The library, volume 2

The library, volume 2

The second volume of the report on raz. The first, `doc/raz.md`, defines the language; this one defines the modules that come with it, and refers to the first's sections by number, as "section 6.8". Only the core binds every implementation (appendix A of the first volume); the rest is the standard library, `Rand`, `Check` and `Prop` among it.

The library grows only by what breaks no program written for it. If a change must break programs, the old form stays for one minor version beside the new; the release notes list the old forms, and the next version removes them. Nothing marks an old form in the source.

Modules written in raz, in the compiler's `lib` directory:

Module	What
`Rt`	ending the program, `exit`, `exitStatus` and `atExit`; `stop`, which ends it with a message; `trace`, the calls under way; what the compiler's checks call when they fail
`Sys`	system calls: `open`, `openMode`, `openRead`, `create`, `read`, `write`, `seek`, `position`, `truncate`, `sync`, `syncAll`, `lock`, `lockShared`, `close`, `closeChecked`, `dup`, `dupTo`, `dupAbove`, `keepOnExec`, `mmap`, `munmap`, `mapRead`, `protect`, `isTerminal`, `yield`, `exit`; files and directories, `rename`, `remove`, `removeDir`, `makeDir`, `makeDirMode`, `chmod`, `setTime`, `access`, `umask`, `symlink`, `readLink`, `info`, `linkInfo`, `infoOf`, `openDir`, `nextEntry`, `entryKind`, `closeDir`; the terminal, `rawMode`, `restoreMode`, `getMode`, `setMode`, `termSize`, `ready`; the environment, `env`, `findEnv`, `envAt`, `setEnv`, `workDir`, `setWorkDir`; programs, `spawn`, `exec` and `execve`, pipes and signals; the user, the machine, time used and limits; time, `clock`, `now`, `sleep`; `random`; errors, by name, and what one means, `errorText`; sockets, among them datagrams served and files passed; and memory shared through a file. One file per target, and `Sys.raz` over the C library for the others, which takes what each C library fixes from `CAbi`
`Out`	writing text and numbers to standard output, buffered; safe to call from several threads
`C`	the parts of the C library the others use, and `printf`, `fflush`, `stdout`, `pthread_create`; a program that imports it uses C
`Lock`	a lock for short sections, for C's threads and `Thread`'s alike
`Thread`	threads, `spawn`, `join` and `signal`, and `Mutex`, in any program; in one that uses C, made by C's `pthread_create`

Module	What
`Task`	a thread given a value of its own, `Task(T)`: `spawn` copies it, with no cast
`Guarded`	a value with its lock, `Guarded(T)`: reached only through `with`
`Mem`	`Allocator`, the interface of every allocator, and pages from the system
`Heap`	a general allocator, `Heap.heap`, safe to call from several threads; freeing a block twice, or one it never gave out, stops the program; `RAZ_HEAP=debug` finds writes past a block's end and after free
`Arena`	an allocator that frees everything at once; `ADR(arena.base)` is its allocator
`Str`	operations on `[]U8` strings: comparing, hashing, finding, trimming and cutting, case, joining and replacing, UTF-8, numbers to and from text
`Bytes`	searches of bytes, many at a time: `index`, `last`, `count` and `find` in a `[]U8`; one file per target where the target has instructions for it
`Buf`	a growable byte buffer
`Fmt`	formatted output: `Fmt.print("{} of {}\n", [Fmt.int(n), Fmt.real(x, 2)])`, into a `Buf` with `Fmt.write`, or to standard error with `Fmt.eprint`
`List`	a generic singly linked list, `List(T)`
`Map`	a generic hash map, `Map(K, V; hash, eq)`
`Vec`	a generic growable array, `Vec(T)`, indexed as a slice
`Sort`	sorting a slice in place, `Sort(T; less)`
`Math`	roots, powers, logarithms, rounding and trigonometry, and `pi`, for `F64`
`Files`	an argument as a string, and a whole file read or written
`Opt`	a command line read through one table, by `getopt`'s rules with long options; its help and its mistakes
`Width`	how many columns of a terminal a character, or a string, takes: 0, 1 or 2, as `wcwidth` says
`Rand`	a seeded random number generator, splitmix64: the same seed, the same numbers; for tests and simulations, not secrets
`Check`	checks for a test: `ok`, `str` and `int` say what failed on standard error, and the program ends with status 1 if any did
`Prop`	property tests and fuzzing: a property draws values, a failing case is shrunk and saved, and run first next time

Lengths, counts, file descriptors, process ids and statuses are `INT`s, and a character is an `I32`. What may pass an `INT` on a small target is an `I64`: a file's size and position, a time in nanoseconds, and a number read from text.

What follows is each module's interface, as `tools/Doc.raz` prints it from the module's file; `...` stands for fields that are the module's own. A module's `main`, where it has one, is its test, and is left out here.

```
module Rt;
const Full*: ERROR = "too many procedures to run at exit";
proc atExit*(p: proc): ERROR;   (* p run when the program ends by exit *)
proc exit*(status: INT);        (* the atExit procedures, the last first, then the end
*)
proc exitStatus*: INT;          (* the status exit was given, in an atExit procedure;
0 before *)
proc stop*(msg: []U8);          (* msg, and a line feed, on standard error, then exit
with status 2 *)
proc trace*;                    (* the calls under way, and their variables, on
standard error *)
```

`Rt` also holds what the compiler's checks call when they fail, and the handlers of the signals that stop a program (section 10). `exit` takes each procedure off the list before it runs it, so it runs once even if it fails or calls `exit` itself; in a program that uses C, it ends through C's `exit`, so that C flushes its buffers too. `exitStatus` is the status `exit` was given, which a procedure `atExit` registered can read, to note a crash: a failed check and `stop` both end with 2. It is 0 before `exit` is called.

```
module Mem;
type
    Allocator* = record
        alloc*: proc(a: ^Allocator; size, align: INT): ^U8;   (* nil if out of room *)
        free*:  proc(a: ^Allocator; p: ^U8);
    end;
const Page* = 4096;
proc pages*(size: INT): ^U8;         (* size bytes, in whole pages, zeroed; or nil *)
proc unmap*(p: ^U8; size: INT);      (* pages given back *)
proc alloc*(a: ^Allocator; size, align: INT): ^U8;   (* a^.alloc(a, size, align) *)
proc free*(a: ^Allocator; p: ^U8);   (* a^.free(a, p) *)

module Heap;
proc debug*(on: BOOL);       (* debug mode on or off, for the blocks given from now on
*)
var heap*: ^Mem.Allocator;
var sure*: ^Mem.Allocator;   (* heap, but running out of memory stops the program *)

module Arena;
type Arena* = record
    base*: Mem.Allocator;     (* first, so that ^Arena(a) finds the rest *)
    ...
end;
proc init*(var ar: Arena; size: INT): BOOL;   (* an arena of size bytes; false if the
system has none *)
proc mark*(const ar: Arena): INT;             (* where ar stands now, for release *)
proc release*(var ar: Arena; m: INT);         (* all allocated since mark m freed; a
mark past where it stands stops the program *)
proc reset*(var ar: Arena);                   (* every allocation forgotten; the
memory used again *)
proc done*(var ar: Arena);                    (* its memory given back *)
```

`NEW(p, a)` calls `a^.alloc`. `Heap` is the general allocator; `Heap.heap` gives `nil` when memory runs out, and `Heap.sure` stops the program instead, as `Buf` does, so that `NEW(p, Heap.sure)` needs no check.

`Heap` has a debug mode, a light AddressSanitizer, chosen by `RAZ_HEAP=debug` in the environment or by `Heap.debug(true)`. Normal mode pays nothing for it. In debug mode each block given has 8 guard bytes after its last, and a freed block is filled with `0DEH` bytes and held in a quarantine, the last 1,024 frees, before it may be given again. A broken guard, a write past the block's end, is found when the block is freed; a block that leaves the quarantine with a byte not `0DEH`, a write after free, is found then. Either stops the program, saying what was found and the block's address and size, with the calls under way, and status 2. A big block, which has pages of its own, is not quarantined: its guard is checked and its pages unmapped, so a later use of it faults. `debug` changes only the blocks given after it, so call it before the first allocation; a block is freed as it was given, in either mode. Only writes are found, not reads, and not those past the guard or after the quarantine. An arena hands out memory from one block: its `free` does nothing, and `reset` forgets everything.

```
module Lock;
type Lock* = record ... end;
proc take*(var l: Lock);
proc give*(var l: Lock);

module Thread;
const Failed*: ERROR = "can't start a thread";
type
    Thread* = record ... end;
    Mutex* = record ... end;     (* zero is free *)
proc spawn*(p: proc(arg: ^U8); arg: ^U8): (^Thread, ERROR);   (* p(arg) in a new
thread *)
proc join*(t: ^Thread);         (* waits for t to end, then frees its stack *)
proc signal*(t: ^Thread; sig: INT): INT;   (* sig sent to t alone: 0, or minus an
errno *)
proc lock*(var m: Mutex);
proc unlock*(var m: Mutex);

module Task(T);
proc spawn*(p: proc(var x: T); x: T): (^Thread.Thread, ERROR);   (* p run on a copy of
x in a new thread *)

module Guarded(T);
type Guarded* = record ... end;       (* a value and its Thread.Mutex; zero is ready
*)
proc with*(var g: Guarded; p: proc(var x: T));   (* p run on the value, under the lock
*)
```

`Task(T)` starts a thread without a cast: `import Jobs := Task(Job); t, e := Jobs.spawn(work, job);` is the safe form of `Thread.spawn(work, ^U8(ADR(job)))`. `spawn` copies `x` into a block from `Heap` before it returns, so the caller may change or drop its `x` at once. The thread owns the copy: `work` changes it freely, and the thread frees it when `work` returns. The thread is joined with `Thread.join`; `Thread.Failed` comes back if there is no memory for the copy or the thread. A result goes back through a pointer in `x`, read after `join`.

`Guarded(T)` is a `T` with its `Thread.Mutex`, for sharing a container between threads: the value is reached only through `with`, which holds the lock while `p` runs. A zeroed `Guarded` is ready. `p` must not call `with` on the same `Guarded`, nor keep a pointer into its `x`.

A `Lock` is for short sections: a thread that finds it taken tries again, giving up the processor now and then. A `Mutex` sleeps. Each of `Thread`'s threads runs on a stack of its own, with a page below it that can't be touched, so that running off the stack stops the program with a message, as in the main thread. In a program that uses C, `Thread` makes its threads with C's `pthread_create`, so that C's library knows of them; in one that doesn't, with the system's own calls. The program is the same either way.

`Sys.kill` signals a program, and the system gives the signal to any of its threads; `signal` gives it to `t` alone, until `t` is joined, and is -3 (ESRCH) once `t` has ended. With `Sys.handle(sig)`, a call `t` waits in, a `read` or a `ready`, fails with -4, so that one thread can break into another's wait, as `^C` breaks into the main thread's. A signal sent before `t` begins its wait is only noted, so a program sends it again until `t` says it has stopped.

```
module Out;
proc str*(s: []U8);
proc char*(c: U8);
proc ln*;                          (* a line feed *)
proc int*(n: I64);                 (* n in decimal *)
proc hex*(n: U64; digits: INT);  (* n in hexadecimal, digits digits at least *)
proc flush*;
proc lines*(on: BOOL);             (* written at each line feed, or only when full *)
```

Out and C's stdio. `Out` keeps a buffer of `Sys.outSize` bytes, 4 KB on Linux. It writes it out when it is full, at each line feed when standard output is a terminal, at `Out.flush`, and when the program ends by `Rt.exit`. `lines(true)` has it written at each line feed whatever standard output is, as a filter of a stream that comes slowly wants through a pipe, as grep's `--line-buffered`; `lines(false)`, only when full, at a terminal too. A lock keeps what each call writes together when threads write at once. C's `stdio` keeps a buffer of its own for the same standard output. A program that writes with both flushes one before writing with the other, or the text comes out of order:

```
module Both;
import Out, C;      (* C: printf, fflush and C's FILE *stdout *)

proc main*;
begin
    Out.str("from raz\n");
    Out.flush;
    C.printf("from C\n");
    C.fflush(C.stdout);
    Out.str("from raz again\n");
end;

end Both.
```

Sys gives every target the same procedures, so that a program that needs only these runs wherever raz does: through system calls on Linux, through the C library elsewhere. A result that can fail is a number, minus an errno, as on Linux; 0 is success. The number is Linux's on every target: `Sys` over another system's C library gives the Linux errno of the same name, and a program tests it by the names below; one with no Linux twin keeps its own number. `Sys.raz`, the `Sys` over C, keeps to POSIX's names; what a C library fixes for its target,

the layouts of its structs, its numbers and the names that differ, it takes from `CAbi`, one file per target, which `tools/cabi.c`, compiled there, writes.

```
module Sys;
const
    stdin* = 0; stdout* = 1; stderr* = 2;
    outSize* = 4096;    (* the bytes Out keeps; each target has its own *)
    (* signals, by their numbers on Linux; each target has its own *)
    sigHup* = 1; sigInt* = 2; sigQuit* = 3; sigKill* = 9; sigPipe* = 13;
    sigTerm* = 15; sigChld* = 17; sigCont* = 18; sigStop* = 19;
    sigTstp* = 20; sigTtin* = 21; sigTtou* = 22; sigWinch* = 28;
    sigUsr1* = 10; sigUsr2* = 12; sigAlrm* = 14;
    (* the limits limit and setLimit take, by their numbers on Linux; each target has
its own *)
    limitCpu* = 0; limitFileSize* = 1; limitData* = 2; limitStack* = 3;
    limitCore* = 4; limitProcs* = 6; limitFiles* = 7; limitMemory* = 9;
    (* errors, minus Linux's errnos on every target *)
    errPerm* = -1; errNoEntry* = -2; errNoProcess* = -3; errIntr* = -4;
    errIo* = -5; errBadFd* = -9; errChild* = -10; errAgain* = -11;
    errNoMemory* = -12; errDenied* = -13; errExists* = -17;
    errNotDir* = -20; errIsDir* = -21; errInvalid* = -22;
    errNotTerminal* = -25; errNoSpace* = -28; errPipe* = -32;
    errNameTooLong* = -36; errNotEmpty* = -39; errLoop* = -40;
    errAddrInUse* = -98; errNetUnreach* = -101; errConnAborted* = -103; errConnReset*
= -104;
    errTimedOut* = -110; errRefused* = -111; errHostUnreach* = -113;
    errInProgress* = -115;

type
    Access* = (reading, writing, creating, exclusive, truncating, appending,
executing, nonblocking);
    Accesses* = set of Access;
    Info* = record
        size*: I64;      (* in bytes *)
        mtime*: I64;     (* when its contents last changed, in nanoseconds since 1970
*)
        seconds*: I64;   (* mtime in whole seconds, rounded down: exact past 2262 and
before 1677, where mtime wraps *)
        mode*: INT;      (* its permission bits, as chmod takes them: 1EDH is rwxr-xr-
x *)
        dir*: BOOL;      (* a directory *)
        file*: BOOL;     (* a regular file: not a directory, device, pipe or socket *)
        symlink*: BOOL;  (* the path itself is a symbolic link *)
        ino*: U64;       (* its inode, which a rename keeps and a new file doesn't *)
        dev*: U64;       (* the file system it is on; with ino, it names the file *)
        links*: U64;     (* how many names it has: 1 for a file no hard link shares *)
        special*: INT;   (* its set-user-id, set-group-id and sticky bits, as chmod
takes them: 800H, 400H, 200H *)
        uid*: INT;       (* its owner's user number *)
        gid*: INT;       (* its group's number *)
    end;
    Kind* = (unknown, isFile, isDir, isLink, isOther);   (* what a directory says a
name in it is *)
    Dir* = record ... end;   (* an open directory; its fields are the target's *)
    Program* = record
```

```
        path*: []U8;
        args*: [][]U8;      (* the first is its name *)
        env*: [][]U8;       (* NAME=value, each added to this program's environment *)
        onlyEnv*: BOOL;     (* env is its whole environment *)
        dir*: []U8;         (* its working directory; "" for this program's *)
        fds*: [3]INT;       (* its standard input, output and error; 0 for this
program's own *)
        group*: INT;        (* the process group it joins: 0 for this program's, -1 a
new one it leads *)
        foreground*: BOOL;  (* its group takes the terminal on standard input *)
        tty*: INT;          (* a pseudo-terminal's fd, to start a session on; 0 for
none *)
        more*: []INT;       (* fds above 2 passed on to it, at the same numbers *)
        keepIgnored*: BOOL; (* signals this program ignores stay ignored in it *)
    end;
    Status* = (running, exited, killed, stopped);
    Mode* = record ... end;   (* a terminal's modes; its fields are the target's *)
    Peer* = record ... end;   (* whom a datagram came from *)
    Remote* = record          (* who is at the other end of a connected socket *)
        unix*: BOOL;          (* a Unix socket: uid and pid say who *)
        ipv6*: BOOL;          (* addr is 16 bytes of IPv6, else 4 bytes of IPv4 then
zeros *)
        addr*: [16]U8;
        port*: INT;
        uid*, pid*: INT;
    end;
    Poll* = record            (* a descriptor for poll to wait on *)
        fd*: INT;
        read*, write*: BOOL;  (* wait until it may be read, written *)
        ready*: BOOL;         (* set by poll *)
    end;

proc write*(fd: INT; s: []U8): INT;     (* the bytes written *)
proc read*(fd: INT; buf: []U8): INT;    (* the bytes read; 0 at the end *)
proc open*(path: []U8; how: Accesses): INT;   (* its fd, closed on exec *)
proc openMode*(path: []U8; how: Accesses; mode: INT): INT;   (* open, a file it makes
given mode *)
proc openRead*(path: []U8): INT;         (* open(path, {reading}) *)
proc create*(path: []U8): INT;           (* open(path, {writing, creating,
truncating}) *)
proc seek*(fd: INT; pos: I64; fromEnd: BOOL): I64;  (* the new position *)
proc position*(fd: INT): I64;            (* where the next read or write is *)
proc truncate*(fd: INT; size: I64): INT; (* the file made size bytes long *)
proc sync*(fd: INT): INT;                (* what was written to it, on the disk *)
proc syncAll*(fd: INT): INT;             (* all that was written to its file system,
on the disk *)
proc lock*(fd: INT; wait: BOOL): INT;    (* the file's lock, held until fd is closed
*)
proc lockShared*(fd: INT; wait: BOOL): INT;   (* lock, shared with others that ask it
so *)
proc close*(fd: INT);
proc closeChecked*(fd: INT): INT;        (* close, and 0 or minus the errno it failed
with *)
proc dup*(fd: INT): INT;                 (* a new fd for the same file, the lowest
free *)
proc dupTo*(fd, into: INT): INT;         (* into made the same file as fd, closing
```

```
what it was *)
proc dupAbove*(fd, least: INT): INT;     (* a new fd for the same file, the lowest
free from least on, closed on exec *)
proc keepOnExec*(fd: INT; yes: BOOL): INT;   (* fd left open, or not (as at first for
a file made by open or pipe), in the program exec runs *)
proc isTerminal*(fd: INT): BOOL;
proc mapRead*(fd, size: INT): ^U8;     (* the first size bytes of fd, which may be
open only to read, mapped read only; or nil *)
proc protect*(p: ^U8; size: INT; read, write: BOOL): INT;   (* the pages at p readable
and writable as asked; neither, and touching them stops the program *)
proc mmap*(size: INT): ^U8;              (* size bytes of zeroed memory, in whole
pages; or nil *)
proc munmap*(p: ^U8; size: INT);
proc yield*;                             (* the processor given to another thread *)
proc exit*(status: INT);                 (* the program ended at once, without Rt's
atExit *)

proc rename*(old, new: []U8): INT;       (* replaces what new names *)
proc remove*(path: []U8): INT;           (* a file, not a directory *)
proc removeDir*(path: []U8): INT;        (* an empty directory *)
proc makeDir*(path: []U8): INT;          (* readable by all, writable by its owner *)
proc makeDirMode*(path: []U8; mode: INT): INT;   (* makeDir, given mode *)
proc chmod*(path: []U8; mode: INT): INT;
proc chown*(path: []U8; uid, gid: INT): INT;   (* its owner and group; -1 leaves one
as it is *)
proc makeFifo*(path: []U8; mode: INT): INT;    (* a named pipe, given mode *)
proc setTime*(path: []U8; sec: I64; nsec: INT): INT;   (* its mtime, and the time it
was read, made sec and nsec *)
proc access*(path: []U8; how: Accesses): INT;   (* 0 if this user may read, write or
execute it *)
proc umask*(mask: INT): INT;             (* the bits taken from every file made; the
mask it was *)
proc symlink*(target, path: []U8): INT;  (* path made a link to target *)
proc readLink*(path: []U8): ([]U8, INT); (* what the link names, good until the next
call *)
proc info*(path: []U8): (Info, INT);     (* following a symbolic link *)
proc linkInfo*(path: []U8): (Info, INT); (* the link itself *)
proc infoOf*(fd: INT): (Info, INT);      (* the open file's *)
proc space*(path: []U8): (I64, I64, INT);   (* the bytes free and in all on path's
file system *)

proc openDir*(var d: Dir; path: []U8): INT;
proc nextEntry*(var d: Dir): ([]U8, BOOL);  (* a name, good until the next call; false
at the end *)
proc entryKind*(const d: Dir): Kind;        (* what the name nextEntry gave last is *)
proc closeDir*(var d: Dir);

proc rawMode*(fd: INT): BOOL;      (* the terminal on fd: no echo, no lines, no
signals *)
proc restoreMode*(fd: INT);        (* as rawMode found it *)
proc getMode*(fd: INT): (Mode, INT);     (* the terminal's modes, to set again *)
proc setMode*(fd: INT; m: Mode): INT;    (* once what was written to it is sent *)
proc termSize*(fd: INT): (INT, INT);   (* columns and rows; 80 and 24 if unknown *)
proc ready*(fd, ms: INT): BOOL;          (* something to read within ms, or ever if ms
< 0 *)
proc readyOf*(fds: []INT; ms: INT): INT;   (* the first of fds with something to read
```

```
within ms, or -1 *)
proc readyFor*(reads, writes: []INT; ms: INT): INT;   (* readyOf, and the first of
writes with room to write *)
proc poll*(var fds: []Poll; ms: INT): INT;   (* sets each fd's ready, as each read and
write say; the number ready, 0 on a timeout, or minus an errno *)
proc setBlocking*(fd: INT; yes: BOOL): INT;   (* if not yes, a read or write that
would wait is -11 at once *)
proc blocking*(fd: INT): INT;            (* 1 if reads and writes wait, 0 if they are
-11 at once, or minus an errno *)

proc errorText*(err: INT): []U8;   (* what errno err, or minus it, means, as C's
strerror says *)

proc env*(name: []U8): []U8;       (* the environment variable's value; "" if unset *)
proc findEnv*(name: []U8): ([]U8, BOOL);   (* the value and true; "" and false if
unset, "" and true if empty *)
proc envAt*(i: INT): ([]U8, BOOL); (* the environment's ith NAME=value, from 0; false
past the last *)
proc setEnv*(name, value: []U8): INT;   (* name set to value, for env and the programs
started *)
proc workDir*: ([]U8, INT);        (* the working directory, good until the next call
*)
proc setWorkDir*(path: []U8): INT;

proc pipe*: (INT, INT);            (* the fd to read and the fd to write; or minus an
errno and -1 *)
proc openPty*: INT;                (* a pseudo-terminal: the fd of its controlling
side *)
proc setTermSize*(fd, cols, rows: INT): INT;   (* the size its programs are told of *)
proc spawn*(p: Program): INT;      (* a program started: its process id *)
proc exec*(p: Program): INT;       (* p run in this program's place; returns only if
it can't *)
proc execve*(path: []U8; args, env: [][]U8): INT;   (* as the kernel's execve: no
search *)
proc fork*: INT;                   (* this program copied: 0 in the copy, its process
id in this one *)
proc setsid*: INT;                 (* a session of its own, led by this program: its
process id, or minus an errno *)
proc wait*(pid: INT; block: BOOL): (INT, Status, INT);   (* which program, how, and
its code *)
proc waitExit*(pid: INT): (INT, Status, INT);   (* as wait, but only for a program to
end *)
proc kill*(pid, signal: INT): INT;
proc pid*: INT;                    (* this program's process id *)
proc group*: INT;                  (* the process group this program is in *)
proc setGroup*(pid, group: INT): INT;    (* pid 0 is this program; group 0 a new one
pid leads *)
proc foreground*(fd, group: INT): INT;   (* the terminal on fd gives its keys to group
*)
proc foregroundOf*(fd: INT): INT;  (* the group the terminal on fd gives its keys to
*)
proc uid*: INT;                    (* the user this program runs as *)
proc euid*: INT;                   (* the user whose rights it has: 0 is root *)
proc hostName*: ([]U8, INT);       (* this machine's name, good until the next call *)
proc processors*: INT;             (* the processors this program may run on, at least
1 *)
```

```
proc cpuTime*(children: BOOL): (I64, I64);   (* ns used in the program and in the
system; or by its children *)
proc limit*(which: INT): (I64, I64, INT);    (* the soft limit and the hard; -1 for
none *)
proc setLimit*(which: INT; soft, hard: I64): INT;

proc catch*(sigs: []INT): INT;     (* sigs caught: an fd to wait on *)
proc caught*(fd: INT): INT;        (* the next signal caught, or 0 if none is waiting
*)
proc ignore*(sig: INT; yes: BOOL): INT;  (* sig ignored; or, if not yes, back to what
it does at first *)
proc ignored*(sig: INT): BOOL;     (* whether sig does nothing now *)
proc handle*(sig: INT): INT;       (* sig noted, for came; a system call it comes
during is -4 *)
proc came*(sig: INT): BOOL;        (* whether sig came since the last time it was
asked *)

proc clock*: I64;                  (* nanoseconds, on a clock that only goes forward
*)
proc now*: I64;                    (* nanoseconds since 1970-01-01, UTC *)
proc sleep*(ms: INT);
proc random*(b: []U8);             (* b filled with random bytes; stops the program if
it can't be *)

proc connect*(path: []U8): INT;    (* a stream socket, connected to the Unix socket at
path: its fd *)
proc connectTcp*(addr: []U8; port: INT): INT;   (* addr an IPv4 or IPv6 address, as
numbers *)
proc connectUdp*(addr: []U8; port: INT): INT;
proc listen*(path: []U8): INT;     (* a Unix socket made at path, taking connections
*)
proc listenTcp*(addr: []U8; port: INT): INT;
proc accept*(fd: INT): INT;        (* the next connection, waiting for one *)
proc send*(fd: INT; s: []U8): INT; (* write, to a socket: -32 (EPIPE), not a signal,
if it is closed *)
proc sendNow*(fd: INT; s: []U8): INT;   (* send, but -11 (EAGAIN) where it would wait
*)
proc connectTcpWithin*(addr: []U8; port, ms: INT): INT;   (* connectTcp, or -110
(ETIMEDOUT) after ms *)
proc listenUdp*(addr: []U8; port: INT): INT;   (* a socket of datagrams on port at
addr *)
proc receive*(fd: INT; buf: []U8): (INT, Peer);   (* the next datagram, in buf, and
who sent it *)
proc reply*(fd: INT; peer: Peer; s: []U8): INT;   (* s, as one datagram, to peer *)
proc localPort*(fd: INT): INT;     (* the port the socket fd is bound to *)
proc closeWrite*(fd: INT): INT;    (* no more writing to the socket fd; the other end
reads an end *)
proc peer*(fd: INT): (Remote, INT);   (* who is at the other end of the connected
socket fd *)
proc socketPair*: (INT, INT, INT); (* two connected stream sockets, and 0 *)
proc sendFds*(fd: INT; s: []U8; fds: []INT): INT;   (* send, and with it fds, to a
Unix socket *)
proc receiveFds*(fd: INT; buf: []U8; fds: []INT): (INT, INT);   (* read, and the fds
that came with it *)

proc memFile*(size: INT): INT;     (* a file of size zero bytes, in memory, with no
```

```
name: its fd *)
proc mapFile*(fd, size: INT): ^U8; (* the first size bytes of fd, shared; munmap gives
them back; or nil *)
```

`info` follows a symbolic link: `size`, `dir` and `file` are those of what it names, and `symlink` says the path was a link, so that a program walking a tree can choose not to follow one back up it. A link to nothing is `-ENOENT`, as a missing file is. `file` lets a walk open only what holds data: opening a pipe would wait for a program to write to it.

`nextEntry` gives every name in the directory except `.` and `..`, in no particular order. `entryKind` says what the name it gave last is, a file, a directory, a symbolic link (whatever it names) or something else, as the directory itself records it, so that a program walking a tree needn't ask `linkInfo` of every name, a system call each; where the file system doesn't record it, it is `unknown`, and `linkInfo` must be asked. `rawMode` turns off what the terminal does with keys before the program reads them (echo, line editing, signals, flow control), and NL to CR NL on output. A program that turns it on restores it before it ends: `Rt.atExit` is the place.

Files. `open` gives a file for what `how` asks: `reading`, `writing` or both; `creating` it if it is missing, readable by all and writable by its owner; with `exclusive`, failing with -17 (EEXIST) if it is there, as a program does that makes a lock or a file of its own; `truncating` it to nothing; `appending`, each write going to its end, however other programs write it too. `openMode` is `open` that gives a file it makes the bits `mode`, less those the umask takes away: 1EDH (755) for a program, 124H (444) for a file never to be written; `open` gives 1A4H (644). `seek` moves to `pos` from the start of the file, or from its end if `fromEnd`, where a negative `pos` reads back from the end; `position` says where the file is, without moving. `truncate` makes the file `size` bytes long, cutting it or adding zeros, and leaves the position where it was. `sync` returns once what was written to the file is on the disk, as a program needs before it says a file is saved; `syncAll` returns once everything written to the file system that holds the file is on the disk, at once rather than a file at a time, and may flush other file systems too. `lock` takes the file's lock for one open file alone: another that asks for it waits if `wait`, and otherwise fails with -11 (EAGAIN). The lock goes when the file is closed or the program ends, however it ends, so it never goes stale as a file made with `exclusive` does. `lockShared` takes it shared, as a program that only reads asks it: any number may hold it so at once, and `lock` waits, or fails, until the last has let it go, as a shared one waits for the one that `lock` took. Both are `flock`'s, so they agree with any other program that takes the file's lock so. `closeChecked` closes as `close` does, and says if that failed: a file system may say only then that what was written never reached it, a full disk or a quota over NFS, so a program that must know its writes landed, as one that copies a backup, asks it. The fd is closed either way, so a failed `closeChecked`, -4 (EINTR) too, is never tried again. `infoOf` is `info` of an open file: its size is what it holds now, however it was written; `linkInfo` is `info` of a symbolic link itself, not what it names, so that `symlink` is true, `dir` and `file` false, and `size` the length of what it names, which need not be there. `Info.mtime` is what a program checks to see whether a file changed under it, and `Info.ino` whether it is still the same file, not one renamed over it; 0 on a target whose files have none. `Info.dev` is the file system the file is on: two files are one only if both `dev` and `ino` agree, which is how a program following links finds that one has led it back into a directory it is already in. `Info.links` is how many names the file has: more than 1 if a hard link shares it, so that writing to it writes to a file named elsewhere too; 0 on a target that doesn't say. `Info.mode` holds the nine bits of `rwxrwxrwx`, and `chmod` sets them, 1EDH (755) for a program, 1A4H (644) for data; `Info.special` holds the three above them, set-user-id 800H, set-group-id 400H and sticky 200H, which `chmod` sets too, with `mode | special`, as a program that copies a file keeps them. `Info.uid` and `Info.gid` are the owner's and the group's numbers; `chown` sets them, with -1 for one to leave as it is, so that a program that saves a file by writing another and renaming it over the first can give it the first's owner (only root may give a file to another user). `makeFifo` makes a named pipe, given the bits `mode` as `makeDirMode` is; `Info` of one has `file` and `dir` false, and `open`

with `nonblocking` doesn't wait for its other end. `makeDirMode` is `makeDir` that gives the directory the bits `mode`, less those the umask takes away, as `openMode` gives a file: 1C0H (700) for one only its owner may enter. Of the three above them, Linux gives a new directory only sticky; it takes set-group-id from the directory it is made in. `setTime` sets the file's time, following a symbolic link, and the time it was last read with it, as a program that copies or restores a file keeps its time, so that a later run sees it unchanged: `sec` seconds and `nsec`, from 0 to 999999999, nanoseconds since 1970. `Info.mtime` holds the time in nanoseconds, which an `I64` holds from 1677 to 2262 only; `Info.seconds` holds it in whole seconds, rounded down, whatever it is, so a program that copies times exactly takes `Info.seconds`, and the nanoseconds as `mtime - seconds * 1000000000`, which is right in any year, since the wrapping cancels out (section 6.1).

Programs. `openPty` makes a pseudo-terminal, a terminal that a program, a terminal emulator, plays the screen and the keyboard of: what it writes to the fd is typed, and what the programs on the terminal write it reads.

`spawn` starts the program at `p.path`, with `p.args` as its arguments, the first being its name. A `p.path` without a `/` is looked for as a shell looks for a command, and as C's `execvp` does: in each directory of this program's `PATH` in turn, an empty one being the working directory, or of `/bin:/usr/bin` if `PATH` is unset or empty; the first that runs is the program. If there is none, `spawn` says so, -2 (ENOENT) or -13 (EACCES), before it makes a child: no program is started, and no process id given for a program that was never there. A `p.path` with a `/` is not looked at first, and a program that is there but that the kernel then refuses to run (not a program, a bad interpreter) is a child that ends with status 127. Its environment is this program's with each `NAME=value` of `p.env` added or put in place of the old value, or, if `p.onlyEnv`, `p.env` alone. It starts in `p.dir`. Its standard input, output and error are `p.fds`; a 0 gives it this program's own, so `{path: "ls", args: ["ls"]}` runs `ls` as a shell runs it, with every other field zero. It gets no other open file but those of `p.more`, each at its own number, as a shell passes `3<file` on, every signal does what it does at first, and none is blocked; with `p.keepIgnored`, a signal this program ignores stays ignored in it, as the kernel's `execve` leaves it, which is what a program run under `nohup` wants of the programs it starts. If `p.tty` is a pseudo-terminal's fd, the program starts a session on that terminal: it is its controlling terminal, and its standard input, output and error, but that a `p.fds[i]` that is not 0 is that file instead, as for a job that reads `/dev/null`; `p.group` is not used.

`p.group` puts the program in a process group, the programs that a shell's job is made of: -1 makes it the first of a new group, whose number is its process id, and the other programs of the job join that number. The terminal gives its keys, and the signals of Ctrl-C and Ctrl-Z, to one group, the one in its foreground: `p.foreground` puts the program's group there before it starts, and `foreground` puts any group there, as a shell does to take the terminal back. `setGroup` moves a program to a group, as the shell does for each program it starts, for whichever of the two runs first; a shell runs in a group of its own, `setGroup(0, 0)`.

`exec` runs the program `p` describes in this program's place, as a shell's `exec cmd` does, found and started as `spawn` would start it, `p.tty`, `p.group`, `p.foreground`, `p.fds` and `p.dir` included, except that it keeps this program's other open files: each stays open in the program unless it is closed on exec, as the files `open`, `pipe`, a socket or `memFile` gives are, unless `keepOnExec(fd, true)` was said of them; `dup` and `dupTo` give files that stay open; and those of `p.more` are kept. It returns only if the program can't run: -2 (ENOENT) or -13 (EACCES) if there is none to run, with nothing done; or, if the program is found but won't start, minus the errno, with what `p` asks of the terminal, the group, the files and the directory done, and this program's signals as they were.

`execve(path, args, env)` is the kernel's execve, for a shell, which has looked for the program along its own `PATH` and set up the child already: the program at `path` is run with `args` and with `env`, `NAME=value`, for its whole environment. Nothing is searched for, and every fd and signal stays as it is, but that a handled signal

does what it does at first: an ignored one stays ignored, as `trap '' HUP` wants of the command it comes before. It returns only if it can't: minus an errno, with nothing changed.

A shell moves its own files with `dup` and `dupTo`, for `2>&1` and `exec >log` and around a builtin: `dupTo(fd, into)` makes `into` the same open file as `fd`, closing what `into` was, and `dup` gives a new fd for it, the lowest free, as a shell saves a file to give it back. `dupAbove(fd, 10)` gives one at 10 or above, closed on exec, out of the way of the fds a script names (`3>log`), as a shell keeps its own. `fork` copies this program, which goes on from there in both: it is 0 in the copy and the copy's process id in this one, as a shell runs a builtin or a procedure as a stage of a pipe. Only the thread that calls it is copied, so a program with threads mustn't; and what `Out` keeps is copied too, to be written twice, so `Out.flush` comes first. `setsid` gives the copy a session of its own, with no controlling terminal, as a daemon does so that closing its terminal doesn't reach it; it is -1 (EPERM) for a program that leads a process group already, which the copy doesn't. `group` is the process group this program is in: a shell starting on a terminal waits until it is `foregroundOf(0)` before it takes it. `access` says whether this user, by its real ids, may read, write or execute a file, as `test -r`, `-w` and `-x` ask, which `Info.mode` alone can't say; an empty `how` asks only whether it is there. `open` refuses `executing`, with -22 (EINVAL). `nonblocking` opens a file without waiting: a fifo opened to write waits for a reader and a fifo opened to read waits for a writer, but with `nonblocking` the open returns at once (-6, ENXIO, for a write with no reader), and a read or write that would wait is -11 later, as `setBlocking(fd, false)` makes it, and `blocking(fd)` says which: 1 if they wait, 0 if not. A file `open`, `openMode`, `openRead` or `create` gives is closed in a program `exec` or `execve` starts, as `pipe`'s are, so that a fifo or a log one program holds doesn't reach every program it runs; `keepOnExec(fd, true)` keeps one open, and `spawn` is unchanged, since it passes its `fds` and `more` as before. `umask` sets the bits taken away from the mode of every file this program, and those it starts, make, and says what they were. `envAt` gives the environment one variable at a time, from 0, for a shell to list it and to pass it on whole with `onlyEnv`. `env` gives "" for a variable that is unset and for one set to nothing; `findEnv` tells them apart, as `TZ` needs, unset meaning the local time and empty meaning UTC. `setEnv` refuses a name that is empty or holds an `=`, and a 0 in either, with -22; as C's `setenv`, it isn't safe while another thread reads the environment, and what `env` gave for a variable may be gone once it sets that variable again. `getMode` reads a terminal's modes, and `setMode` sets them again once what was written is sent, so that a shell keeps them for each job: a program stopped by Ctrl-Z with its terminal raw gets it back raw, and the shell its own between. `cpuTime` says how long the processor ran this program, in its own code and in the system for it, or, if `children`, the programs it started and has waited for, as `time` and `times` say. `limit` gives a limit, `limitFiles` say, soft and hard, and `setLimit` sets them, as `ulimit` does: a program may raise its soft limit up to the hard one, and lower either. `uid` is the user this program runs as, `euid` the one whose rights it has, 0 for root, as a prompt shows `#`; `hostName` is the machine's name. `processors` counts the processors the program may run on, as the system's affinity mask allows, for a program that makes a thread to each; it is never less than 1.

`wait` waits for the program `pid` to change, for any of this program's if `pid` is -1, or for any of the group `g` if `pid` is `-g`, as a shell waits for a job's programs, and says which one, and how: `exited`, with its exit status; `killed`, with the number of the signal that ended it; or `stopped`, by Ctrl-Z or a signal, with the signal's number. Without `block` it doesn't wait, and is 0 and `running` if no program changed. A program that has nothing to wait for is -10 (ECHILD). `waitExit(pid)` is `wait` that blocks until a program ends, and not for one that is stopped, so that a program waiting for its editor isn't woken by Ctrl-Z. `kill` sends a program a signal, `sigTerm` to end it; to a group, if `pid` is minus its number.

Signals. `catch` takes the signals `sigs` away from what they would do, ending the program most of them, and gives an fd that is ready to read when one comes, so that a program waits for signals with `ready` and `readyOf` as it waits for keys or a server. `caught` gives each signal that came, in turn. A later `catch` adds to the signals caught and gives the same fd. Catching `sigKill` or `sigStop`, or the signals `Rt` reports, SIGSEGV and SIGFPE,

is -22 (EINVAL). `ignore` makes a signal do nothing, as a shell ignores `sigTtou` so that it can take its terminal back, and `ignore(sig, false)` gives it back what it does at first. On a signal that `catch` took, either gives it up: the signal is no longer caught, with its default action or ignored, and `catch`'s fd gives the signals still caught. A program that `spawn` starts begins with neither. `ignored(sig)` says whether a signal does nothing now, because the program ignored it or because the one that started this one left it so, as a program in the background or under `nohup` finds `sigInt` and `sigHup`: it should leave those alone, and `handle` or `catch` the rest.

To wait for a child, a signal or a time, whichever is first, `catch` `sigChld` with the other signals and wait in `ready` on `catch`'s fd: a child that ends makes `sigChld` ready there, as a signal that came before the wait does, since `catch` holds it until `caught` takes it, so none falls between a `wait` that found nothing and the sleep after it. When the fd is ready, call `caught`, then `wait(pid, false)`. `handle` can't do this: a signal noted just before the call it should break isn't seen by it.

`handle` is for a program that must see a signal while it waits in a system call, as a shell waiting for its foreground program, or for a key, sees Ctrl-C: the signal is noted, and the call it comes during, `wait`, `read`, `ready` and the rest, fails with -4 (EINTR) rather than going on; `sleep` sleeps its time all the same. `came(sig)` says whether it came since the last time it was asked. Nothing else runs when the signal comes, so the program does what it does after it, in its own time; `Out` and `Files` try again after -4. `ignore(sig, false)` gives the signal back what it does at first, and the signals `catch` refuses, `handle` refuses too.

Time. `clock` is for measuring how long something takes: it never goes back, and says nothing of the date. `now` is the date and time, which a person or the network may set back. `sleep` waits `ms` milliseconds.

Sockets. `connect` is for talking to a server on the same machine, an X server at `/tmp/.X11-unix/X0` say; `connectTcp` for one across the network, given its address as numbers, `"127.0.0.1"` or `"::1"`: turning a name into an address is a library's work over these. `connectUdp` gives a socket of datagrams, each `send` one message to that address and each `read` one message from it, as a program asks a name server. What a socket receives comes by `read`, what it sends goes by `send`, and `close` ends it. `send` is `write` for a socket, except when the other end has closed it: `write` would stop the program with the signal SIGPIPE, where `send` returns -32 (EPIPE), so that a program learns its server has gone and can end as it wants. `listen` and `listenTcp` are the other side: a server's socket, on `path`, which must not be there yet, or on `port` at `addr`, `"0.0.0.0"` or `"::"` for every address the machine has; `accept` gives each connection that comes, as a socket of its own. A server started on port 0 gets a port the system chooses, which `localPort` says, as a test does that must not take a port in use. `connectTcp` waits as long as the system does, minutes for a host that never answers; `connectTcpWithin` gives up after `ms` milliseconds, with -110 (ETIMEDOUT).

`listenUdp` is the server's side of `connectUdp`: a socket of datagrams from anyone, as a name server has. `receive` gives the next datagram, at most `LEN(buf)` bytes of it, the rest lost, and a `Peer` that `reply` sends an answer to.

`closeWrite` ends this side's writing to a socket, so that the other end reads an end of input and can still answer on the same socket: a client sends its request, says it is done, and reads the reply. `peer` says who is at the other end of a connected socket: for a TCP one its address and port; for a Unix one the user and process that made the connection, as the system recorded them, so that a server can check that its client is the same user instead of trusting a directory's mode. `sendNow` is `send` that never waits: where the socket has no room it gives -11 (`errAgain`), without making reads on it non-blocking too. `socketPair` makes two connected stream sockets at once, as a `listen`, `connect` and `accept` would, with no name on disk. Each is closed in a program exec starts. `accept` gives -103 (`errConnAborted`) for a client that went before it was taken.

Passing files and sharing memory. A program can give another an open file over a Unix socket: `sendFds` sends `s`, at least one byte, and with it `fds`, at most 16, each a file the other program then has open too; `receiveFds` reads into `buf` as `read` does, puts the fds that came with it in `fds`, as many as there is room for, closing the rest, and says how many bytes and how many fds. Each is closed in a program exec starts. `memFile` makes a file that lives in memory, of `size` zeroed bytes, and `mapFile` maps it, or any file, so that what one program writes there another sees at once: so a program hands an X server or a Wayland compositor its pixels, by `sendFds` of a `memFile` both map, rather than by copying them down a socket. `mapFile` needs a file open to read and write; `mapRead` maps one that is open only to read, a dictionary that isn't the user's to write, for reading only: a write there stops the program. `protect` changes what may be done with pages `mmap` gave, from a page boundary: read, write, both, or neither, so that touching them stops the program, as a guard page after a buffer does, or a reserve mapped in full and opened as it fills. It takes the address as the system does, so with a C library it is that library's whether an address not on a page boundary is -22 (EINVAL) or rounded down.

`readyOf` waits on several file descriptors at once, as a program does that listens to a server and a terminal both; a descriptor at its end, or failed, counts as ready, since a read of it won't wait. It looks at the first 64 of `fds`, and `connect` and `listen` take a path of at most 107 bytes, as `struct sockaddr_un` holds; a longer one is -36 (ENAMETOOLONG). A path that starts with a zero byte is one of Linux's abstract names (D-Bus's `unix:abstract=`), which has no file: the address is as long as the name, with no zero after it.

`poll` is `readyFor` without its limits. Each `Poll` names a descriptor `fd` and whether to wait to `read` from it, to `write` to it, or both; `poll` waits up to `ms` milliseconds (as long as it takes if `ms < 0`), sets every one's `ready`, and gives how many are: all of them, in one call, not the first. It takes any number of descriptors (the list the system wants is on the stack for 64 or fewer, and in memory from `mmap`, given back before it returns, for more; -12 if there is none). A descriptor at its end or failed is ready, as for `readyOf`; so is one that isn't open, and a read or a write of it is -9 (EBADF); a negative `fd` is never ready. A timeout is 0 and a failure minus an errno (-4, if a signal `handle` took broke in), where `readyFor` gives -1 for both. `readyFor` and `readyOf` are written over it.

A program that both reads from a descriptor and writes much to it, as a terminal writes what is pasted to its program while it reads what the program writes back, must not wait on either: the program may be waiting for its own writes to be read. `setBlocking(fd, false)` has a read or a write on `fd` that would wait return -11 (EAGAIN) at once, a write having written what there was room for; `readyFor` waits, as `readyOf` does, for something to read on any of `reads` or for room to write on any of `writes`. A descriptor in both is given once, when it is ready for either; the 64 it looks at are `reads`, then `writes`.

Strings, buffers and collections.

```
module Str;
proc cmp*(a, b: []U8): INT;           (* -1, 0 or 1 as a sorts before, with or after
b, byte by byte *)
proc eq*(a, b: []U8): BOOL;
proc eqFold*(a, b: []U8): BOOL;       (* eq, an ASCII letter's capital and small the
same *)
proc copy*(a: ^Mem.Allocator; s: []U8): ([]U8, BOOL);   (* s's bytes in memory from a;
false if a has none *)
proc free*(a: ^Mem.Allocator; s: []U8);                 (* what copy gave from a,
given back; "" is none *)
proc hash*(s: []U8): U64;             (* FNV-1a, 64 bits *)
proc find*(s, sub: []U8): INT;        (* where sub first starts in s, or -1 *)
proc findFrom*(s, sub: []U8; from: INT): INT;   (* where sub first starts in s at or
after from, or -1 *)
proc findLast*(s, sub: []U8): INT;    (* where sub last starts in s, or -1 *)
proc hasPrefix*(s, p: []U8): BOOL;
proc hasSuffix*(s, x: []U8): BOOL;
proc trim*(s: []U8): []U8;            (* s without the spaces at either end *)
proc trimLeft*(s: []U8): []U8;        (* s without the spaces at its start *)
proc trimRight*(s: []U8): []U8;       (* s without the spaces at its end *)
proc cut*(s, sep: []U8): ([]U8, []U8, BOOL);   (* what comes before the first sep, and
after it *)
proc field*(s: []U8): ([]U8, []U8);   (* the first run of bytes in s that aren't
spaces, and what follows it *)
proc isDigit*(c: U8): BOOL;           (* 0 to 9 *)
proc isLetter*(c: U8): BOOL;          (* A to Z, a to z *)
proc isSpace*(c: U8): BOOL;           (* a space, as trim takes it *)
proc isPunct*(c: U8): BOOL;           (* printable, and neither a letter, a digit nor
the blank *)
proc hexVal*(c: U8): INT;             (* c's value as a hexadecimal digit, or -1 *)
proc lower*(s: []U8);                 (* s's ASCII capitals made small, in place *)
proc upper*(s: []U8);                 (* s's ASCII small letters made capitals, in
place *)
proc join*(var b: Buf.Buf; parts: [][]U8; sep: []U8);   (* the parts added to b, sep
between each two *)
proc replace*(var b: Buf.Buf; s, old, new: []U8);       (* s added to b, with new for
each old in it *)
proc encode*(c: I32): ([4]U8, INT);   (* c in UTF-8: its bytes, and how many *)
proc decode*(s: []U8): (I32, INT);    (* the character s starts with, and its length
in bytes *)
proc decodeLast*(s: []U8): (I32, INT);   (* the character s ends with, and its length
in bytes *)
proc next*(s: []U8; i: INT): (I32, INT);  (* the character at i, and where the next
starts; -1 at the end *)
proc valid*(s: []U8): BOOL;           (* s is UTF-8, every character rightly spelled
*)
proc chars*(s: []U8): INT;            (* the characters in s, as decode reads them *)
proc toInt*(s: []U8): (I64, BOOL);    (* s, in decimal; false if it isn't a number an
I64 holds *)
proc toHex*(s: []U8): (U64, BOOL);    (* s, in hexadecimal; false if it isn't a number
a U64 holds *)
proc toU64*(s: []U8): (U64, BOOL);    (* s, decimal digits only; false if it isn't
one, or a U64 can't hold it *)
```

```
proc fromC*(p: ^U8): []U8;            (* the bytes at p up to its zero byte, not
copied; "" for nil *)
proc toC*(s, out: []U8): ^U8;         (* s copied into out with a zero byte after it,
which holds LEN(s) + 1 bytes at least: PTR(out) *)
const Base64*, Base64Url*;            (* the digits of RFC 4648, sections 4 and 5 *)
proc base64With*(s, out, alphabet: []U8; pad: BOOL): INT;   (* s in base64 with
alphabet's 64 digits into out, which holds base64Size(LEN(s)) bytes at least, padded
with = if pad: the bytes written *)
proc unbase64With*(s, out, alphabet: []U8; pad: BOOL): (INT, BOOL);   (* s, base64 as
base64With wrote it, into out, which holds (LEN(s) + 3) div 4 * 3 bytes at least: the
bytes written; false if s is not that *)
proc base64Size*(n: INT): INT;        (* the bytes base64 makes of n bytes: 4 for each
3, or part of 3 *)
proc base64*(s, out: []U8): INT;      (* s in base64 into out, which holds
base64Size(LEN(s)) bytes at least: the bytes written *)
proc unbase64*(s, out: []U8): (INT, BOOL);   (* s, base64, into out, which holds
LEN(s) div 4 * 3 bytes at least: the bytes written; false if s is not base64 *)
proc fromInt*(n: I64): ([20]U8, INT); (* n in decimal: its bytes, and how many *)
proc le*(s: []U8): U64;               (* s, at most 8 bytes, the first the lowest *)
proc be*(s: []U8): U64;               (* s, at most 8 bytes, the first the highest *)
proc putLE*(s: []U8; v: U64);         (* v into s, the lowest byte first *)
proc putBE*(s: []U8; v: U64);         (* v into s, the highest byte first *)

module Buf;
type Buf* = record len*: INT; ... end;
proc init*(var b: Buf; a: ^Mem.Allocator);
proc room*(var b: Buf; n: INT): BOOL;    (* room for n more bytes; false if the
allocator has none *)
proc add*(var b: Buf; c: U8);
proc str*(var b: Buf; s: []U8);
proc spare*(var b: Buf; n: INT): []U8;   (* room for n more bytes, and those bytes, to
fill *)
proc bytes*(const b: Buf): []U8;         (* what it holds, until it next grows *)
proc cut*(var b: Buf; n: INT);           (* only the first n bytes kept, its memory
kept *)
proc reset*(var b: Buf);                 (* emptied, its memory kept *)
proc done*(var b: Buf);                  (* its memory given back *)

module Fmt;
type Arg* = variant
    int(i: I64);
    uint(u: U64);
    hex(h: U64);
    lowHex(h: U64);
    str(s: []U8);
    char(c: U8);
    bool(b: BOOL);
    real(r: F64; places: INT);
end;
const ShortMost* = 24;                      (* the longest Fmt.short *)
proc short*(x: F64; var out: [ShortMost]U8): []U8; (* x in the fewest digits that read
back, in out *)
proc print*(f: []U8; args: []Arg);             (* to standard output *)
proc eprint*(f: []U8; args: []Arg);            (* to standard error, once Out is
flushed *)
proc write*(var b: Buf.Buf; f: []U8; args: []Arg); (* added to b *)
```

```
proc into*(out: []U8; f: []U8; args: []Arg): (INT, BOOL); (* into out: the bytes, and
false if cut short *)

module List(T);
type
    Node* = record next*: ^Node; val*: T; end;
    List* = record head*: ^Node; len*: INT; ... end;
proc init*(var l: List; a: ^Mem.Allocator);
proc push*(var l: List; v: T): BOOL;     (* v at the front; false if the allocator has
no room *)
proc pop*(var l: List): (T, BOOL);       (* the front value, taken off; false if the
list is empty *)
proc reverse*(var l: List);
proc clear*(var l: List);

module Map(K, V; hash: proc(k: K): U64; eq: proc(a, b: K): BOOL);
type Map* = record len*: INT; ... end;
proc init*(var m: Map; a: ^Mem.Allocator);
proc put*(var m: Map; k: K; v: V): BOOL;         (* k now maps to v; false if there
was no room *)
proc remove*(var m: Map; k: K): BOOL;            (* false if k wasn't there *)
proc get*(const m: Map; k: K): (V, BOOL);        (* what k maps to, and whether it
maps to anything *)
proc next*(const m: Map; i: INT): (INT, K, V);   (* the entry after i *)
proc done*(var m: Map);                          (* its memory given back, leaving it
empty *)

module Vec(T);
type Vec* = record len*: INT; ... end;
proc init*(var v: Vec; a: ^Mem.Allocator);
proc room*(var v: Vec; n: INT): BOOL;    (* room for n more; false if the allocator
has none *)
proc push*(var v: Vec; x: T);            (* x added at the end *)
proc pop*(var v: Vec): (T, BOOL);        (* the last element, taken off; false if the
array is empty *)
proc add*(var v: Vec): ^T;               (* a new element at the end, zero, to fill in
place *)
proc tryPush*(var v: Vec; x: T): BOOL;   (* push; false, and v as it was, if there is
no room *)
proc tryAdd*(var v: Vec): (^T, BOOL);    (* add; nil and false, and v as it was, if
there is no room *)
proc resize*(var v: Vec; n: INT);        (* n elements: cut short, or the new ones
zero *)
proc insert*(var v: Vec; i, n: INT);     (* n zero elements opened at i; stops the
program if there is no room *)
proc remove*(var v: Vec; i, n: INT);     (* the n elements from i taken out, those
after moved down *)
proc all*(const v: Vec): []T;            (* the elements, until it next grows *)
proc reset*(var v: Vec);                 (* emptied, its memory kept *)
proc done*(var v: Vec);                  (* its memory given back *)

module Sort(T; less: proc(a, b: T): BOOL);
proc sort*(s: []T);

module Math;
const pi* = 3.141592653589793;
```

```
proc sqrt*(x: F64): F64;
proc pow*(x, y: F64): F64;            (* x to the power y *)
proc floor*(x: F64): F64;
proc ceil*(x: F64): F64;
proc round*(x: F64): F64;             (* to the nearest whole number; a half away from
0 *)
proc roundEven*(x: F64): F64;         (* to the nearest whole number; a half to the
even one *)
proc exp*(x: F64): F64;
proc ln*(x: F64): F64;
proc log2*(x: F64): F64;
proc log10*(x: F64): F64;
proc sin*(x: F64): F64;
proc cos*(x: F64): F64;
proc tan*(x: F64): F64;
proc tanh*(x: F64): F64;              (* from -1 to 1 *)
proc asin*(x: F64): F64;              (* from -pi/2 to pi/2 *)
proc acos*(x: F64): F64;              (* from 0 to pi *)
proc atan*(x: F64): F64;              (* from -pi/2 to pi/2 *)
proc atan2*(y, x: F64): F64;          (* the angle of (x, y) from the x axis, from -pi
to pi *)
proc bits*(x: F64): U64;              (* the bits of x, as IEEE 754 lays them out *)
proc fromBits*(u: U64): F64;          (* the F64 whose bits are u *)
proc mul*(a, b: U64): (U64, U64);     (* a times b: the low 64 bits of the product,
and the high *)
proc divMod*(hi, lo, d: U64): (U64, U64); (* hi, lo divided by d: the quotient and the
remainder; hi < d *)
proc toReal*(s: []U8): (F64, BOOL);   (* s as a real, the F64 nearest it; false if it
isn't one *)
```

A `Buf`, a `List`, a `Map` and a `Vec` take their memory from the allocator `init` gives them. `Buf.add`, `str` and `spare` stop the program, as a failed check does, if the allocator has no room; `room` asks first. A program reads into a `Buf` through `spare`, then adds what it read to `len`: `s := Buf.spare(b, 4096); n := Sys.read(fd, s); if n > 0 then INC(b.len, n); end;`. `cut` takes back what was added, to a length it had: `start := b.len; ...; Buf.cut(b, start);`; a length past `len` stops the program, as an index past the end does. A `Vec` is `Buf` for any `T`: one block, which doubles when it is full, so `push` costs the same on average however long it grows. `all` gives the elements as a slice, indexed and checked as any is, and `v.len` is their number; the slice, and what `add` gives, are good until the array next grows. `pop` takes the last element off, keeping the memory. `push`, `add` and `resize` stop the program, as `Buf.add` does, if the allocator has no room; `room` asks first: `import Lines := Vec([]U8);`. `Fmt`'s format takes the next argument at each `{}`; `{{` and `}}` stand for `{` and `}`. A `Map` hashes and compares its keys with the procedures it is given: `import Count := Map([]U8, INT, Str.hash, Str.eq);`.

`Str` reads and writes UTF-8. `encode` writes a character, of at most 10FFFFH, in one to four bytes; a number that isn't one becomes FFFDH, the character that stands for a wrong one. `decode` reads the first character of `s`, and gives FFFDH and 1 for a byte that doesn't start one rightly, and 0 and 0 for an empty `s`: a continuation, a byte followed by too few continuations, a surrogate, a character beyond 10FFFFH or one spelled in more bytes than it needs. `decodeLast` reads the last character in the same way, to walk a string backwards. `valid` says that `decode` finds no such byte in `s`, and `chars` counts the characters, each such byte one. `next` walks a string a

character at a time, as `decode` reads them: from 0, it gives the character at `i` and where the one after it starts, and 0 and -1 when `i` is at the end, or past it:

```
c, i := Str.next(s, 0);
while i >= 0 do
    ...                                (* c is a character of s *)
    c, i := Str.next(s, i);
end;
```

A string is bytes, and `Str` takes no more of Unicode than UTF-8: `lower`, `upper` and `eqFold` change and match ASCII letters alone, the character tests know ASCII alone, and nothing says how wide a character is on a terminal, nor which spellings of a text are the same.

A space, to `isSpace`, `trim`, `trimLeft`, `trimRight` and `field`, is a blank, a tab, a line feed, a carriage return, a vertical tab or a form feed. `cut` gives what comes before the first `sep` in `s`, what comes after it, and `true`; or `s`, an empty string and `false` if `s` has no `sep`. `field` passes the spaces at the start of `s` and gives the bytes up to the next space or the end, and the rest of `s` from there; two empty strings if `s` holds only spaces. They take a string apart without memory of their own, each part a slice of `s`:

```
line, rest, more := Str.cut(text, "\n");     (* the first line, and the others *)
key, val, ok := Str.cut(line, "=");
w, rest := Str.field(line);                 (* the words of a line *)
while LEN(w) > 0 do
    Out.str(w); Out.ln;
    w, rest := Str.field(rest);
end;
```

`lower` and `upper` write `s`'s bytes, so they take a variable's, not a string literal's (section 5.10). `join` and `replace` add to a `Buf` as `Buf.str` does; an empty `old` adds `s` as it is.

`toInt` reads a decimal number, with a `-` or `+` before it if any, and nothing else. `toHex` reads hexadecimal digits, capitals or not, and nothing else: no sign, `0x` or `H`. `fromInt` writes a number as `Out.int` does, with a `-` if it is negative, its bytes the first of the 20 it gives. `toU64` reads a decimal number as `toInt` does, but digits only (no sign), up to the largest `U64`: what `Fmt.uint` writes.

`fromC` and `toC` are the two ends of a call to C. `fromC` makes a slice of the bytes at a pointer up to its zero byte, without copying them; `nil` gives `""`. `toC` copies `s` into `out`, puts a zero after it and gives `PTR(out)`, so a call reads `C.realpath(Str.toC(path, room), ...)`. `out` holds `LEN(s) + 1` bytes at least; a shorter one stops the program, as any slice does.

`base64With` and `unbase64With` take the 64 digits as `alphabet` and say with `pad` whether `=` pads a last group: `Base64Url` is section 5's, with `-` and `_`, and `Base64` section 4's. Without padding a last group has 2 or 3 digits, and a length 1 more than a multiple of 4 is no base64. `base64` and `unbase64` are `Base64` with padding. In detail, `base64` writes `s` as RFC 4648 (section 4) does: the digits `A` to `Z`, `a` to `z`, `0` to `9`, `+` and `/`, and `=` to pad a last group of 1 or 2 bytes to four. `unbase64` reads that and nothing else: its length a multiple of 4, `=` only as the last one or two bytes, and the bits of the digit before the padding that no byte uses all 0, so a byte string has the one spelling `base64` gives it. Line breaks, missing padding and the `-` and `_` of base64url are the caller's to mend first; `unbase64` of such a string is false, and what it wrote into `out` is not to be used. The caller gives the room, as `Fmt` does: an `out` shorter than the comment says stops the program, as indexing past a slice does.

`le` and `be` read an unsigned number of up to 8 bytes in a fixed order, little-endian or big-endian, whatever the machine's own, and `putLE` and `putBE` write one; more than 8 bytes stops the program. They are how a file format or a network protocol is read and written:

```
len := U32(Str.le(header[4..8]));
Str.putBE(packet[0..2], U64(port));
```

`Math.toReal` reads a real as a constant is written (section 2), with a `-` or `+` before it if any, its point and the digits after it, or its exponent, left out if it has none, and `e` taken as `E`: `2`, `-0.5` and `1.5e-3` are reals to it. It gives the `F64` nearest the number, and `false` for anything else, or a number too big for an `F64`.

`Map.next` walks a map: from 0, it gives the next entry and the number to pass for the one after it, and -1 when there are no more. A walk sees each entry once if the map doesn't change during it:

```
i, k, v := Names.next(names, 0);
while i >= 0 do
    Fmt.print("{} is {}\n", [Fmt.str(k), Fmt.int(v)]);
    i, k, v := Names.next(names, i);
end;
```

`Sort.sort` puts `s` in the order `less` gives, in place, without memory of its own, in time `n log n` at most; two elements equal by `less` may change places. `Math`'s procedures are as C's `math.h` has them, to within the last bit or so; `roundEven` is IEEE 754's default rounding, as Pascal's `Round`, where `round` is C's. `Fmt.real(x, places)` writes `x` with `places` digits after the point, at most 9, rounded to the nearest; from 1E15 on, with an exponent, as `1.50E17`; and `nan`, `inf` and `-inf` as themselves. `Fmt.short(x, out)` writes `x` into `out`, from 0, and gives those bytes: the fewest significant digits that `Math.toReal` reads back as the same `F64`, and of two such the nearer `x`: `0.1`, not `0.10000000000000001`; `42`, with no point; `0.0001`; and below 1E-4 or from 1E15 on with an exponent, as `1.5E17` or `5E-324`. Zero keeps its sign, `0` or `-0`, and `nan` and `inf` are as `real` writes them. The longest is `Fmt.ShortMost` bytes, so a call is `var d: [Fmt.ShortMost]U8; Fmt.print("{}", [Fmt.str(Fmt.short(x, d))]);`, and a width goes with `str`. The bytes are `out`'s: a second call with the same `out` overwrites them. `hex` writes capitals and `lowHex` lower case, neither with a `0x`. A width between the braces pads an argument to that many bytes: `{2}` with spaces on the left, `{02}` with zeros on the left, after a sign (`{04}` of -7 is `-007`), `{-10}` with spaces on the right. A longer argument is written whole. A `{` followed by anything but `}` or a width is dropped, as before. `Fmt.print("{2}:{02} U+{04}", [Fmt.int(3), Fmt.int(7), Fmt.hex(0E9H)])` writes `3:07 U+00E9`. `Fmt.into` formats into a slice the caller gives, with no memory of its own, for a library that has no heap: it gives the bytes written, and false if the slice was too short and the text is cut there.

Files and Opt are what most programs need of their command line. `Files.writeAll` writes again after a short write or a signal, as a pipe or a socket needs, and `write` is made over it. `realPath` adds to `b` the absolute path of what `path` names, with no `.`, `..` or symbolic link in it, and leaves the working directory alone: -2 (`ENOENT`) if a part isn't there, -20 (`ENOTDIR`) if a file is used as a directory, and -40 (`ELOOP`) after 40 links. `b` is left as it was

on an error. `removeTree` removes a file, or a directory with all it holds; a symbolic link is removed itself and never followed.

```
module Files;
proc arg*(i: INT): []U8;                    (* argument i; 0 is the program's name;
"" if there is none *)
proc read*(path: []U8; var b: Buf.Buf): BOOL;   (* the whole file added to b *)
proc load*(path: []U8; var b: Buf.Buf): INT;    (* as read: 0, or minus the errno that
stopped it *)
proc write*(path, data: []U8): BOOL;        (* the file made to hold data; false if
it or its close failed *)
proc writeAll*(fd: INT; s: []U8): INT;      (* all of s written to fd; 0, or minus
the errno *)
proc realPath*(path: []U8; var b: Buf.Buf): INT;   (* the absolute path of path,
plain, added to b; 0, or minus an errno *)
proc removeTree*(path: []U8): INT;          (* the file, or the directory and all in
it, removed; 0, or minus the first errno *)

module Opt;
type
    Row* = variant
        usage(scope, line: []U8);           (* a synopsis line, without "usage: " *)
        sub(scope, name, help: []U8);       (* a subcommand *)
        option(scope: []U8; letter: U8; long, value, help: []U8);
    end;
    Parser* = record
        letter*: U8;    (* the option just read *)
        value*: []U8;   (* its value; "" for a flag *)
        long*: BOOL;    (* it was written by its long name *)
        spelt*: []U8;   (* as it was written, no dashes: "b" or "bar"; to read only *)
        ...
    end;
proc init*(var p: Parser; name: []U8; table: []Row);   (* the arguments from 1 on, by
table *)
proc next*(var p: Parser): BOOL;                    (* the next option; false when
they end *)
proc operand*(var p: Parser): ([]U8, BOOL);         (* the next operand; false when
there are no more *)
proc at*(var p: Parser): INT;                       (* the argument operand would
give next *)
proc scope*(var p: Parser; name: []U8);             (* subcommand name's options,
from the word after it *)
proc number*(const p: Parser; lo, hi: I64): I64;    (* the value just read, a number
from lo to hi *)
proc refuse*(const p: Parser; wants: []U8);         (* the value just read refused,
as number refuses one *)
proc fail*(const p: Parser; text: []U8);            (* the program's own mistake *)
proc misuse*(const p: Parser; text: []U8);          (* the program's own mistake,
with the synopsis *)
```

`Opt` reads a command line by the rules of the C library's `getopt`, with GNU's long options, so that a program's options behave as every other Unix program's. The program describes itself once, in a table of `Row`s: its synopsis lines, its subcommands and its options. The help and the messages for mistakes are made from the table, so they can't drift from it; the program's test of `p.letter` is the only other place an option appears. The

parser is a record the program owns, and the table a slice it keeps as long as the parser: a literal written in the call to `init` lives as long as the procedure that makes it, and a constant, `const Rows: []Opt.Row = [...]`, as long as the program.

A row's `scope` is `""` for the program itself, or a subcommand's name: `"start"`, or `"remote.add"` for a subcommand of `remote`. An option has a `letter`, a `long` name, or both. One with a long name only has a key of its own below `'!'` for a letter, so that the program can tell it from the others. With a `value`, the name the help and the messages give it, the option takes a value; with `""` it is a flag. A value written in brackets, `"[when]"`, may be left out, and is `""` then: it is given only in the option's own word, `--color=always` or `-calways`, never as the next, as `getopt`'s `::` has it, and the help shows it as `--color[=when]`.

```
const wait = 1;     (* the key of --wait, which has no letter *)
...
Opt.init(p, "keep", [
    Opt.usage("", "keep [-v] [--wait secs] start ..."),
    Opt.sub("", "start", "start a job"),
    Opt.option("", 'v', "verbose", "", "say more"),
    Opt.option("", wait, "wait", "secs", "give the job secs to start"),
    Opt.usage("start", "keep start [-b label] name command [argument ...]"),
    Opt.option("start", 'b', "bar", "label", "show the job in the bar as label")]);
while Opt.next(p) do
    if p.letter = 'v' then verbose := true;
    elsif p.letter = wait then secs := Opt.number(p, 1, 600);
    end;
end;
w, ok := Opt.operand(p);
if not ok or (w # "start") then Opt.fail(p, "no command; see keep -h"); end;
Opt.scope(p, "start");
while Opt.next(p) do
    if p.letter = 'b' then bar := p.value; end;
end;
job, ok := Opt.operand(p);
```

The rules are `getopt`'s:

- Flags may share a dash: `-lw` is `-l -w`. A letter's value is the rest of its word, `-bdev`, or the whole next word, `-b dev`, even one that starts with a dash or is empty; in a group, the first letter that takes a value takes the rest, `-qn12`.
- A long option is written in full, with its value after a `=` or in the next word: `--bar=dev`, `--bar dev`. An abbreviation is unknown, not guessed, so that adding an option never breaks a script. A value given to a flag, `--follow=x`, is a mistake.
- `--` ends the options, and isn't an operand; `-` alone is one. The options end at the first operand, so in `keep start site python -m http.server` the `-m` is python's.
- In a scope with no option whose letter is a digit, a word of `-` and a digit, such as `-33.87`, is an operand, and so ends the options. A scope that gives a digit a letter, as `head -5` has, reads it as an option.
- Each scope has its own options. An option given twice is read twice.

`number` reads the value as `Str.toInt` does. `operand` reads what options are left first, and the program doesn't see them. So does `at`, which says where the operands are: the arguments from `operand`'s next on are `Files.arg(i)` for `i` from `at(p)` to `ARGC() - 1`, so that an interpreter hands a script its arguments as they are:

```
script, ok := Opt.operand(p);       (* tery file arg ... *)
first := Opt.at(p);                 (* file's own arguments, from here on *)
```

`-h` or `--help` among a scope's options prints its help on standard output and ends the program with status 0. `init` and `scope` answer it, looking through the scope's options before `next` gives one, so that the program has done nothing yet: `keep -v -h -x` prints the help, and the program never sees `-v`, nor the mistake. The help is the scope's synopsis lines, a blank line, and its subcommands, its options and the help itself, each with its line of help:

```
usage: keep start [-b label] name command [argument ...]

  -b, --bar label   show the job in the bar as label
  -h, --help        print this and exit
```

A scope with several synopsis lines has the others indented under the first; one with none, `usage:` and its words, as `usage: keep start`. A scope may give `-h` to an option of its own, as grep's "no file names"; help there is `--help` alone, and its row in the help has no letter.

A mistake on the command line is a sentence on standard error, the program's name first, then the scope's synopsis lines, as the help begins, and status 2:

```
keep: unknown option -x
usage: keep start [-b label] name command [argument ...]
```

The sentences are `unknown option -x` (or `--name`, up to its `=`), `-b needs a label`, `--follow takes no value` and `--wait wants a number from 1 to 600, not '0'`; for a number with no top, given the largest `I64` for `hi`, `--jobs wants a number from 1 up, not '0'`. `refuse` refuses a value the program reads itself, as `number` refuses one, naming the option as the user wrote it, `-b` or `--bar`: `Opt.refuse(p, "a label with no spaces")` writes `keep: -b wants a label with no spaces, not 'a b'`, and the synopsis lines. `p.spelt` is that spelling, for the program's own sentences. `fail` writes the program's own mistakes the same way, but alone, so that the sentence can say where the help is: `keep: no command; see keep -h`. `misuse` writes one with the scope's synopsis lines after it, for a mistake in the operands: `Opt.misuse(p, "no tests named")`.

A mistake in the table is the program's, not its user's. `init` checks the table before it reads a word, and ends the program with status 1: `keep: option table, row 3: the option is in the scope already`. The others are an option with neither a letter nor a long name, a long name with `=` in it, a letter `-`, a value with one bracket but not the other, and `--help` in the table, since it is `Opt`'s.

Width says how many columns of a terminal text takes, for a program with a screen that wraps its lines and cuts them at the window's edge.

```
module Width;
proc width*(c: I32): INT;      (* the columns the character c takes: 0, 1 or 2 *)
proc columns*(s: []U8): INT;   (* the columns the UTF-8 text s takes *)
```

A combining mark, a format character, a control and Hangul's medial and final jamo take 0 columns; an ideograph, a kana, most emoji and the rest that Unicode calls East Asian Wide or Fullwidth take 2; every other character takes 1, the ambiguous ones among them, as most terminals draw them. These are the C library's `wcwidth`'s answers, but for a control, which is -1 there. `Width` knows characters, not clusters: an emoji joined to another by a zero width joiner, or a flag, counts as its parts. Its tables, about 4 KB, come from Unicode's data, made again by `tools/Widths.raz` for a new version of Unicode; a program that imports `Width` and calls neither procedure carries none of them.

Rand, Check and Prop are for testing a program, and are standard library, not core: a target may leave them out. `raztest(1)` builds and runs the tests that use them.

`Rand` is a seeded generator of numbers, splitmix64: one `U64` of state. The same seed gives the same numbers on every machine, so a run can be replayed. It is not for secrets.

```
module Rand;
type Rand* = record ... end;
proc init*(var r: Rand; seed: U64);     (* starts r at seed; any seed, 0 too *)
proc next*(var r: Rand): U64;           (* the next number, all 64 bits of it *)
proc below*(var r: Rand; n: U64): U64;  (* 0 to n - 1, each as likely as another; n =
0 stops the program *)
proc range*(var r: Rand; lo, hi: I64): I64;   (* lo to hi, both included; lo must not
exceed hi *)
proc seed*(): U64;                      (* a fresh seed from the kernel, to print and
give back to init *)
```

```
module Dice;
import Rand, Check;

(* Five rolls of a die from seed, as the digits of a number. *)
proc rolls(seed: U64): INT;
var r: Rand.Rand; i, v: INT;
begin
    Rand.init(r, seed);
    for i := 1 to 5 do v := v * 10 + INT(Rand.range(r, 1, 6)); end;
    return v;
end;

proc main*;
begin
    Check.int("the same seed, the same rolls", rolls(7), rolls(7));
    Check.ok(rolls(7) >= 11111, "every die shows 1 or more");
    Check.ok(rolls(7) <= 66666, "every die shows 6 or less");
end;

end Dice.
```

`Check` says what a test expects. A check that fails writes `FAIL` and its words on standard error, with what it got and what it wanted where they differ; the program goes on. At its end, if any check failed, it says how many and exits with status 1, and a run with none failed says nothing and keeps its own status. No call is needed at

the end: the first check asks `Rt.atExit` to make it. A program that stops after a check failed, by `Rt.stop` or a failed check of the language, ends with status 1 too. `mute` is for `Prop`, which searches with the checks silenced.

```
module Check;
proc ok*(cond: BOOL; what: []U8);                (* fails, saying what, when cond is
false *)
proc str*(what, got, want: []U8);                (* fails when got is not want; shows
both, escaped, and the byte where they differ *)
proc int*(what: []U8; got, want: I64);           (* fails when got is not want; shows
both *)
proc mute*(on: BOOL);                            (* while on, checks say nothing and
are not counted *)
proc mutedFailures*(): INT;                      (* how many failed since mute was
turned on *)
```

```
module Digits;
import Check;

(* How many decimal digits n, not negative, takes. *)
proc count(n: INT): INT;
var d: INT;
begin
    d := 1;
    while n >= 10 do n := n div 10; INC(d); end;
    return d;
end;

proc main*;
begin
    Check.int("0 is one digit", count(0), 1);
    Check.int("9 is one digit", count(9), 1);
    Check.int("10 is two digits", count(10), 2);
    Check.ok(count(1000000) = 7, "a million is seven digits");
end;

end Digits.
```

`Prop` tests a property: a procedure that draws values and checks them with `Check`. A case is the bytes the property drew, and `Source` gives them as a number, a choice, a flag or a string of bytes; fewer and smaller

bytes give simpler values. When a case fails, `Prop` shrinks it by changing its bytes and running the property again, so no type needs a shrinker of its own, and it saves the case. A property that draws only bytes is a fuzzer.

```
module Prop;
type Source* = record ... end;
type Property* = proc(var s: Source);              (* draws from s, checks with Check
*)
proc bits*(var s: Source): U64;                    (* 64 bits, from 8 bytes, the first
the highest *)
proc int*(var s: Source; lo, hi: I64): I64;        (* lo to hi, both included; zeros
give lo *)
proc pick*(var s: Source; n: INT): INT;            (* 0 to n - 1; n at least 1 *)
proc bool*(var s: Source): BOOL;
proc bytes*(var s: Source; max: INT; var b: Buf.Buf);   (* b, emptied, then given at
most max bytes *)
proc check*(name: []U8; p: Property);              (* runs p on the saved cases, then
on fresh ones *)
```

`check` runs `p` on each case saved in the directory `$PROP_DIR/name` (`test/seeds/name` if `PROP_DIR` is unset), then on `$PROP_CASES` fresh cases (100 if unset), from the seed `$PROP_SEED` or a new one, or for as many seconds as `$PROP_TIME` says, if it is set. Each is a whole number. A property that holds says nothing and writes no file. The first case that fails is shrunk and saved in a file named by its hash; standard error says the seed and the file; the shrunk case runs once more with `Check` heard, and the program ends with status 1. A saved case is run first the next time, so a failure stays found until it is mended; keep the directory with the tests. A case that

stops the program, by `Rt.stop`, a failed check of the language or a signal, can’t be shrunk; it is saved as it was, when the program ends. `raztest --seed` and `--fuzz` set `PROP_SEED` and `PROP_TIME`.

```
module Reverse;
import Prop, Check, Buf, Heap;

var
    b, was: Buf.Buf;    (* the bytes drawn, and a copy of them *)

(* Turns s round, in place. *)
proc reverse(s: []U8);
var i: INT; c: U8;
begin
    for i := 0 to LEN(s) div 2 - 1 do
        c := s[i]; s[i] := s[LEN(s) - 1 - i]; s[LEN(s) - 1 - i] := c;
    end;
end;

(* Reversing any string twice gives it back. *)
proc twice(var s: Prop.Source);
begin
    Prop.bytes(s, 64, b);
    Buf.reset(was);
    Buf.str(was, Buf.bytes(b));
    reverse(Buf.bytes(b));
    reverse(Buf.bytes(b));
    Check.str("reversed twice", Buf.bytes(b), Buf.bytes(was));
end;

proc main*;
begin
    Buf.init(b, Heap.heap);
    Buf.init(was, Heap.heap);
    Prop.check("reverse", twice);
end;

end Reverse.
```