function indexes, giving whatever sits in the slot already, which is why the argument-count helpers of chapter 44 exist for bindings that must refuse.

This example is also the cheapest illustration of how little output has to be handled: the script's two `print` calls go to `vm.output`, which `uc_vm_init()` sets to `stdout`, and the host prints nothing of its own. Nothing prevents the reverse, either, and the section headed *Collecting what a run produced* describes what changes once a host takes that stream over.

Watching a failure: `exception-handler`

The point of this example is the third argument to `printf`, which is why its handler serialises rather than formats by hand:

```
static void
log_exception(uc_vm_t *vm, uc_exception_t *ex)
{
        char *trace = ucv_to_jsonstring_formatted(vm, ex->stacktrace, ' ',
2);

        printf("Program raised an exception:\n");
        printf("  type=%d\n", ex->type);
        printf("  message=%s\n", ex->message);
        printf("  stacktrace=%s\n", trace);

        free(trace);
}
```

`ex->stacktrace` is an ordinary array of objects; converting it with `ucv_to_jsonstring_formatted()` turns frames into data a log shipper understands, and the freed pointer is the json-c allocation. The implementation is the single setter:

```
        /* register custom exception handler */
        uc_vm_exception_handler_set(&vm, log_exception);
```

Since a VM arrives with `uc_vm_output_exception` installed, this replaces the human-oriented report with the machine-readable one; a server that wants both reads the old handler with `uc_vm_exception_handler_get()` and calls it too. What the replacement receives for a failure reached through a higher-order function is the whole value of the example, reproduced in full:

```
$ ./build/examples/exception-handler
Program raised an exception:
  type=3
  message=left-hand side is not a function
  stacktrace=[
  {
    "filename": "my program",
    "line": 1,
    "byte": 35,
    "function": "fail",
    "tco": 1,
    "context": "In fail(), file my program, line 1, byte 35:\n  (1 tail call
frames omitted)\n  called from function map ([C])\n  called from anonymous
function (my program:1:61)\n\n `{% function fail() { doesnotexist(); }
map([1], x => fail(x)); %}`\n  Near here ----------------------^\n"
  },
  {
    "function": "map"
  },
  {
    "filename": "my program",
    "line": 1,
    "byte": 61
  }
]
An error occurred while running the program
$ echo $?
1
```

Four things to read out of that transcript. `type=3` is `EXCEPTION_REFERENCE`, spelled out immediately below under *Type is a number first*, and the record also carries the printable wording, so the number is the stable thing to match on. The frames come innermost first. `map` contributes a bare `{"function":"map"}` entry because a native function has no source position; a `[C]` frame is told apart from a script frame by having no `filename`. And `context` holds the same text the default handler would have printed, carriage-return-newline encoded inside a json string, which means the machine-readable form loses nothing a human reader needed.

Type is a number first

`ex->type` is a `uc_exception_type_t`; the enumeration runs from `EXCEPTION_NONE` through `EXCEPTION_USER`, printed here as a bare decimal. Nothing in the public headers gives the words the reports use for those numbers except `exception_type_strings[]` (in `internal/vm.h`) — a second reason to serialize the record rather than format numbers into a log line nobody can interpret.

The report is assembled elsewhere

Look again at what the example prints, and at what it does not. There is no `filename`, no line, and no caret drawing in its own code — the example asks for three fields of the record and everything richer than that was assembled by the machinery it displaced. Concretely: the `context` string above was produced by `uc_traceback_format_head()` (in `vm.c`, reachable through `uc_traceback_from_frame()` and the traceback array's `tostring` metamethod) when the frames were captured, and it is stored per frame. A handler therefore gets a complete report already in hand; assembling positions itself from `struct uc_refframe` data would be reimplementing that.

One VM for every event: `state-reuse`

A daemon runs one program many times — once per packet, ubus message, or timer tick. This example is that loop with five iterations and nothing else:

```
        /* execute compiled program function five times */
        for (int i = 0; i < 5; i++) {
                printf("Iteration %d: ", i + 1);

                /* execute program function */
                int return_code = uc_vm_execute(&vm, program, NULL);

                /* handle return status */
                if (return_code == ERROR_COMPILE || return_code ==
ERROR_RUNTIME) {
                        printf("An error occurred while running the
program\n");
                        exit_code = 1;
                        break;
                }

                /* perform GC step */
                ucv_gc(&vm);
        }
```

Its companion script reads a value out of the global object, doubles it, and stores it back, which is the only way a repeatedly run program can carry anything forward:

```
        static const char *program_code = MULTILINE_STRING(
                {%
                        let n = global.value || 1;

                        print("Current value is " + n + "\n");

                        global.value = n * 2;
                %}
        );
```

Run it and the doubling compounds:

```
$ ./build/examples/state-reuse
Iteration 1: Current value is 1
Iteration 2: Current value is 2
Iteration 3: Current value is 4
Iteration 4: Current value is 8
Iteration 5: Current value is 16
```

One property deserves emphasis because it surprises people who know Lua: `let` has no effect on persistence across two runs. The declaration binds a name in the current scope, and for a chunk-level run that scope is the global object, so `let n` on the next run re-reads the key that `global.value = ...` wrote. Everything a run declares and everything it assigns lives on until something clears it. That is convenient for passing results, and it is also why the next example, Fresh state each time, exists.

The `ucv_gc(&vm)` call in the loop is optional in substance and instructive in placement: collection is reference counted, so values die when the scope forgets them, and this is the cheap incremental sweep on top (chapter 18 and the closer discussion under Collection cycles). Doing it once per event bounds residency for services that churn a lot of strings.

Fresh state each time: `state-reset`

Same loop, same five iterations, opposite construction: the VM is created inside the loop and released at the end of it:

```
        /* initialize default module search path */
        uc_search_path_init(&config.module_search_path);

        /* execute compiled program function five times */
        for (int i = 0; i < 5; i++) {
                /* initialize VM context */
                uc_vm_t vm = { 0 };
                uc_vm_init(&vm, &config);
```

with the matching `uc_vm_free(&vm)` closing the iteration, and with `uc_stdlib_load()` called inside for the same reason. Its script asserts the amnesia it is meant to produce:

```
$ ./build/examples/state-reset
Iteration 1: Global variable is null? true
Iteration 2: Global variable is null? true
Iteration 3: Global variable is null? true
Iteration 4: Global variable is null? true
Iteration 5: Global variable is null? true
```

The economics follow from the split the pair demonstrates: the compilation sits outside the loop and the VM inside it, because compiling a program once for many events is free and initialising a VM is comparatively costly. A VM per request buys a guarantee no careful clearing can really match — the second request cannot observe anything the first one wrote — at the price of rebuilding the scope and the stdlib's registrations each time. Where a deployment needs the guarantee and wants to know what it costs, the section headed How much a VM costs measures both sides; for the middle ground, Resetting without throwing the VM away below sets out the technique the pair points toward.

Neither example releases the last expression value, because neither requests one: they pass NULL, and with NULL there is nothing to drop. Both do test the two error codes together rather than branching across the full switch: a looping host normally restarts the service on the same signal either way, and it breaks out rather than iterating on a program that just failed.

What the two modes do to a source

Under the shared configuration the six examples are all template-mode hosts, which suits programs whose embedded source is delimited visibly, and misleads anyone who then feeds a script file to `execute-file`. The rule is worth pinning down with one program that tries one source under both readings and captures what the script prints instead of mixing it into the host's own lines:

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

#include <ucode/ucode.h>


static const char script[] = "print(\"sum: \", x + y, \"\\n\");\nreturn x +
y;\n";

static void
try(const char *mode, bool raw)
{
        uc_parse_config_t config = {
                .raw_mode = raw,
                .strict_declarations = false,
                .lstrip_blocks = true,
                .trim_blocks = true
        };
        uc_vm_t vm = { 0 };
        uc_source_t *source;
        uc_program_t *program;
        uc_value_t *rv = NULL;
        FILE *captured;
        char *error = NULL;
        long produced;
        char *emitted;

        uc_vm_init(&vm, &config);

        /* Point the stream at a scratch file after uc_vm_init(), which sets
it to stdout. */
        captured = tmpfile();
        vm.output = captured;

        uc_stdlib_load(uc_vm_scope_get(&vm));
        ucv_object_add(uc_vm_scope_get(&vm), "x", ucv_int64_new(123));
        ucv_object_add(uc_vm_scope_get(&vm), "y", ucv_int64_new(456));

        source = uc_source_new_buffer("example", strndup(script,
strlen(script)), strlen(script));
        program = uc_compile(&config, source, &error);
        uc_source_put(source);

        if (program == NULL) {
                printf("%s: compiled=no\n", mode);
```

```
                free(error);
        } else {
                uc_vm_execute(&vm, program, &rv);
                fflush(captured);
                vm.output = stdout;

                produced = ftell(captured);
                emitted = malloc(produced + 1);
                fseek(captured, 0, SEEK_SET);
                fread(emitted, 1, produced, captured);

                printf("%s: returned=%s, emitted=%s\n", mode,
                       rv == NULL ? "null" : ucv_to_string(&vm, rv),
                       produced == (long)strlen(script) && !memcmp(emitted,
script, produced) ?
                        "the source text verbatim" : "something else");

                free(emitted);
                ucv_put(rv);
                uc_program_put(program);
        }

        fclose(captured);
        uc_vm_free(&vm);
}

int main(void)
{
        try("raw-mode, plain script ", true);
        try("template-mode, same   ", false);

        return 0;
}
```

```
raw-mode, plain script : returned=579, emitted=something else
template-mode, same   : returned=null, emitted=the source text verbatim
```

Read the second line against the transcript near the top of *Running a file*: a source that has no tags, compiled as a template, is one long piece of text: it is *emitted*, unchanged, and nothing in it runs, so the returned value is absent. Raw mode is the reading that executes it. Neither mode is wrong, and one flag settles which mode a host uses; what to watch for is the silent case, where template mode accepts your script, compiles cleanly, and simply declines to run it.

The example also enforces two mechanics stated elsewhere: the assignment to `vm.output` goes **after** `uc_vm_init()`, because the initializer unconditionally sets the stream to `stdout`, and the stream is restored before

the host prints again, since script output and host output are otherwise interleaved by buffering rather than by meaning.

Collecting what a run produced

With the stream pointed at a scratch file the host owns, a run's output becomes a string — usable as the body of a template response or as a log record's payload. The mechanics of taking that stream over are in *Collecting, tracing, output* in chapter 42. The summary a host needs there is that the redirect has to happen after `uc_vm_init()`, that the stream must be handed back to `stdout` before normal printing resumes, and that exceptions still travel to standard error, since the default handler writes there and not to the stream, so capturing output does not capture failures. The experiment under *What the two modes do to a source* exercises the same redirection in passing.

Resetting without throwing the VM away

The section headed *Fresh state each time* establishes that a new VM starts blank. A service wanting that same clean start point while keeping its long-lived VM can obtain it out of the scope itself, which is an ordinary object. Three things determine what a reset involves, two of them properties of the standard library's registration scheme:

- Registered functions and predefined constants become flat keys of the global object. This tree has eighty-one of them, counted by a script (`length(keys(global))` answers 81 after `uc_stdlib_load()` with the examples' configuration): a mix of builtin functions such as `print`, module tables like `math`, and entries added when the scope came into existence, like `NaN`, `Infinity` and `global` itself. None arrives lazily and none arrives tagged, so scanning those keys will not tell you which of them are yours.
- Program state likewise arrives as ordinary keys. The write `global.value = ...` that both examples perform inserts one key into this very same table, so there is literally no structural separation between data from the interpreter's own plumbing and data written by application code: the distinction exists only in naming discipline.
- Deleting a key destroys just that binding, without disturbing any sibling. Hence the sound shape of a reset is to delete the specific names the program is authorised to own. Anything the host registers itself, as in *Giving scripts a C function* above, belongs on that inventory too if it is supposed to vanish.

The test below runs the writer three times and deletes exactly `tick` between each round, leaving the whole library surface intact:

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

#include <ucode/ucode.h>


/* Written by the program on every run; the thing a reset has to undo. */
static const char *userkeys[] = { "tick", NULL };

static const char bump[] = "{% global.tick = (global.tick == null) ? 1 :
global.tick + 1; %}";

int main(void)
{
        uc_parse_config_t config = { .raw_mode = false, .strict_declarations
= false };
        uc_vm_t vm = { 0 };
        uc_source_t *source;
        uc_program_t *program;
        uc_value_t *tick;
        size_t i;

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));

        source = uc_source_new_buffer("bump.uc", strdup(bump), strlen(bump));
        program = uc_compile(&config, source, NULL);
        uc_source_put(source);

        for (int round = 1; round <= 3; round++) {
                uc_vm_execute(&vm, program, NULL);

                tick = ucv_object_get(uc_vm_scope_get(&vm), "tick", NULL);

                printf("round %d saw tick=%s\n", round,
                       tick ? ucv_to_string(&vm, tick) : "(absent)");
                if (tick)
                        ucv_put(tick);

                for (i = 0; userkeys[i] != NULL; i++)
                        ucv_object_delete(uc_vm_scope_get(&vm),
userkeys[i]);
        }

        printf("the registered print() survived: %s\n",
```

```
                ucv_object_get(uc_vm_scope_get(&vm), "print", NULL) ? "yes" :
"no");

        uc_program_put(program);
        uc_vm_free(&vm);

        return 0;
}
```

```
round 1 saw tick=1
round 2 saw tick=1
round 3 saw tick=1
the registered print() survived: yes
```

Compare against what `state-reuse` prints across the same three rounds without the deletion loop, `1`, `2`, `4`; deleting precisely an inventory of owned keys buys `state-reset`'s cleanliness while retaining compiled programs, registered natives, modules already demanded into the cache, and everything else kept besides the scope's mutable data (chapter 42 describes what the VM's registry retains between runs). If the host has no authoritative list of its own keys, the scope's contents do not supply one, because standard-library registrations are mixed in; either keep a list elsewhere or use fresh VMs.

How much a VM costs

Measuring this pair is cheap: constructing and destroying a whole VM takes roughly ten microseconds, so freshening one for every incoming message is feasible. The timings below average three hundred iterations, with three consecutive runs of the same binary against a plain, unoptimised build of the same source tree:

```
compiling the program: 2.2 us per compile
one whole VM lifetime: 12.2 us per VM
compiling the program: 2.7 us per compile
one whole VM lifetime: 14.3 us per VM
compiling the program: 1.5 us per compile
one whole VM lifetime: 11.8 us per VM
```

Successive trials stay within a factor of two; cross-machine differences are naturally larger. The proportions matter more than the absolute numbers. Constructing one VM costs about five to eight compiles of this small program. Populating the scope accounts for most of that, because `uc_stdlib_load()` installs eighty-one entries (the resetting section above counts them), each requiring a heap allocation. Even so, ten to fifteen microseconds per VM leaves considerable headroom under realistic loads, so choosing VM-per-request is mainly a question of isolation requirements rather than throughput.

Collection cycles

Only `state-reuse` asks for an incremental collection turn, inside its loop body; its counterpart never emits such a call, since disposing the whole VM discards everything reachable from scope en masse:

```
                /* perform GC step */
                ucv_gc(&vm);
```

The detailed workings are in chapter 18. Reference counting reclaims most values when their last owner releases them, so the explicit `ucv_gc()` call is not what frees ordinary strings. The collector matters mainly for reference cycles, which reference counts cannot break on their own. Calling it once per handled request spreads the work more evenly than collecting only after a batch. A long-running daemon that holds a steady set of values should still collect periodically, because cycles can otherwise retain memory without appearing in ordinary inspections. `state-reset` sidesteps the issue by releasing the whole VM.

Reading the six as a set

Put the six side by side and a few patterns emerge that no single file states:

Example	Lives to show	The line to copy
`execute-string`	the whole lifecycle, exit-code extraction	`exit_code = (int)ucv_int64_get(...)`
`execute-file`	sources from disk, and what a mode flag changes	`uc_source_new_file(argv[1])` with its NULL test
`native-function`	fixed and variadic argument access	`for (size_t n = 0; n < nargs; n++)`
`exception-handler`	records as data rather than as text	`ucv_to_jsonstring_formatted(vm, ex->stacktrace, ' ', 2)`
`state-reuse`	the persistent scope, and stepping collection in the loop	one `uc_vm_init()` outside the loop
`state-reset`	isolation by construction, compile reused across VMs	`uc_vm_init()` and `uc_vm_free()` inside the loop

None of the six touches the pieces of the API that belong to larger daemons: pushing arguments and calling a named function (`uc_vm_push_args()` , `uc_call()`), the resource types of chapter 45, the interrupting machinery of chapter 46, the precompiled program loading of chapter 47, or exporting a C extension as a module at all (chapter 48).

The natural path outward from these six is along those subjects in turn, ending where a daemon that reacts to external events needs all of the pieces at once. That composition is the subject of chapter 50, which threads them into a socket-facing service.

A worked embedding

Source files referenced in this chapter: `examples/execute-string.c`, `examples/native-function.c`, `examples/exception-handler.c`, `include/ucode/lib.h`, `include/ucode/types.h`.

Chapters 40 to 49 examined the embedding interface in detail: compiling sources, values and their representations, VM state, the exception and break machinery, native functions, resource types, and finally what the six shipped example hosts each exercise separately. This chapter assembles those pieces into small, self-contained hosts. Three hosts take a mode argument to demonstrate contrasting policies, and a fourth gives a compact resource illustration checked by the automated example runner. Each host addresses decisions an integrator faces before writing production code:

Question	Listing	What it demonstrates
How should a host respond to the different ways a single execution can end?	Program 1	naming every status code, forwarding requested exit codes, recovering cleanly after an interruption
Should interpreter state survive between successive invocations?	Program 2	persistent versus freshly built versus surgically wiped scopes around the very same script
Whose memory backs objects visible to scripts?	Program 3	structures residing purely in C exposed via typed resources carrying methods, host-raised faults folded into one log line, symmetric construction/destruction counts across normal, failing and cancelled paths

Build any listed program using the standard pattern against the local tree build directory:

```
$ cc -std=gnu11 -Wall -I include /tmp/name-of-file.c -o name \
      -L build -lucode -Wl,-rpath,"$PWD"/build
```

Three of the listings take command-line arguments, since an argument is what selects the behaviour being shown; each is noted with the arguments that bring it to life, and the outputs below are what those runs print. The fourth takes none and runs as it stands.

Before reading the individual implementations, consider the recurring shape in all four samples, which matches the order introduced in chapter 40:

```
configure the parser
compile the source into a program
create the VM and install the standard library
install host functions, types, or an exception handler
execute the program and inspect its status and value
release the program and the VM
```

The configuration matters through compilation and VM setup. The later steps are the same whether the host keeps one VM and runs many programs on it or creates a fresh VM for each request.

Source text modes revisited under practical pressure

Every shipped example leaves `raw_mode` unset, so the parser treats its input as template text: statements run only inside `{% ... %}`, while text outside the braces is emitted. For a host that only runs scripts written in ucode, `raw_mode = true` is simpler and makes the embedded source look like an ordinary script file. The choice matters most when the source arrives from another program, such as a JSON field containing a user-authored snippet. A host that means raw scripts but leaves the flag unset normally does not get a parse error; it gets the script emitted as text. The three hosts here set `.raw_mode = true` explicitly rather than accidentally inheriting template behaviour.

Classification vocabulary used repeatedly ahead

Each sample turns status and exception constants into words before printing them. A transcript that names the codes is easier to compare than one carrying bare integers, so the listings use the following classification consistently:

Constant	Meaning	Usual host response
`STATUS_OK`	the run returned a value	use the returned value
`STATUS_EXIT`	the program called `exit()`	propagate the requested exit code
`STATUS_BREAK`	a break request stopped the run	drain residual stack values, then continue
`ERROR_COMPILE`	compilation failed	report the compiler diagnostics
`ERROR_RUNTIME`	the run raised an exception	inspect the exception record

The mapping is an explicit `switch` in the first listing. There are only five statuses, so testing each one is straightforward and avoids confusing a break with an error.

Program 1: classifying terminations, including recoverable ones

Calling `uc_vm_execute` can return one of five statuses: `STATUS_OK`, `STATUS_EXIT`, `STATUS_BREAK`, `ERROR_COMPILE`, and `ERROR_RUNTIME`. `STATUS_OK` carries the value the program returned; `STATUS_EXIT` carries the argument given to `exit()`; `STATUS_BREAK` means a break request stopped the run, possibly with values left on the operand stack; `ERROR_COMPILE` means the source never became a program; and `ERROR_RUNTIME` means an exception stopped it. The first listing names each status explicitly and uses a small helper to turn it into text.

The interruption is triggered by a native function rather than by a signal, so the example is deterministic and portable. The script starts some work and then calls `stophere()`, whose C implementation calls `uc_vm_break_request()`. The VM notices the request between instructions and returns `STATUS_BREAK`. The partially evaluated expression leaves its values on the operand stack, so the host drains them before reusing the VM. After draining, the listing runs a new program on the same VM to show that recovery works cleanly. The same pattern is useful for event-loop timeouts and administrative cancellation.

The listing takes the name of the source to compile as its argument, one of `compute`, `crash`, `exit` or `interrupt`, and defaults to `compute`; the runs below are one per name.

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

#include <ucode/ucode.h>

/* A run reports which of the five ways it ended by returning one status
code. */
static const char *
statusname(uc_vm_status_t status)
{
        switch (status) {
        case STATUS_OK:
                return "STATUS_OK";

        case STATUS_EXIT:
                return "STATUS_EXIT";

        case STATUS_BREAK:
                return "STATUS_BREAK";

        case ERROR_COMPILE:
                return "ERROR_COMPILE";

        case ERROR_RUNTIME:
                return "ERROR_RUNTIME";
        }

        return "unknown";
}

/* Four programs exercising the four interesting outcomes of a run. */
static const char *sources[] = {
        "return length(\"hello\");",
        "let f = 1;\nf();",
        "exit(7);",
        "let work = \"a\" + \"b\";\nstophere();\nreturn work + \"!\";",
};

static const char *labels[] = { "compute", "crash", "exit", "interrupt" };

#define NSOURCES (sizeof(sources) / sizeof(sources[0]))


static uc_value_t *
stophere(uc_vm_t *vm, size_t nargs)
```

```
{
        (void)nargs;

        uc_vm_break_request(vm);

        return NULL;
}

/* Turn one source text into a program, or return NULL with the parse error
in *error. */
static uc_program_t *
compile(uc_parse_config_t *config, const char *name, const char *code, char
**error)
{
        uc_source_t *source = uc_source_new_buffer(name, strdup(code),
strlen(code));
        uc_program_t *program;

        if (source == NULL) {
                *error = NULL;

                return NULL;
        }

        program = uc_compile(config, source, error);

        uc_source_put(source);

        return program;
}

static void
drain(uc_vm_t *vm)
{
        while (vm->stack.count > 0)
                ucv_put(uc_vm_stack_pop(vm));
}

int main(int argc, char **argv)
{
        /* Interleaves correctly with script-reported failures on standard
error. */
        setvbuf(stdout, NULL, _IOLBF, 0);

        /* raw_mode: this host evaluates scripts rather than template
documents. */
        uc_parse_config_t config = { .raw_mode = true };
```

```
        uc_program_t *program;
        uc_value_t *retval = NULL;
        uc_vm_status_t status;
        char *error = NULL;
        int rc = 1;
        const char *which = argc > 1 ? argv[1] : "compute";
        size_t i;

        for (i = 0; i < NSOURCES; i++)
                if (strcmp(which, labels[i]) == 0)
                        break;

        if (i == NSOURCES) {
                fprintf(stderr, "Usage: %s [%s|%s|%s|%s]\n", argv[0],
labels[0], labels[1], labels[2], labels[3]);

                return 2;
        }

        program = compile(&config, labels[i], sources[i], &error);

        if (program == NULL) {
                printf("%-10s compiled=no\n", which);
                free(error);

                return 1;
        }

        uc_vm_t vm = { 0 };

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));
        uc_function_register(uc_vm_scope_get(&vm), "stophere", stophere);

        status = uc_vm_execute(&vm, program, &retval);

        printf("%-10s status=%s", which, statusname(status));

        if (status == STATUS_OK || status == STATUS_EXIT)
                printf(" value=%s", retval ? ucv_to_string(&vm, retval) :
"(none)");

        if (status == STATUS_BREAK) {
                size_t residue = vm.stack.count;

                printf(" residue=%zu", residue);
```

```
                drain(&vm);
        }

        printf("\n");

        if (status == STATUS_BREAK) {
                /* The break stopped the first program mid-expression; a
fresh run on the
                   same VM completes normally once the residue is gone. */
                uc_program_t *again = compile(&config, "after-break",
"return 6 * 7;", &error);
                uc_value_t *second = NULL;
                uc_vm_status_t second_status;

                second_status = uc_vm_execute(&vm, again, &second);

                printf("recovered  status=%s value=%s\n",
statusname(second_status),
                       second ? ucv_to_string(&vm, second) : "(none)");

                ucv_put(second);
                uc_program_put(again);
        }

        /* exit() reports its argument back to the process, like the
interpreter does. */
        if (status == STATUS_OK)
                rc = 0;
        else if (status == STATUS_EXIT)
                rc = retval ? (int)ucv_int64_get(retval) : 1;

        ucv_put(retval);
        uc_program_put(program);
        uc_vm_free(&vm);

        return rc;
}
```

Program binaries are named arbitrarily below (`prog` , `state` , `service`); pick whatever filenames suit the build. The transcripts interleave script-reported failures on standard error with host output, so every invocation pipes both streams together.

```
$ ./prog compute
compute    status=STATUS_OK value=5
$ echo $?
0
$ ./prog crash
Type error: left-hand side is not a function
In crash, line 2, byte 3:

 `f();`
    ^-- Near here


crash      status=ERROR_RUNTIME
$ echo $?
1
$ ./prog exit
exit       status=STATUS_EXIT value=7
$ echo $?
7
$ ./prog interrupt
interrupt  status=STATUS_BREAK residue=3
recovered  status=STATUS_OK value=42
$ echo $?
1
```

The example chooses three process policies. Success exits `0`; `exit()` forwards the number requested by the script; a break exits `1` without encoding the reason in the status. Recording the residue count before draining makes the interruption visible in the transcript. Leaving residue behind is tolerable when the VM is freed immediately, but draining keeps the diagnostics consistent and helps when a VM may be reused.

The default exception handler prints a rich report with the offending line and position. Program 3 later replaces it with a one-line formatter, illustrating the choice between detailed developer diagnostics and compact entries for a log pipeline.

Program 2: choosing lifetime granularity for interpreter state

An application that runs one program repeatedly has to decide how long interpreter state lives. A batch job may want a counter carried from one round to the next; a service handling independent requests may need the opposite. This example supplies three modes and compiles the program once, so the state policy rather than the source or parse mode is what changes:

- `reuse` keeps one VM for the whole sequence, so data written in one round is visible in the next. It matches the shipped `state-reuse` example.

- `fresh` initialises and releases a VM for each round. It mirrors the isolation strategy of `state-reset`.
- `wipe` reuses the VM but deletes keys that the host knows belong to the script, giving much of the visible effect of a fresh VM without paying to rebuild the whole VM.

The wipe loop deletes scope keys only; it does not clear the registry. That distinction matters because the registry is intended for longer-lived host state (chapter 42). Treating the scope and registry as one namespace can either leak application values across runs or discard cached host state that was meant to survive. The loop also calls `ucv_gc()` once per round. Reference counting already reclaims ordinary values when their last owner drops them, but periodic stepping bounds the memory held by reference cycles.

The listing takes one of `reuse`, `fresh` or `wipe` as its argument and defaults to `reuse`; the runs below are one per mode, with a name from outside the set to show what the usage line looks like.

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

#include <ucode/ucode.h>


/* One program, five rounds; the rounds' data either accumulates or does not.
*/
static const char *script = "global.seen = (global.seen == null) ? 1 :
global.seen + 1;\n"
                            "return global.seen;\n";

/* Keys this host considers its own, i.e. fair game for a wipe between runs.
*/
static const char *owned[] = { "seen" };

#define NOWNED (sizeof(owned) / sizeof(owned[0]))


static void
cleanup(uc_value_t *scope)
{
        size_t i;

        for (i = 0; i < NOWNED; i++)
                ucv_object_delete(scope, owned[i]);
}

int main(int argc, char **argv)
{
        uc_parse_config_t config = { .raw_mode = true };
        const char *mode = argc > 1 ? argv[1] : "reuse";
        uc_source_t *source;
        uc_program_t *program;
        int round;

        setvbuf(stdout, NULL, _IOLBF, 0);

        if (strcmp(mode, "reuse") != 0 && strcmp(mode, "fresh") != 0 &&
strcmp(mode, "wipe") != 0) {
                fprintf(stderr, "Usage: %s [reuse|fresh|wipe]\n", argv[0]);

                return 2;
        }
```

```
        source = uc_source_new_buffer("counter.uc", strdup(script),
strlen(script));
        program = uc_compile(&config, source, NULL);
        uc_source_put(source);

        if (program == NULL) {
                fprintf(stderr, "the shipped script must compile\n");

                return 1;
        }

        if (strcmp(mode, "fresh") == 0) {
                for (round = 1; round <= 5; round++) {
                        uc_vm_t vm = { 0 };
                        uc_value_t *val = NULL;

                        /* A VM exists only for the duration of one event.
*/
                        uc_vm_init(&vm, &config);
                        uc_stdlib_load(uc_vm_scope_get(&vm));

                        uc_vm_execute(&vm, program, &val);

                        printf("fresh round %d value=%s\n", round, val ?
ucv_to_string(&vm, val) : "(none)");

                        ucv_put(val);
                        uc_vm_free(&vm);
                }
        } else {
                uc_vm_t vm = { 0 };

                /* One VM for the whole batch. */
                uc_vm_init(&vm, &config);
                uc_stdlib_load(uc_vm_scope_get(&vm));

                for (round = 1; round <= 5; round++) {
                        uc_value_t *val = NULL;

                        if (strcmp(mode, "wipe") == 0)
                                cleanup(uc_vm_scope_get(&vm));

                        uc_vm_execute(&vm, program, &val);

                        printf("%s round %d value=%s\n", mode, round, val ?
ucv_to_string(&vm, val) : "(none)");
```

```
                        ucv_put(val);

                        /* Reference counting reclaims immediately; this
pass picks up cycles. */
                        ucv_gc(&vm);
                }

                uc_vm_free(&vm);
        }

        uc_program_put(program);

        return 0;
}
```

The transcripts make the behavioural difference easy to read: `reuse` accumulates across rounds, while both reset modes return the same value every time:

```
$ ./state reuse
reuse round 1 value=1
reuse round 2 value=2
reuse round 3 value=3
reuse round 4 value=4
reuse round 5 value=5
$ echo $?
0
$ ./state fresh
fresh round 1 value=1
fresh round 2 value=1
fresh round 3 value=1
fresh round 4 value=1
fresh round 5 value=1
$ echo $?
0
$ ./state wipe
wipe round 1 value=1
wipe round 2 value=1
wipe round 3 value=1
wipe round 4 value=1
wipe round 5 value=1
$ echo $?
0
$ ./state nonsense
Usage: ./state [reuse|fresh|wipe]
$ echo $?
2
```

The choice is driven by isolation requirements rather than assumed performance differences. Chapter 49 measures whole-VM construction and finds it inexpensive enough for a fresh VM per event on typical hosts. Choose the strategy that expresses the required lifetime. `wipe` puts the maintenance burden on the host: every script-owned name must be on the delete list, and missing one can produce state that appears at the start of the next run.

Program 3: handing C-owned objects to scripts safely

Resource types are the clearest way to expose C-owned data to scripts without exposing pointers. The example keeps a `counter_t` in C memory, gives the type a constructor and methods, and counts constructions and destructions so the transcript can check cleanup. Before declaring the type, the host looks for an existing one. Declaring the same name again would leave the new prototype unused and can make the script see the older type instead.

```
        countertype = ucv_resource_type_lookup(&vm, "Counter");

        if (countertype == NULL)
                countertype = uc_type_declare(&vm, "Counter",
counter_methods, counter_free);
```

`uc_type_declare()` builds the type's function table and installs its prototype members. Methods retrieve their receiver with `uc_fn_this("Counter")`, which checks that the value really is a resource of the named type before giving the host the C data slot:

```
static uc_value_t *
counter_bump(uc_vm_t *vm, size_t nargs)
{
        void **slot = uc_fn_this("Counter");
        counter_t *self;
        uc_value_t *amount;

        if (nargs < 1) {
                uc_vm_raise_exception(vm, EXCEPTION_TYPE, "bump() needs a
number");

                return NULL;
        }

        amount = uc_fn_arg(0);
        self = (counter_t *)(*slot);
        self->qty += ucv_to_double(amount);

        return ucv_double_new(self->qty);
}
```

One compile-time pitfall deserves its own warning because the diagnostic does not mention strings. Numeric argument conversions can usually be nested, but `ucv_string_get()` is a macro that takes the address of its argument. Writing `ucv_string_get(uc_fn_arg(1))` therefore takes the address of a temporary and produces `error: lvalue required as unary '&' operand`. Assign `uc_fn_arg(1)` to a named local first, then pass that local to the accessor:

```
char *_ucv_string_get(uc_value_t **);
#define ucv_string_get(uv) _ucv_string_get((uc_value_t **)&uv)
```

Consequently, writing `strncpy(dst, ucv_string_get(uc_fn_arg(1)), n)` produces the lvalue diagnostic rather than a message about strings, because the expression supplied to `&` is a temporary. The listing avoids the trap by storing fetched arguments in locals. It crosses the C/script boundary both ways: methods mutate the C structure and

return summary strings built from it, and the destructor count confirms that the resource is released once the last script handle disappears.

The checked example adjusts the C quantity, renders a summary from it, and reports the construction and destruction counts. Both counts are equal before the VM is released, showing that the resource was reclaimed deterministically by reference counting rather than waiting for a collection pass:

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

#include <ucode/ucode.h>


/* A number carried in C memory that scripts may read and move. */
typedef struct {
        double qty;
        char label[16];
} counter_t;

static uc_resource_type_t *countertype;
static unsigned int born = 0, died = 0;

static void
counter_free(void *mem)
{
        free(mem);

        died++;
}

static uc_value_t *
counter_bump(uc_vm_t *vm, size_t nargs)
{
        void **slot = uc_fn_this("Counter");
        counter_t *self;
        uc_value_t *amount;

        if (nargs < 1) {
                uc_vm_raise_exception(vm, EXCEPTION_TYPE, "bump() needs a
number");

                return NULL;
        }

        amount = uc_fn_arg(0);
        self = (counter_t *)(*slot);
        self->qty += ucv_to_double(amount);

        return ucv_double_new(self->qty);
}

static uc_value_t *
```

```
counter_text(uc_vm_t *vm, size_t nargs)
{
        void **slot = uc_fn_this("Counter");
        counter_t *self = (counter_t *)(*slot);
        char text[48];

        (void) nargs;

        snprintf(text, sizeof(text), "%s=%.1f", self->label, self->qty);

        return ucv_string_new(text);
}

static uc_function_list_t counter_methods[] = {
        { "bump", counter_bump },
        { "text", counter_text },
};

static uc_value_t *
new_counter(uc_vm_t *vm, size_t nargs)
{
        counter_t *self;
        uc_value_t *arg;

        if (nargs < 2)
                return NULL;

        arg = uc_fn_arg(0);

        self = calloc(sizeof(*self), 1);
        self->qty = ucv_to_double(arg);

        arg = uc_fn_arg(1);
        strncpy(self->label, ucv_string_get(arg), sizeof(self->label) - 1);

        born++;

        return ucv_resource_new(countertype, self);
}

int main(void)
{
        uc_parse_config_t config = { .raw_mode = true };
        static const char *script =
                "let c = newcounter(10, \"ticks\");\n"
                "c.bump(2.5);\n"
                "c.bump(2.5);\n"
```

```
                "return c.text();\n";
        uc_source_t *source;
        uc_program_t *program;
        uc_value_t *retval = NULL;
        uc_vm_status_t status;
        uc_vm_t vm = { 0 };

        setvbuf(stdout, NULL, _IOLBF, 0);

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));

        countertype = ucv_resource_type_lookup(&vm, "Counter");

        if (countertype == NULL)
                countertype = uc_type_declare(&vm, "Counter",
counter_methods, counter_free);

        uc_function_register(uc_vm_scope_get(&vm), "newcounter",
new_counter);

        source = uc_source_new_buffer("counter.uc", strdup(script),
strlen(script));
        program = uc_compile(&config, source, NULL);
        uc_source_put(source);

        status = uc_vm_execute(&vm, program, &retval);

        printf("status=%d returned=%s\n", (int) status, retval ?
ucv_to_string(&vm, retval) : "(none)");
        printf("born=%u died=%u\n", born, died);

        ucv_put(retval);
        uc_program_put(program);
        uc_vm_free(&vm);

        printf("after release: born=%u died=%u\n", born, died);

        return status == STATUS_OK ? 0 : 1;
}
```

The returned summary reflects the native adjustment. The equal `born` and `died` counts show that the script reference did not keep the C object alive past the script's last handle.

```
status=0 returned=ticks=15.0
born=1 died=1
after release: born=1 died=1
```

The complete listing combines several techniques at once: a factory for resource objects, a compact exception formatter, validation in native methods, cooperative cancellation, and cleanup by reference counting:

```
typedef struct {
        int64_t id;
        double qty;
        char label[24];
} ledger_t;

static uc_resource_type_t *ledgers;
static unsigned int born = 0, died = 0;

static void
ledger_free(void *mem)
{
        free(mem);

        died++;
}

static void
log_exception(uc_vm_t *vm, uc_exception_t *ex)
{
        (void) vm;

        printf("[exception] %s: %s\n", exception_type_strings[ex->type],
               ex->message ? ex->message : "(no message)");
}

static uc_value_t *
ledger_withdraw(uc_vm_t *vm, size_t nargs)
{
        void **slot = uc_fn_this("Ledger");
        ledger_t *self;
        uc_value_t *amount;

        if (nargs < 1) {
                uc_vm_raise_exception(vm, EXCEPTION_TYPE, "withdraw() needs
a number of units");

                return NULL;
        }

        amount = uc_fn_arg(0);
        self = (ledger_t *)(*slot);

        if (ucv_to_double(amount) > self->qty) {
                uc_vm_raise_exception(vm, EXCEPTION_USER, "withdraw(%g)
exceeds the %g units on '%s'",
```

```
                                        ucv_to_double(amount), self->qty,
self->label);

                return NULL;
        }

        self->qty -= ucv_to_double(amount);

        return ucv_double_new(self->qty);
}

/* Any mode's leftovers get popped here so the VM goes back to an empty
stack. */
static size_t
drain(uc_vm_t *vm)
{
        size_t n = 0;

        while (vm->stack.count > 0) {
                ucv_put(uc_vm_stack_pop(vm));

                n++;
        }

        return n;
}
```

The installation block selects a lookup-guarded type binding, substitutes a lightweight reporter for the verbose default presenter, arranges factories, and grants the ability to cancel by composing scenario controls. This orchestrates selection between three prepared script bodies that exercise contrasting routes:

```
        ledgers = ucv_resource_type_lookup(&vm, "Ledger");

        if (ledgers == NULL)
                ledgers = uc_type_declare(&vm, "Ledger", ledger_methods,
ledger_free);

        uc_function_register(uc_vm_scope_get(&vm), "openledger",
open_ledger);
        uc_function_register(uc_vm_scope_get(&vm), "stopwork", stop_work);
        uc_vm_exception_handler_set(&vm, log_exception);
```

The full implementation, preserving everything omitted above. Its argument is one of `stock`, `fault` or `halt` and defaults to `stock`; the runs below are one per mode.

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

#include <ucode/ucode.h>


/*
 * A host that hands scripts objects whose contents live in C memory.  The
 * `Ledger` type below is registered like a shipped module would register
one,
 * carries two methods reachable as script methods, and reports what it
raises.
 */

typedef struct {
        int64_t id;
        double qty;
        char label[24];
} ledger_t;


static uc_resource_type_t *ledgers;
static unsigned int born = 0, died = 0;


/* The teardown path every instance takes, however the run ended. */
static void
ledger_free(void *mem)
{
        free(mem);

        died++;
}

static void
log_exception(uc_vm_t *vm, uc_exception_t *ex)
{
        (void) vm;

        printf("[exception] %s: %s\n", exception_type_strings[ex->type],
               ex->message ? ex->message : "(no message)");
}

static uc_value_t *
ledger_adjust(uc_vm_t *vm, size_t nargs)
```

```
{
        void **slot = uc_fn_this("Ledger");
        ledger_t *self;
        uc_value_t *delta;

        if (nargs < 1) {
                uc_vm_raise_exception(vm, EXCEPTION_TYPE, "adjust() needs a
number of units");

                return NULL;
        }

        delta = uc_fn_arg(0);

        self = (ledger_t *)(*slot);
        self->qty += ucv_to_double(delta);

        return ucv_double_new(self->qty);
}

static uc_value_t *
ledger_withdraw(uc_vm_t *vm, size_t nargs)
{
        void **slot = uc_fn_this("Ledger");
        ledger_t *self;
        uc_value_t *amount;

        if (nargs < 1) {
                uc_vm_raise_exception(vm, EXCEPTION_TYPE, "withdraw() needs
a number of units");

                return NULL;
        }

        amount = uc_fn_arg(0);
        self = (ledger_t *)(*slot);

        if (ucv_to_double(amount) > self->qty) {
                uc_vm_raise_exception(vm, EXCEPTION_USER, "withdraw(%g)
exceeds the %g units on '%s'",
                                      ucv_to_double(amount), self->qty,
self->label);

                return NULL;
        }

        self->qty -= ucv_to_double(amount);
```

```
        return ucv_double_new(self->qty);
}

static uc_value_t *
ledger_describe(uc_vm_t *vm, size_t nargs)
{
        void **slot = uc_fn_this("Ledger");
        ledger_t *self = (ledger_t *)(*slot);
        char text[80];

        (void) nargs;

        snprintf(text, sizeof(text), "#%lld %-6s %5.1f", (long long) self-
>id,
                 self->label, self->qty);

        return ucv_string_new(text);
}

/* Methods reach instances through the prototype the type registration
builds. */
static uc_function_list_t ledger_methods[] = {
        { "adjust", ledger_adjust },
        { "withdraw", ledger_withdraw },
        { "describe", ledger_describe },
};

static uc_value_t *
open_ledger(uc_vm_t *vm, size_t nargs)
{
        ledger_t *self;
        uc_value_t *arg;

        if (nargs < 2) {
                uc_vm_raise_exception(vm, EXCEPTION_TYPE, "openledger()
needs an id and a label");

                return NULL;
        }

        arg = uc_fn_arg(0);

        self = calloc(sizeof(*self), 1);
        self->id = ucv_int64_get(arg);

        arg = uc_fn_arg(1);
```

```
        strncpy(self->label, ucv_string_get(arg), sizeof(self->label) - 1);

        born++;

        return ucv_resource_new(ledgers, self);
}

static uc_value_t *
stop_work(uc_vm_t *vm, size_t nargs)
{
        (void) nargs;

        uc_vm_break_request(vm);

        return NULL;
}

/* Any mode's leftovers get popped here so the VM goes back to an empty
stack. */
static size_t
drain(uc_vm_t *vm)
{
        size_t n = 0;

        while (vm->stack.count > 0) {
                ucv_put(uc_vm_stack_pop(vm));

                n++;
        }

        return n;
}

int main(int argc, char **argv)
{
        uc_parse_config_t config = { .raw_mode = true };
        const char *mode = argc > 1 ? argv[1] : "stock";
        static const char *code_stock =
                "let racks = [];\n"
                "push(racks, openledger(7, \"bolts\"));\n"
                "push(racks, openledger(9, \"wire\"));\n"
                "for (let r in racks) r.adjust(2.5);\n"
                "return join(\", \", map(racks, r => r.describe()));\n";

        static const char *code_fault =
                "let racks = [];\n"
                "push(racks, openledger(7, \"bolts\"));\n"
```

```
                "push(racks, openledger(9, \"wire\"));\n"
                "for (let r in racks) r.adjust(2.5);\n"
                "racks[0].withdraw(900);\n"
                "return join(\", \", map(racks, r => r.describe()));\n";

        static const char *code_halt =
                "let racks = [];\n"
                "push(racks, openledger(7, \"bolts\"));\n"
                "push(racks, openledger(9, \"wire\"));\n"
                "for (let r in racks) r.adjust(2.5);\n"
                "stopwork();\n"
                "return \"not reached\";\n";

        const char *code = strcmp(mode, "fault") == 0 ? code_fault :
                           strcmp(mode, "halt") == 0 ? code_halt :
code_stock;
        uc_source_t *source;
        uc_program_t *program;
        uc_vm_status_t status;
        uc_value_t *retval = NULL;
        size_t residue;
        char *error = NULL;
        int rc = 1;

        setvbuf(stdout, NULL, _IOLBF, 0);

        if (strcmp(mode, "stock") != 0 && strcmp(mode, "fault") != 0 &&
strcmp(mode, "halt") != 0) {
                fprintf(stderr, "Usage: %s [stock|fault|halt]\n", argv[0]);

                return 2;
        }

        source = uc_source_new_buffer("service.uc", strdup(code),
strlen(code));
        program = uc_compile(&config, source, &error);
        uc_source_put(source);

        if (program == NULL) {
                printf("result compiled=no error=%s\n", error ? error :
"?");

                free(error);

                return 1;
        }
```

```
        uc_vm_t vm = { 0 };

        uc_vm_init(&vm, &config);
        uc_stdlib_load(uc_vm_scope_get(&vm));

        /* Registering the same name twice drops the second prototype, so
hosts
           that may be initialised more than once look first. */
        ledgers = ucv_resource_type_lookup(&vm, "Ledger");

        if (ledgers == NULL)
                ledgers = uc_type_declare(&vm, "Ledger", ledger_methods,
ledger_free);

        uc_function_register(uc_vm_scope_get(&vm), "openledger",
open_ledger);
        uc_function_register(uc_vm_scope_get(&vm), "stopwork", stop_work);
        uc_vm_exception_handler_set(&vm, log_exception);

        status = uc_vm_execute(&vm, program, &retval);

        residue = drain(&vm);

        printf("result status=%d returned=%s cleaned=%zu\n", (int) status,
               retval ? ucv_to_string(&vm, retval) : "(none)", residue);

        printf("instances born=%u died=%u\n", born, died);

        if (status == STATUS_OK)
                rc = 0;

        ucv_put(retval);
        uc_program_put(program);
        uc_vm_free(&vm);

        printf("released   born=%u died=%u\n", born, died);

        return rc;
}
```

The three transcripts cover normal work, a validation failure, and a cancelled run. In all three, the number of instances born equals the number destroyed. The fault path emits one compact exception line and returns `ERROR_RUNTIME`; the cancelled path reports the number of values drained from the stack and returns `STATUS_BREAK`.

```
$ ./service stock
result status=0 returned=#7 bolts    2.5, #9 wire     2.5 cleaned=0
instances born=2 died=2
released   born=2 died=2
$ echo $?
0
$ ./service fault
[exception] Error: withdraw(900) exceeds the 2.5 units on 'bolts'
result status=4 returned=(none) cleaned=0
instances born=2 died=2
released   born=2 died=2
$ echo $?
1
$ ./service halt
result status=2 returned=(none) cleaned=3
instances born=2 died=2
released   born=2 died=2
$ echo $?
1
```

The fault path ties together mechanisms introduced earlier. `uc_vm_raise_exception()` records an exception type and message, and the installed handler prints one compact line using `exception_type_strings[type]`. A native method returns `NULL` after raising, allowing the run to report the pending exception. The host then drains any values left on the operand stack and records how many it removed. Equal construction and destruction counts show that resources owned by the script were released on every path without requiring the script to cooperate.

The examples run one VM in one thread. Threaded hosts still use the same per-VM lifecycle, but the sharing policy is a host decision rather than part of the VM's contract; see chapter 42 for the state each VM keeps.

Composing the choices

For a new embedding, the decisions are:

- Choose the source mode deliberately. Test template mode and raw-script mode if the host accepts source from other programs.
- Branch on all five run statuses. Drain the stack after a break before reusing the VM.
- Choose one VM lifetime policy: persistent VMs, fresh VMs, or a documented key list for scoped resetting.
- Expose privileged C objects through resource types rather than raw pointers, and look an existing type up before declaring its name again.
- Install a compact exception handler for production logs while keeping detailed reports available for development builds.
- Measure VM creation and recompilation on the target before making lifetime choices on performance grounds.

Natural next steps are to package features as modules (chapters 47 and 48), connect the VM to an event loop (uloop, chapter 36), integrate sockets or ubus, and consider precompiled deployment when startup cost justifies it. None changes the embedding structure: configure the parser, compile once, create a VM, expose selected capabilities, run, classify the status, and release the VM.

Inside the interpreter

Source files referenced in this chapter: `lexer.c`, `include/ucode/internal/lexer.h`, `compiler.c`, `include/ucode/internal/compiler.h`, `chunk.c`, `program.c`, `vm.c`, `include/ucode/internal/vm.h`, `include/ucode/internal/chunk.h`, `include/ucode/internal/program.h`.

The previous ten chapters treated the interpreter as a machine you program against: you compiled a source, you ran a program, you handed values in and got values out. This chapter opens the machine. It matters for three practical jobs: reading a crash or a traceback down to the instruction that caused it, deciding what to believe about performance and memory, and reading the bytecode listings the debugger and the trace output produce. None of the later chapters need what is here, and nothing here changes what they say.

Everything below is specific to this implementation. It is also specific to the internal headers: the opcode numbers, the operand formats and the chunk layout are declared in `include/ucode/internal/`, which is installed with the source tree but is not part of the interface chapters 40 to 50 use. Internal names carry `__hidden` visibility, which means they are not exported from `libucode.so`; a program of the kind this chapter shows reaches the remaining structures through the header itself rather than through library calls.

The pipeline

Four stages, in this order:

```
source text → tokens → (parse and emit, one pass) → bytecode chunks in a
program → the VM
```

The lexer turns the bytes of one `uc_source_t` into tokens. The parser is a precedence-climbing parser — a table of rules, one per token type, each rule a prefix function for when the token starts an expression and an infix function for when it continues one. Code generation happens inside the same recursive descent: there is no syntax tree anywhere in the tree of this repository, and no separate "compile" walk over one. When the parser decides what a construct means it emits the instructions for it, directly into the chunk of the function being compiled. Chapter 43 dealt with what the front end receives and what it hands back; here is what happens in between.

One consequence of the single pass is worth having plainly in mind: a construct can only be compiled with what the parser has already seen and what the compiler has recorded about the enclosing scopes. That is why a forward-declared function must be announced before its call site (chapter 5), why a `break` outside a loop is a syntax error and not a run-time one, and why error recovery skips ahead rather than back. When a syntax error is reported, `uc_compiler_parse_synchronize()` drops tokens until it reaches something that can begin a statement — `}`, `;`, `else`, `endif`, `endwhile`, `endfor`, `endfunc`, `return`, `break`, `continue`, `let` — which is the boundary at which a second diagnostic can be trusted.

Tokens

`uc_lex_state_t` names the states the scanner moves through, and the list explains the two modes the language has. In raw mode the scanner is in `UC_LEX_IDENTIFY_TOKEN` between the outermost pair of braces it recognises as program text; in template mode it alternates between identifying a tag opener and copying literal text:

```
UC_LEX_IDENTIFY_BLOCK   looking for the next tag or the end of file
UC_LEX_BLOCK_EXPRESSION_EMIT_TAG   inside {{ … }}
UC_LEX_BLOCK_STATEMENT_EMIT_TAG    inside {% … %}
UC_LEX_BLOCK_COMMENT               inside {# … #}
UC_LEX_IDENTIFY_TOKEN              scanning one ordinary token
UC_LEX_PLACEHOLDER_START/END       substituting an inline placeholder
UC_LEX_EOF
```

Chapter 16 covers the template tags from the language side. From the scanner side what matters is that a template file produces the token type `TK_TEXT` for its literal runs, and that the compiler emits a `PRINT` instruction per run; a template and the script that would build the same output by hand differ only in where those `PRINT` instructions come from.

Token types are one enum, `uc_tokentype_t`, and they are the parser's whole alphabet. Numbers, strings and regular expressions arrive as prepared `uc_value_t *` payloads on the token rather than as text to be converted later: `TK_NUMBER` carries an integer, `TK_DOUBLE` a floating-point value, `TK_STRING` an interned string, `TK_REXP` a compiled pattern. The keyword set is not a separate token type each; words such as `if`, `while` and `let` do have their own types (`TK_IF`, `TK_WHILE`, `TK_LOCAL`), while contextual words are matched against the lexeme with `uc_compiler_keyword_check()`. That distinction is why the brace-free block syntax of chapter 7 works: `endif` is matched as a keyword where a block-end is wanted, not as a reserved word that would then collide with a variable of that name.

The precedence table

The parser's notion of precedence is one enum, in ascending binding strength. The order is the whole of the rule; there is no second table to disagree with it:

```
P_COMMA      ,
P_ASSIGN     = += -= *= /= %= <<= >>= &= ^= |= **= &&= ||= ??=
P_TERNARY    ?:
P_OR         || ??
P_AND        &&
P_BOR        |
P_BXOR       ^
P_BAND       &
P_EQUAL      == === != !==
P_COMPARE    < <= > >= in
P_SHIFT      << >>
P_ADD        + -
P_MUL        * / %
P_EXP        **
P_UNARY      ! ~ + - ++ -- (prefix)
P_INC        ++ -- (postfix)
```

`uc_compiler_parse_precedence()` climbs this ladder, taking operators while the next token binds tighter than the level it was called at, which is what gives the familiar shape of expression parsing. The two details that are visible from the language are where associativity comes from and where it does not. `**` sits above `*` so that `2 ** 3 ** 2` groups to the right, and assignment sits below everything it could take on the right — chapter 6 has the resulting table with examples. The table also shows `in` grouping with the comparisons rather than with the bitwise operators, and `??` grouping with `||`, which is what makes `a ?? b || c` parse as `(a ?? b) || c`.

Chunks, the constant pool, and locals

Each function gets one `uc_chunk_t`: a byte vector of instructions, plus debug information. The debug part is a record of statement spans and of variable spans, and it is what allows a name to be attached to an operand when a listing is printed; it is also what the debugger turns a breakpoint line into an offset with (chapter 60), and what the exception report quotes the offending source line from (chapter 14).

Numbers and strings do not travel in the instruction stream. They are added to the program's constant pool and the instruction carries the index. `uc_vallist_add()` maintains the pool with separate sorted indexes for integers, floating-point values and strings, so a repeated literal is not duplicated:

```
let a = "same";
let b = "same";
print(a == b, " ", a === b, "\n");
```

```
true true
```

The two bindings receive the same interned string, which is why the identity comparison holds. Values that are neither numbers nor strings have no place in the pool; `true`, `false` and `null` are their own instructions, and literal containers are built by the instructions of the next section.

The other structural fact about compiled code is that **a local variable is a stack slot**. Entering a function sets a frame base; local slot *n* is then stack position *base* + *n*. A `let` therefore compiles to "evaluate the initialiser and leave its value where it lands", with no store instruction at all:

```
0000  LOAD8   {0x1}
0002  LOAD8   {0x2}
0004  LOAD8   {0x3}
0006  MUL
0007  ADD
```

Those five instructions are the whole of `let x = 1 + 2 * 3;`: the value seven is computed and left in the slot the new local occupies. A later read is `LLOC` of that slot, and the assignment to such a variable is `SLOC`, which writes the slot and leaves the value in place because the assignment is itself an expression — the `POP` that follows a statement-level assignment discards the value the statement is not interested in:

```
0008  LLOC    {0x1}
000d  LOAD8   {0x1}
000f  ADD
0010  SLOC    {0x1}
0015  POP
```

That is `x = x + 1;` — read the slot, add, store it back, drop the expression's value. A block pops its locals when it exits, and the same popping at the end of a function body is what makes the frame reusable, as it will be in the tail-call section.

The instruction encoding

An instruction is an opcode byte followed by nothing, or by one operand of one, two, or four bytes, big endian — the most significant byte first. There are seventy opcode values, and which of them take an operand, and of what width and sign, is one table, `uc_vm_insn_format[]`, indexed by opcode: a positive value is the operand width in bytes and a negative value marks the operand as signed. The table is the whole of the encoding, and it is exported, which makes it readable from outside the library:

```
[I_LOAD] = 4      [I_LOAD8] = 1     [I_LOAD16] = 2    [I_LOAD32] = 4
[I_LREXP] = 4
[I_LLOC] = 4      [I_LVAR] = 4      [I_LUPV] = 4
[I_CLFN] = 4      [I_ARFN] = 4
[I_SLOC] = 4      [I_SUPV] = 4      [I_SVAR] = 4
[I_ULOC] = 4      [I_UUPV] = 4      [I_UVAR] = 4      [I_UVAL] = 1
[I_NARR] = 4      [I_PARR] = 4
[I_NOBJ] = 4      [I_SOBJ] = 4
[I_JMP] = -4      [I_JMPZ] = -4     [I_JMPNT] = 4
[I_COPY] = 1
[I_CALL] = 4
[I_IMPORT] = 4    [I_EXPORT] = 4    [I_DYNLOAD] = 4
```

Anything absent from the table takes no operand, including every arithmetic and comparison operator. Four operands are not plain numbers:

Instruction	Operand
`JMP`, `JMPZ`	a branch distance, biased: the stored value minus `0x7fffffff` is the distance from the opcode byte to the target, so `target = offset + distance`
`ULOC`, `UUPV`, `UVAR`	the low 24 bits are a slot, upvalue index or name index; the high 8 bits are an opcode naming the operation, and the handler dispatches through it
`CALL`	bits 0–15 the argument count, bits 16–28 the number of spread arguments, bit 31 "method call"
`CLFN`, `ARFN`	a 1-based function id, followed in the stream by four bytes per upvalue the function captures

The bias in the branch operand and in a capture word is the same trick: an unbiased signed value cannot be distinguished from an unpatched zero, and the compiler patches a jump after it has emitted the block it branches over. A capture word is `-(slot + 1)` for an enclosing *local* — always negative — and the enclosing function's *upvalue index* for a name that was already an upvalue — never negative.

`NOOP` is opcode zero, is absent from the format table, and has no case in the dispatch switch. Encountered as an instruction it would raise "unknown opcode". It exists because the compiler emits a single zero byte immediately after certain `RETURN`s, as the next section describes; nothing executes it.

The instruction set

Grouped by what they are for. The name column is the mnemonic as the trace and the debugger print it, and is the identifier `I_` plus that name in the internal header.

Loading values:

Mnemonic	Operand	Effect
`LOAD`	constant index	push the constant
`LOAD8`, `LOAD16`, `LOAD32`	immediate	push the integer
`LNULL`, `LTRUE`, `LFALSE`	—	push the respective value
`LREXP`	constant index	push a fresh regexp from the pooled pattern, its first byte the flag bits
`LTHIS`	—	push the current frame's `this`
`LVAR`	constant index naming a variable	look the name up through the scope chain; push `null` if absent, raise a reference error in strict mode
`LLOC`, `LUPV`	slot, upvalue index	push a local, push an upvalue (an open one reads through to its slot)
`LVAL`	—	pop a key and the value below it, push the keyed read — dispatching `__get__`
`PVAL`	—	as `LVAL` but leaving the container in place, for compound assignment
`COPY`	depth	push again the value *depth* places down the stack

Storing:

Mnemonic	Operand	Effect
`SVAR`	name index	store into the scope that owns the name, creating it in the outermost reachable one, then push the value back
`SLOC`, `SUPV`	slot, upvalue index	write the top of the stack into a local or upvalue, leaving the value there
`SVAL`	—	pop container, key and value, store through `ucv_key_set()` (dispatching `__set__`), push the result
`ULOC`, `UUPV`, `UVAR`, `UVAL`	packed index and operator	read, combine with the popped right operand, store, push the new value; the operator is carried in the operand

Containers:

Mnemonic	Operand	Effect
`NARR`	capacity	push a new empty array with room reserved for that many elements
`PARR`	count	append the *count* values above the array beneath them, then pop them
`MARR`	—	pop an array and append its elements to the array on the stack top; anything else raises "is not iterable"
`NOBJ`	(ignored)	push a new empty object
`SOBJ`	count, even	raw-store the *count*/2 key and value pairs above the object beneath them, then pop them
`MOBJ`	—	spread an object's keys, or an array's elements under index keys, into the object below

`SOBJ` writes with `ucv_key_rawset()`, so the `__set__` metamethod of chapter 12 does not see the keys an object literal is built from; a spread by contrast goes through the ordinary object store.

Operators: `ADD`, `SUB`, `MUL`, `DIV`, `MOD`, `EXP` take two values and push one, with the division and integer-overflow behaviour of chapter 6; `BOR`, `BXOR`, `BAND`, `LSHIFT`, `RSHIFT` do the same for the bitwise operators, producing unsigned results; `EQ`, `NE`, `EQS`, `NES`, `LT`, `LE`, `GT`, `GE` push a boolean; `IN` asks whether the lower value is a key or member of the higher one; `NOT` is the logical negation of truthiness, `COMPL` the bitwise complement, `PLUS` and `MINUS` the unary coercions.

Control:

Mnemonic	Operand	Effect
`JMP`	biased distance	branch unconditionally; out of range raises "jump target out of range"
`JMPZ`	biased distance	pop a value and branch when it is falsy
`JMPNT`	packed type set, depth and distance	pop a value and branch when its type is outside the packed set, pushing `null` in that case
`CALL`	packed counts	call, as below
`RETURN`	—	end the frame, leaving its result on the caller's stack
`CUPV`	—	close the open upvalues of the frame about to be left, then pop
`POP`	—	pop one value, and take one step of the collector
`PRINT`	—	pop one value and write it to the VM output, strings raw and containers as JSON

`PRINT` is what a template's literal text compiles to; `print()` as a language function is a normal call to a native function, which is why a listing of `print(x, "\n")` shows `LVAR`, `LLOC`, `LOAD`, `CALL`.

Iteration and lifetime:

Mnemonic	Operand	Effect
`NEXTK`, `NEXTKV`	—	advance the iterator state on the stack, pushing the next key (with the value, for `NEXTKV`) and the new state; at the end pushing nulls
`DELETE`	—	pop key and container, remove the key, push whether it was removed; a non-object raises a reference error
`EXPORT`	slot	capture a local as an upvalue and append it to the program's export list
`IMPORT`	packed export index and target upvalue	bind an imported name; a target of `0xffff` means the wildcard form, whose following bytes list the exported names to collect into a constant object
`DYNLOAD`	packed count and base upvalue	pop a module name, load it, then bind the listed exports, or a copy of the whole module scope when the count is zero

`IMPORT` and `EXPORT` belong to the compile-time module model of chapter 17 and run once, when the module body runs; `DYNLOAD` is what a dynamic `import()` or `require()` compiles to, and it can raise. A missing named export in the static case binds `null` rather than failing — an import that does not resolve is not a compile error in that form.

Closures and upvalues

A function value is made by `CLFN`, or by `ARFN` for an arrow function, carrying the function id; the two differ only in the flag set on the closure. A function that references a name from an enclosing function declares one upvalue each, and the closure instruction is followed by one four-byte word per upvalue. That word is the whole of the link between the two frames, and the listing prints it:

```
function counter() {
        let n = 0;

        return function () {
                n++;

                return n;
        };
}

let next = counter();

print(next(), next(), "\n");
```

```
12
```

(`print()` joins its arguments without a separator, chapter 9.)

Compiled, the same program reads:

```
; function counter: 0 args, 0 upvalues, 14 bytes
0000  LOAD8   {0x0}
0002  CLFN    {0x3}      ; (anonymous), 1 upvalue
             capture -2
000b  RETURN
```

The capture word decodes to `-2`, and the encoder's rule is `-(slot + 1)`, so slot 1 of `counter`'s frame is what is being captured — which is `n`. Had `counter` itself been nested one level deeper and `n` reached through an upvalue rather than a local, the word would have been non-negative and named the enclosing upvalue index instead.

While the enclosing frame is live the upvalue is *open*: it names a slot in that live frame rather than holding a value, so both functions see the one variable, and a write through either is visible to the other. When the enclosing frame is about to be left, the upvalues pointing into it are closed: the reference copies the slot's value into itself and stops aliasing the stack. The closing is driven by the stack height, so it catches every local that a block is leaving, and the trace prints each one as a `{!slot}` line when `-t` is on. Once closed, an upvalue is a value cell; the returned closure above keeps `n` alive exactly because the reference that closed owns it.

Reading and writing an upvalue from the inner function is visible in the listing of the same example:

```
; function : 0 args, 1 upvalues, 19 bytes
0000  LOAD8   {0x1}
0002  UUPV    {0x0}     ; with PLUS
0007  LOAD8   {0x1}
0009  SUB
000a  POP
000b  LUPV    {0x0}     ; n
0010  RETURN
```

The body `n++; return n;` is the instruction sequence for a postfix increment: push one, `UUPV` adds it to upvalue 0 and leaves the new value, subtract one to recover the value the expression evaluates to, discard that with `POP` because it is a statement, then read the upvalue back for the return.

The calling convention

At a call site the callee and its arguments are already on the stack in this order, lowest first: an optional receiver, the function value, then the arguments. `CALL` says how to read that arrangement. The function value sits at depth *nargs* — one more, `nargs + 1`, when the method bit is set, which is also where the receiver is taken from. Arrow functions take no receiver at all: they inherit the enclosing frame's `this`, which is the mechanism behind chapter 8's "arrow functions have no `this`".

Spread arguments complicate the arrangement, because `f(...a, 1, ...b)` does not know its final arity until it runs. The packed spread count tells the call how many of the pushed arguments are arrays to be flattened; the call builds a temporary array of what it has, splices each marked argument's elements in, and reshuffles the stack to the real argument list. Excess arguments beyond a fixed-arity function's parameters are discarded; a variadic function instead finds an internal ellipsis marker in place of the extras, which is what `arguments` reports.

The frame that a call pushes records the base of its arguments in the stack, the closure or native function being run, the receiver, the argument count, whether it is a method call, and whether the function was compiled in strict mode. Native functions get a frame of the same shape. The count of frames is capped at 1000, and exceeding it raises the runtime error `Too much recursion`; chapter 14 shows the shape of that report and chapter 8 the depth a recursive script reaches before it.

Tail calls

A call whose result is returned directly — `return f();` — has nothing left to do in the calling frame, so the compiler marks the return and the VM replaces the current frame rather than pushing a new one. The mark is a zero byte after the return instruction, which is why a listing shows a `NOOP` there:

```
function direct() {
        return indirect();
}

function guarded() {
        try {
                return indirect();
        } catch (e) {
                return e;
        }
}
```

The two functions compile as follows. `indirect()` is not defined here, and that does not matter: the question is what the compiler emits, not what happens when the code runs.

```
; function direct: 0 args, 0 upvalues, 14 bytes
0000  LVAR    {0x0}
0005  CALL    {0x0}     ; 0 args, 0 spreads
000a  RETURN
000b  NOOP
000c  LNULL
000d  RETURN

; function guarded: 0 args, 0 upvalues, 25 bytes
0000  LVAR    {0x0}
0005  CALL    {0x0}     ; 0 args, 0 spreads
000a  RETURN
000b  JMP     {+12}
0010  LLOC    {0x1}     ; e
0015  RETURN
0016  POP
```

`direct` has the marker; `guarded` does not, and it cannot have one. A `return` inside a `try` block is a return the block still has to be protected for, because an exception from the callee has to find the handler; the compiler tracks a nesting count of enclosing `try` blocks and suppresses the marker while that count is above zero. So the presence or absence of the `NOOP` byte in a listing tells you whether a frame is kept, and a recursion that is a tail call in one function is a frame per level in another whose body has acquired a `try` around it. Collapsed frames are counted in

the frame, so a backtrace over tail calls reports the depth that was avoided rather than pretending the frames were never there.

Exceptions are addressed out of line

A `try` block costs nothing inside the instruction stream: there is no handler instruction to execute on the way through protected code. What the compiler adds is an entry to the chunk's exception-handling range table, giving a span of offsets, the offset of the handler code, and the stack slot to unwind to. When an exception is raised the VM searches the current chunk's ranges for one containing the current offset, and a hit restores the stack to the recorded slot and sets the instruction pointer to the handler. Nothing on the normal path is examined at all.

The handler is compiled as code placed after the protected block, which is what the listing of `guarded` above shows: the `JMP` after the return skips over `LLOC` / `RETURN` to the continuation past the block, and that skipped section is the handler, entered only by the range table. A `catch` variable is a local of the handler's own scope, read with `LLOC`, and it goes out of existence with it. A handler whose span has ended is no longer found, which is why an exception escapes an inner `try` and reaches an outer one without any instruction being involved. The range table travels in the debug part of the chunk, so a chunk written without debug information keeps it: it is not decoration, and without it the exception would have nowhere to land.

Running the code

`uc_vm_execute()` builds the closure over the program's entry function (chapter 43), pushes a frame based at slot zero, pushes a placeholder for the result, and enters the dispatch loop. The loop decodes one instruction — reading its operand into a union and advancing the instruction pointer — and switches on the opcode. Nothing else drives the interpreter: there is no scheduler and no callback between instructions, although a breakpoint check does run inside the decode step, which is how a breakpoint at an offset fires when control reaches that offset (chapter 60).

The collector is driven from that same loop, from one instruction: `POP` calls `uc_vm_gc_step()`, which does nothing until the VM's allocation count reaches the interval, at which point the cycle collector runs and the count resets. The interval is the one chapter 18 discusses and `-g` changes: `UC_GC_DEFAULT_INTERVAL`, one thousand allocations. A program whose allocation rate is high but whose `POP` rate is low can therefore hold a cycle longer than the interval suggests, and a `gc()` call from the script runs the collector immediately rather than waiting for a `POP`.

`uc_vm_execute()` finishes by translating the run's end into one of the five statuses of chapter 46, and on an exception it invokes the installed handler. Nothing in this chapter changes that contract; a host still sees only the status and the value.

The compiled file

Chapter 47 has the format in full: the magic word, the version carried in the top byte of the flags word, the sources, the constant pool, the exports, and then the function chunks in the same byte arrangement this chapter describes. Two

facts from it belong here because they explain listings. A loaded program is a program in every respect — it is dispatched and run the same way, more than once — and the debug information is what carries the statement spans, the variable name spans and the exception ranges along with the instructions. A program compiled with `debug = false` is smaller chiefly because that information is absent, and the consequence is not only cosmetic: the exception still lands on its handler, but nothing can say which name a slot held.

Reading a listing

Two facilities produce these listings. The first is the trace: `ucode -t` runs the program while printing every step, and is the instrument for a question of the form "what is this actually doing". Its output goes to standard error, and it prints three kinds of line. A source-context line names the file, line and the bytes of the statement about to run, with the statement highlighted; a frame line prints a frame when one is pushed; and an instruction line gives the offset, the mnemonic, the operand, and an annotation naming what the operand means. Under the instruction lines, each stack push, pop and slot write is reported with the slot it touched. Colour codes surround the source-context lines, dropped from the next listing along with the rest of the terminal's escapes:

```
          f.uc:1  let total = 0;
  [*] CALLFRAME[0]
   |- stackframe 0/0
   |- ctx null
   `- 0 upvalues
  [+0] null
          f.uc:1  let total = 0;
00000000  LOAD8 {0}
  [+1] 0
          f.uc:1  let total = 3;
00000002  LOAD8 {3}
  [+2] 3
          f.uc:2  total += 3 * 4;
00000004  LOAD8 {4}
  [+3] 4
          f.uc:2  total += 3 * 4;
00000006  MUL
  [-3] 4
  [-2] 3
  [+2] 12
00000007  ULOC {0x2d000001}      ; "total" (ADD)
  [-2] 12
  [+2] 12
  [!1] 12
0000000c  POP
  [-2] 12
```

The annotation on `ULOC` shows both halves of its packed operand, the index and the operation; the three bracket forms are `[+slot]` for a push, `[-slot]` for a pop and `[!slot]` for a slot write. A call adds a frame line at the call and a `CALLFRAME[n]` block giving the new frame's base, receiver and upvalue count:

```
0000000f  CALL {0x1}
  [*] CALLFRAME[1]
   |- stackframe 3/5
   |- ctx null
   `- 0 upvalues
          g.uc:1  function twice(v) {
00000000  LLOC {0x1}    ; "v"
  [+5] 21
```

The second facility is a disassembly of a whole function rather than of a run. `udbg`'s `disassemble` command does it against a live program, and chapter 60 covers it. It needs a process being debugged, so the listing below comes from a small program of this chapter's own, which compiles a script and walks the bytes of every chunk, decoding them with the exported format table and naming locals through the exported variable-span lookup. It is deliberately a straightforward linear sweep, which means it reads the extra words an instruction carries — the capture words after a closure, the counts after a spread call — from the width it just decoded, and it prints operand values the way the table gives them, biased where the table is biased:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <ucode/ucode.h>
#include <ucode/internal/vm.h>
#include <ucode/internal/chunk.h>
#include <ucode/internal/program.h>

/* The single list of mnemonics is an X-macro in the internal header;
expanding it
 * a second time here yields the names belonging to the opcode numbers. */
#undef __insn
#define __insn(_name) #_name,
static const char *insn_names[] = { __insns };
#undef __insn

/* Function ids are 1-based positions in the program's function list. */
static uc_function_t *
function_by_id(uc_program_t *program, size_t id)
{
        size_t i = 1;

        uc_program_function_foreach(program, fn)
                if (i++ == id)
                        return fn;

        return NULL;
}

static void
disassemble(uc_program_t *program)
{
        uc_program_function_foreach(program, fn) {
                uc_chunk_t *chunk = &fn->chunk;

                printf("; function %s: %zu args, %zu upvalues, %zu bytes\n",
                       fn->name, fn->nargs, fn->nupvals, chunk->count);

                for (size_t off = 0; off < chunk->count; ) {
                        uint8_t op = chunk->entries[off];
                        int8_t fmt = op < __I_MAX ? uc_vm_insn_format[op] :
0;

                        size_t wide = (size_t)(fmt < 0 ? -fmt : fmt);
                        size_t captures = 0;
                        uint32_t arg = 0;
```

```
                        char note[96] = "";

                        if (op >= __I_MAX) {
                                printf("%04zx  <invalid opcode 0x%02x>\n",
off, op);

                                break;
                        }

                        /* operands are big endian, most significant byte
first */
                        for (size_t i = 0; i < wide; i++)
                                arg = arg * 0x100 + chunk->entries[off + 1
+ i];

                        switch (op) {
                        case I_LLOC:
                        case I_SLOC:
                        case I_LUPV:
                        case I_SUPV:
                                {
                                        uc_value_t *name =
uc_chunk_debug_get_variable(
                                            chunk, off, arg, op == I_LUPV
|| op == I_SUPV);

                                        snprintf(note, sizeof(note), "; %s",
                                                 name ?
ucv_string_get(name) : "(?)");

                                        break;
                                }

                        case I_ULOC:
                        case I_UUPV:
                        case I_UVAR:
                                snprintf(note, sizeof(note), "; with %s",
insn_names[arg >> 24]);
                                arg &= 0x00ffffff;

                                break;

                        case I_JMP:
                        case I_JMPZ:
                                snprintf(note, sizeof(note), "; to %04zx",
                                         off + (size_t)((int32_t)arg -
0x7fffffff));
```

```
                                break;

                        case I_CLFN:
                        case I_ARFN:
                                {
                                        uc_function_t *fn2 =
function_by_id(program, arg);

                                        captures = fn2 ? fn2->nupvals : 0;

                                        snprintf(note, sizeof(note), "; %s,
%zu upvalue%s",
                                                 fn2 && *fn2->name ? fn2-
>name : "(anonymous)",
                                                 captures, captures == 1 ?
"" : "s");

                                        break;
                                }

                        case I_CALL:
                                snprintf(note, sizeof(note),
"; %u arg%s, %u spread%s%s",
                                         arg & 0xffff, (arg & 0xffff) ==
1 ? "" : "s",
                                         (arg >> 16) & 0x7fff,
                                         ((arg >> 16) & 0x7fff) == 1 ? "" :
"s",
                                         (arg & 0x80000000) ? ", method" :
"");

                                break;

                        default:
                                break;
                        }

                        if (fmt < 0)
                                printf("%04zx  %-7s {%+d}\t%s\n", off,
insn_names[op],
                                       (int32_t)arg - 0x7fffffff, note);
                        else if (wide)
                                printf("%04zx  %-7s {0x%x}\t%s\n", off,
insn_names[op], arg, note);
                        else
                                printf("%04zx  %-7s\t\t%s\n", off,
```

```
insn_names[op], note);

                        for (size_t i = 0; i < captures; i++) {
                                uint32_t raw = 0;

                                for (size_t j = 0; j < 4; j++)
                                        raw = raw * 0x100 +
                                              chunk->entries[off + 1 + wide
+ 4 * i + j];

                                printf("             capture %+zd\n",
                                       (ssize_t)((int32_t)raw -
0x7fffffff));
                        }

                        off += 1 + wide + 4 * captures;
                }

                printf("\n");
        }
}

static const char *script =
        "function counter() {\n"
        "       let n = 0;\n"
        "       return function () {\n"
        "               n++;\n"
        "               return n;\n"
        "       };\n"
        "}\n"
        "let next = counter();\n"
        "print(next(), next(), \"\\n\");\n";

int
main(void)
{
        uc_parse_config_t config = { 0 };
        uc_program_t *program;
        uc_source_t *source;
        char *error = NULL;

        config.raw_mode = true;

        source = uc_source_new_buffer("counter.uc", strdup(script),
strlen(script));
        program = uc_compile(&config, source, &error);
        uc_source_put(source);
```

```
        if (!program) {
                fprintf(stderr, "%s", error);

                free(error);

                return 1;
        }

        printf("; %d opcodes\n\n", __I_MAX);

        disassemble(program);

        uc_program_put(program);

        return 0;
}
```

```
; 70 opcodes

; function main: 0 args, 0 upvalues, 54 bytes
0000  CLFN    {0x2}     ; counter, 0 upvalues
0005  LLOC    {0x1}     ; counter
000a  CALL    {0x0}     ; 0 args, 0 spreads
000f  LVAR    {0x0}
0014  LLOC    {0x2}     ; next
0019  CALL    {0x0}     ; 0 args, 0 spreads
001e  LLOC    {0x2}     ; next
0023  CALL    {0x0}     ; 0 args, 0 spreads
0028  LOAD    {0x1}
002d  CALL    {0x3}     ; 3 args, 0 spreads
0032  RETURN
0033  NOOP
0034  LNULL
0035  RETURN

; function counter: 0 args, 0 upvalues, 14 bytes
0000  LOAD8   {0x0}
0002  CLFN    {0x3}     ; (anonymous), 1 upvalue
             capture -2
000b  RETURN
000c  LNULL
000d  RETURN

; function : 0 args, 1 upvalues, 19 bytes
0000  LOAD8   {0x1}
0002  UUPV    {0x0}     ; with PLUS
0007  LOAD8   {0x1}
0009  SUB
000a  POP
000b  LUPV    {0x0}     ; n
0010  RETURN
0011  LNULL
0012  RETURN
```

Read from the top: the main function makes the closure for `counter`, calls it twice through the `next` local, and calls `print` with three arguments; the `NOOP` after the last `RETURN` is the tail-call marker for the final `print` call. `counter` loads zero — that is `n`'s initialiser, left in the slot — and makes the inner closure, capturing one upvalue at the encoded `-2`, slot 1. The anonymous function has the one upvalue and no arguments, and its body is the postfix increment and read traced above. The names shown for slots come from the chunk's variable spans, so they are absent where the chunk was compiled without debug information; the `LVAR` and `LOAD` lines name no constant because the constant pool is not readable through the public header, which is where the trace output has the advantage: it annotates constant operands with the value, as its `; "print"` above shows.

A last observation on which of the three names is printed where. A function value's own name comes from the program's function table, and an anonymous function has an empty one — the third chunk is headed `; function :` because nothing names it. The debugger resolves the same thing from the variable spans instead, which is why chapter 60's backtrace can call that closure by the name it was bound to.

Summary

- Parsing and code generation are one pass: a precedence-climbing parser emits instructions directly into the chunk of the function being compiled, so a syntax error is a fact about text the parser cannot use and recovery means skipping to a statement boundary.
- A local variable is a stack slot of its frame, so an initialiser is a computation left in place rather than a store; a block exit pops its locals.
- An instruction is an opcode byte plus an optional big-endian operand of one, two or four bytes; the exported `uc_vm_insn_format[]` table gives every width and sign. Branch distances and closure capture words are biased by `0x7fffffff`. Compound updates pack an operator into the top byte of the operand and calls pack argument, spread and method counts.
- A closure instruction carries four bytes per upvalue: negative for a captured enclosing local, encoded as `-(slot + 1)`, non-negative for an inherited upvalue index. Open upvalues alias a live slot; closing one copies the value out.
- `NOOP` after a `RETURN` is the tail-call marker, and a `return` inside a `try` never carries one — the frame has to stay for the handler.
- Exception handling lives in a per-chunk range table of spans, handlers and unwind targets, consulted only when something is raised; it travels with the chunk even when the rest of the debug information is dropped.
- `POP` drives the collector, so the allocation interval of chapter 18 is measured in allocations between pops.
- Read a run with `ucode -t`; read a function with `udbg`'s `disassemble`, or with a sweep over the chunk bytes against the exported format table.

Deployment models

Source files referenced in this chapter: `CMakeLists.txt`, `main.c`, `program.c`, `include/ucode/program.h`, `openwrt/ucode/Makefile`, `debian/rules`, `include/ucode/vm.h`.

A ucode program reaches a device in five shapes: a source file that the interpreter reads when it is wanted, an executable script started by its shebang line, a precompiled bytecode file carrying its own shebang, a host process that holds a virtual machine inside itself, and a resident server that keeps one machine warm across many requests. The shapes are not ranked; they differ in what has to be present at run time, what a start-up costs, whether anything survives between two invocations, how an error reaches a human, and who reaps the process when it is done.

The measurements in this chapter are from a debug build of the source tree, on one machine, at one moment; they are there to show orders of magnitude, and a size-optimised build — which is what `BUILD_OPTIMIZE_SIZE` turns on by default, and what both packaging routes turn off deliberately — measures differently.

The standalone interpreter

The plain shape is a script and the interpreter named in the same command line:

```
$ ./build/ucode -e 'print("hello\n")'
hello
$ ./build/ucode -p '6 * 7'
42
$ ./build/ucode script.uc one two
```

With `-e` the argument is the program; `-p` is the same with its result printed, on the value's string form and without a trailing newline, which is why the transcript above shows the shell's prompt running on from `42`; naming files makes them the program, with anything after them landing in `ARGV`, and with `-` standing for the standard input. A script run this way sees the file name it was given as `SCRIPT_NAME` and the trailing words as `ARGV`; chapter 20 owns those two variables and the rules for populating them from a `ucode` invocation of your own. Nothing about the deployment shows up in either of them, which is one reason the shape transfers so well: the same file runs from a command line, from a cron entry and from another script.

The exit status is the part a caller acts on, and it has a small, fixed set of values. A program that returns from its last statement exits `0`; `exit(n)` exits with `n` taken modulo 256; an uncaught runtime exception or a `die()` exits `254`; a syntax error exits `255`; and a file that cannot be opened exits `1` after a message on the standard error.

```
print("status when this file ends: 0\n");
```

```
status when this file ends: 0
```

```
$ for probe in 'exit(3)' 'exit(259)' 'die("nope")' 'nosuchfunction()' 'let
=' ; do
        ./build/ucode -e "$probe" 2>/dev/null
        printf '%-22s %s\n' "$probe" "$?"
done
exit(3)                 3
exit(259)               3
die("nope")             254
nosuchfunction()        254
let =                   255
```

`254` and `255` are the two values worth remembering, because they are how a supervision layer learns that a script went wrong rather than finished, and because neither of them is the status a shell reports for a signal — a script killed by `SIGKILL` reports through the shell's own notation instead. A file that cannot be opened is a third case, reported by the driver rather than by the parser:

```
$ ./build/ucode /tmp/nope.uc
Failed to open "/tmp/nope.uc": No such file or directory
$ echo $?
1
```

What must be installed for the shape to work is the interpreter and, beyond the built-in functions, the extension modules the script imports. Module resolution is a search path, and the compiled-in default is built from the install prefix — `<prefix>/<libdir>/ucode/*.so`, then `<prefix>/share/ucode/*.uc`, then `./*.so` and `./*.uc` — and chapter 17 has the whole of `-L` and the path's precedence; what matters for deployment is that the path is baked in at configure time, that a relative entry in it makes a script's working directory part of its behaviour, and that a `.uc` file found on the path satisfies an import just as a shared object does, which is how pure-ucode libraries are distributed without touching a build system.

Executable scripts

A file with a shebang line and the executable bit set is started by the kernel, which reads the line, splits it on spaces, applies at most one argument, and execs the named program with the script as the argument after it. Both usual spellings work:

```
#!/usr/bin/env ucode
#!/usr/local/bin/ucode
```

The first finds the interpreter wherever the invoking environment puts it, and is the form to use in a tree whose install prefix is not settled; the second is exact and does not consult a path. Because the interpreter derives a good deal of its behaviour from the name it was called under, a shebang script named after one of the other two names inherits that mode: `utpl` switches the source to template mode, and `ucc` makes a bare invocation write `./uc.out`.

```
$ printf '#!/usr/bin/env utpl\nName: {{ name }}\n' > /tmp/greet.uc
$ chmod +x /tmp/greet.uc
$ PATH=./build:$PATH /tmp/greet.uc
Name:
```

The interpolation evaluates to nothing in hand, since nothing defined `name`, and an empty interpolation comes out as nothing; `-D` is how a value is given to a template from the command line, which chapter 16 covers.

That convenience is also the shape's one real trap: a script that must run in template mode depends on the name of the file it happens to be installed as. Naming it `greet.uc` and calling it through `utpl` is the reliable spelling, and passing `-T` explicitly in a wrapper is the explicit one.

Strict mode cannot ride on a `#!/usr/bin/env ucode` line, because `env` receives the remainder of the line as a single word:

```
$ printf '#!/usr/bin/env ucode -S\nprint("never runs\n")\n' > /tmp/bad.uc
$ chmod +x /tmp/bad.uc && PATH=./build:$PATH /tmp/bad.uc
env: ‘ucode -S’: No such file or directory
env: use -[v]S to pass options in shebang lines
```

Naming the interpreter directly does take the one flag, which is the form to use when a tree wants its scripts strict:

```
#!/usr/local/bin/ucode -S
```

and the portable alternative is a one-line wrapper that execs `ucode -S` with the real script as its argument. Strict mode is a parse-time setting rather than a runtime one, so it cannot be requested from inside a file; chapter 5 has what it changes.

Precompiled programs

`ucc` turns sources into a bytecode file, and the file is a program in its own right: it begins with a shebang line naming the interpreter, it is exec-able, and running it needs no compiler in the process — which is the shape to pick when the run time must not carry the compiler's code, or when parse time is wanted once at build instead of once per start.

```
$ printf 'print("precompiled\n");\n' > hello.uc
$ ./build/ucc -o hello.uc.bin hello.uc
$ ls -l hello.uc hello.uc.bin
-rw-rw-r-- 1 jow jow  23 Sep 24 13:33 hello.uc
-rwxrwxr-x 1 jow jow 221 Sep 24 13:33 hello.uc.bin
$ chmod +x hello.uc.bin && ./hello.uc.bin
precompiled
$ head -c 8 hello.uc.bin
#!/usr/b
```

The size difference is the honest one: twenty-three bytes of text become two hundred and twenty-one of program, so a precompiled file is bigger than its source and smaller than what the same statement costs in a process that has to parse it. The executable bit is set by `ucc` itself. The shebang at the front is what makes direct execution work, and it names the interpreter as the build knew it; `-I` sets it, for a tree whose interpreter lands somewhere else. Chapter 47 owns the format — the magic, the sections, the debug information, `-s` for stripping it and `-g` for keeping source line mappings — and three things there are deployment decisions rather than format details: a program built with debug information reports an error at the source line it came from, since the line-to-opcode mapping rides in the file; a program built with `-s` reports against the program's own coordinates instead, which is a poorer message and a smaller file; and a deployed program whose sources are absent still names those sources in its messages, so shipping the programs without the sources leaves a diagnosis that points at a file nobody has. A build that strips the debug information and keeps the sources is worse off than one that keeps neither.

Loading is cheap and parsing is not free, which is the argument for the shape on a device that starts often. Four hundred starts of each of three forms, on the debug build of this tree:

```
ucode -e 1                  0.60 ms per start
ucode big.uc  (2000 stmts)  1.40 ms per start
./hello.uc.bin               0.97 ms per start
```

A start-up is sub-millisecond either way on this machine; what parsing costs grows with is the size of the program, and a deployment that starts a large script once per event is the case where precompiling earns its keep.

As a library

The fourth shape has no `ucode` process at all: a host links `libucode`, holds a `uc_vm_t` in its own memory, and compiles or loads programs through the API of part IV. What has to be installed is the shared library — `libucode.so` with its soname, which is what an embedder links against, its `RUNPATH` set to `@loader_path/../lib` so that it finds its own modules — and, when the host compiles at run time, the headers, which the install rule places under `include/ucode/` and which is what the `libucode-dev` and `ucode-dev` packages exist to carry.

Two costs decide the shape's economics, and both are small enough that the decision is usually about state rather than speed. Starting a machine, loading the standard library into its scope and running a program that loops five hundred

times costs about a sixth of a millisecond, and re-using one machine across many runs saves little of that — two thousand runs of the same compiled program on a debug build:

```
fresh vm per run:  0.1609 ms
reused vm per run: 0.1527 ms
```

The saving of holding a machine is not in the milliseconds; it is that a machine that stays alive keeps its scope, its loaded modules, its caches and its open resources, and can therefore answer the second request without paying the whole cost of getting to know the system again. A host that creates one machine per request and destroys it afterwards has a deployment that is simple and leak-resistant, and is paying the start-up of a module load on every request; a host that shares one machine between threads has to serialize it, since a machine is not re-entrant — chapter 42 and chapter 50 are about exactly that trade and about the audit that tells an embedder which side of that trade it is on.

Resident services

The fifth shape is a daemon that embeds the interpreter and runs for as long as the system does: uhttpd serving requests through scripts (chapter 53), uwsd holding sockets and handlers in one process (chapter 54), rpcd answering bus calls from scripts under `/usr/share/rpcd/ucode/` (chapter 55), and netifd running protocol handlers in scripts. The plain-ucode version of the same idea is a script that loads `uloop`, subscribes to something on the bus, and never returns:

```
uloop.run();
```

The consequences that matter for deployment are that a fault in a script is a fault in the service rather than a failed command, that a long-lived process holds its memory and its file descriptors and therefore has to be written to let them go — the discipline of chapter 42, with its checklist for a reused virtual machine, is written for this shape — and that configuration and code arrive at different times, so a service normally needs a way to be told that a script changed: re-reading a directory of handlers, or a bus call that reloads them, or simply a restart. Between two runs of a standalone script nothing survives; inside a resident service a great deal survives, and deciding which of the two you want is mostly deciding where your state is allowed to live.

Choosing

Shape	What must be installed	Start-up	State between runs	Reporting an error	Who reaps it
Source and interpreter	`ucode`, plus the modules imported	Parse each time, sub-millisecond for a small file	None	Status 254 or 255, message with source line	The caller
Shebang script	The same, and the interpreter findable from the shebang	Same	None	Same	The caller
Precompiled program	`ucode`, and its sources for good messages	No parse	None	Depends on debug information	The caller
Linked library	`libucode`, plus headers when compiling at run time	About 0.16 ms per machine, less re-used	As much as the host keeps	Return status and the raised value	The host
Resident service	The same, plus whatever the host needs	Paid once	Everything, until it exits	Service-wide; a fault is a service fault	Supervision

The repository shows the rule of thumb in its own deployments. A ruleset generator runs once per action and is a template, started fresh, and is the first shape: firewall4's four template files and its helper module of chapter 56 are the case, and its statelessness is a virtue. A board's configuration change is a script run once and is the second shape. A management interface that answers many requests a second is the fifth, which is why the LuCI back ends of chapter 57 are installed as scripts loaded by a server rather than as scripts run per request. And anything that must not depend on a toolchain on the target, or that must not pay a parse per invocation, is the third shape, at the price of having to be rebuilt whenever it changes.

Two further facts round out the picture. A device image chooses its module set package by package, so a script's `import` is a package dependency and not a build-time fact — chapter 59 has how the packages split. And a host that links the library must find it: the build sets an `RUNPATH` on the library, the OpenWrt package installs the development headers for embedders, and there is no exported package manifest or pkg-config file, which is why the examples in `examples/` link with a plain `-lucode` and a search path.

Reading on

- Chapter 2 — obtaining and installing the interpreter, and what a package contains.
- Chapter 5 — strict mode, and why it is a parse-time setting.
- Chapter 16 — templates and the delimiters; chapter 20 covers `utpl`, which is template mode chosen by name.
- Chapter 17 — the module search path, `-L`, and importing a `.uc` library.
- Chapter 20 — `ARGV`, `SCRIPT_NAME` and the start-up environment.
- Chapter 42 and chapter 50 — the shared-machine trade and the audit that shows it.
- Chapter 47 — the precompiled file format, debug information and `-s`.
- Chapter 42 — the state a reused virtual machine keeps, and the checklist for a host that keeps one.
- Chapters 53 to 58 — the resident services this chapter points at.

uhttpd: ucode as a web backend

Source files referenced in this chapter: upstream `openwrt/uhttpd` — `CMakeLists.txt`, `ucode.c`, `uhttpd.h`, `main.c`, `examples/ucode/handler.uc`, `examples/ucode/dump-env.uc` — read from revision `373145f72c88` of the `master` branch, committed 2026-08-24, whose line numbers these are; Appendix G gives an address for each of them. A handler needs a machine running uhttpd; the pieces written in the language on its own are readable and runnable anywhere this interpreter is.

uhttpd is a small HTTP server that treats ucode the way it treats CGI: a handler is a script, and a request is answered by a process that writes its response to its standard output. The difference from a plain CGI setup is where the interpreter lives. uhttpd embeds it in its own address space as a loadable plugin, and it creates the virtual machine once per configured URL prefix when the plugin initialises, keeps that machine alive for the life of the daemon, and runs every matching request inside it. Everything about how a ucode handler is written follows from that sentence: the file's top level is an initialisation phase that happens once and whose values stay, and the per-request part is a callback that receives one fresh object and runs in a machine that has been running since before the first request.

The prefix table and the callback

The plugin is `uhttpd_ucode`, built from the single file `ucode.c` when the project is configured with `UCODE_SUPPORT`, which defaults to `ON` (`CMakeLists.txt:13,77-81`). Handlers are declared in pairs of command-line flags: `-o` gives a URL prefix and `-0` the handler file for it, and both may be repeated in pairs (`main.c:159-162`, collected by `add_ucode_prefix()` at `main.c:255-270`). OpenWrt's init script expresses the same through a UCI list option, `ucode_prefix`, whose entries are `prefix=handler-file` strings translated into those flag pairs, and which is only consulted when the plugin object is present on the system — a configuration that is quietly inert on a build without the plugin is a fact worth knowing before debugging a handler that is never called.

A request whose path matches a prefix is dispatched to that handler: the first matching prefix wins, the remainder of the path after the prefix becomes `PATH_INFO`, and the part after `?` is kept separately (`ucode.c:329-387`, `389-416`). The dispatch is CGI-shaped — a per-request process created through the same machinery that serves `.cgi` files — so a handler that dies takes its request with it rather than the server; a request for which no prefix matches, and a handler that fails to start, are answered with a `500` of uhttpd's own making.

A handler file must provide a global function named `handle_request`, which is the name the plugin looks for (`UH_UCODE_CB` at `ucode.c:28`), and it is called with one argument, the request object, once the plugin has run the file's top level exactly once (`ucode.c:230-318`, execution at `:301`). The shipped example is a whole handler in five lines, and it shows both halves of the shape at once:

```
{%

'use strict';

global.handle_request = function(env) {
        include("dump-env.uc", { env });
};
```

The assignment to `global.handle_request` is what the plugin needs; the include inside the function body is what makes the per-request work happen in a template, and `{ env }` is the scope-passing form of `include()`, which is how a template that names `env` gets hold of the value — chapter 16 owns the three forms of that call and this is the third of them.

Because the top level runs once, a handler can compute anything that costs something to compute — a parsed configuration, a prepared lookup table, an open bus connection — and the second half of the arrangement is that nothing about it is reset between requests. Chapter 42 makes the same argument for a host that reuses one machine and chapter 50 measures what a long-lived machine keeps; in a web handler those effects are the first things to look for, because the machine accumulates for as long as the service does and one handler file is the whole of the accumulation's source.

The request object

The object handed to the callback is assembled per request (`ucode.c:329-387`) and carries four kinds of thing: `PATH_INFO`, the part of the path after the matched prefix; the CGI-style variables that uhttpd hands a script process — the request method, the query string, the content length and the `HTTP_`-prefixed headers — copied out of the same table a CGI process would see; `HTTP_VERSION` as a floating-point number, computed as `0.9 + version / 10.0` (`ucode.c:371-372`); and `headers`, an object of the parsed request headers in their own casing (`ucode.c:374-377`). It is a plain dictionary, which means it can be walked, and the shipped companion template does exactly that in order to dump a request:

```
<h1>Headers</h1>

{% for (let k, v in env.headers): %}
<strong>{{ replace(k, /(^|-)(.)/g, (m0, d, c) => d + uc(c)) }}</strong>:
{{ v }}<br>
{% endfor %}

<h1>Environment</h1>

{% for (let k, v in env): if (type(v) == 'string'): %}
<code>{{ k }}={{ v }}</code><br>
{% endif; endfor %}

{% if (env.CONTENT_LENGTH > 0): %}
<h1>Body Contents</h1>

{% for (let chunk = uhttpd.recv(64); chunk != null; chunk = uhttpd.recv(64)):
%}
<code>{{ replace(chunk, /[^[:graph:]]/g, '.') }}</code><br>
{% endfor %}
{% endif %}
```

Three things in that file are worth taking as instruction rather than as example. Header names arrive in the dashed upper-case convention and the template turns one into a title with a regular-expression substitution carrying a function replacement, which is the idiom chapter 21 gives for `replace()`. Only strings are listed from the environment, because a walk over the object also yields numbers and a dump that printed `0` where a variable is unset is a dump that misleads. And the body is read in chunks of sixty-four bytes until a read returns null, because the body arrives as a stream and not as a value; there is no whole-body member on the request object, and a handler that wants the body as a value has to assemble it.

The `uhttpd` object

The plugin installs one global under the name `uhttpd` (`ucode.c:248-256`), and its members are the handler's whole interface to the connection:

Member	Behaviour
`send(...)`	Writes each argument to the standard output: strings as they are, anything else through its string conversion. Returns the number of bytes written.
`sendc`	An alias of `send`, registered as a second name for the same function.
`recv(len)`	Reads up to `len` bytes of the request body, or the rest of it with no argument; blocks with a one-second poll; returns a string, or null once the body is exhausted.
`flush()`	Flushes the standard output.
`urlencode(s)`	Percent-encodes.
`urldecode(s)`	Percent-decodes.
`docroot`	A read-only string with the configured document root.

The response is what the handler writes to its standard output, and the convention is the header-block convention of the shipped examples: a `Status:` line, one or more headers, an empty line, and the entity. The `Status:` line is not part of the protocol; it is the convention by which uhttpd's CGI machinery learns the status, and the dump template begins with `Status: 200 OK` for that reason. A handler that forgets the blank line has produced headers all the way down, and a handler that writes before them has produced a response that cannot be understood; there is no diagnostic for either case, because at that point the server is merely a pipe.

The `Status: 500 Internal Server Error` that an exception produces is generated by the plugin's own exception handler, which adds the exception class and the first frame of the trace to the body (`ucode.c:196-222`); an `EXIT` exception — what `exit()` raises — is deliberately not reported. The consequence for a handler author is comfortable: an uncaught error in a handler is a status `500` with a class name rather than a hung or half-written response, and the detail is in the server's log rather than in the client's page.

The pieces that can be exercised without the server can be exercised here, and it is worth seeing that the encoding members have plain ucode counterparts for the cases that matter:

```
let path = "/api/status?name=a b&tag=%2f";
let qs = split(path, /\?/)[-1];

print(qs, "\n");
print(join(" ", map(split(qs, /&/), p => replace(p, /%([0-9a-f]{2})/gi,
        (m0, h) => chr(+`0x${h}`)))), "\n");
```

```
name=a b&tag=%2f
name=a b tag=/
```

Templates as the page layer

The arrangement in the shipped pair — a handler that assigns the callback and delegates to a template, and a template that is mostly literal text with control blocks — is the one the plugin is built around, and its logic is that a page is text with holes in it rather than a program that concatenates. The template's literal text is the response because in template mode text outside the delimiters goes to the standard output, and the plugin therefore points the machine's output stream at the standard output before serving and at the bit bucket while the handler's top level runs (`ucode.c:295-303,320-323`), which is why `printf()` in a handler's initialiser is invisible rather than leaked into a page.

Chapter 16 has the mode itself; the parts that are specific to being a page layer are these. A template included with a scope can name values that are not its own, which is the route by which `env` reaches the page, and the same mechanism means that a page included from two handlers with different scopes can see different values under one name. A page that needs the request body reads it through `uhttpd.recv()`, so a page which is included twice in one request would find the body already drained — the body is a stream belonging to the request, not a value belonging to the scope. And a template can be the handler file itself, since a template can assign `global.handle_request` inside a code block just as readily as a raw script can; the shipped example keeps the two apart, which is the arrangement to copy, because a file whose literal text is a page has an initialiser that emits a page when the plugin runs it once.

Per-request semantics

What a fresh request object resets is the request: a handler gets new path information, new headers and a new body on every call, and cannot see the previous request's copies of them. What it does not reset is the machine, which is the same instance holding the same scope, the same loaded modules, the same registry of named values and whatever a module keeps in itself. Three concrete results:

- A variable assigned at the top level of the handler file is computed once and read on every request; that is the intended way to prepare things, and it is also how a handler keeps growing.
- A counter incremented inside the callback is a request counter with no lifecycle at all, and there is no place in this arrangement where it is reset — a handler that accumulates without intending to is holding a dictionary in a top-level binding that grows by request.
- A module imported by the handler file is loaded once, and any state it keeps is per prefix: two prefixes served by two handler files in two machines get two independent instances, and the same is true of the registry, which chapter 42 describes as per machine rather than per process.

The scope-lifetime question that chapters 42 and 50 work through for embedders is therefore settled here in one particular way: the scope lives as long as the daemon, and the per-request work runs in it. An embedder who wants a per-request scope has to construct one — a child scope per request, entered and left around the callback — and uhttpd does not, because it buys the cheapness of an already-warm machine with the retention of that machine's data. Two consequences of that choice are worth naming for anyone hardening a deployment: memory that a handler leaks is reclaimed only by restarting the server, so a long-lived handler wants an occasional look at `debug.memstats()`; and a handler's initialiser runs at daemon start rather than per request, so a startup failure there is a plugin that fails to initialise and a daemon that will not start, not a page that does not render.

Configuration and limits

Beyond the two flags, the knobs that bear on ucode handlers are few: `script_timeout`, `-t`, applies to the script handlers, including these; the document root is exposed to a handler as `uhttpd.docroot` and is the value that decides which files a handler's own path arithmetic can reach; and the number of worker processes the daemon runs determines how many copies of each prefix's machine a deployment holds, which is the multiplied form of every retention figure this chapter has discussed. What the daemon's tree leaves open is worth recording with it: the interaction between the one-second poll inside a body read and the script timeout is not documented anywhere in the daemon's tree, so a handler that reads a body a client never finishes sending should be assumed to be able to occupy a process for the timeout's duration; and the daemon carries no manual page for any of this, the usage text printed by `-h` being the closest thing to one.

Reading on

- Chapter 16 — templates, the three forms of `include()` and what a page is made of.
- Chapter 17 — the module search path a handler file uses to import its libraries.
- Chapter 42 — the machine's registry, the host-side store that lives as long as the machine.
- Chapter 42 — the shared-machine trade, in the embedder's terms.
- Chapter 42 — the state a reused virtual machine keeps, and how a host lets go of it.
- Chapter 50 — the scope-lifetime audit, applied to a long-lived handler.
- Chapter 54 and chapter 55 — the other two embedded servers, one of them keeping sockets open across requests and the other answering bus calls.

uwsd: a persistent ucode web server

Source files referenced in this chapter: upstream `jow-/uwsd` — `script.c`, `include/script.h`, `include/config.h`, `config.c`, `main.c`, `example/chat.conf`, `example/handler.uc`, `example/chat-server.uc` — read from revision `1450021ba05a` of the `master` branch, committed 2026-09-10, whose line numbers these are; Appendix G gives an address for each of them. A handler of the shape shown here needs a running uwsd to be connected to.

uhttpd, the previous chapter's host, keeps the interpreter at arm's length: a request becomes a process that writes a response, and nothing of the machine remains when the process is gone. uwsd turns that around. It is a single-process HTTP and WebSocket server and proxy whose routing logic is a ucode script, its connections are values a script can hold on to, and its interpreter keeps running between events. Its build treats the language as a hard dependency rather than an optional feature — the configuration header includes the VM header directly (`include/config.h:26`) and the project links the library unconditionally — which is the clearest possible statement of how central the interpreter is to the thing.

A uwsd deployment runs one server process and, for each configured script, one worker process holding that script's machine: when the environment carries `UWSD_WORKER_SOCKET` and `UWSD_WORKER_SCRIPT` the binary becomes a script worker rather than a server (`main.c:81-85`), so a script's fault costs the one worker and the connections routed to it, not the listening process. The worker's virtual machine is created once, in `script_context_run()` (`script.c:2148`), with `uc_vm_init(&ctx.vm, NULL)`, an exception handler, the standard library loaded into its scope and the server's own API installed, and it then runs the bootstrap program described next and stays in its event loop.

The bootstrap and its five callback slots

The script a deployment names in its configuration is not run directly. The host generates a short bootstrap around it (`script.c:2159-2163`) which reads, in the host's own quoting, as:

```
{%
import * as cb from '<the configured script path>';
return [ cb.onConnect, cb.onData, cb.onRequest, cb.onBody, cb.onClose ];
%}
```

Two things in three lines carry the whole contract. The import is the aliased wildcard form of chapter 17, so the script is a module: its exported bindings arrive as the members of `cb`, and its file-scope code runs as an import does, once per worker rather than once per event. And the return value is a positional array of five slots, which the host stashes as its own dispatch table after recording it in the machine's registry under the name `uwsd.cb` (`script.c:2189`). The five slots are read by position (`script.c:2191-2195`), and none of them is required: a name the script does not export is simply read as `null`, and every call site tests its slot before invoking it (`script.c:`

`317,440,548,592,769`). A script exporting one callback is therefore a perfectly valid script which handles one kind of event.

What is optional is the coverage rather than the shape, and what each omission costs is set by the guard at the call site rather than by any general rule. `onData` returning without a handler (`script.c:440`) leaves the frames of an open WebSocket being read and thrown away, so the socket stays up and silent; an absent `onBody` does the same to the entity of a streamed request (`script.c:769`); an absent `onClose` skips the notification (`script.c:548`); and an absent `onConnect` lets the upgrade complete untouched, which is a WebSocket accepted with no subprotocol agreed (`script.c:317`, `script.c:386-396`). Only one absence is loud: the host answers an HTTP request with a status 501 and the text `Backend script does not implement an onRequest() handler.` when there is no `onRequest` (`script.c:619-629`). That asymmetry is worth knowing, because it is the difference between a handler script which appears to work and one which is visibly answering nothing.

The language's own contribution to that arrangement is small enough to write out. This is the bootstrap's expression over a script which exports only `onData`, with the host's guard applied to each slot:

```
// the namespace `import * as cb from '<script>'` would yield for a partial
script
let cb = {
        onData: function (conn, message) { return `got ${message}`; }
};

let slots = [ cb.onConnect, cb.onData, cb.onRequest, cb.onBody, cb.onClose ];

printf("slots: %d\n", length(slots));

for (let i = 0; i < length(slots); i++)
        printf("%d: %s\n", i, slots[i] ? "handler" : "null");

let event = 2;
printf("dispatching an event to slot %d: %s\n", event,
        slots[event] ? slots[event]("c1", "x") : "501 Not Implemented");
```

```
slots: 5
0: null
1: handler
2: null
3: null
4: null
dispatching an event to slot 2: 501 Not Implemented
```

The array is not short and is not an error, it is complete and mostly empty; the emptiness is what the guards above turn into behaviour.

The five callbacks and the arguments each receives are fixed by the call sites: `onConnect` with the connection and the offered subprotocols (`script.c:340-344`), `onData` with the connection and the payload — a string buffer or a decoded object for a WebSocket message, or a string and a final flag for streamed data (`script.c:465-469,504-518`) — `onRequest` with the connection, the method and the request target (`script.c:593-598`), `onBody` with the connection and one chunk of the entity (`script.c:772-776`), and `onClose` with the connection and, for a WebSocket close, the status code and reason (`script.c:552-562`). The shape of the host's side is a table of slots looked up by name and invoked with a connection as the first argument throughout, which is the same pattern a plain ucode program uses for any set of hooks:

```
let cb = {
        onConnect: function (conn, protocols) { return `connect ${conn} $
{protocols}`; },
        onData: function (conn, data, final) { return `data ${conn} ${data} $
{final}`; },
        onRequest: function (conn, method, uri) { return `request ${conn} $
{method} ${uri}`; },
        onBody: function (conn, chunk) { return `body ${conn} ${chunk}`; },
        onClose: function (conn, code, msg) { return `close ${conn} ${code} $
{msg}`; }
};

print(cb.onConnect("c1", [ "chat" ]), "\n");
print(cb.onData("c1", "ping", true), "\n");
print(cb.onRequest("c1", "GET", "/index.uc"), "\n");
print(cb.onBody("c1", "chunk"), "\n");
print(cb.onClose("c1", 1000, "bye"), "\n");
```

```
connect c1 [ "chat" ]
data c1 ping true
request c1 GET /index.uc
body c1 chunk
close c1 1000 bye
```

The shipped handler shows the same discipline with the language's own conveniences, at `example/handler.uc:10-26` ; it needs a host to run in, since every path in it goes through the host's objects, and it is reproduced for its shape — a guard on the offered protocols, per-connection state hung on the connection itself, a re-arming timer, and an explicit accept:

```
export function onConnect(connection, protocols)
{
        warn(`Connect! ${connection} ${protocols}\n`);

        if (!('shell' in protocols))
                return connection.close(1003, 'Unsupported protocol
requested');

        connection.data({
                counter: 0,
                n_messages: 0,
                n_fragments: 0
        });

        timer(1000, () => timer_cb(connection));

        return connection.accept('shell');
};
```

The `export` keyword is what makes a binding visible to the aliased import, chapter 17 has it, and `connection.data({...})` is the idiom by which a handler attaches its per-connection state to the connection rather than to a table it would have to maintain itself.

What can stop a worker

An absent callback is a handled case, so the ways a script worker fails to serve are few and all of them belong to loading the script rather than to dispatching events. The first is that the configured path cannot be resolved by the import, which the compiler reports against the generated bootstrap: a script named by a path that does not exist fails before the worker ever reaches its loop.

```
import * as cb from './handler.uc';
```

The second is that the script is found but does not compile: the import compiles the module as part of running the bootstrap, so a syntax error inside the script arrives as an error of the bootstrap and names the script's own path in the process, which is the one case where the generated wrapper does not obscure the fault. The host then reports `Failed to compile handler script:` (`script.c:2169-2172`) and the child exits without entering the loop. The third is that the program runs and then leaves the OK status: a top-level `exit()` prints `Handler script exited with code N` and the worker takes that as its own status (`script.c:2201-2205`), and any other status or a fault of the bootstrap itself ends the child without a message (`script.c:2208-2209`). Exceptions raised later, from inside a callback, are a different matter: those are answered per event, which the events section below covers.

The practical reading is that a deployment which serves nothing has usually mistaken its path rather than its exports, and that the diagnostic to look for is the compiler's, naming either the bootstrap or the script.

Routing

Configuration decides which connection reaches which script, and its shape is a listener block holding matches around backends; from `example/chat.conf:15-23`, a whole routing declaration:

```
listen :8080 {
        match-protocol http { use-backend chat-client; }
        match-protocol ws   { use-backend chat-server; }
}
```

The match axes the configuration understands are the protocol — `http` or `ws` (`config.c:851-864`) — the hostname (`config.c:867-874`) and the request target, and a backend is a named bundle of actions drawn from `serve-file`, `serve-directory`, `run-script`, `use-backend`, `proxy-tcp`, `proxy-udp` and `proxy-unix` (`config.c:175-188`). A script backend is a `run-script` with a path, plus an `environment`, a `ws-message-format` drawn from `raw`, `buffered` and `json`, and a `ws-message-limit` (`config.c:286-288`, `example/chat.conf:7-13`). The message format is the setting that decides what `onData` receives, and the three choices are worth being exact about: `raw` hands over the frame as it came, `buffered` re-assembles a fragmented message into one string (`script.c:834-846`) and `json` decodes the payload into a value before the callback sees it (`script.c:504-508`) — a script written against one of the three is not a script that runs under another, and the format is a property of the deployment rather than of the script.

The `uwsd` object and its resources

The API the host installs before running the bootstrap is one global, `uwsd`, with four members (`script.c:2133-2145`), and two resource types declared alongside it (`script.c:2187-2188`):

Name	Kind	Notes
`uwsd.connections()`	function	The live connections, which is how a handler reaches every client it has rather than only the one it was called for.
`uwsd.sha1digest(data)`	function	The digest the WebSocket handshake needs, exposed rather than required from a module.
`uwsd.uuid()`	function	An identifier, for connection or request labels.
`uwsd.spawn(...)`	function	Starts a child process and yields a handle for it.
`uwsd.connection`	resource	One client connection.
`uwsd.spawn`	resource	One child process, with `stdin`, `stdout` and `close`.

The connection type carries both halves of the request and the handler's controls over the socket. Its request fields, assembled into the object the handler sees (`script.c:1855-1912`, created at `:2073`), are the local and peer address and port, the TLS flag and cipher, the peer certificate's issuer and subject, the HTTP version, the method, the request target and the headers; its methods (`script.c:1297-1312`) are `version`, `protocol`, `method`, `uri`, `header`, `info` for reading those, and `data`, `store`, `reply`, `accept`, `expect`, `send` and `close` for acting. The method names say the shape of the model, which is a great deal closer to a socket than to a request-response pair: `accept` and `close` are the endpoints of a negotiated connection, `expect` declares what the handler wants handed to it next, `data` is the association of state to the connection, and `reply` is the one member that speaks HTTP.

`reply` is worth its own paragraph because it is where the two worlds meet. It takes a header object and builds a whole response from it, reading a `Status:` entry for the status line, defaulting the media type to `application/octet-stream` when none is given, and patching a placeholder in the content length once the body length is known (`script.c:1169-1295`). A handler that answers an HTTP request therefore composes a description of a response rather than writing bytes, which is the opposite of uhttpd's convention of writing to a standard output, and is what makes the same handler able to serve a WebSocket frame with `send`. Resources are resources: chapter 45 is about how a host declares a type with a close callback and a method set and about what it means for the lifetime of the thing, and everything this section says about `uwsd.connection` and `uwsd.spawn` is an application of it — the close callbacks registered at `script.c:2187-2188` being the reason a connection whose last reference is dropped is a connection that gets closed rather than leaked.

The broadcast pattern the shipped chat server shows is the clearest illustration of what the resource inventory buys, quoted from `example/chat-server.uc:19-24`:

```
function broadcast(msg) {
        for (let client in uwsd.connections())
                unicast(client, msg);
}
```

with the single send composing its payload as an interpolated object, so that the JSON encoding is the language's business and the framing the server's.

Events, timers and children

Nothing in a uwsd handler blocks the loop, because the handler is the loop's callback: control returns to the server between events, and a handler that wants something to happen later arranges for it rather than waiting for it. Timers are the host's `timer()` — the shipped handler re-arms one of a second on each expiry, which is the idiom for periodic work in this shape — and a child process is arranged for through `uwsd.spawn()`, whose handle is a resource whose `stdin` and `stdout` members are the ordinary file handles of chapter 25, so the output of a child arrives as another event rather than as a blocking read. The event-loop discipline is the one chapter 36 lays out for `uloop`: keep callbacks short, do not wait inside one, hold on to what you registered, and remember that a callback which retains its arguments retains them until the next event replaces them — with the added obligation of a script that runs for days that a connection's state must be released with the connection, which is what the close callbacks on the resources do for the host's own side and what letting go of the connection value does for the script's.

From a uhttpd handler to a uwsd handler

The two arrangements answer the same question in opposite directions, and the differences are all in what has a lifetime.

	uhttpd handler	**uwsd handler**
Entry point	`handle_request(env)`, looked up by name	Up to five callbacks, exported and collected positionally by a generated bootstrap
The request	The argument: a per-request object of path information, CGI-style variables and headers	Held by the connection: address, TLS and request members read through its methods
The response	Bytes written to the standard output, with a `Status:` header by convention	`connection.reply(headers)` for a response, `connection.send()` for a frame
Continuity	None; a request is a process	Total; one machine per worker, holding its scope, its modules and its open connections
State	Recomputed per request	Attached to the connection with `data()`, or held in the machine between events
Failure	A status 500 for that request	The worker, and the connections routed to it

Written as advice: a uhttpd handler is a function that is told about a request and finishes it, and moving the same logic to uwsd is mostly the work of deciding where its state lives, since the continuity that makes uwsd able to hold a socket open is the same continuity that makes a counter, a cache or a table of connections persist without anybody having chosen to keep them. Chapter 53's per-request section and chapter 50's worked example of a reused machine read as the two halves of that sentence, and chapter 42's checklist is what a uwsd handler wants for the same reason it wants a test.

Reading on

- Chapter 17 — module import, `export`, and the aliased wildcard form the bootstrap relies on.
- Chapter 25 and chapter 30 — file handles, and starting a child and talking to it.
- Chapter 36 — the event-loop discipline these handlers live under.
- Chapter 45 — how a host declares a resource type, and what a close callback guarantees.
- Chapter 42 — the state a reused virtual machine keeps, which is where the callback table and every connection's state live between events.
- Chapter 53 — the other shape of embedded web server, with its per-request semantics.

rpcd: ucode as an ubus service

Source files referenced in this chapter: upstream `openwrt/rpcd` — `ucode.c`, `examples/ucode/example-plugin.uc` — read from revision `e37ed9d81469` of the `master` branch, committed 2026-07-19, whose line numbers these are; Appendix G gives an address for each of them. For the side of the arrangement that is language rather than daemon, the repository's own `ubus` module, the `ubus` command-line client and a bus daemon. A provider or a client of the kind shown below needs a bus to speak on: an OpenWrt system runs one by default, and elsewhere `ubusd -s /tmp/bus.sock` starts one.

rpcd is a process that publishes a set of remote procedures on the message bus: it is the thing that answers `ubus call` for the higher-level services of a device, and it is built as a core plus loadable plugins. One of those plugins is written in the language rather than using it: `ucode.c` turns a directory of ucode files into bus objects, so that a device service is a script dropped into a directory rather than a C program compiled into the daemon. For a book about this language that is the arrangement with the most to explain, because the script does not merely get called — it *declares* the interface it will be called through, in a value the host reads back out of the machine.

What the plugin loads

The directory is fixed at build time: `RPC_UCSCRIPT_DIRECTORY` is the install prefix with `/share/rpcd/ucode` on it (`ucode.c:35`), which is why OpenWrt speaks of `/usr/share/rpcd/ucode/` . At plugin initialisation the code opens the directory and runs every regular file in it (`ucode.c:1084-1098`), with a security check on the way: a file writable by anyone but its owner is skipped with a warning (`ucode.c:1090`), because a world-writable file in that directory is a way to get code into a privileged daemon. Before any script runs, the plugin re-opens the interpreter's own shared object globally (`ucode.c:1076`) and initialises the module search path (`ucode.c:1082`) — the first of those is what lets a native extension module imported by one of these scripts resolve the interpreter's symbols, which is a detail worth knowing when a `require` of a compiled module succeeds under one host and fails under another.

Each script gets a virtual machine of its own. Its parse configuration is fixed (`raw_mode` on, block stripping on, strict declarations off — `ucode.c:84-89`), its scope gets the standard library (`ucode.c:961`) and the thirteen status constants (`ucode.c:947-959`), and the plugin declares into it a request type and a deferred type of its own (`ucode.c:963,965`). A script that imports the `ubus` module gets the module's own richer set of status names alongside those — the module exposes fifteen of them plus an access-control marker, so the constants a script sees depend on whether it reached for the module; using the module's spelled forms avoids depending on which subset the host injected.

The signature: a script that declares its own interface

The script's return value is the whole of its configuration. After running a file the plugin takes the value and stores it in that machine's registry under the name `rpcd.ucode.signature` (`ucode.c:1012`); a later registration pass reads it back and builds the bus objects (`ucode.c:687-798`), one object per top-level key, each with the type name `rpcd-`

`plugin-ucode-` followed by the object's key (`ucode.c:767`), its methods attached one by one (`ucode.c:665`), and the object added to the bus (`ucode.c:791`). There is no configuration file for this plugin: the returned dictionary is the config format, and there is accordingly no `config/` and no JSON file in the plugin's area of the tree.

The shape is a three-level dictionary — object name, then method name, then a descriptor — and the validator (`ucode.c:564-641`) accepts exactly that: a `call` member that is callable, and an optional `args` member whose values are *type hints* rather than defaults. The hint is read for its type, and the mapping is mechanical (`ucode.c:666-724`):

Hint value in `args`	**Wire type**
the integer 8, 16 or 64	an eight, sixteen or sixty-four bit integer
any other integer	a thirty-two bit integer
a boolean	an eight bit integer
a string	a string
a double	a double
an array	an array
an object	a table

That is why the shipped example's hints read as odd numbers: `foo: 32` and `bar: 64` are not defaults, they are the way an author says "thirty-two bit integer" and "sixty-four bit integer". Arguments that a method did not declare are rejected with the invalid-argument status, with one exception for the session name the bus attaches for accounting (`ucode.c:240-328`). The validation happens on the way in, before the callback runs, so a malformed call is a status rather than an exception.

Here is the shipped example, abridged from `examples/ucode/example-plugin.uc` ; it cannot run in this environment, because it needs the plugin to provide the request type, the status constants and the bus:

```
'use strict';

let ubus = require('ubus').connect();

return {
        example_object_1: {
                method_1: {
                        call: function() {
                                return { hello: "world" };
                        }
                },

                method_2: {
                        args: {
                                foo: 32,
                                bar: 64,
                                baz: true,
                                qrx: "example"
                        },

                        call: function(request) {
                                return {
                                        got_args: request.args,
                                        got_info: request.info
                                };
                        }
                },

                method_4: {
                        call: function() {
                                die("An error occurred");
                        }
                }
        },

        example_object_2: {
                method_a: {
                        args: { number: 123 },

                        call: function(request) {
                                request.reply({ got_number:
request.args.number });
                        }
                }
        }
};
```

Four things in it are the whole of the host's contract. The `return` at top level is unusual in a script and is the hinge of the design. A method may answer by returning a plain object, or by calling `reply()` on the request object it is handed — the file shows both, and a method that does neither leaves the caller waiting for the guard timeout. `request.args` is the call's named arguments and `request.info` is the metadata of the call, with the caller's user and group under `acl`, the object's identification under `object` and the method name beside them (`ucode.c:384-408,452-460`), which is what makes an in-script authorisation check like `method_3`'s comparison of `request.info.acl.user` possible. And an error is expressed by `exit()` with a status, or by an ordinary runtime exception, which the host catches and turns into the unknown-error status with a diagnostic on its standard error (`ucode.c:466-531`).

The same interface without rpcd

The mechanism underneath — declaring objects, methods and argument types to a bus from ucode — is not rpcd's property: the `ubus` module does it directly and chapter 37 documents it, which makes it the place to see the shapes move. Below is a provider that publishes an object with one declared method and then sits in the event loop, the way the plugin's registered objects sit in rpcd's loop:

```
import * as ubus from "ubus";
import * as uloop from "uloop";

let conn = ubus.connect("/tmp/ucode-ch55-bus.sock");

if (conn == null) {
        die("connect: " + ubus.error() + "\n");
}

let obj = conn.publish("demo.echo", {
        echo: {
                args: { text: "" },

                call: function (req) {
                        req.reply({ echoed: req.args.text });
                }
        }
});

if (obj == null) {
        die("publish: " + ubus.error() + "\n");
}

print("serving demo.echo\n");

uloop.run();
```

The provider's declaration is the same three-level shape rpcd's signature uses — a method name, an `args` dictionary of hints, a `call` function — and it resolves the same way on a call. Started against a bus on a socket, the object appears in the enumeration and answers a call; a call carrying an argument the method did not declare is refused before the callback runs, which is the behaviour rpcd's own validator implements:

```
$ ubus -s /tmp/ucode-ch55-bus.sock list | grep -x demo.echo
demo.echo
$ ubus -s /tmp/ucode-ch55-bus.sock call demo.echo echo '{"text":"hi"}'
{
        "echoed": "hi"
}
$ ubus -s /tmp/ucode-ch55-bus.sock call demo.echo echo
'{"text":"hi","spurious":1}'
Command failed: ubus call demo.echo echo {"text":"hi","spurious":1} (Invalid
argument)
```

The client prints its usage text after a failed call, which is why the third transcript is cut at the failure line; the interesting part is the status the call came back with.

The bus used here is one started by hand against a socket path, and the socket path is passed to `ubus.connect()` explicitly; a deployment on a device reaches the system bus without naming a path, since the client library's own default is the run-directory socket. On the calling side the same exchange in ucode is a `call()` on the connection, whose reply's fields land as members of the returned object:

```
import * as ubus from "ubus";

let conn = ubus.connect("/tmp/ucode-ch55-bus.sock");
let res = conn.call("demo.echo", "echo", { text: "hi" }, 2000);

if (res == null) {
        die("call: " + ubus.error() + "\n");
}

print("keys: ", keys(res), " value: ", res.echoed, "\n");
```

```
$ ./build/ucode -L build client.uc
keys: [ "echoed" ] value: hi
```

Two shapes differ between the module and the plugin, and both differences matter to a script author who moves between them. The module's `publish()` is imperative and gives a resource that removes the object when the script lets it go, whereas the plugin's signature declares objects that live as long as the script's machine; and the module's deferred calls are values of a type with `await()` and `completed()`, while the plugin's `defer()` on the request

object is the way a handler releases a call it cannot answer yet (`ucode.c:775-797`), with the reply eventually sent through `reply()` or failed through `error()` (`ucode.c:799-871`). The plugin's guard is a timeout derived from the daemon's execution-timeout setting (`ucode.c:1071`), so a handler that neither returns nor replies is failed rather than left hanging.

Reading the arrangement as a deployment

The rpcd arrangement has properties that follow from the directory rather than from the language, and they are the ones to have in mind when a service behaves surprisingly. A script is loaded when the daemon starts, so installing or changing a file needs a restart of the daemon — the arrangement is not a watcher, and the plugin carries no reload mechanism. A script's file name is irrelevant except as an identity in the daemon's own reporting, which is why several services can live in several files of one directory without colliding, since their object names are what they declare rather than what they are called. A script that throws at load time takes its objects with it rather than the daemon with them: the objects are added in a registration pass over the recorded signature, so a signature that never arrived is a script that contributes nothing, which is a quieter failure than a crash and correspondingly easier to miss — the way to see it is to enumerate the bus and look for the object, which is exactly what the transcript above demonstrates.

On privileges: the plugin hands the caller's user and group to the script and leaves the decision to it, so an authorisation test inside a method is a policy the script chose, and the object's presence on the bus is not by itself a permission; the one argument name the validator lets through undeclared, because the bus attaches it, is the host making the same kind of choice on the script's behalf. On LuCI: the management framework's server-side back ends are installed under this very directory and are written as scripts of exactly this shape, which is why chapter 57 can describe a LuCI application's back end as "a file in the ucode plugin's directory" — the reachability of those objects through the bus and through the framework's own call layer is that chapter's subject rather than this one's.

Reading on

- Chapter 36 — the event loop a provider script sits in.
- Chapter 37 — the ubus module: connections, published objects, deferred calls and the status constants.
- Chapter 53 — uhttpd, which answers HTTP requests instead of bus calls, with a fresh process per request.
- Chapter 54 — uwsd, which keeps connections as values and handlers as exported callbacks.
- Chapter 57 — the LuCI back ends that are installed into this plugin's directory.
- Chapter 45 — resource types and their close callbacks, which is what a published object is.
- Chapter 42 — the state a reused virtual machine keeps, which is the discipline every host in part V needs.

Case study: firewall4

Source files referenced in this chapter: upstream `openwrt/firewall4` — `root/sbin/fw4`, `root/usr/share/ucode/fw4.uc`, `root/usr/share/firewall4/main.uc`, `root/usr/share/firewall4/templates/`, `root/etc/init.d/firewall`, `tests/lib/mocklib/` — read from the `master` branch at revision `c2ae8c8940a8`, committed 2026-08-27, whose file names these are; Appendix G gives an address for each of them. The real thing needs a machine with nftables and the `fw4` package; the miniature reproduction in the middle of the chapter stands in for it, giving the same shapes with the calls taken out.

Firewall4 is the largest program written in this language that ships in OpenWrt, and it is not a daemon. It is a configuration compiler: it reads the board's configuration, and it writes a ruleset on its standard output, and the thing that consumes that output is `nft`. Its core is one module of three and a half thousand lines and an entry file of a couple of hundred, and the interpreter is what joins the configuration to the ruleset. Studying it is worth a chapter of a language manual because it settles, for the largest program in the tree, the questions a program of that size has to answer: how the entry point is shaped, how a big module is organised, how output is produced, and how a program that shells out to a system utility stays testable.

The invocation

The whole of the start path is a shell script, and its heart is one line of it, from `root/sbin/fw4`:

```
ACTION=start \
        utpl -S $MAIN | nft $VERBOSE -f $STDIN
```

Every part of that line is a decision about the language. `ACTION=start` is the argument passing convention: the program is told what to do by an environment variable rather than by an argument, and the entry file dispatches on it without parsing a command line at all. `utpl` is not a second program; it is a symlink to the interpreter which the build creates next to it, and the interpreter looks at the name it was invoked under and enters template mode when the name is that one — which is why the line carries no template-flag option, and why chapter 52 calls a file's name part of its deployment. `-S` turns on strict declarations, so a misspelled name in a ruleset template is an error rather than a null value. `$MAIN` is `/usr/share/firewall4/main.uc`, a template rather than a program, whose text output is the ruleset. And the pipe into `nft -f /dev/stdin` hands the rendered text to the kernel's ruleset loader, so the interpreter never speaks to netlink and never validates a ruleset: the contract between the two halves of the system is text.

Read as a deployment, the shape is the cheapest one there is: a process per action, nothing resident, state only in the files it reads and the one state file it maintains. The procd script delegates into the same command entirely, with `start_service()` running `fw4` with the start action, `stop_service()` running the flush action, and a configuration trigger re-running it when the firewall configuration changes — which is what a configuration compiler wants as a supervision arrangement.

The tree

Seventeen `.uc` files on that branch, of which thirteen ship and four belong to the tests. There are no `.tpl` files at all: the templates are `.uc` files too, because a template in this language is a file in the language rather than a file in a dialect of it.

```
root/
├── sbin/fw4                              the shell entry point
├── etc/init.d/firewall                   the procd service script
├── usr/share/ucode/
│   └── fw4.uc                            the core module, 3455 lines
└── usr/share/firewall4/
    ├── main.uc                           the entry template
    └── templates/
        ├── ruleset.uc   rule.uc          redirect.uc
        ├── mangle-rule.uc                zone-match.uc
        ├── zone-jump.uc                  zone-verdict.uc
        └── zone-masq.uc  zone-mssfix.uc  zone-notrack.uc  zone-drop-
invalid.uc

tests/lib/mocklib/{fs,uci,ubus}.uc      four files of stand-ins
```

The core module is installed under `usr/share/ucode/` rather than under the program's own directory, and that is not tidiness: the interpreter's compile-time search path ends with a `.uc` entry under the share directory, so installing a module there is what lets a file in another directory `require()` it by bare name. There is no build file at the root of the repository at all, because the packaging lives in the OpenWrt tree; what matters here is only where the files land, since that is what the program's imports depend on.

The entry template and its dispatch

The entry file begins with a template delimiter, and its first act is the import of the core module:

```
{%
        const fw4 = require("fw4");
%}
```

The value `require()` hands back for a `.uc` file is the value of the file's last top-level statement, and the core module ends by returning a dictionary of its own making; that is how the module hands its interface to the template. It is also why there is no command-line parsing to read anywhere in the program: the dispatch at the end of the entry file is a switch over the environment, so the actions are the case labels.

```
switch (getenv("ACTION")) {
case "start":
        return render_ruleset(true);

case "print":
        return render_ruleset(false);

case "reload-sets":
        return reload_sets();

case "network":
        return lookup_network(getenv("OBJECT"));
...
}
```

Four things follow from that shape, and all four are worth copying or refusing deliberately in a program of your own. A second action costs one case label and no plumbing. An action's own arguments arrive the same way its name does, as members of the environment, which is why `OBJECT` appears beside `ACTION`. A program whose whole interface is the environment is trivially drivable from a shell script, from a service script and from a test, and correspondingly hard to misuse. And because the interface is not a command line, there is nothing to print as usage text: the actions and their environment variables are documented by the case labels and the reads, which is a real cost of the arrangement and worth knowing before adopting it.

The output layer

A rendered ruleset comes out of templates, and the entry file reaches a template by the scoped form of `include()`, from the renderer of the ruleset itself:

```
function render_ruleset(use_statefile) {
        fw4.load(use_statefile);

        include("templates/ruleset.uc", { fw4, type, exists, length,
include });
}
```

The first argument is a path taken relatively to the including file, which is why the templates sit in a directory next to the entry file rather than needing to be named absolutely; the second argument is the scope of the included template, and what it contains is the interesting part. The included template is given the core module under the name `fw4`, so its own code can ask the module questions, and it is given four built-in functions by name — which is necessary because a template's code does not see the includer's file-scope bindings, only what the scope object hands it, so a template that wants `length()` or a nested `include()` has to be handed them. Chapter 16 works through the three

forms of the call; what matters for the shape of a program this size is that the second form is what makes a template a unit with a declared interface rather than a fragment that happens to run in whatever context surrounded it.

Because the template's text output is the thing the program exists to produce, the two halves of the design fall out cheaply: the configuration work is code, the ruleset grammar is text, and nothing has to be converted between them. The reload action shows the same economy from the other side: it emits raw netfilter commands — flushes of the sets followed by element additions — straight on the standard output, so a set update needs no template at all, only a loop. And the include action shows the same economy applied to a user's files: the action runs a user's include script with a shell function named `config` defined to print a refusal and fail, which is a neat piece of work — a capability is removed from a child by shadowing the command that would have provided it, in a program that is otherwise all text.

The module

`fw4.uc` is one file of three and a half thousand lines, and its first fifteen lines are the whole of how it begins:

```
const fs = require("fs");
const uci = require("uci");
const ubus = require("ubus");

const STATEFILE = "/var/run/fw4.state";

const PARSE_LIST   = 0x01;
const FLATTEN_LIST = 0x02;
const NO_INVERT    = 0x04;
const UNSUPPORTED  = 0x08;
const REQUIRED     = 0x10;
const DEPRECATED   = 0x20;
```

Six observations on those lines, and they generalise. The module imports the native modules it needs at its own top level and holds them in `const` bindings, so nothing downstream has to know how a module is reached. There is no command-line handling, because there is no command line at the module's level. The state file's path is a constant at the top rather than a string repeated at its uses. And the flag group is spelled as bit positions of a constant integer, the way a C header would spell it, because the flags select how the program reads a configuration section: list-valued, flattened, not invertible, unsupported, mandatory, deprecated. A configuration reader of that shape needs exactly this vocabulary of bit flags; that the language has no enumeration facility does not trouble it, because a group of named constants with bit values is the same machine with fewer letters, and chapter 6's bitwise operators are how such flags are read back.

Below the head the file is large lookup tables — the ICMP type name tables for both protocol families, a table of tens of entries each — then the readers of the configuration, then the rendering helpers. Two of its habits are worth naming. Its diagnostics read the environment rather than a flag parameter, so the warning function tests `QUIET` and `TTY` to decide how much to say and whether to colour its output — the same consequence of the environment-driven

interface as at the entry point, arriving again at the other end of the program. And its dialogue with netfilter goes through one small function that assembles a command out of a fixed program path, a terse flag, a JSON flag and the caller's arguments and runs it through a pipe, so the rest of the file never spells a command line: every query the firewall asks of the kernel — the counters, the sets, the flow tables — comes back as JSON and is read with the notations of chapter 15.

Here is the whole of that structure at a scale you can run, with the module standing where the installed one stands and the template doing to it what the entry file does:

```
const fs = require("fs");

const POLICY = {
        input: "drop",
        output: "accept",
        forward: "accept"
};

function describe(chain) {
        return `${chain} -> ${POLICY[chain]}`;
}

return {
        policy: POLICY,
        describe: describe
};
```

```
{%
        const demo = require("policies");
        const mode = getenv("MODE") ?? "print";
%}
# rendered in {{ mode }} mode
table inet filter {
{% for (let chain, pol in demo.policy) { %}
        chain {{ chain }} policy {{ pol }};
{% } %}
}
# last: {{ demo.describe("forward") }}
```

Run them as two files in one directory, with the interpreter invoked under its template name and in strict mode, which is the arrangement of the real program's one line:

```
$ ls
main.uc  policies.uc
$ MODE=apply utpl -S main.uc
# rendered in apply mode
table inet filter {
        chain input policy drop;
        chain output policy accept;
        chain forward policy accept;
}
# last: forward -> accept
```

The miniature leaves out the configuration store, the state file and the kernel, and it keeps every structural decision: a module that ends by returning a dictionary, a template that requires it by bare name and relies on the search path, dispatch on the environment, and text output as the product. One thing it shows that a description can hide: the interpolation delimiters. A template's text is interpolated by the double-brace form, while the dollar-brace form belongs to strings and not to template text, so a template written with the wrong delimiter emits the delimiter rather than the value, three times in a row and with no complaint.

What the case teaches

Five things the program settles for a program of its size, each of which is a language decision as much as an architectural one.

One module and a thin entry. The whole of the logic is in a module and the whole of the interface is a template that imports it and dispatches on the environment. The alternative — a program that is also the library — is what makes shell-embedded logic that cannot be imported by anything else.

Strict mode as the default of a deployment. The one command line passes `-S`. A ruleset generator that silently rendered a null where a name was misspelled would be a firewall with a hole in it, and the flag is how the deployment says so. Chapter 5 has what the flag costs, which is a declaration discipline in a file that reads configuration from a store.

The search path is the packaging contract. Nothing in the invocation passes a module path. The program finds its module because the module is installed at a directory the interpreter was configured to search, and that is the same agreement that governs where a native module's object file lands — chapter 17 for the path, chapter 59 for the packages, and here for the consequence that a program's install layout is a part of its source, not a detail of whoever builds it.

Templates are the output layer, not a view layer. The ruleset is text and the templates emit text, with the module supplying predicates and data to the interpolation. Where a template would have wanted a loop over something the module has not exposed, the design adds a method rather than reaching into a structure — which is what keeps a template of some hundred lines a maintainable artefact.

A program that shells out is testable by standing the shelling out. The program's tests run against mock objects for the three native modules it uses and against fixtures recorded as the JSON that the real utilities answer — that is what the test tree's four mock files are for. A program written that way has its system interfaces at three named seams, which is the same discipline as the command assembler in the module: one place where the outside is touched.

Reading on

- Chapter 5 — strict mode, what it catches and what it costs.
- Chapter 16 — templates, the interpolation delimiters and the three forms of `include()`.
- Chapter 17 — the module search path, what `require()` hands back for a file, and installing a module.
- Chapter 20 — the environment as seen by a program.
- Chapter 38 — the configuration store the module reads.
- Chapter 52 — deployment models, of which this is the configuration-compiler instance.
- Chapter 59 — the packaging that decides where these files land.

Case study: the LuCI ucode runtime

Source files referenced in this chapter: upstream `openwrt/luci` — `modules/luci-base/ucode/` (`uhttpd.uc`, `http.uc`, `dispatcher.uc`, `runtime.uc`), `modules/luci-base/src/lib/luci.c`, `modules/luci-base/Makefile`, the `contrib/package/` module packages and the application packages' `Makefile`s — read from revision `06e111ab07b9` of the `master` branch, committed 2026-09-16, whose paths and line numbers these are; Appendix G gives an address for each of them. A page of the shape this chapter describes needs a device running LuCI; the parts written in the language on its own are readable anywhere.

LuCI is the web management framework of OpenWrt, and its page runtime is written in this language: the request handler, the dispatcher, the HTTP layer, the page templates and a great deal of the application back ends are ucode files. The framework arrived at the language from Lua, and the shape of what it carries still shows that history — a native module that supplies the few things the old runtime got out of its host, a template engine that is the interpreter's own, and a dispatch layer that reads declarative menu files and maps a request path onto an action. As a case study it answers a question the earlier chapters of this part have been circling: what does it take to run a program the size of a web interface in this language, when the language has no objects, no standard library of note, and no async runtime.

Where the code is, and what the native part does

The runtime is a directory of ucode files — the dispatcher, the HTTP layer, the request entry point, the template layer, the system-information and authorisation modules — beside a directory of seven page templates with the `.ut` extension, all installed under the share tree's ucode directory. Its native part is a single module, built as `core.so` into a directory of its own beneath the module directory and reaching it as `luci.core`; the module registers eighteen functions in that namespace, and they are the operations a page runtime needs that the language itself has no business providing: the loading and querying of translation catalogs and the two translation functions, a hash, the shadow and password-entry lookups and the crypt primitive, the identity getters and setters, `kill`, and the three system-information calls.

The division is worth looking at as a decision rather than as a list. The parts of the old runtime that were routing, templating, session logic and page composition were rewritten in the scripting language, and the parts that were bindings to `libc` and to the translation machinery were left as native registrations, so a script has to reach the namespace for them:

```
import { hash, load_catalog, translate } from 'luci.core';
```

The rest of the namespaces a page sees are ucode modules rather than native code — `luci.http`, `luci.runtime`, `luci.dispatcher`, `luci.authplugins`, a version module generated at build time — which means that the boundary between the compiled and the interpreted halves of the framework is one file wide, and that a feature added to the framework is normally a change to a `.uc` file rather than a change to a module. Two further packages maintained in the same tree are modules of the ordinary kind, one adding an HTML tokenizer and entity codec under

the name `html` and the other a bridge that embeds a Lua interpreter as a resource type with a single constructor, and the second of them is what makes the legacy path below work at all.

How a request becomes a page

The web server is uhttpd, and LuCI installs itself into it as a handler for one prefix by way of the configuration hook of chapter 53: a UCI list entry naming the prefix and the handler file, which is the handler's whole registration. That handler file is twelve lines, and it is a good illustration of how thin a uhttpd handler can be when the work lives in modules:

```
import dispatch from 'luci.dispatcher';
import request from 'luci.http';

global.handle_request = function(env) {
        let req = request(env, uhttpd.recv, uhttpd.send);

        dispatch(req);

        req.close();
};
```

The request object is what the HTTP module builds around the environment and the two transfer functions of the host — a plain object with the request's data and methods of its own, which is the same layering chapter 53 recommends for a handler that intends to grow. Everything after that is the dispatcher, and the dispatcher's work divides into building a tree, finding a node in it, and carrying out what the node says.

The tree comes from files installed by packages. Menu descriptions are JSON documents in a directory of their own, and the controller files the old system left behind are found beside them, so the tree is the union of everything installed on the board. The build reads the JSON directly; a Lua controller can only be read through the bridge, and the dispatcher checks for the bridge and warns rather than failing when it is absent, which is the legacy path's visible seam. The result is cached on the filesystem under a name derived from the set of files it was built from, so the whole traversal of the installed tree happens once per change rather than once per request — an important property for a page runtime on flash-backed storage, and one that is eight lines of hashing rather than a service.

Finding a node is a path walk over that tree, with the language of the page chosen per request from the translation catalogs and with the authorisation checks done against the node's own access specification. Carrying out what the node says is a switch, and the switch is where the framework's whole mixture of past and present lives:

Action type	What the dispatcher does
`template`	Renders a template, choosing the template engine by asking whether a ucode template of that name exists and falling back to the Lua path if not.
`view`	Renders the framework's generic view template with the node's view named in its scope.
`call`	Calls a named function of the dispatcher's own runtime.
`function`	Requires a module by name and calls a member of it.
`cbi` , `form`	Invokes the model-driven form layers.
`alias` , `rewrite`	Restarts the dispatch at another path.
`firstchild` , `none`	Ends with the not-found page.

Read that table as a migration plan and it is a complete one: every row that can be served by the language is served by the language, and the rows that name the older machinery are guarded by the availability of the bridge rather than by a version test. A framework that has to keep an ecosystem of third-party pages running while it changes its runtime has to have a table shaped like that, and the interesting property of this one is that the fallbacks are a runtime lookup of a template file's existence rather than a configuration setting.

Rendering a page

A ucode template is found by asking whether a file of its name with the template extension exists in the framework's template directory, and it is rendered by compiling it in template mode and capturing what it writes. The language's own contribution is the second of those two steps, and it is one function: `render()` runs its first argument — a template path or a callable — with the machine's output diverted into a string it returns, and a path argument carries the scope convention of `include()` with it, so the second argument supplies the values the template interpolates. Here is that mechanism running against this interpreter, with a template of four lines and a renderer that hands it a title and a table of links:

```
import { writefile, unlink } from "fs";

writefile("/tmp/ucode-ch57-page.ut",
        "<h1>{{ title }}</h1>\n" +
        "{% for (let name, url in links) { %}" +
        "<li><a href=\"{{ url }}\">{{ name }}</a></li>\n" +
        "{% } %}");

let page = render("/tmp/ucode-ch57-page.ut", {
        title: "Network",
        links: { lan: "/admin/network", wan: "/admin/firewall" }
});

print(page);

unlink("/tmp/ucode-ch57-page.ut");
```

```
<h1>Network</h1>
<li><a href="/admin/network">lan</a></li>
<li><a href="/admin/firewall">wan</a></li>
```

That is the page runtime's rendering step entire: text out of a file, values in through a scope, and the result a string that the HTTP layer sends. The template's control blocks are ordinary code, which is why a page can call a function in the middle of a loop rather than needing the template language to anticipate every computation a page wants — the property chapter 16 spends most of its length establishing — and note that the values arrived as a scope object rather than as globals, which is what keeps one template render from seeing another's state in a process that serves one request after another inside one machine.

The framework's own template set is seven files of this kind — the page furniture, the generic view, the two error pages and the two authorisation fragments — and its applications add their own: the revision carries thirty-four template files and thirty module files over the whole repository, in packages ranging from a status page to container management. The back ends of the newer applications are of the other kind described by chapter 55: files installed into the procedure daemon's ucode directory, reached through the bus rather than through the page layer, which is the division of labour the framework settled on — a page renders and asks the bus, and a service answers without ever knowing that a page asked.

Porting from Lua

The framework's migration is a decade deep, and the parts of it a page author meets are mechanical. The table below is the mapping for the constructs that appear in a controller or a template; the left column is what the older pages say, the right what the same thing is in this language, with the chapter that owns the construct.

Lua	ucode	Chapter
`for k, v in pairs(t) do`	`for (let k, v in t) { }`	7
`a .. b`	`a + b`, or an interpolated string	9
`#t`	`length(t)`	22
`require "mod"`	`require("mod")` for a value, `import` for bindings	17
`local x`	`let x`, with the declaration discipline of strict mode	5
metatables and `__index`	prototypes and the metamethods	12
`nil`	`null`, with `undefined` as the not-a-value	4
one table for everything	arrays and objects as two types	10, 11
`ngx`-style host globals	the host's object and the request scope	53
`assert(x, msg)`	`assert(x, msg)`, spelled the same and behaving the same	14

Two of those rows deserve more than a table cell. The `nil` row is where a ported page most often goes wrong, because the distinction chapter 4 draws between a value that is absent and a name that has no value at all is a distinction Lua does not make, and a page that tests a field's presence one way in the old runtime has to be read again before it is trusted in the new one. And the metatable row is a change of shape rather than a change of spelling: what a metatable does for a table in Lua, a prototype with metamethods does for an object here, with the semantics of chapter 12 — and, since the rewrite, with the delegation rule of chapter 12 that an object-valued metamethod is the only kind that delegates to the parent. The framework itself is indifferent to all of this, being a set of data structures and functions rather than a hierarchy, which is the same lesson chapter 56 drew from a program of comparable size.

What the case teaches

Three things follow from the shape of this runtime, and they are general about programs of this kind.

A page runtime needs an output-capturing renderer, and this language has one. `render()` is a small function and it is the thing that makes template files composable — a page is a function from a scope to a string — and it is the same function that lets a fragment be included in a page or in a mail body or in a log line. A framework that had to build that itself out of pipes would be a framework with a slower template layer.

The line between native and interpreted is worth moving deliberately. Eighteen functions in one file is a small surface, and every one of them is a binding to something the language has no business owning. The framework's history is instructive on that point: it kept the translation catalogs, the identity lookups and the hashes native and moved everything above them, and the seam between the halves is a table of eighteen registrations.

A migration survives by probing at run time. Whether a page is rendered by one engine or the other is decided by looking for a file, and whether a legacy controller can be read at all is decided by whether a module loaded. Both are run-time facts about the installed system rather than build-time facts about the framework, which is why the same framework serves a board that has the legacy packages and a board that has not, and why the framework's package list is the interesting document about which of the two paths a board is on: the base package pulls the interpreter, the file, log, configuration, bus and HTML modules and the HTTP library's binding, and the legacy runtime package is the one that adds the Lua bridge.

Reading on

- Chapter 12 — prototypes and metamethods, the mechanism behind the metatable row.
- Chapter 16 — templates, and the scoped form of `include()` that a renderer is built on.
- Chapter 17 — the two import systems, which is what a runtime built out of modules organises itself by.
- Chapter 19 — the language's differences from the languages it resembles, in its compressed form.
- Chapter 53 — uhttpd, the host this runtime plugs into, and its handler contract.
- Chapter 55 — the procedure daemon's ucode directory, where these applications' back ends live.
- Chapter 59 — the packaging landscape, including the modules this framework's packages name.

Case study: Wi-Fi

Source files referenced in this chapter: upstream `openwrt/openwrt` — the `wifi-scripts` package at `package/network/config/wifi-scripts/` (both its `files/` and its `files-ucode/` file sets) and the `hostapd` package at `package/network/services/hostapd/`, including `patches/601-ucode_support.patch`, `src/src/utils/ucode.h` and `files/hostapd.uc` — read from the `master` branch at revision `e2aa1d759647`, with the older `wifi-scripts` layout taken from revision `352c0791754a` of the `openwrt-24.10` branch; Appendix G gives an address for each of them. The detection logic of the scripts needs no radio to be understood; the scripts themselves need a device with one.

Wireless configuration is the second place in OpenWrt where a subsystem's logic moved into this language, and it did so in a way different from the firewall's. The firewall is one program with a template front end; wireless is a family of cooperating programs — a detector that learns what radios the board has, a generator that turns that knowledge plus the user's configuration into a device configuration, and a daemon that drives the access-point software and answers questions about it. The language appears in each of the three, and in the third of them it appears as an embedded interpreter inside a long-lived C daemon. Reading the three together shows what the language is used for when the job is hardware.

The scripts package

The package that carries the reconfiguration logic is `wifi-scripts`, and its dependency line is the clearest statement of what wireless scripting needs: the interpreter, and the netlink modules for wireless and for routing, the bus, the configuration store and the file system, declared as packages rather than assumed.

```
  DEPENDS:=+netifd +ucode +ucode-mod-nl80211 +ucode-mod-rtnl +ucode-mod-ubus
+ucode-mod-uci
```

The user-facing command is still a shell script, and it shells out to the interpreter for the two halves of its work — detection, then generation piped into a batch of configuration changes:

```
wifi_config() {
        [ -e /tmp/.config_pending ] && return
        ucode /usr/share/hostap/wifi-detect.uc
        [ ! -f /etc/config/wireless ] && touch /etc/config/wireless
        ucode /lib/wifi/mac80211.uc | uci -q batch
        ...
}
```

Read that as a deployment and it is the first shape of chapter 52 twice over, joined by a pipe: two runs of the interpreter, the first writing a file and the second reading it and emitting commands that another program consumes.

The pending-file test at the top is the reentrancy guard a run-on-demand program needs and a daemon does not. A three-line hot-plug hook calls the same entry when a device appears, which is how the detection half gets triggered on a board whose radios show up late.

The package as it stands on the main branch carries two file sets, and the difference between them is a history of the migration in one directory listing. The older set, which is the one the earlier research read on the 24.10 branch, is the three modules under `usr/share/hostap/`, the generator at `lib/wifi/mac80211.uc` and the shell helpers alongside them. The newer set, under a sibling directory of the package, is a reorganisation into a module namespace with data files beside it:

```
files-ucode/usr/share/ucode/
├── iwinfo.uc                   a reporting program in its own right
└── wifi/
    ├── common.uc               the shared helpers
    ├── iface.uc                interface construction
    ├── ap.uc                   access-point specifics
    ├── supplicant.uc           the station back end
    ├── hostapd.uc              the hostapd back end
    ├── netifd.uc               the network-daemon interface
    └── validate.uc             schema-driven option checking

files-ucode/usr/share/schema/
├── wireless.wifi-device.json   one schema per configuration section type
├── wireless.wifi-iface.json
├── wireless.wifi-station.json
└── wireless.wifi-vlan.json

files-ucode/usr/share/
├── wifi_devices.json           the driver capability database
└── iso3166.json                the country codes
```

Six things about that layout are worth more than a listing. The modules are in the share tree's ucode directory, so they are reachable by bare import names from anywhere, which is the arrangement chapter 17 recommends for a set of modules that several entry points share. The driver-specific behaviour is a module per daemon — `wifi/hostapd.uc` against `wifi/supplicant.uc` — so a daemon is a back end selected by name rather than a case in a conditional. The knowledge is in JSON files rather than in conditionals: the driver database and the country list are data installed next to the code, and the four schema files describe what the configuration sections may contain. And `iwinfo.uc` is not part of the configuration path at all: it is a reporting program, which is what makes it a good illustration of the same modules serving two purposes.

The schema files earn their place by being read as data rather than being duplicated as code. The validation module loads the four schemas at its top level and derives its option table from them, so a check of the wireless configuration's vocabulary is a lookup in a file that a human can diff:

```
const schemas = {
        device: json(fs.readfile('/usr/share/schema/wireless.wifi-
device.json')).properties,
        iface: json(fs.readfile('/usr/share/schema/wireless.wifi-
iface.json')).properties,
        vlan: json(fs.readfile('/usr/share/schema/wireless.wifi-
vlan.json')).properties,
        station: json(fs.readfile('/usr/share/schema/wireless.wifi-
station.json')).properties,
};
```

and the same table answers a query from the outside world, since the module also offers the option names and their encoded types on its standard output through the JSON conversion of `sprintf` — a capability published by a script out of a data file, with no interface defined anywhere but in the schema.

Detection, and the board database

The detector's job is to turn what the kernel says about the radios into a description that the generator can read, and the file it maintains is the `wlan` section of the board's JSON description. Its opening is the shape of a ucode program of this kind entire: imports of what it needs, two reads of the existing description, and functions that walk the sys filesystem.

```
#!/usr/bin/env ucode
'use strict';
import { readfile, writefile, realpath, glob, basename, unlink, open,
rename } from "fs";
import { is_equal } from "/usr/share/hostap/common.uc";
let nl = require("nl80211");

let board_file = "/etc/board.json";
let prev_board_data = json(readfile(board_file));
let board_data = json(readfile(board_file));
```

The shebang line is there although every caller names the interpreter explicitly, which is the convention chapter 52 recommends for a file that is both a program and an inclusion target. The import of `fs` picks individual names out of the module rather than aliasing the whole of it, the import of the shared module is by absolute path — the form chapter 17 describes for programs that are started from a working directory nobody controls — and the native module comes through the older function. Then two reads of one file: the previous copy and the working copy, because the detector has to know whether it changed anything in order to say so.

The rest of the file's first eighty lines is sysfs traversal and arithmetic, and it is worth reading for what it does not use. Paths are resolved with `realpath` and enumerated with `glob` and reduced with `basename`, so a radio's identity is

built out of a device path and an index read out of a sys file rather than out of a name guessed at. A list of physical devices is sorted with a comparator function, which is a closure rather than a sort key. And a chunk of it is pure arithmetic in the open, the mapping of a frequency to a channel number:

```
function freq_to_channel(freq) {
        if (freq < 1000) {
                return 0;
        }

        if (freq == 2484) {
                return 14;
        }

        if (freq < 2484) {
                return (freq - 2407) / 5;
        }

        if (freq < 5950) {
                return (freq - 5000) / 5;
        }

        return 0;
}

for (let freq in [ 2412, 2437, 2484, 5180 ]) {
        printf("%-6d channel %d\n", freq, freq_to_channel(freq));
}
```

```
2412   channel 1
2437   channel 6
2484   channel 14
5180   channel 36
```

The real function carries a few more bands than this excerpt, including the divisions for the ultra-wide channels and the sixty-gigahertz band's odd stride, and the point of showing the arithmetic is that the detector has no helper for it: a regulatory table would be a fine thing to have, and what the program has is a ladder of conditions over integers, which is a perfectly ordinary way for this language to be written. A function of that shape is also trivially testable, which is the other half of why the arithmetic is kept in functions rather than inlined at its uses.

The query that gathers the hardware data is one call of the wireless netlink module, asking for a dump of the physical devices and asking the module to break the answer up per device — and that is the one call in the program that means nothing without a radio to describe. The module's data surface is where a call of that shape goes looking for its argument values:

```
import * as nl from "nl80211";

let c = nl.const;

print("members: ", keys(nl), "\n");
print("constants: ", length(c), "\n");
print("get-wiphy command: ", c["NL80211_CMD_GET_WIPHY"], "\n");

let dump = sort(filter(keys(c), k => match(k, /DUMP/)));

print("dump flags: ", join(", ", map(dump, k => `${k}=${c[k]}`)), "\n");
```

```
members: [ "listener", "waitfor", "request", "error", "const" ]
constants: 188
get-wiphy command: 1
dump flags: NLM_F_DUMP=768, NLM_F_DUMP_FILTERED=32, NLM_F_DUMP_INTR=16
```

Five members and a table of a hundred and eighty-eight constants: the module is small and its vocabulary is big, which is the usual shape of a netlink binding and the reason chapter 35 spends as much space on the answer structures as on the calls. A script that reads that table by name, as the detector does, is insulated from the numbers; a script that hard-codes `768` is not, and the constant table's presence is what makes the first of those two spellings cost nothing.

The daemon half

The third piece of the wireless stack is the access-point daemon, and its relationship to the language took some untangling, because the answer is different on the two sides of the boundary between upstream and packaging.

Upstream hostap — the daemon maintained at `w1.fi`, covering both the access-point daemon and the supplicant — carries no ucode at all. That is a checked negative: a search of the whole tree over each of its branches turns up no directory of ucode glue, no build knob for it and no binding of it, and the one place the word appears is in commit messages about network-card firmware. Anyone reading an account of a ucode-enabled access-point daemon and going to the upstream sources to read the interface will find nothing, and should not conclude that the account was wrong.

The packaging side is where the integration lives. OpenWrt's hostapd package carries a patch whose subject line says what it does, "Add ucode support, use ucode for the main ubus object", together with the sources it applies to, a header apiece for the daemon, the access-point code and the supplicant, and two installed scripts. The patch's own description of its purpose is that it improves dynamic reconfiguration, in that it can cope with a change to one wireless interface and with interfaces appearing and going away — which is the same motivation that put a scripting language into the router's main daemon, and which is why the daemon adopted this particular one.

The interface between the daemon and the script is readable from the header the package carries: functions to create a machine, to run a named script, to prepare and perform a call into it, to release it, and to keep lists of script values in the machine's registry, along with a constant giving the directory the scripts are found in and a handful of native functions registered for the scripts' use — printing into the daemon's own log, hashing, asking the frequency tables, reading the debug level. The installed script shows the other side of the same interface: it opens with a mixture of the older and the newer import forms, attaches an exception handler to the bus module so that a failed call answers with a diagnostic rather than with silence, and publishes the daemon's bus objects itself:

```
let libubus = require("ubus");
import * as uloop from "uloop";
import { open, readfile, access } from "fs";
import { wdev_remove, is_equal, vlist_new, phy_is_fullmac, phy_open,
         wdev_set_radio_mask, wdev_set_up } from "common";

let ubus = libubus.connect(null, 60);

function ex_handler(e)
{
        e = split(`${e}\n${e.stacktrace[0].context}`, '\n');
        for (let line in e)
                hostapd.printf(line);
        return libubus.STATUS_UNKNOWN_ERROR;
}
libubus.guard(ex_handler);
```

The import from `"common"` is the module the wireless scripts package installs, reached by bare name — one directory shared between the scripts and the daemon's script, which is a fact about the packaging rather than about the language and an easy one to miss while reading either tree alone. The handler is worth two remarks for a book reader: the value a `catch` binds is an object with the three members `type`, `message` and `stacktrace`, which is what the handler's use of it assumes and what this interpreter confirms, and `libubus.guard()` is the mechanism by which a script that answers bus calls decides what a fault costs — here, the unknown-error status and a line in the daemon's log for each line of the diagnostic.

Further into the same file, the daemon's whole bus surface is script:

```
hostapd.data.obj = ubus.publish("hostapd", main_obj);
hostapd.data.auth_obj = ubus.publish("hostapd-auth", auth_obj);
```

which is the arrangement of chapter 55 wearing different clothes: a script declaring objects and methods, the host doing the bus work, and the object's methods being closures over the script's state. What differs is the lifetime and the reason for it. The procedure daemon's scripts are loaded from a directory so that installing a service installs a file; this script is started by the daemon that embeds it, because it is part of that daemon rather than a service that lives beside it, and because the objects it publishes are the daemon's own control interface rather than an independent service's.

Both use the same two mechanisms — the bus module and the event loop — which is the practical benefit of the language having been designed with the embedding as one of its modes rather than as an afterthought.

What the case teaches

Four things, in the order a reader is likely to need them.

A description file is worth more than a protocol. The detector and the generator do not speak to each other; they read and write one JSON document, and a third program, the reporting tool, reads it as well. No version negotiation, no socket, and no ordering constraint beyond the file's having been written before it is read, which for programs that run at different moments of a device's life is a better interface than any call.

Data files beat conditionals as the place where hardware knowledge lives. Four JSON schemas, one driver capability database and a country list carry most of what the newer wireless stack knows about the world, and the code that reads them is short. The validation module's option table and the published option list are derived from the schemas rather than maintained beside them, which is the only arrangement in which they stay true.

A netlink binding's constants are part of its usability. The wireless module's five functions are not much use without its table of a hundred and eighty-eight names, and a script that reads names out of that table can be read by somebody who does not have the kernel headers in mind. This is the same argument chapter 35 makes, arriving at the place where it matters most.

The embedding story has to be read at the packaging layer. Upstream has no ucode; the distribution's patch adds a machine to the daemon and moves the daemon's bus objects into a script, with a directory shared with the reconfiguration scripts; the sources that would confirm the split are in the package rather than in the upstream tree, so a reader who reads only the upstream tree learns the wrong thing. When part V cites an embedded interpreter, the file and line it names is the authority, and where it names none, that is the finding rather than an omission.

Reading on

- Chapter 14 — exception objects and their `stacktrace` member, which the daemon's handler relies on.
- Chapter 15 — JSON, the notations, and the `%J` conversion that publishes a value as data.
- Chapter 17 — the two import systems, the bare-name and absolute-path import forms, and installing modules in the share directory.
- Chapter 25 — the file-system module whose sysfs traversals fill the detector.
- Chapter 34 and chapter 35 — the routing and wireless netlink modules the package depends on.
- Chapter 36 — the event loop the daemon's script runs in.
- Chapter 37 — publishing bus objects from a script.
- Chapter 38 — the configuration store the generator writes into.

- Chapter 52 — deployment models, of which the detector and the generator are the standalone instance and the daemon is the embedded one.
- Chapter 55 — the procedure daemon's ucode directory, the other way of putting a script behind a bus object.

The wider ecosystem

Source files referenced in this chapter: `README.md`, `debian/control`, `debian/rules`, `openwrt/ucode/Makefile`, `.github/workflows/`, `udbg.c`, `debug_highlight.c`, `debug_highlight.h`, `debug_lineedit.c`, `docs/debugger.md`. The embedder quoted in the second section, `ucode.c`, is read out of `openwrt/netifd`, branch `master`. The build file quoted beside it, `CMakeLists.txt`, is read out of `openwrt/rpcd`, branch `master`. The other files named here belong to the ucode repository.

The chapters before this one describe ucode as something you program against or embed. This one describes the surroundings: which programs take the interpreter in, how the source tree becomes installable packages, how a release is identified, what editing and analysis tools exist outside the repository, and how to recognise ucode inside a tree you did not write.

What is asserted about this repository can be checked in a checkout of it: the paths, the file counts and the transcripts are those of the tree, and your own copy answers the same questions with the same commands. What is asserted about other projects is attributed to the upstream file and line it rests on, and Appendix G gives the revision those line numbers belong to. Web-scale numbers decay, so the counts quoted below carry the dates they were taken, 2026-07-19 and 2026-09-16, and are to be read as observations of those days rather than as properties of the projects.

Where ucode is consumed

Consumers fall into two shapes, separated by who owns the interpreter process. In the first, a C program links `libucode` and owns the virtual machine: it calls `uc_vm_init()`, compiles or loads a program, registers native functions and resource types, and drives the event loop itself. netifd, uhttpd, rpcd and the hostapd build carried by the OpenWrt trees are of this kind. The embedding interface is the subject of part IV — chapter 40 gives its shape and chapter 49 walks the six example hosts that ship in `examples/`.

```
#include <string.h>

#include <ucode/vm.h>
#include <ucode/lib.h>
#include <ucode/compiler.h>
#include "netifd.h"

uc_vm_t vm;
```

That is the whole of what the embedding side looks like from outside the host's own logic: three headers from the installed `include/ucode/` set and one VM object at file scope (`ucode.c:23-27` in openwrt/netifd). rpcd shows the same arrangement from its build side; the plugin target exists only when the project is configured with `UCODE_SUPPORT`, which is declared `ON` (`CMakeLists.txt:12,69-74` in openwrt/rpcd):

```
OPTION(UCODE_SUPPORT "ucode plugin support" ON)
...
IF(UCODE_SUPPORT)
  FIND_LIBRARY(ucode NAMES ucode)
  SET(PLUGINS ${PLUGINS} ucode_plugin)
  ADD_LIBRARY(ucode_plugin MODULE ucode.c)
  TARGET_LINK_LIBRARIES(ucode_plugin ${ucode})
  SET_TARGET_PROPERTIES(ucode_plugin PROPERTIES OUTPUT_NAME ucode PREFIX "")
```

The second shape leaves the interpreter binary as the entry point. A `.uc` file is installed by a package and then either runs as a command through its shebang line, as netifd's protocol handler does, or is loaded by a host that watches a directory of them — rpcd loads plugins from `/usr/share/rpcd/ucode/`, and that is where LuCI's application back ends are installed (chapter 57). In that shape the script is a data file to its installer and a program to whoever invokes it, and nothing in it refers to C.

```
let handlers = {
        init: function (name) { return `hello ${name}`; },
        run: function (job) { return job * 2; }
};

print(keys(handlers), "\n");
print(handlers.init("world"), "\n");
print(handlers.run(21), "\n");
```

```
[ "init", "run" ]
hello world
42
```

The contract a hosted script fulfils is the small structure above seen from the other side: the script leaves a value — a dictionary of functions is the usual form — where host code can find it, and the host calls into it per request or per event. Chapters 53, 54 and 55 cover the three hosts that do that in OpenWrt.

The census below is reduced to what a reader can act on: the consumer, the shape of its use, and the job the language does there. It is a pointer to the case-study chapters, not an analysis of them.

Consumer	Shape	What ucode is used for
`openwrt/openwrt`	both	42 `.uc` package scripts (netifd's library and protocol handler, `wireguard-tools`, `umdns`, the wireless scripts), plus an embedded VM in the hostapd build
`openwrt/luci`	hosted scripts	ucode runtime libraries under `modules/luci-base/ucode/` and about thirty `.uc` back-end scripts installed under `/usr/share/rpcd/ucode/`
`openwrt/packages`	scripts	eleven shebang scripts across packages; packages such as `shunt` and `uneighbord` declare `DEPENDS:=+ucode`
`openwrt/netifd`	embedded	`ucode.c`, `ucode.h`, `proto-ucode.c` run protocol handlers in script; `proto-ucode.uc` is the shipped handler
`openwrt/uhttpd`	embedded	`ucode.c` binds the VM to the request path (chapter 53)
`openwrt/rpcd`	embedded and hosted	a `ucode` plugin module loads scripts from `/usr/share/rpcd/ucode/` (chapter 55)
`jow-/uwsd`	embedded	a single-process HTTP and WebSocket server with script handlers (chapter 54)
hostapd as built in OpenWrt trees	embedded	`src/utils/ucode.c` initialises the VM so host control can be scripted (chapter 58)
`immortalwrt/immortalwrt`, `coolsnowwolf/lede`, `lede-project/source`, `istoreos/istoreos`, `Lienol/openwrt`, `Entware/Entware`	both	carry the same integrations as `openwrt/openwrt`, file for file

Two searches give the relative weight of the two shapes: `uc_vm_init` occurs in 18 results across eight repositories — the language's own tree, netifd, uhttpd, rpcd, openwrt/openwrt and three OpenWrt-derived trees — while the shebang line `#!/usr/bin/env ucode` occurs in 115 results across eight repositories of OpenWrt-derived trees. The same families appear in both lists, and the fork rows are not separate ecosystems: file-level checks found the same netifd handler, the same hostapd `ucode.c` and the same package scripts in each of them. Measured by repository attention the largest consumers are OpenWrt's own trees: `coolsnowwolf/lede` at 31,583 stars, `openwrt/openwrt`

at 28,425, `immortalwrt/immortalwrt` at 11,598 and `openwrt/luci` at 7,841, against 174 for the language repository itself (figures as of 2026-07-19).

Packaging

Everything that gets installed comes out of one CMake project at the repository root: the shared library, the two programs, the symlinks that give the same binary its other two names, and one loadable object per extension module. The install half of `CMakeLists.txt` is short enough to quote in full:

```
add_executable(udbg udbg.c debug_highlight.c debug_lineedit.c)
target_link_libraries(udbg PRIVATE libucode ${JSONC_LINK_LIBRARIES})
install(TARGETS ucode udbg RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR})
install(TARGETS libucode LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR})
install(TARGETS ${LIBRARIES} LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}/
ucode)

add_custom_target(utpl ALL COMMAND ${CMAKE_COMMAND} -E create_symlink ucode
utpl)
install(FILES ${CMAKE_CURRENT_BINARY_DIR}/utpl DESTINATION $
{CMAKE_INSTALL_BINDIR})

if(COMPILE_SUPPORT)
  add_custom_target(ucc ALL COMMAND ${CMAKE_COMMAND} -E create_symlink ucode
ucc)
  install(FILES ${CMAKE_CURRENT_BINARY_DIR}/ucc DESTINATION $
{CMAKE_INSTALL_BINDIR})
endif()

file(GLOB UCODE_HEADERS "include/ucode/*.h")
install(FILES ${UCODE_HEADERS} DESTINATION include/ucode)
```

The core library target is `add_library(libucode SHARED ...)`, given `OUTPUT_NAME ucode` and the `SOVERSION` cache variable, so with the default prefix the artefact is `libucode.so.<soversion>` and its ELF soname says the same; `SOVERSION` defaults to `0` and its cache entry reads `"Override ucode library version"`. Each extension module is a `MODULE` target with `OUTPUT_NAME` set to the bare module name and `PREFIX` cleared, which is why the installed files are `fs.so` and `ubus.so` rather than `libfs.so`, and why they are installed into a directory of their own, `${CMAKE_INSTALL_LIBDIR}/ucode/`. In a build directory of a checkout:

```
$ objdump -p build/libucode.so.0 | grep -i soname
  SONAME               libucode.so.0
```

The soname is part of what an embedder links against, so a packager who wants it to track a release sets `SOVERSION` at configure time — both packaging routes in this tree do, as shown below. The headers are installed by the globbing rule above, which is not recursive, so the `internal/` subtree is not covered by it:

```
$ ls include/ucode/*.h | wc -l
10
$ ls include/ucode/internal/*.h | wc -l
11
```

At run time a script names a module the way it names any other import; the mapping from that name to a file is the search path, and the default is compiled in from the install prefix:

```
set(LIB_SEARCH_PATH "${CMAKE_INSTALL_PREFIX}/${CMAKE_INSTALL_LIBDIR}/ucode/
*.so:${CMAKE_INSTALL_PREFIX}/share/ucode/*.uc:./*.so:./*.uc" CACHE STRING
"Default library search path")
```

Chapter 17 deals with that path and with the `-L` flag that extends it. What the tree does not install is discovery metadata for build systems: there is no export set, no package configuration file and no pkg-config file.

```
$ grep -c "install(EXPORT\|EXPORT_NAME\|PKG_CONFIG\|configure_file"
CMakeLists.txt
0
```

An embedder therefore locates the library and the header directory on its own terms, which is what the rpcd fragment earlier in this chapter does with its single `FIND_LIBRARY` line, and what the examples in `examples/` do with a plain `-lucode`.

The `debian/` tree is a full upstream Debian source package in native format, producing four binary packages:

```
$ grep "^Package:" debian/control
Package: ucode
Package: ucode-modules
Package: libucode
Package: libucode-dev
```

The interpreter and `udbg` go to `ucode` through `usr/bin/*`, the extension modules to `ucode-modules` through `usr/lib/*/ucode/*.so`, the runtime to `libucode` through `usr/lib/*/libucode.so.*` and the headers to `libucode-dev`. The source stanza declares `Section: interpreters`, `Standards-Version: 4.7.2` and a build dependency on debhelper compat level 13, cmake, `libjson-c-dev`, `libmd-dev` and `zlib1g-dev`; the maintainer of record is Paul Spooren, not the upstream author. `debian/rules` is a plain `dh` sequence with one configure override, and it is where the version and the feature set are decided:

```
SOVERSION = $(word 3,$(subst ., ,$(DEB_VERSION_UPSTREAM)))

override_dh_auto_configure:
	dh_auto_configure -- \
		-D SOVERSION=$(SOVERSION) \
		-D BUILD_OPTIMIZE_SIZE=OFF \
		-D ZLIB_CHUNK_SIZE=131072 \
		-D NL80211_SUPPORT=OFF \
		-D RTNL_SUPPORT=OFF \
		-D UBUS_SUPPORT=OFF \
		-D UCI_SUPPORT=OFF \
		-D ULOOP_SUPPORT=OFF
```

`SOVERSION` is read out of the upstream version string as its third dot-separated word, so the version `0.0.20250529` yields the library name `libucode.so.20250529`; link-time optimisation is enabled on an explicit architecture list and `hardening=+all` is exported for all of them. The five `*_SUPPORT` flags switched off there are the two netlink modules, the message bus, the configuration store and the event loop. The twelve that stay available are `debug`, `digest`, `ffi`, `fs`, `io`, `log`, `math`, `resolv`, `serial`, `socket`, `struct` and `zlib`; their options default to `ON` in the source, except `ZLIB_SUPPORT`, `DIGEST_SUPPORT` and `FFI_SUPPORT`, which default to whatever the library probes found and which the declared build dependencies supply. Whether these packages have been accepted into a distribution is not stated by the tree, and no published package index answers it.

OpenWrt is packaged from `openwrt/ucode/Makefile`, which builds out of the same source directory (`CMAKE_SOURCE_DIR=$(CURDIR)/../../`) and splits the result into eleven packages:

```
$ grep -o "Package/[a-z0-9-]*" openwrt/ucode/Makefile | sed -n "s|Package/||p" | sort -u
libucode
ucode
ucode-mod-fs
ucode-mod-math
ucode-mod-nl80211
ucode-mod-resolv
ucode-mod-rtnl
ucode-mod-struct
ucode-mod-ubus
ucode-mod-uci
ucode-mod-uloop
```

`PKG_VERSION` and `PKG_ABI_VERSION` are taken from the committer date of the checked-out commit, formatted `%Y-%m-%d` and `%Y%m%d`, and the date form doubles as the library soname through `CMAKE_OPTIONS += -DSOVERSION=$(PKG_ABI_VERSION)`. The `ucode` package is in `SECTION:=lang`, depends on `+libucode` and installs `usr/bin/u*` — `ucode`, `ucc`, `utpl` and `udbg` together; `libucode` is in `SECTION:=libs`, carries the

ABI version and depends on `+libjson-c`; a `Build/InstallDev` step exposes `usr/include/ucode/*.h` and `libucode.so*` so that a package which embeds the VM can build against it, which is how rpcd's package pulls it in. Each `ucode-mod-*` package depends on `ucode` plus whatever library its module was gated on, and installs one file into `usr/lib/ucode/`. The host-side configure options enable `fs`, `math` and `struct` and switch off `nl80211`, `resolv`, `rtnl`, `ubus`, `uci` and `uloop`, so a cross build does not need the OpenWrt libraries present on the build host; the modules that do need them ship as their own subpackages.

The set of packages is smaller than the set of modules the build can produce:

```
$ for m in $(ls build/*.so | xargs -n1 basename | sed 's/\.so$//' | grep -v
'^libucode$' | sort); do
        grep -q "ucode-mod-$m" openwrt/ucode/Makefile || printf '%s\n' "$m"
done
debug
digest
ffi
io
log
serial
socket
zlib
```

Those eight have no subpackage in this Makefile. Two of them are nonetheless depended upon downstream: `+ucode-mod-log` is named by LuCI's `modules/luci-base/Makefile` (`Makefile:20-27` in openwrt/luci). Where that package is built from is not recorded in this tree, and two attempts to reach a `lang/ucode` Makefile, in `openwrt/packages` and in `openwrt/openwrt`, both came back 404. The practical consequence is that the module set of a device image has to be read from the image's package list rather than inferred from the CMake feature toggles.

Automation lives in five workflow files plus a legacy GitLab definition:

```
$ for f in .github/workflows/*.yml; do
        printf '%-32s %s\n' "${f##*/}" "$(sed -n '1s/^name: //p' "$f")"
done
debian.yml                       Build .deb package
jsdoc.yml                        GitHub pages
macos.yml                        Build on macOS
openwrt-ci-master.yml            OpenWrt CI master branch testing
openwrt-ci-pull-request.yml      OpenWrt CI pull request testing
```

`debian.yml` runs on pushes of tags matching `v*.*.*`, installs the debhelper toolchain, derives a version from `git describe --long --tags`, adds a changelog entry with `dch` and runs `dpkg-buildpackage -b -us -uc`, keeping the resulting `*ucode*.deb` files as an artefact. `macos.yml` builds on every push and pull request with the netlink, bus, configuration-store and event-loop modules switched off, after installing `json-c` and `libmd` through

Homebrew; it is the continuing check that the core compiles without any OpenWrt library. `jsdoc.yml` regenerates the API reference under `docs/` with `npm run doc` and publishes it as GitHub Pages on pushes to `master`, and it is gated on the repository name so that forks skip it. The two `openwrt-ci-*` workflows run the native test matrix through `ynezz/gh-actions-openwrt-ci-native` with `CI_ENABLE_UNIT_TESTING=1` and `CI_TARGET_BUILD_DEPENDS="libnl-tiny ubus uci"`; the pull request variant pins the Clang leg to version 11. `.gitlab-ci.yml` at the root includes the same OpenWrt CI templates for the GitLab mirror. The test suites those jobs drive are the subject of chapter 61.

Releases and versioning

The repository identifies a release by the date it was cut, and it has done so from the beginning. Twelve lightweight tags exist, of the form `v0.0.YYYYMMDD`, the last of them `v0.0.20250529`; the first is `v0.0.20220322`. They are lightweight rather than annotated, so a release carries no message of its own beyond the date in its name. The major and minor components have stayed at zero through all of them, so the third component is the release, and it is a date.

Neither packaging route invents a version of its own; both read the tree:

```
$ git tag --list | tail -3
v0.0.20230606
v0.0.20231102
v0.0.20250529
```

The Debian route reads the version out of the changelog and takes its third dot-separated word as the library version, which is why the rules file carries the `SOVERSION = $(word 3,$(subst .,, $(DEB_VERSION_UPSTREAM)))` line the previous section quoted; and the Debian workflow derives its version from `git describe` and writes a changelog entry with `dch`, so a tag of the tree's own form is what produces a package of the matching number. The OpenWrt route takes the committer date of the commit being built as both its package version and its ABI version, and passes the second of those to the build as `SOVERSION`, so a device's library soname tells you which commit's date it came from.

Two absences complete the picture, and both are worth knowing when you need to answer the question "which version of ucode is this". The interpreter has no version flag: an unknown option is reported as an invalid option rather than as a version, and nothing in the command-line driver prints a version string. And the sources carry no version macro to read out of a header, so the two ways to date a build are the library's soname, which both routes set from the date, and the build's own provenance: the `BUILD_INFO` the CMake probe records from `git describe` in a checkout, which is a cache entry rather than a queryable property of a binary. A deployment that cares about versions therefore cares about the soname and the package version, and a build from a tarball of a tagged tree has to set them by hand.

Editor and tooling support

Inside the repository there is one highlighter, `debug_highlight.c`, which emits terminal escapes for both the language and template files and is what gives a debugger session its coloured source; chapter 61 has it and the line-editing module beside it. It exists for the debugger rather than as a general filter, though a terminal will accept its output as readily as the debugger does.

Outside it, on the searches of 2025-07 and 2026-09, the language has editor support of three kinds and no official one of them. There are two tree-sitter grammars: one carried by the language's own author, and one by another contributor which covers template files and documentation comments in addition to the language, and which is therefore the more complete of the two for somebody who wants to edit templates with indentation that makes sense. There is one language server, with its own package on the node package registry and the assets of a Visual Studio Code extension; it carries the two TextMate grammars that editors of that family use, one for the language and one for templates, so an editor that reads TextMate grammars can highlight both file kinds out of that package. And there are TypeScript bindings for the grammar, one on the package registry and one published to the Visual Studio Code marketplace under the identifier `ucode`, which is what makes the grammar usable from the editors that embed tree-sitter.

The absences are worth listing with the same care, because an integrator looks for them: there is no Pygments or Rouge lexer, so a documentation generator that uses either of those two frameworks will render a ucode fence as plain text, and the work-around of borrowing the JavaScript lexer is imperfect for the template delimiters and the pattern literals in particular; there is no Emacs mode and no standalone Vim plugin, with the tree-sitter route being the way Neovim gets its support; and there is no formatter, the indentation rules at the root's editor configuration file being the whole of the enforced style — tabs, a width of four, trailing whitespace removed, a final newline. That the style is carried by a file rather than by a program means that a contribution's formatting is a matter of the editor's configuration rather than of a build step's pass, which is a smaller loss than it sounds with a style as small as this one's, and is the kind of gap that tends to be filled by whoever minds it most.

Finding ucode in a codebase

Finding the language in a tree you did not write is a matter of two patterns, one for the scripts and one for the embeddings, and the two look nothing like each other.

A script is a file with a `.uc` extension, usually with the shebang line at its front and always with imports, and the forms of those are what distinguish the language's files from a JavaScript file that happens to share the extension:

```
import * as fs from "fs";
import { readfile, glob } from "fs";
import { is_equal } from "/usr/share/hostap/common.uc";
const ubus = require("ubus");
```

The first three are the import forms of chapter 17 and the fourth is its older companion; seeing the pair of them in one tree is normal rather than a sign of confusion, as chapter 58's wireless files showed. A template file has no imports at all and is found instead by its delimiters, `{%` and `{{`.

An embedding is C that includes the language's headers and calls into it, and the markers are the include prefix and the entry point of chapter 40:

```
#include <ucode/vm.h>
#include <ucode/lib.h>

uc_vm_init(&vm, NULL);
```

and a native module rather than an embedding shows the same prefix together with the registration entry point of chapter 48, `uc_module_init`. A build system announces its use of the library in whichever of the three styles the project's build system speaks: a `FIND_LIBRARY(ucode)` in a CMake project, `-lucode` in a make rule or the link line of a configure script, and `+ucode` or `+ucode-mod-`*name* in the dependency line of an OpenWrt package, which last form is the one that answers the practical question "does this device have the module I want" without consulting anything but the package list.

One discrimination is needed, because the name is not exclusive. A repository search for the word returns somewhere about fifteen hundred results, of which the great majority are other things that are also spelled `ucode`: the microcode of a processor family, and a header of the Nintendo 64 development libraries that has held that name since the nineties. Neither of them is related, and neither is hard to tell apart, since the language's own files carry the include prefix `ucode/` or the module-registration symbols and the others do not carry either.

What a search returns

The counts below come from the search interfaces of the forge, on the dates named with each of them, and they are reproduced here because they are the only quantitative answer to "how much of this language is out there". Treat them as observations of those days: a star count is a popularity measurement rather than a usage one, and some of the timestamps the forge's interface returned for repository activity were not consistent with the dates of the queries.

Query	Result
Repositories containing the text `ucode`	About 1500, of which the majority are the unrelated projects above
Repositories defining or calling `uc_vm_init`	18 across 8 repositories, the language's own tree, netifd, uhttpd, rpcd, OpenWrt's main tree and three of its derivatives
Files with the interpreter's shebang line	115 across 8 repositories, all of them OpenWrt or an OpenWrt derivative
`.uc` files in `openwrt/openwrt`	42
`.uc` files in `openwrt/luci`	30
Stars of the language's own repository	174
Stars of `openwrt/openwrt`, `coolsnowwolf/lede`, `immortalwrt/immortalwrt`, `openwrt/luci`	28425, 31583, 11598, 7841

Read together, those numbers say something specific about the language's position: that it is used at a considerable scale and in a narrow place, that the embeddings and the scripts are roughly balanced rather than one being an accessory of the other, and that the disparity between the language's own repository and the trees that consume it is a factor of a hundred and a half rather than of two. A reader who came to part V wondering whether the language has a life outside the firewall that started it has the answer in the second and third rows of that table; a reader who came wondering whether it has a life outside OpenWrt has it in the absence of any other name from those rows.

Reading on

- Chapter 2 — obtaining and installing the interpreter, and what the packages contain.
- Chapter 17 — the module search path, which is what makes the installed layout work.
- Chapter 20 — the two flavours of embedded start-up that an embedding host presents.
- Chapter 48 — writing a native module, which is what the module packages name.
- Chapter 49 and chapter 50 — the example hosts in this tree, and a worked embedding.
- Chapter 52 — the deployment shapes that the consumers of this chapter illustrate.
- Chapters 53 to 58 — the consumers themselves, one chapter each.
- Chapter 60 — the debugger, which is the tool the language carries for itself.
- Chapter 61 — the test suites, the documentation generation and the tooling the repository carries.

The debugger

Source files referenced in this chapter: `lib/debug.c`, `lib/debug_remote.c`, `lib/debug_proto.c`, `lib/debug_proto.h`, `udbg.c`, `debug_highlight.c`, `debug_lineedit.c`, `main.c`, `docs/debugger.md`.

The `debug` module's own functions are documented in the generated reference at ucode-lang.org: *the debug module*.

The debugger is a source-level one: breakpoints by file, line and instruction offset, stepping, a stack with source context, expression evaluation in the suspended frame, and a disassembly of what is running. It is split into two programs. The debug core lives in the interpreter, in the `debug` module built from `lib/debug.c`, `lib/debug_remote.c` and `lib/debug_proto.c`; it decides when execution stops and answers questions about the suspended state. The client is the `udbg` binary built from `udbg.c`, `debug_highlight.c` and `debug_lineedit.c`; it reads your keystrokes, renders source with colour and formatting, and keeps the command history. Between them is a line-oriented text protocol: one uppercase verb, an optional space and a JSON object, terminated by a newline.

The split is the reason the debugger has the shape it does. The core emits no escape sequences and formats no columns; it exchanges structured data only. Everything presentational is in the client, which means a different client — an editor plugin, an IDE adapter, a test harness — drives the same core over the same lines, without reimplementing breakpoint resolution or frame walking. `udbg` is one client, not the only one the design contemplates.

Sessions are entered three ways, and once entered they are identical: a connected descriptor is handed to the same command loop, which announces the stop, then reads and dispatches commands until it is told to resume or to quit. What differs is only how the descriptor arrives.

Starting a session

The ordinary way is `-x`, which debugs the program locally:

```
$ build/ucode -L build -x greet /tmp/dbg1.uc
```

`greet` here is an optional breakpoint location, given attached to the flag; it is resolved with the same rules as the `break` command accepts, and when it is given the program stops there rather than at its first instruction. Without a location, `-x` stops at the first instruction of the program. The other spelling, `-X`, arms the debugging infrastructure but does not open a session by itself: it installs the `SIGUSR1` handler and waits for someone to ask for a session, and with an attached location it installs a breakpoint at that location, so that hitting it stops the program and waits for a client to connect. Both flags accept their location argument attached, as in `-xgreet` and `-Xworker`; a separated word is parsed as the file name argument.

Under the hood `-x` does not run a client in-process. It creates a socket pair, forks, and executes `udbg --fd 3` in the child with one end of the pair installed on descriptor 3, so that the child owns the real terminal — its own standard input and output remain the terminal that invoked `ucode`, and descriptor 3 is used for nothing but the protocol. The parent keeps the other end and runs the program. This is why a `-x` session responds to window resizing, history and line editing exactly as `udbg` run on its own does, and why the debug core never has to know which of the three transports it is talking over.

`udbg` itself can be run directly, in three forms. With a process id it sends `SIGUSR1` to that process and connects to the attach socket derived from the id, which is the way an already-running program is taken over. With a path it connects to that Unix socket, which is the peer of a program that called `debug.listen(path)`. With `--fd N` it uses descriptor N as an established connection, which is the form the `-x` flag's child process uses.

The first stop

A session opens with a report of where execution is, followed by a window of source. This is a session on the program of the same name, started without a location:

```
$ printf 'print who\nc\n' > cmds
$ build/ucode -L build -xgreet /tmp/dbg1.uc < cmds
Connected to ucode debugger

Paused (breakpoint) in greet(), /tmp/dbg1.uc:2:2
  breakpoint #1
[/tmp/dbg1.uc] main » greet
   1 function greet(who) {
   2     let msg = "hello " + who;
   3     return msg;
   4 }
dbg > "world"
dbg > hello world
*** program finished ***

Connection closed
```

The quoted file `dbg1.uc` is the seven-line program that the earlier sections use; its rendering of tabs is explained below. The report line names the stop reason, the function and the position; the indented `breakpoint #1` line names the breakpoint responsible, and it is present only when one is. The bracketed line is the frame path, `main » greet` reading as `greet` called from `main`. The source lines carry line numbers, and a stop within a function of another file shows that file. Colour surrounds all of it: the current statement's span is shaded, the exact instruction position is underlined, and a tab is drawn as `<->` in place of the character, expanded to four columns. The transcript above has the escape sequences removed, which is what makes the tabs visible as `<->` and the shading absent.

Commands are read on the session connection; `c` above is `continue`. Each command's output follows the `dbg >` prompt, and a session ends when the program ends or on `quit`. Feeding commands from a file, as the transcript does, works unchanged: nothing in the client requires a terminal, which is what makes scripted sessions of this kind possible at all. Chapter 61's debugger test suite drives itself the same way.

Where to stop

A location is given as one of:

Form	Meaning
`path`	the first instruction of the named file
`path:line`	the statement at that line of that file
`path:line:offset`	the statement at that byte offset within the line
`line` or `line:offset`	as above, in the file of the frame that is currently stopped
`name`, `object.method`	the first instruction of the named function
`(expression)`	an expression evaluated in the stopped frame, which must yield a function

The path form and the expression form are told apart on the first character: anything containing a slash or a colon, or beginning with a digit, is a location; anything else is looked up as a name first and evaluated as an expression if no function of that name exists. A parenthesised expression is therefore the way to name something the shape rules would read as a path, and it is also the way to address a value rather than a name, as in `(handlers.dispatch)`. An expression needs a stopped frame to be evaluated in; with no session open, only names resolve. That is why the `-x` and `-X` location arguments, which are resolved before the program starts, are limited to names and paths.

A location resolves to an instruction offset, and the resolution is the reason the debugger stops where it does rather than where you pointed. The statement boundaries the compiler recorded in the chunk — chapter 51's span records — are walked to find the statement containing the requested source position, and the breakpoint goes on the statement's first instruction. A breakpoint put on the line of a `for` header stops at the condition test of the loop rather than at its body, because that is the statement that line's position falls in; a breakpoint on a line that only continues an expression from the previous line resolves to the enclosing statement. Listing the breakpoints shows the resolved position, so when a stop lands somewhere adjacent to what was asked for, the listing says where the breakpoint went:

```
dbg > break 3
Breakpoint #4 added
dbg > list
#1    /tmp/dbg1.uc:3:14 - main()
(step) /tmp/dbg1.uc:1:1 - main()
(uncaught) <next instruction>
```

The listing carries your breakpoints with a number and the debugger's own with a name in parens, the name being the kind of the entry — `once`, `step`, `catch` and `uncaught` — and only a user entry carries the number at all. `step` is the internal breakpoint used by the single-step commands, and `uncaught` is the one that catches an exception that no handler takes; both of them are the mechanism of the two automatic stop reasons, and neither can be deleted by number. An entry with no function behind it prints as `<next instruction>` rather than as a position, which is what an `uncaught` stop on the way out looks like. The number in the acknowledgement of an added breakpoint counts the internal entries as well, while `list` and `delete` number user breakpoints among themselves from one, so `delete 1` refers to the entry listed as `#1`, which is the one just added.

A breakpoint that would have to sit inside a function nested in another one resolves to the innermost statement of the enclosing chunk whose recorded span covers the position, which for a line inside a function body is the declaration of that function. The reliable form inside a function is therefore the function name:

```
dbg > break greet
Breakpoint #4 added
dbg > c
Paused (breakpoint) in greet(), /tmp/dbg1.uc:2:2
  breakpoint #1
[/tmp/dbg1.uc] main » greet
   1 function greet(who) {
   2     let msg = "hello " + who;
   3     return msg;
   4 }
dbg >
```

`delete` removes the breakpoint the session stopped on when given no argument, and the numbered one when given one. A deleted breakpoint is out of the VM's list at once, and the command reports `OK`.

Moving

Four commands move execution. `step` — `s` — advances a single instruction, so it goes into a call; `next` — `n` — runs to the next statement of the current frame at the same nesting or shallower, so it walks over a call; `return` runs until the current frame returns; `continue` — `c` — runs until the program ends or some breakpoint is taken. `step` and `next` are the same routine with one flag switched, and all four leave the session open: the next thing printed is the next report, which is a `Paused` line for a step or breakpoint stop, and for an exception the `uncaught` report

shown below. Three of them have the short alias shown, and `return` is the one movement command with none. A command that has nothing to move to answers `ERROR` and stays paused, so a `step` at the last instruction of the outermost frame reports the failure and returns the prompt rather than resuming the program unattended.

The other two commands end the program. `throw` raises an exception at the current position, taking an optional type and a message as its payload: `throw "boom"` raises an uncaught user exception, which is the usual way to check what a handler will see. `quit` — `q` — terminates the program as `exit()` does, and the protocol verb has no confirmation of its own; what asks first is the client, which on a terminal prints `Terminate program? (y/n) >` and needs a `y` before it sends the verb, unless the command is given as `quit -f`. A piped session has no terminal to prompt on, which is why the scripted transcripts of this chapter simply end on `c` rather than on `quit`.

A step into a call that has no source is reported as a native frame; a step that leaves a frame whose file differs from the file you were in is reported with the frame path, which is how a step into library code is made legible.

Asking about the frame

```
dbg > backtrace
#2  [/tmp/dbg1.uc] greet()
     1 function greet(who) {
     2     let msg = "hello " + who;
     3     return msg;
#1  [/tmp/dbg1.uc] main()
     5
     6 let name = "world";
     7 print(greet(name), "\n");
     8
dbg > print who
"world"
dbg > eval who = "there"
OK
dbg > print who
"there"
dbg > c
hello there
```

`backtrace` — `bt` — lists frames innermost first, each with its file and function and a few lines of source around its position, with the numbers aligned so a frame that is deeper is visibly indented past its own number. `print` evaluates an expression in the stopped frame and shows the rendered value; this is the same rendering `print()` and `printf()` produce, sent as a pre-rendered string, which is what lets a closure, a resource or a regular expression appear in the reply at all. Names resolve in the stopped frame, so a `print` of a name that the current position has not brought into scope answers `null` rather than failing: above, `who` answers the argument's value because the stop is inside `greet`.

`eval` assigns as well as reads: it accepts an assignment expression, applies it to the stopped frame's storage and answers `OK`. The second `print` and the eventual program output both show the new value, which is the point: a debugger session can take a running program somewhere its input cannot reach. Assigning a local changes the slot from that point onward only if the name is still live, and that assigning to a global from a stopped frame is indistinguishable from the program having done it.

`variables` lists what the frame holds. It answers one line per entry with its name on the left and its value on the right, where a slot that the current position has no value in reports `<out of range>` rather than a value:

```
dbg > variables
(callee)          : <out of range>
greet             : <out of range>
```

Two further commands cover source. `sources` — `src` — lists the source buffers the program carries, numbered; `source` fetches and prints the raw text the core holds for a given file path, without highlighting, which is how a mismatch between the running program's text and the file on disk is spotted — a precompiled program keeps its source names and line tables without keeping the text, so the text shown is whatever the path now holds. `lines` — `ln` — prints the region around a location given as a file and line, as an offset, as `+n` or `-n` relative to the current position, or as an expression naming a function, with optional counts of surrounding lines.

Disassembling from the debugger

`disassemble` prints the instructions of a function, of a statement containing an offset, or of an expression, in a layout that pairs each instruction with its raw bytes:

```
dbg > disassemble
Function: greet
000000: 01 00 00 00 00     LOAD {0x0 : "hello "}
000005: 0a 00 00 00 01     LLOC {0x1 : local who}
000010: 2d                  ADD
```

The location forms are a name, `name+n` for the first *n* bytes of a function, `#offset` for the statement containing an instruction, `#from-to` for an instruction range, and a parenthesised expression. A listing that ends mid-instruction is a range request, not a truncated one. The names after each operand — `local who`, the printed constant — come from the debug information of the chunk, so they are absent for a program compiled without it, and the bytes themselves read with chapter 51's rules: an operand of one, two or four bytes, most significant first.

The value of this command is a question of the form "is the machine executing what I think this line means". The `ADD` above with no operand of its own, the constant folded into a `LOAD`, and the single `LLOC` are what `let msg = "hello " + who;` compiles to, and reading them against the source has no substitute when a computation is not the one written.

Inspecting from script

The debug core is a module, so a program can be inspected by itself. The names below are the module's, and they answer about the frame that calls them, which makes a debug module call the thing to put inside a suspected function rather than something to run against one.

```
import * as debug from "debug";

function inner(x) {
        let label = "frame-local";
        let pos = debug.sourcepos();
        let info = debug.getinfo(label);

        printf("line %d, byte %d\n", pos.line, pos.byte);
        printf("%s %s, refcount %d, length %d\n",
               info.type, info.value, info.refcount, info.length);

        for (let frame in debug.traceback()) {
                printf("frame %s at line %d\n", frame.callee, frame.line);
        }

        return x;
}

inner("value");
```

```
line 5, byte 28
string frame-local, refcount 3, length 11
frame function inner(x) { ... } at line 13
```

The three calls are the three kinds of question a program asks about itself.

- `sourcepos()` answers the position of the calling instruction as an object of file, line and byte, or `null` when the position has no source behind it.
- `getinfo(value)` answers a description of a value: type and value as the printer renders them, plus the reference count, an address, and the type's own measures — a length for a string, entries for a container. The reference count is a diagnostic of the counting rules of chapter 18 and nothing else, and the address is not a handle; both differ between runs of the same program, so the transcript above prints the fields that do not.
- `traceback(level)` answers an array of frames, innermost first, each with the callee, its `this`, whether it is a method call, whether it was compiled in strict mode, and its file, line and byte, plus the highlighted `context` text that an exception report prints. It is the same walk that builds the report of chapter 14, offered as data.

`getlocal(level, variable)` reads and `setlocal(level, variable, value)` writes a local of the frame chosen by level, level one being the caller of the call. The variable is given by name or by slot index, and the answer is an object giving the index, the name, the value and the extent of source over which the name is live; the extent comes from the same records the debugger matches names with:

```
import * as debug from "debug";

function inner(x) {
        let label = "frame-local";

        printf("by name: %J\n", debug.getlocal(1, "label"));
        printf("by index: %J\n", debug.getlocal(1, 1));
}

inner("value");
```

```
by name: { "index": 2, "name": "label", "value": "frame-local", "linefrom":
3, "bytefrom": 10, "lineto": 9, "byteto": 11 }
by index: { "index": 1, "name": "x", "value": "value", "linefrom": 3,
"bytefrom": 10, "lineto": 9, "byteto": 11 }
```

Both forms answer the same shape of thing, addressed two ways: by the name, which resolves to the entry of that name, and by the index, which resolves to whatever occupies that slot at this point. `getupval(target, variable)` and `setupval(target, variable, value)` do the same for a captured name, with the closure given as a value rather than a level; they are how the closure state of chapter 51 is read and changed from outside the closure:

```
import * as debug from "debug";

function counter() {
        let n = 5;

        let inner = function () {
                return n;
        };

        printf("captured: %J\n", debug.getupval(inner, 0));
        debug.setupval(inner, 0, 42);
        printf("now returns: %d\n", inner());
}

counter();
```

```
captured: { "index": 0, "name": "n", "closed": false, "value": 5 }
now returns: 42
```

The `closed` field says whether the upvalue still aliases a live slot of an enclosing frame, which here it does because `counter` has not returned; the write shows through the closure because the slot the upvalue aliases is the one `inner` reads.

Three calls reach a session. `debugger()` opens a local session now, exactly as `-x` does — it spawns the client, takes over `SIGINT` and installs the system breakpoints — or, with a function as its argument, puts its stop at that function's first instruction so that the session opens when the function is first entered. `listen()` with no argument arms `SIGUSR1` and returns; with a truish argument it arms it and then waits on the spot for a client, under the same thirty-second bound the signal path uses; with a string it binds that path as a socket and blocks on the accept there, for as long as it takes, unlinking the path once a client has arrived — which is the form to use when a service should offer a debugger on a known path. `attach()` is the module's own form of `-X`: it arms attach mode and installs the `SIGUSR1` handler that stops the program and brings up the session on the pid-derived socket, and a function argument puts the same stop on that function's first instruction that `debugger()` would. `breakpoint(spec)` installs a breakpoint from script with the location grammar of the command, taking the entry function as an optional second argument for the case where no frame is stopped yet, and answering the identifier or `false`. `break()` is not that: it takes no argument, and it stops the program where it is called, opening a session on the spot, which makes it the statement form of a breakpoint that is already placed. `notifyExit(status, code, exception)` is for a host that runs the interpreter and wants to hear about the program's end over the session, and it is what produces an exit event to a client that is attached when the program finishes.

The last call is `memdump(file)`, which writes a heap picture to a file: every slot of every frame, every argument and receiver, each named value with what its reference count then is. A file handle, a pipe handle or a socket may be given in place of a name. The same report is written on a signal, and the arrangement is worth knowing because it is how a wedged script is read out without having armed a session beforehand: the signal is `SIGUSR2` and the file is `ucode-memdump-<pid>.txt` in `/tmp`, and the three knobs are the environment variables `UCODE_DEBUG_MEMDUMP_SIGNAL`, `UCODE_DEBUG_MEMDUMP_PATH` and `UCODE_DEBUG_MEMDUMP_ENABLED`, the last of which disables the handler entirely on any value other than `1`, `yes` or `true`. Installing that handler installs an event-loop watcher for the signal, which is why loading the debug module makes the loop of chapter 36 handle signals — an effect to know about, since a program that installs its own disposition for `SIGUSR2` will have the debug module's installation replaced by it.

Being taken over from outside

The path that needs no foresight is `SIGUSR1`. A program started with `-X`, or one that called `listen()`, installs a handler; sending the signal stops it at the next instruction, opens the attach socket at `/tmp/ucode-debug-`*pid*`.sock` and waits for a client for up to thirty seconds, then runs a session on the connection. `udbg` given a process id performs both halves, signalling and connecting:

```
$ build/ucode -L build -X /tmp/dbg7.uc &
[1] 388196
$ printf 'print i\nc\n' > cmds
$ build/udbg 388196 < cmds
Debugger socket already present, connecting...
Connected to ucode debugger

Paused (step) in main(), /tmp/dbg7.uc:1:1
[/tmp/dbg7.uc] main
   1 let i = 0;
   2
   3 while (i < 40) {
dbg > 0
dbg > *** program exited (code -256) ***

Connection closed
```

The status of `-256` is the process having been killed by the signal that ends a session on a client that hangs up; the point of the transcript is the sequence, and it is the same sequence a developer uses against a daemon that cannot be restarted. The thirty seconds bound matters on a device: a `SIGUSR1` whose client never arrives resumes the program when it expires. `listen(path)` with a path is the alternative when the pid-derived name is inconvenient, and it is also the form for a process whose process id is not reachable from where the client runs, since the path may be any socket path the two agree on.

While a session is open the program is suspended, not running with a debugger beside it; the exception handler the debug core installs forwards to whatever handler was installed before it, so an exception reaches both the report the host would have printed and the attached client, which sees an event for it.

The protocol

Every exchange is a line. A verb may stand alone or be followed by a space and a JSON object; the payload is always an object when it is anything, which is what allows fields to be added to a verb without breaking a client that was written before they existed. The commands and their replies are:

Command	Payload	Reply
`BREAK`	`{"spec":...}`	`BREAKPOINT_ADDED {"id"}` or `ERROR`
`DELETE`	`{"id":N}`, omitted for the current one	`OK` or `ERROR`
`LIST_BREAKPOINTS`	—	`BREAKPOINTS {"items":[…]}`
`NEXT`, `STEP`, `CONTINUE`, `RETURN`	—	nothing of their own
`BACKTRACE`	`{"full":bool}`	`BACKTRACE {"frames":[…]}`
`VARIABLES`	—	`VARIABLES {"vars":[…]}`
`PRINT`	`{"expr":"…"}`	`VALUE {"repr"}` or `ERROR`
`EVAL`	`{"expr":"…"}`	`OK` or `ERROR`
`LINES`	`{"spec"?,"before"?,"after"?}`	`SOURCE_RANGE {"file","from","to","cursor"?}`
`SOURCE`	`{"file"}`	`SOURCE {"file","text"`
`SOURCES`	—	`SOURCES {"items":[{"index","file"}]}`
`THROW`	`{"type"?,"message"}`	no reply of its own
`DISASSEMBLE`	`{"spec"?}`	`DISASSEMBLY {"function","instructions":[…]}`
`HELP`	`{"command"?}`	`HELP {"commands":[{"verb","help"}]}`
`QUIT`	—	none

The core sends three verbs of its own. `PAUSED` opens every stop and carries the reason — `entry`, `breakpoint`, `step`, `exception`, `uncaught` or `interrupt` — with the position, the function, the identifier of the breakpoint when one is responsible, and the exception type and message when the stop is an exception. `EVENT` reports something that happened irrespective of where execution was: an exception being handled, a signal arriving, the program exiting. `ERROR` is the single failure shape for every command. The four movement commands have no acknowledgement of their own deliberately: the next thing a client sees is whatever really happened next, so there is no moment of "the step finished" for the core to report before the next stop or exit exists to report.

A frame in `BACKTRACE` is `{"kind","index","file"?,"line"?,"col"?,"insn"?,"function"?,"module"?}` with `kind` distinguishing a script frame from a native one, and carries a `variables` array when `full` was asked for. A variable is `{"name","kind","value_repr"}`, `kind` being `this`, `local`, `internal` or `upvalue`, and the value is carried as an already-rendered string because a ucode value may be a closure, a resource or a pattern, which have no JSON form. `LINES` carries no text either: it names a range of a file, and the text is fetched with `SOURCE`, which keeps the source of a big file out of every reply that mentions a line of it. The `file` field is the path as the core resolved it, relative to the working directory where it can be, and a client passes it back unchanged rather than reasoning about it.

Driving a session from a tool

Nothing in this design assumes a person at a terminal, and three properties make a tool session ordinary. The client reads its commands from its standard input when it has no terminal, so a scripted session needs no pty. The replies are structured, so a tool can parse them instead of reading the rendering — which is what it must do, since the rendering is the client's business and not the protocol's. And the transport for a session you start yourself is a socket, so a tool may hold either end.

For automation the practical shape is to keep a program's stop under control and drive it from one process: start it with `-X` and a location that stops it at a point you chose, connect `udbg` to its process id, and feed the commands on its standard input. That is what the transcript of the previous section does, and the test suite runs the debugger the same way over a table of scripts, commands and expected transcripts; chapter 61 has it.

Two things are worth knowing before automating a session. A command that moves execution does not report completion, so a driver reads until the next `PAUSED` or `EVENT` rather than expecting a reply per command. And a session that is abandoned leaves the program waiting to resume, so a driver ends with `CONTINUE` when the program should run on, and with `QUIT` when it should not; after `QUIT` the exit status the parent reports is the exit-status translation of chapter 14 rather than the program's own.

Summary

- The debug core is the `debug` module inside the interpreter and the client is `udbg`; between them is a line protocol of uppercase verbs with optional JSON object payloads, and the core renders nothing.
- A session opens with `-x` — locally, with `udbg` executed as a child over a socket pair — with `-X` and a `SIGUSR1` later, or from script with `debugger()`, `listen()` and `breakpoint()`. All three reach the same command loop.
- Locations are paths with line and byte offset, or function names, or expressions in parens; they resolve to a statement boundary from the compiled chunk's span records, which is why a stop can land on the adjacent statement, and inside a nested function a location is more reliably given as a function name.
- `step` takes one instruction and enters calls, `next` walks a statement at a time over them, `return` runs to the return, `continue` runs to the next stop or the end; `throw` raises and `quit` terminates.

- `backtrace`, `variables` and `print` ask about the stopped frame, `eval` writes into it, and `disassemble` shows what the machine is executing, with names that come from the debug information.
- The module's own calls — `sourcepos`, `getinfo`, `traceback`, `getlocal`, `setlocal`, `getupval`, `setupval` — are the same inspection, from inside the program; `memdump` writes the heap picture, on call or on a signal.
- Taking a running process over is `udbg <pid>`: the signal stops it, `/tmp/ucode-debug-`*pid*`.sock` is waited on for thirty seconds, and `listen(path)` is the alternative when that name does not suit.
- Automated sessions are ordinary: commands come from standard input, replies are structured, movement commands have no acknowledgement, and a driver closes with `CONTINUE` or `QUIT`.

Testing and tooling

Source files referenced in this chapter: `tests/CMakeLists.txt`, `tests/cram/CMakeLists.txt`, `tests/custom/CMakeLists.txt`, `tests/custom/run_tests.uc`, `tests/custom/99_debugger/run_debugger_tests.uc`, `tests/fuzz/CMakeLists.txt`, `jsdoc/conf.json`, `jsdoc/c-transpiler.js`, `package.json`, `.github/workflows/`, `debug_highlight.c`.

The repository tests the interpreter three ways, documents itself from its own sources, and ships the highlighting engine that an editor can borrow. Knowing where each piece sits pays off twice: a change can be checked in the way appropriate to it, and a strange result can be reproduced for whoever has to read about it.

Everything test-related is behind one build switch, `UNIT_TESTING`. It is off by default, and configuring with it on is what makes CTest know that there are tests at all:

```
$ cmake -S . -B build -DUNIT_TESTING=ON
$ cmake --build build
$ cd build && ctest -N
Test project .../ucode/build
  Test #1: cram
  Test #2: custom
  Test #3: debugger

Total Tests: 3
```

Three suites, three ways of working. `cram` runs black-box scenarios of the command line. `custom` runs the functional corpus through a runner written in ucode. `debugger` drives the debug protocol of chapter 60 against live processes. Beyond the switch, `UNIT_TESTING` also turns on `-DUNIT_TESTING` for the build and, under Clang, adds a second interpreter binary, `ucode-san`, built with the address, leak and undefined behaviour sanitisers, and a matching pair of test targets that run the same corpora through that binary.

The command line suite

Cram tests are text files with indented shell lines and their expected output underneath, and the file extension marks them: `tests/cram/test_basic.t`. CMake creates a Python virtual environment in the build tree, installs cram into it, and registers a test that runs the `test_*.t` files with two environment settings: `BUILD_BIN_DIR` set to the directory the built binaries live in, and `UCODE_BIN` set to the command that should be invoked as `ucode` — the sanitised binary under Clang, and the ordinary one wrapped in `valgrind --quiet --leak-check=full` otherwise.

The first lines of the file set up the environment for every scenario in it, and a scenario is a block of comment-indented commands with the output that must come back:

```
setup common environment:

  $ [ -n "$BUILD_BIN_DIR" ] && export PATH="$BUILD_BIN_DIR:$PATH"
  $ alias ucode="$UCODE_BIN"

  $ for m in $BUILD_BIN_DIR/*.so; do
  >   ln -s "$m" "$(pwd)/$(basename $m)"; \
  > done

check that ucode provides exepected help:

  $ ucode | sed 's/ucode-san/ucode/'
  Usage:
    ucode -h
```

The module symlinks in the setup are what let a scenario write `import * as fs from "fs"` without a `-L` argument: cram runs each file in a directory of its own, so the search path finds the module beside the script. This suite is the right home for anything that is really about the command line — an option, the shape of a usage message, the behaviour of a script run as an executable file — because it is the only suite that runs the binary the way a user does.

The functional suite

The bulk of the coverage is `tests/custom`, and its runner is itself a ucode program, `run_tests.uc`. The CTest target invokes it in strict mode with the module directory on the search path, and passes the interpreter to exercise through the environment rather than the command line, which is how the same corpus runs against the plain binary, the sanitised one or a wrapped one:

```
COMMAND $<TARGET_FILE:ucode> -L $<TARGET_FILE_DIR:fs_lib>/*.so -S
run_tests.uc
WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
```

Coverage is directories, and the layout is a numbered category per area followed by one file per subject:

```
tests/custom/
├── 00_syntax/          01_arithmetic/       02_runtime/
├── 03_stdlib/          04_modules/          06_metamethods/
├── 05_lib_serial/      06_lib_uloop/        17_lib_ffi/
└── 99_bugs/            99_debugger/
```

The runner globs the category directories, skips any whose name names a library the build does not have, globs the subjects inside each of them, and prints one line per subject; the last line totals the run, and the runner's own exit status

is the count of failed subjects, which is what CTest reads. Passing the path of a subject restricts the run to it, which is how a single subject is reworked:

```
$ cd tests/custom
$ ../../build/ucode -L ../../build -S run_tests.uc 01_arithmetic/01_division
##
## Running arithmetic tests
##

01_division ............................. OK
```

A category whose library is missing is announced rather than failed — "Skipping `uloop` tests (no library found)" — which is what lets a full run pass on a machine without every optional dependency.

The subject file

A subject file is a prose description of a behaviour followed by one or more test cases. The header is documentation and nothing else, and it is written in plain prose, because it is read when a case fails:

```
While arithmetic divisions generally follow the value conversion rules
outlined in the "00_value_conversion" test case, a number of additional
constraints apply.

-- Expect stdout --
Division by zero yields Infinity:
1 / 0 = Infinity
...
-- End --

-- Testcase --
print("Division by zero yields Infinity:\n");
printf("1 / 0 = %s\n", 1 / 0);
...
-- End --
```

Sections are introduced by a marker line and closed by `-- End --`. The recognised openers are `-- Args --` for the arguments to pass the interpreter, `-- Vars --` for environment settings of the form `NAME=value`, `-- Expect stdout --`, `-- Expect stderr --` and `-- Expect exitcode --` for what must come back, `-- File name --` for the contents of a helper file, and `-- Testcase --` for the program itself. Several `-- Testcase --` sections in one file give several cases sharing one set of expectations, which is the shape most subjects use. A closing marker written `-- End (no-eol) --` keeps the section's trailing newline out of the comparison.

The interpreter is started with `-T','`, so a test case is compiled as a template: code goes between `{%` and `%}`, values between `{{` and `}}`. A file whose subject is template behaviour is therefore written directly as a template, and one whose subject is ordinary code reads naturally all the same. Two globals are defined before the case runs: `TESTFILES_PATH`, the absolute path of the case's own `files` directory, and `UCODE_BIN`, the command line of the binary under test. A case that needs a second script — a module to import, a file to `include()` — keeps it in that directory and names it through `TESTFILES_PATH` rather than building paths from the working directory:

```
-- Testcase --
{%
        let real_printf = printf;

        include(TESTFILES_PATH + "/include.uc");
%}
-- End --
```

The two definitions are visible to the case as ordinary globals, which is what the following case prints when it is run the way the runner runs it, with `-T`, a module path and the two `-D` definitions:

```
{%
        printf("bin=%s files=%s\n", UCODE_BIN, TESTFILES_PATH);
%}
```

```
bin=build/ucode files=/tmp/ucode-ch61-demo/files
```

Each case runs as a child whose standard input is the case text, whose standard output and error are captured to temporary files, and whose working directory is the subject's own directory; the comparison rewrites that directory out of the captured text before comparing it, so a message that quotes the script's path is comparable between machines. A mismatch is reported as a labelled unified diff of expectation against result, on standard output, which is the one part of a failing run worth reading closely:

```
90_subject_name ......................... !
Testcase #1: Expected stdout did not match:
---
90_subject_name ......................... FAILED (1/1)
```

The `!` marks the case that diverged, the diff follows, and the line closes with the count.

Adding coverage is a directory entry and a file: the subject goes beside the ones on the subject's behavioural area, with a name continuing the numbering, and the runner finds it without anything being registered. The suite of chapter 60's protocol is kept apart from the rest for that reason, being self-selecting by location. The `99_bugs` category is where a

case goes when it was written against a specific report; its subjects are named after the behaviour they pin down, and a failing one of those reads as a regression rather than as a curious result.

The debug protocol suite

`tests/custom/99_debugger/run_debugger_tests.uc` is a second runner, registered as its own CTest target and also reachable on its own the way the functional runner is. It tests the protocol of chapter 60 rather than the rendering of `udbg`, and it does it without a terminal: each case starts the target program with `-X1` — attach mode with an initial breakpoint at line one — connects to the process's attach socket with the `socket` module itself, writes protocol lines, and asserts on the parsed replies and on the target's own standard output.

That shape is deliberate. Asserting on rendered text would tie the suite to the client's presentation; driving it over a pty would tie it to a terminal; and starting the target through `debug.listen()` from inside the script would exercise the nested-resume path rather than the one the interactive flows take, so a breakpoint installed during a session and reached after a later `CONTINUE` would not behave as it does in practice. The cases cover breakpoint installation and removal, the four movement commands, variable and upvalue inspection, backtraces, the source range requests, disassembly and the error shapes. Where the functional suite pins down what the language computes, this one pins down what a debugger sees.

Fuzzing

`tests/fuzz` holds the harness: a `test-*.c` per target, each built as a LibFuzzer binary with the fuzzer, address, leak and undefined-behaviour sanitisers, registered as a test that runs the binary against the `corpus` directory with a maximum input length of 256 bytes, a ten-second per-input timeout and a total budget of five minutes. The subdirectory is added only under Clang, and in the current tree it is commented out of `tests/CMakeLists.txt`, so configuring and building it is a manual step; the corpus directory carries no seeds, and the one target present is a skeleton whose entry point discards its input. Fuzzing the parser is therefore an activity to set up rather than one that runs in place.

Documentation from the sources

The reference pages on the website are generated from the repository, from the same block comments that sit above the C functions the modules register. The toolchain is JSDoc, driven by `package.json` and `jsdoc/conf.json`, and the one custom part is a JSDoc plugin, `jsdoc/c-transpiler.js`, which transpiles each C file — keeping line numbers aligned — so that the doc comments above the registration functions are read as JSDoc blocks. The configuration takes every `.c` file in the tree, and the output goes to `docs`. The theme is its own repository, `ucode-lang/ucode-jsdoc-theme`, wired in as a dev dependency of the toolchain:

```
$ npm run doc
$ ls docs/*.html | wc -l
76
```

Two families of page come out. `module-` *name* `.html` is the module reference — the functions, the resource types with their methods, the properties, the constants — and `lib_` *file* `.c.html` carries the same material with its source attached, which is why reading the two side by side is possible. The tutorials in `docs/tutorials` are Markdown with a manifest beside them, and they are published as `tutorial-` *nn*-.html`.

This matters here because the comments and the modules are one artefact: a function documented with `@function module:name#name` in its C file appears in the generated reference, and the same text is what a manual chapter must agree with. Where this book and the comments disagree, one of the two is wrong, and the comments have the advantage that they are regenerated; where the generated page documents a behaviour that the interpreter does not have, the C comment is the thing to correct. The chapters of part II and part III were checked against the modules by running them, which is the only authority, and against the reference pages for wording.

The generation runs in continuous integration on every push to the main branch, publishing the directory as a Pages site, and the workflow is `.github/workflows/jsdoc.yml` ; the other workflows build the Debian packages, the macOS build and the OpenWrt build against both the main branch and open pull requests.

Highlighting

The interpreter's own syntactic highlighter lives in `debug_highlight.c` , together with the line editor that `udbg` uses, and it is what produces the coloured source in a debugger session. It highlights both the language and template files, in the terminal dialect of the client; the client itself supplies the column layout and the frame path shown in chapter 60. It is in the repository for the debugger's sake rather than as a filter, and its command line use is the reason a plain `highlight` -style tool is not needed to see a script in colour where a terminal is all there is.

Third-party editor support — the tree-sitter grammars, the language server, the TextMate grammars, the TypeScript typings — is surveyed in chapter 59, together with what is absent: there is no Pygments or Rouge lexer, no Emacs mode, no formatter in the repository, and the `EDITORCONFIG` at the root is the extent of the in-tree formatting configuration. A chapter of this book that shows a script with a tab in it is showing what the file holds, and chapter 60 noted the debugger's way of drawing it.

Summary

- Tests are behind `UNIT_TESTING` ; with it on, CTest knows three of them: `cram` , `custom` and `debugger` , and under Clang the sanitised `ucode-san` binary and its pair of targets.
- `cram` runs the command line for real, with the built directory on the path and the built modules linked beside the script; it is where options and start-up behaviour are pinned down.

- `custom` is the functional corpus, run by `run_tests.uc` in strict mode: numbered categories, one file per subject, a prose description followed by cases with expectations, a labelled diff on a mismatch, the runner's exit status the count of failures, and categories for absent libraries skipped by name.
- A case is compiled in template mode and runs with `TESTFILES_PATH` and `UCODE_BIN` defined; helper files live in the subject's `files` directory, and the working directory is rewritten out of the comparison.
- `debugger` drives the protocol of chapter 60 over a socket against targets started with `-X1`, asserting on parsed replies and on the target's own output rather than on rendering.
- The fuzzer harness builds a LibFuzzer binary per target against the `corpus` directory; adding its subdirectory is manual today, the corpus carries no seeds and the one target is a skeleton.
- `docs` is generated from the module sources by JSDoc through the `c-transpiler` plugin, yielding the `module-` and `lib_` pages published by the `jsdoc` workflow, and it is the same documentation the manual chapters are written against.
- `debug_highlight.c` is the in-tree highlighter, used by the debugger client; editor integrations live outside the repository and are listed in chapter 59.

Appendix G: Source revisions and links

This appendix fixes what the book's file quotations refer to. A statement of the form `vm.c:1240` counts lines from a particular commit of a particular repository, so each repository is pinned here to the revision its files were read at, and every quoted file is given an address that carries that revision. Addresses are printed in full rather than tucked behind a label, because the book is meant to be readable on paper as well as on screen.

Two conventions are worth knowing. An address of the form `origin/blob/revision/path` names one file at one commit, so it keeps describing the text the chapter quoted however far the branch has moved since, and it ends in `#Lfirst-Llast` where the chapter quoted a span of lines. A path beginning with a slash, such as `/usr/share/firewall4/main.uc`, names a file as a running system holds it rather than a path in a repository; those are marked as install paths, and the file they are installed out of is what gets an address.

The list is drawn out of the chapters themselves rather than kept by hand: every path named in them is looked for in the pinned tree, a span that ran past the end of the file it names would be an error rather than a typo, and a chapter that starts quoting a new file or a new repository appears here the next time the appendix is generated.

The revisions

Repository	Branch	Revision	Committed	Quoted by
`ucode-lang/ucode`	`master`	`c05d2187547b309f64c5429739b2dfdc891c5d0c`	2026-10-03	the interpreter this book documents
`jow-/uwsd`	`master`	`1450021ba05a6a801e8c0ad446e3b2d8458ba976`	2026-09-10	the resident websocket and HTTP server
`openwrt/firewall4`	`master`	`c2ae8c8940a89407da32fbd662d4010ee2c9bbe6`	2026-08-27	the nftables frontend built as ucode programmes
`openwrt/luci`	`master`	`06e111ab07b902f2a746fcd465763eb78e2acf35`	2026-09-16	the web management interface built on ucode
`openwrt/netifd`	`master`	`06d06c86d757196e56bb13a1a8d6d69df7cad04d`	2026-09-11	an interface daemon embedding the library
`openwrt/openwrt`	`master`	`e2aa1d759647837a9e82e22ed5db9a3b79b27cc2`	2026-09-24	the distribution tree carrying the packages
`openwrt/openwrt`	`openwrt-24.10`	`352c0791754aba986a2ed7504be7a033189cfca7`	2026-09-23	the released branch of the same tree
`openwrt/rpcd`	`master`	`e37ed9d814699098eb7e26c8b33c054840782dfb`	2026-07-19	the RPC bus daemon with the

Repository	Branch	Revision	Committed	Quoted by
				ucode plugin
`openwrt/uhttpd`	`master`	`373145f72c884c36a2b16f7f47e74ffae06bd754`	2026-08-24	the HTTP server the ucode plugin serves

Each revision is the head of its branch as the quoted files were read. The file quotations of the book address these commits and no other.

Chapter 06, Operators

Read out of `ucode-lang/ucode`, branch `master`, at revision `c05d2187547b309f64c5429739b2dfdc891c5d0c`. Every address below is that repository's address of the file named, at that revision.

- `types.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/types.c#L1968-L1973`
- `vm.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/vm.c#L150-L151`
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/vm.c#L1947-L1949`
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/vm.c#L2042-L2048`

Chapter 13, Regular expressions

Read out of `ucode-lang/ucode`, branch `master`, at revision `c05d2187547b309f64c5429739b2dfdc891c5d0c`. Every address below is that repository's address of the file named, at that revision.

- `compiler.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/compiler.c#L76`
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/compiler.c#L676`

- `types.c`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/types.c#L1485`

Chapter 15, JSON and other notations

Read out of `ucode-lang/ucode`, branch `master`, at revision `c05d2187547b309f64c5429739b2dfdc891c5d0c`. Every address below is that repository's address of the file named, at that revision.

- `lib.c`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/lib.c#L3699-L3789`
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/lib.c#L6276`
- `types.c`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/types.c#L1708`

Chapter 19, Idiosyncrasies

Read out of `ucode-lang/ucode`, branch `master`, at revision `c05d2187547b309f64c5429739b2dfdc891c5d0c`. Every address below is that repository's address of the file named, at that revision.

- `vm.c`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/vm.c#L145-L154`

Chapter 25, fs

Read out of `ucode-lang/ucode`, branch `master`, at revision `c05d2187547b309f64c5429739b2dfdc891c5d0c`. Every address below is that repository's address of the file named, at that revision.

- `lib/fs.c`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/lib/fs.c#L119-L122`

Chapter 49, The six example programs

Read out of `ucode-lang/ucode`, branch `master`, at revision `c05d2187547b309f64c5429739b2dfdc891c5d0c`. Every address below is that repository's address of the file named, at that revision.

- `examples/execute-string.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/examples/execute-string.c`
- `examples/execute-file.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/examples/execute-file.c`
- `examples/native-function.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/examples/native-function.c`
- `examples/exception-handler.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/examples/exception-handler.c`
- `examples/state-reuse.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/examples/state-reuse.c`
- `examples/state-reset.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/examples/state-reset.c`
- `examples/CMakeLists.txt`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/examples/CMakeLists.txt`
- `main.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/main.c`

Chapter 50, A worked embedding

Read out of `ucode-lang/ucode`, branch `master`, at revision `c05d2187547b309f64c5429739b2dfdc891c5d0c`. Every address below is that repository's address of the file named, at that revision.

- `examples/execute-string.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/examples/execute-string.c`

- `examples/native-function.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/examples/native-function.c`
- `examples/exception-handler.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/examples/exception-handler.c`
- `include/ucode/lib.h`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/include/ucode/lib.h`
- `include/ucode/types.h`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/include/ucode/types.h`

Chapter 51, Inside the interpreter

Read out of `ucode-lang/ucode`, branch `master`, at revision `c05d2187547b309f64c5429739b2dfdc891c5d0c`. Every address below is that repository's address of the file named, at that revision.

- `lexer.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/lexer.c`
- `include/ucode/internal/lexer.h`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/include/ucode/internal/lexer.h`
- `compiler.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/compiler.c`
- `include/ucode/internal/compiler.h`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/include/ucode/internal/compiler.h`
- `chunk.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/chunk.c`
- `program.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/program.c`

- `vm.c`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/vm.c`
- `include/ucode/internal/vm.h`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/include/ucode/internal/vm.h`
- `include/ucode/internal/chunk.h`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/include/ucode/internal/chunk.h`
- `include/ucode/internal/program.h`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/include/ucode/internal/program.h`

Chapter 52, Deployment models

Read out of `ucode-lang/ucode`, branch `master`, at revision `c05d2187547b309f64c5429739b2dfdc891c5d0c`. Every address below is that repository's address of the file named, at that revision.

- `CMakeLists.txt`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/CMakeLists.txt`
- `main.c`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/main.c`
- `program.c`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/program.c`
- `include/ucode/program.h`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/include/ucode/program.h`
- `openwrt/ucode/Makefile`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/openwrt/ucode/Makefile`
- `debian/rules`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/debian/rules`

- `include/ucode/vm.h`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/include/ucode/vm.h`

Chapter 53, uhttpd: ucode as a web backend

Read out of `openwrt/uhttpd`, branch `master`, at revision `373145f72c884c36a2b16f7f47e74ffae06bd754`. Every address below is that repository's address of the file named, at that revision.

- `CMakeLists.txt`:
  - `https://github.com/openwrt/uhttpd/blob/373145f72c884c36a2b16f7f47e74ffae06bd754/CMakeLists.txt#L13`
  - `https://github.com/openwrt/uhttpd/blob/373145f72c884c36a2b16f7f47e74ffae06bd754/CMakeLists.txt#L77-L81`
- `ucode.c`:
  - `https://github.com/openwrt/uhttpd/blob/373145f72c884c36a2b16f7f47e74ffae06bd754/ucode.c#L28`
  - `https://github.com/openwrt/uhttpd/blob/373145f72c884c36a2b16f7f47e74ffae06bd754/ucode.c#L196-L222`
  - `https://github.com/openwrt/uhttpd/blob/373145f72c884c36a2b16f7f47e74ffae06bd754/ucode.c#L230-L318`
  - `https://github.com/openwrt/uhttpd/blob/373145f72c884c36a2b16f7f47e74ffae06bd754/ucode.c#L248-L256`
  - `https://github.com/openwrt/uhttpd/blob/373145f72c884c36a2b16f7f47e74ffae06bd754/ucode.c#L295-L303`
  - `https://github.com/openwrt/uhttpd/blob/373145f72c884c36a2b16f7f47e74ffae06bd754/ucode.c#L320-L323`
  - `https://github.com/openwrt/uhttpd/blob/373145f72c884c36a2b16f7f47e74ffae06bd754/ucode.c#L329-L387`
  - `https://github.com/openwrt/uhttpd/blob/373145f72c884c36a2b16f7f47e74ffae06bd754/ucode.c#L371-L372`
  - `https://github.com/openwrt/uhttpd/blob/373145f72c884c36a2b16f7f47e74ffae06bd754/ucode.c#L374-L377`
- `uhttpd.h`:
  - `https://github.com/openwrt/uhttpd/blob/373145f72c884c36a2b16f7f47e74ffae06bd754/uhttpd.h`

- `main.c`:
  - `https://github.com/openwrt/uhttpd/blob/373145f72c884c36a2b16f7f47e74ffae06bd754/main.c#L159-L162`
  - `https://github.com/openwrt/uhttpd/blob/373145f72c884c36a2b16f7f47e74ffae06bd754/main.c#L255-L270`
- `examples/ucode/handler.uc`:
  - `https://github.com/openwrt/uhttpd/blob/373145f72c884c36a2b16f7f47e74ffae06bd754/examples/ucode/handler.uc`
- `examples/ucode/dump-env.uc`:
  - `https://github.com/openwrt/uhttpd/blob/373145f72c884c36a2b16f7f47e74ffae06bd754/examples/ucode/dump-env.uc`

Chapter 54, uwsd: a persistent ucode web server

Read out of `jow-/uwsd`, branch `master`, at revision `1450021ba05a6a801e8c0ad446e3b2d8458ba976`. Every address below is that repository's address of the file named, at that revision.

- `script.c`:
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L317`
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L340-L344`
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L386-L396`
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L440`
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L465-L469`
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L504-L508`
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L504-L518`
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L548`
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L552-L562`
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L592`

    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L593-L598
    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L619-L629
    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L769
    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L772-L776
    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L834-L846
    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L1169-L1295
    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L1297-L1312
    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L1855-L1912
    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L2133-L2145
    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L2148
    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L2159-L2163
    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L2169-L2172
    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L2187-L2188
    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L2189
    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L2191-L2195
    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L2201-L2205
    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/script.c#L2208-L2209
- include/script.h:
    - https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/include/script.h

- `include/config.h`:
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/include/config.h#L26`
- `config.c`:
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/config.c#L175-L188`
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/config.c#L286-L288`
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/config.c#L851-L864`
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/config.c#L867-L874`
- `main.c`:
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/main.c#L81-L85`
- `example/chat.conf`:
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/example/chat.conf#L7-L13`
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/example/chat.conf#L15-L23`
- `example/handler.uc`:
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/example/handler.uc#L10-L26`
- `example/chat-server.uc`:
  - `https://github.com/jow-/uwsd/blob/1450021ba05a6a801e8c0ad446e3b2d8458ba976/example/chat-server.uc#L19-L24`

Chapter 55, rpcd: ucode as an ubus service

Read out of `openwrt/rpcd`, branch `master`, at revision `e37ed9d814699098eb7e26c8b33c054840782dfb`. Every address below is that repository's address of the file named, at that revision.

- `ucode.c`:
  - `https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L35`
  - `https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L84-L89`

- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L240-L328
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L384-L408
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L452-L460
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L466-L531
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L564-L641
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L665
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L666-L724
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L687-L798
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L767
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L775-L797
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L791
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L799-L871
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L947-L959
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L961
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L963
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L965
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L1012
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L1071
- https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L1076

    - `https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L1082`
    - `https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L1084-L1098`
    - `https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/ucode.c#L1090`
- `examples/ucode/example-plugin.uc` :
    - `https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/examples/ucode/example-plugin.uc`

Chapter 56, Case study: firewall4

Read out of `openwrt/firewall4` , branch `master` , at revision `c2ae8c8940a89407da32fbd662d4010ee2c9bbe6` . Every address below is that repository's address of the file named, at that revision.

- `root/sbin/fw4` :
    - `https://github.com/openwrt/firewall4/blob/c2ae8c8940a89407da32fbd662d4010ee2c9bbe6/root/sbin/fw4`
- `root/usr/share/ucode/fw4.uc` :
    - `https://github.com/openwrt/firewall4/blob/c2ae8c8940a89407da32fbd662d4010ee2c9bbe6/root/usr/share/ucode/fw4.uc`
- `root/usr/share/firewall4/main.uc` :
    - `https://github.com/openwrt/firewall4/blob/c2ae8c8940a89407da32fbd662d4010ee2c9bbe6/root/usr/share/firewall4/main.uc`
- `root/usr/share/firewall4/templates/` :
    - `https://github.com/openwrt/firewall4/tree/c2ae8c8940a89407da32fbd662d4010ee2c9bbe6/root/usr/share/firewall4/templates`
- `root/etc/init.d/firewall` :
    - `https://github.com/openwrt/firewall4/blob/c2ae8c8940a89407da32fbd662d4010ee2c9bbe6/root/etc/init.d/firewall`
- `tests/lib/mocklib/` :
    - `https://github.com/openwrt/firewall4/tree/c2ae8c8940a89407da32fbd662d4010ee2c9bbe6/tests/lib/mocklib`
- `root/sbin/fw4` , quoted as `fw4` :
    - `https://github.com/openwrt/firewall4/blob/c2ae8c8940a89407da32fbd662d4010ee2c9bbe6/root/sbin/fw4`

Chapter 57, Case study: the LuCI ucode runtime

Read out of `openwrt/luci`, branch `master`, at revision `06e111ab07b902f2a746fcd465763eb78e2acf35`. Every address below is that repository's address of the file named, at that revision.

- `modules/luci-base/ucode/`:
  - `https://github.com/openwrt/luci/tree/06e111ab07b902f2a746fcd465763eb78e2acf35/modules/luci-base/ucode`
- `modules/luci-base/ucode/uhttpd.uc`, quoted as `uhttpd.uc`:
  - `https://github.com/openwrt/luci/blob/06e111ab07b902f2a746fcd465763eb78e2acf35/modules/luci-base/ucode/uhttpd.uc`
- `modules/luci-base/ucode/http.uc`, quoted as `http.uc`:
  - `https://github.com/openwrt/luci/blob/06e111ab07b902f2a746fcd465763eb78e2acf35/modules/luci-base/ucode/http.uc`
- `modules/luci-base/ucode/dispatcher.uc`, quoted as `dispatcher.uc`:
  - `https://github.com/openwrt/luci/blob/06e111ab07b902f2a746fcd465763eb78e2acf35/modules/luci-base/ucode/dispatcher.uc`
- `modules/luci-base/ucode/runtime.uc`, quoted as `runtime.uc`:
  - `https://github.com/openwrt/luci/blob/06e111ab07b902f2a746fcd465763eb78e2acf35/modules/luci-base/ucode/runtime.uc`
- `modules/luci-base/src/lib/luci.c`:
  - `https://github.com/openwrt/luci/blob/06e111ab07b902f2a746fcd465763eb78e2acf35/modules/luci-base/src/lib/luci.c`
- `modules/luci-base/Makefile`:
  - `https://github.com/openwrt/luci/blob/06e111ab07b902f2a746fcd465763eb78e2acf35/modules/luci-base/Makefile`
- `contrib/package/`:
  - `https://github.com/openwrt/luci/tree/06e111ab07b902f2a746fcd465763eb78e2acf35/contrib/package`

Chapter 58, Case study: Wi-Fi

Read out of `openwrt/openwrt`, branch `master`, at revision `e2aa1d759647837a9e82e22ed5db9a3b79b27cc2`. Every address below is that repository's address of the file named, at that revision.

- `package/network/config/wifi-scripts/`:
  - `https://github.com/openwrt/openwrt/tree/e2aa1d759647837a9e82e22ed5db9a3b79b27cc2/package/network/config/wifi-scripts`

- `package/network/services/hostapd/`:
    - `https://github.com/openwrt/openwrt/tree/e2aa1d759647837a9e82e22ed5db9a3b79b27cc2/package/network/services/hostapd`
- `package/network/services/hostapd/patches/601-ucode_support.patch`, quoted as `patches/601-ucode_support.patch`:
    - `https://github.com/openwrt/openwrt/blob/e2aa1d759647837a9e82e22ed5db9a3b79b27cc2/package/network/services/hostapd/patches/601-ucode_support.patch`
- `package/network/services/hostapd/src/src/utils/ucode.h`, quoted as `src/src/utils/ucode.h`:
    - `https://github.com/openwrt/openwrt/blob/e2aa1d759647837a9e82e22ed5db9a3b79b27cc2/package/network/services/hostapd/src/src/utils/ucode.h`
- `package/network/services/hostapd/files/hostapd.uc`, quoted as `files/hostapd.uc`:
    - `https://github.com/openwrt/openwrt/blob/e2aa1d759647837a9e82e22ed5db9a3b79b27cc2/package/network/services/hostapd/files/hostapd.uc`

Chapter 59, The wider ecosystem

Read out of `ucode-lang/ucode`, branch `master`, at revision `c05d2187547b309f64c5429739b2dfdc891c5d0c`. Every address below is that repository's address of the file named, at that revision.

- `README.md`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/README.md`
- `debian/control`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/debian/control`
- `debian/rules`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/debian/rules`
- `openwrt/ucode/Makefile`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/openwrt/ucode/Makefile`
- `.github/workflows/`:
    - `https://github.com/ucode-lang/ucode/tree/c05d2187547b309f64c5429739b2dfdc891c5d0c/.github/workflows`
- `udbg.c`:
    - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/udbg.c`

- `debug_highlight.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/debug_highlight.c`
- `debug_highlight.h`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/debug_highlight.h`
- `debug_lineedit.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/debug_lineedit.c`
- `docs/debugger.md`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/docs/debugger.md`

Read out of `openwrt/netifd`, branch `master`, at revision `06d06c86d757196e56bb13a1a8d6d69df7cad04d`. Every address below is that repository's address of the file named, at that revision.

- `ucode.c`:
  - `https://github.com/openwrt/netifd/blob/06d06c86d757196e56bb13a1a8d6d69df7cad04d/ucode.c#L23-L27`

Read out of `openwrt/rpcd`, branch `master`, at revision `e37ed9d814699098eb7e26c8b33c054840782dfb`. Every address below is that repository's address of the file named, at that revision.

- `CMakeLists.txt`:
  - `https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/CMakeLists.txt#L12`
  - `https://github.com/openwrt/rpcd/blob/e37ed9d814699098eb7e26c8b33c054840782dfb/CMakeLists.txt#L69-L74`

Chapter 60, The debugger

Read out of `ucode-lang/ucode`, branch `master`, at revision `c05d2187547b309f64c5429739b2dfdc891c5d0c`. Every address below is that repository's address of the file named, at that revision.

- `lib/debug.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/lib/debug.c`
- `lib/debug_remote.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/lib/debug_remote.c`

- `lib/debug_proto.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/lib/debug_proto.c`
- `lib/debug_proto.h`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/lib/debug_proto.h`
- `udbg.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/udbg.c`
- `debug_highlight.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/debug_highlight.c`
- `debug_lineedit.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/debug_lineedit.c`
- `main.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/main.c`
- `docs/debugger.md`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/docs/debugger.md`

Chapter 61, Testing and tooling

Read out of `ucode-lang/ucode`, branch `master`, at revision `c05d2187547b309f64c5429739b2dfdc891c5d0c`. Every address below is that repository's address of the file named, at that revision.

- `tests/CMakeLists.txt`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/tests/CMakeLists.txt`
- `tests/cram/CMakeLists.txt`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/tests/cram/CMakeLists.txt`
- `tests/custom/CMakeLists.txt`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/tests/custom/CMakeLists.txt`

- `tests/custom/run_tests.uc`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/tests/custom/run_tests.uc`
- `tests/custom/99_debugger/run_debugger_tests.uc`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/tests/custom/99_debugger/run_debugger_tests.uc`
- `tests/fuzz/CMakeLists.txt`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/tests/fuzz/CMakeLists.txt`
- `jsdoc/conf.json`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/jsdoc/conf.json`
- `jsdoc/c-transpiler.js`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/jsdoc/c-transpiler.js`
- `package.json`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/package.json`
- `.github/workflows/`:
  - `https://github.com/ucode-lang/ucode/tree/c05d2187547b309f64c5429739b2dfdc891c5d0c/.github/workflows`
- `debug_highlight.c`:
  - `https://github.com/ucode-lang/ucode/blob/c05d2187547b309f64c5429739b2dfdc891c5d0c/debug_highlight.c`