

BYOK Script Writer's Guide

This guide describes scripting as of BYOK firmware 1.2.

How to write a script for the BYOK, for people who write prose rather than code.

What a script is

A script is a text file on the memory card, in the `Scripts` folder, ending in `.lua`. The BYOK finds it the next time it starts.

Scripts do small jobs on the document you are writing. Typing a scene break. Replacing `INT` with something longer. Counting the paragraphs. They cannot change how the BYOK itself behaves — a script cannot alter your settings, your keyboard, or another script.

The two kinds

A one-shot runs when you ask for it. You give it a letter, and `Ctrl+T` followed by that letter runs it. It does its job and stops.

```
function main()
  BYOK.doc.insert("\n\n* * *\n\n")
end
```

A watcher runs while you type. It has no letter, because you do not summon it — you switch it on, and it stays on until you switch it off.

```
function init()
  BYOK.sys.registerEvent("WORD")
end

function listen(event, text, from, to)
  ...
end
```

A watcher has two functions because it has two moments. `init` runs once when you switch it on, and says what it wants to be told about. `listen` runs every time that thing happens.

What a watcher can be told about

Event	Happens when	text is	from-to cover	At most
WORD	you finish a word	the word	the word, without the space	64 bytes
LINE	you press Enter	the line you just finished	that line, without the newline	1024 bytes
TAB	you press Tab	the word before the tab	the word and the spaces Tab left	64 bytes

`listen` is told which event it is for, because a watcher may ask for several. If yours asks for only one you can ignore the first argument, but it costs nothing to check.

Past the cap, nothing happens at all. No event, no error, and your `listen` is simply not called. What you get is never a shortened version: the `from-to` would still describe the whole of it, so a script replacing a range it had only seen part of would eat the rest. Silence is the safe answer, and it is the only one.

`LINE` is the big one because it carries a whole **paragraph**. Enter is a paragraph break on the BYOK, not a line break — the screen wraps your text for you — so the line you are handed runs back to the last time you pressed it. Ordinary prose runs several hundred bytes to the paragraph. The other two carry a word, and something past 64 bytes is not a word.

If a `LINE` watcher never seems to fire, this is the first thing to check. The log says when a paragraph was too big to send, and how big it was — take the card out and read `Scripts/scripts.log`. See **When something goes wrong**.

There are no tab characters in your document. Pressing Tab types spaces — five of them — and a tab in a file you copy across is turned into spaces when it loads. Nothing you write will ever find a `\t`, because there is never one there.

So the `TAB` range covers the abbreviation *and* the spaces that Tab just left behind. That is what makes expanding a snippet a single `replaceRange`: both vanish and your text takes their place. Do not count the spaces yourself or assume how many there are — use the `from` and `to` you were given, and it stays right if the number ever changes.

`WORD` goes the other way, and the difference matters the first time you write one. The space you typed is what tells the BYOK the word is finished, and that space is *not* part of the range — it is still sitting there after your replacement lands. So a snippet ending in a space of its own gives you two. Leave it off and let the space you typed do the work:

```
CHR = "CHARACTER:"      -- not "CHARACTER: "
```

A Tab expander is the opposite, and wants the trailing space that a WORD expander must not have — there is nothing left behind for it to inherit.

Pressing Tab does not also finish a word. It sends `TAB` and nothing else, so a snippet watcher on `WORD` and a completion watcher on `TAB` never both fire on the same keypress.

None of these can stop the key. By the time your script runs, the word is typed, the line is broken, the spaces are in. What you get is the chance to change what happened — which is why every event tells you exactly which stretch of text to replace.

Wherever your cursor was when the event fired, that is where it stays. If you were three words further on by the time a `LINE` watcher reformatted the line behind you, you are still three words further on.

Which to write

If you would reach for it deliberately — "expand this", "count that" — write a one-shot. If it should happen without you thinking about it, write a watcher.

Watchers cost more. One is fine. Four is the limit, and the BYOK will refuse the fifth rather than slow your typing down.

Two watchers may both listen for the same event, but **only one of them may change text on it**. The second is refused when you try to switch it on, and told which one holds it. Watchers on *different* events never conflict — a word expander, a line capitaliser and a tab expander all run happily together.

How much a script can hold

64 KB, every time, whatever else the BYOK is doing. Each run of a one-shot, and each watcher that is switched on, gets its own 64 KB. Your script's own words count against it: a long table of phrases uses some of it before anything happens. Go over and the script stops with *The script used too much memory*.

Comments and spacing cost nothing. The BYOK throws them away when it reads your script, so write it as clearly as you like. There is nothing to gain from squeezing it.

What does cost memory is what your script *holds*: its own words and tables, and anything it builds while it runs. Anything that grows — a history, a log, a list of words seen — wants a limit, with the oldest dropped once it is reached. Words you need only now and then can live in the script's data file, read when wanted and let go afterwards.

`BYOK.sys.memory()` says where you stand: two numbers, the bytes your script is using and the most it may use. It needs no capability. The first includes rubbish Lua has not yet cleared away, so call `collectgarbage()` just before if you want only what you are really holding. Lua lets that rubbish build up to about two and a half times what you hold before it clears any away, and it always clears it before the limit would stop you, so a high reading on its own is nothing to worry about. Trust it over Lua's own `collectgarbage("count")`, which leaves out long strings such as those `string.rep` builds, and can read well under the limit you are actually up against.

There is also a limit on how long a script may work without pausing. Waiting for a key or sleeping starts the count again, so a game or a conversation can run as long as you like. A loop that never waits cannot, and stops with *The script ran for too long*.

The header block

Every script starts with a few lines that tell the BYOK what it is. They begin with `---` `@`, and they must come before anything else.

```
--- @name          Scene break
--- @description   Types a centred scene break at the cursor
--- @type          oneshot
--- @key           b
--- @requires      document
--- @caps          doc.edit
```

Line	What it does
@name	What appears in the script list. Required — without it the script is ignored.
@description	The sentence shown when you open the script's details. Keep it under about fifty characters or the end is cut.
@type	oneshot or watcher. Required.
@key	The letter for <code>Ctrl+T</code> . One-shots only; a watcher is switched on instead.
@requires	document if it needs something open, none if not.
@caps	What it is allowed to do. See below.
@shares	Optional, and rarely wanted. Lets another script use this one's data file.
@uses	Optional. Names the script whose data file this one wants instead of its own.

The last two only matter when two scripts have to share one setting between them, and they only work as a pair — see **Sharing one file between two scripts** under **BYOK.data**.

@shares is the only line with nothing after it. That is not a gap where something should go: the line being there IS the answer, because there is only one thing it can say. Writing `@shares y` or `@shares data` changes nothing, and you may do it if a bare line looks unfinished to you.

What you cannot do is name who may use it. `@shares sprintgoal.lua` does **not** restrict anything — every installed script that names this one can still use its data file, and the log will say so. If that is not what you want, the answer is not to share at all.

Spelling is forgiving: `one-shot` and `oneshot` are the same, and capitals do not matter.

@caps is the important one

A script can only call what its `@caps` line asks for. Ask for nothing and there is almost nothing to call: the script will stop at its first line with an error.

Three things are exceptions and need no capability at all, so they work in every script including one with no `@caps` line: **writing to the log** — `print` and `BYOK.sys.log` — `BYOK.sys.uptime()`, which says how long the BYOK has been switched on, and `BYOK.sys.memory()`, which says how much

memory the script is using. None is something done to you. One writes a line to a file you open only when something has already gone wrong; the others are facts about the device rather than about you or your work. Having to edit a header before you could add a `print` or time a loop would be friction at exactly the wrong moment.

There are nine to choose from:

Capability	Lets the script
<code>doc.read</code>	Read the document, what you have selected, and what you last copied
<code>doc.edit</code>	Change the document
<code>doc.navigate</code>	Move you — find a word and take you to it
<code>data.read</code>	Read its own data file
<code>data.write</code>	Write its own data file
<code>ui.alert</code>	Put a message on screen
<code>ui.list</code>	Ask you to choose from a list. A watcher may only ask from a <code>TAB</code>
<code>ui.ask</code>	Ask you to type a line. A watcher may only ask from a <code>TAB</code>
<code>screen.draw</code>	Take the screen and draw on it. One-shots only

List several with commas:

```
--- @caps      doc.read, ui.alert
```

`doc.navigate` is separate from `doc.read` on purpose. A script that scrolls your document away from where you were has done something to you, even though it changed nothing — so it has to say so. `data` is separate from `doc` for a plainer reason: a script's own file and the book you are writing are not the same thing, and a script that only keeps notes for itself should not be asking to touch your manuscript.

The clipboard sits under `doc.read` rather than having its own line. It holds your own words, put there by your own copy, and a script already allowed to read the whole document learns nothing new from it. Worth knowing that it survives switching files, so what it holds may have come from somewhere other than the document you are looking at.

`screen.draw` is one capability rather than a dozen. Nobody should have to list every kind of line and box before they can draw one.

Two things worth knowing before they cost you an afternoon:

- A misspelt capability is not reported when the script loads. It simply grants nothing, and the script fails at the moment it tries to use it.
- Asking for more than you use is not an error, but it is a promise you did not need to make. `@caps doc.edit` on a script that only counts words says it may rewrite your book.

Your letter may not be the one you asked for

If two scripts want `b`, the first one keeps it. The second is left without a letter — it still appears in the list, and you can give it a different one there. You are not asked to resolve it while you are trying to write.

Check that you got the letter you asked for, the first time you run a new script. A script with no letter is easy to mistake for one that is not working, when in fact `Ctrl+T` and your letter are quietly running somebody else's script.

`?` and the space bar are never available: they already mean something when `Ctrl+T` is waiting.

What you can call

Everything the BYOK offers lives under `BYOK`, grouped by what it touches — the document you are writing, the script's own file, asking and telling, the screen, and everything else. Each group has its own section below, and the two after them say what Lua itself gives you and what a script cannot do.

BYOK.doc — the document you are writing

Three capabilities, because there are three different things a script can do to a document: look at it, change it, and move you around in it.

Looking — `doc.read`

`BYOK.doc.chunks()`

Walks the whole document in pieces, front to back:

```
for chunk in BYOK.doc.chunks() do
    ...
end
```

Each chunk is a few thousand characters and always ends at the end of a line, so a word or a phrase is never split between two of them — and no piece of the document ever arrives twice. The chunks lay end to end and make exactly your file. You get the document as it was **when the script started**. One pass, forwards; you cannot ask for a particular page and cannot go back.

What you were typing a second ago is included. The BYOK writes the page out before it runs any script that reads, so a script never sees an older version of your work than the one in front of you. You do not have to save first, and there is no point doing so.

That is what makes it safe to **work something out with** `chunks()` **and then act on it with** `find`, even though one reads the file and the other works on the page in front of you. They are the same document while your script runs. Count occurrences in one and correct them in the other, and the number you counted still means something.

`BYOK.doc.selection()`

The text you have selected, or `nil` if nothing is.

```
local text = BYOK.doc.selection()
if not text then
  BYOK.ui.alert("Nothing selected", "Select some text first")
  return
end
```

Select the usual way — hold shift and use the arrows — then run the script. Your selection survives opening the script list, so it does not matter whether you run the script by its letter or pick it from the list.

Very large selections come back as `nil` rather than as a shortened version, so you are never handed half a paragraph believing it is all of it.

`BYOK.doc.clipboard()`

What you last copied or cut, or `nil` if you have not copied anything since the BYOK started.

```
local clip = BYOK.doc.clipboard()
if clip then
  BYOK.doc.insert("> " .. clip)
end
```

All of it or nothing, like `selection()` — there is no way to ask for the first so many characters, because that would cut in the middle of a word and possibly in the middle of a letter.

It is not always from the file you have open. The clipboard survives switching files, so a script may be handed something cut from somewhere else entirely. If that matters to what you are doing, you cannot tell from here.

Reading it does not disturb it. Your clipboard still holds what it held, and paste still works.

Changing — `doc.edit`

`BYOK.doc.insert(text)`

Types `text` where your cursor is, exactly as if you had typed it. **If something is selected, this replaces it** — which is why there is no separate "replace the selection" call:

```
BYOK.doc.insert(BYOK.doc.selection():upper())
```

It hands back whether the edit was taken, and on anything that matters you should look.

```
if not BYOK.doc.insert(text) then
  -- it did not happen
end
```

`false` almost always means one thing: **your script has a window open**. While a window has the screen, the page will not accept changes — see **Drawing on the screen** for what to do instead. It is also `false` for an empty insert, or text past the size limit.

Worth saying plainly, because it is an easy mistake and a quiet one: a script that counts what it ASKED for rather than what was taken will tell you it fixed six things and have changed nothing at all.

BYOK.doc.replaceRange(from, to, text)

Replaces the stretch between two positions. This is what a watcher uses, because the event tells it exactly which stretch it means — by the time the script runs, your cursor has usually moved on.

Passing empty text deletes the stretch and puts nothing back, which is how a watcher removes something rather than changing it.

Hands back the same thing as `insert`, and for the same reasons.

Both of these will insert characters the screen cannot draw. The BYOK's fonts cover the ordinary keyboard and little else. An em dash, curly quotes, an ellipsis — the kind of punctuation a typographic script reaches for first — go into the file correctly and appear on screen as a filled diamond. Your text is not damaged and the file is right; you simply cannot see what you wrote. Until the fonts grow, write two hyphens rather than an em dash, and straight quotes rather than curly ones.

Moving you — `doc.navigate`

BYOK.doc.find(term)

Searches from your cursor, wraps round the end once, and takes you to the match with it selected. Returns `true` if it found one.

```
if not BYOK.doc.find("TK") then
  BYOK.ui.alert("No TK left", "Nothing still to come")
end
```

Run it again to walk to the next one. It is a separate capability from reading because it moves you: a script that scrolls your document away from where you were has done something to you, even though it changed nothing.

It tells you **whether** it found the term, never **where**.

When it says `true`, the cursor really is on the match — not on its way there. That is what lets the next line edit it.

It matches letters, not words

Worth reading twice, because the obvious use of it damages prose. `find` looks for your letters **anywhere**, including inside longer words:

```
BYOK.doc.find("ot")    -- stops inside lot, not, got, another
```

A script that corrected that would rewrite all four.

Capitals count too. `find("teh")` walks straight past `Teh` at the start of a sentence. List both spellings if you want both — and if you are also counting occurrences yourself, count them the same way, or the number you have and the matches you get will not be the same set.

There is no whole-word option, and you cannot build one. Since `find` tells you whether it matched and never where, your script cannot look at the letters either side and decide for itself.

What works is putting the spaces into the term, and back into the replacement:

```
BYOK.doc.find(" ot ")
```

Be honest with yourself about what that misses. An occurrence at the start of a line, or against a comma or a full stop, has no space on one side and will not be found. Two of the same word separated by a single space only take one pass each time, because the space between them belongs to whichever is found first — run it again for the rest.

Short words are where this is most dangerous and hardest to get right, so a list of corrections is safest when every entry is long enough to be unambiguous on its own.

Changing what you found

`find` leaves the match selected, and `insert` replaces a selection. Those two together are how a script fixes a word it went looking for — needs `doc.navigate` and `doc.edit`:

```
if BYOK.doc.find("teh") then
  BYOK.doc.insert("the")
end
```

Fixing all of them

`find` gives back `false` only when the term is nowhere in the document. Because it wraps, that means it has looked everywhere — so a loop finishes rather than handing you the first match over and over:

```
while BYOK.doc.find("teh") do
  BYOK.doc.insert("the")
end
```

Each turn replaces one, so there is one fewer to find, and it always reaches the end.

Unless your replacement contains the thing you are looking for. Searching for `hte` and inserting `hte` never runs out of matches, and the BYOK eventually stops the script for running too long.

A loop like that cannot let you say no. It only ends because every turn removes a match; leave one alone and the wrap brings you back to it for ever. If you want to ask before each change, count them first with `chunks()` and go round exactly that many times.

What this cannot do

This is a sweep, and sweeping is all of it. `chunks()` and `find` are two separate searches that do not join up: if you read the document and noticed a misspelling in the third paragraph, `find` still takes you to the first one in the file. Changing one particular occurrence and leaving the rest alone is not something a script can say today — see the end of this section.

BYOK.data — the script's own file

A script can keep a file of its own beside itself on the card. The name follows the script's: a script called `scene.lua` reads and writes `scene.dat`, and `count.lua` would use `count.dat`.

8 KB is the ceiling, and there is no way round it. Both reading and writing refuse anything larger, there is no way to read a big file a piece at a time, and a script cannot open a second file — its own `.dat` is the only one it can reach. Eight kilobytes is roughly a thousand short lines: plenty for snippets, settings or a list of names, and nowhere near a dictionary. **If what you are planning does not fit, it does not fit.** Better to know that while you are planning than to meet it as a failed save later on.

You never name the file. There is no filename anywhere in these calls, and that is deliberate — a script cannot reach a file it has no business with, because it has no way to say one. It also means you cannot get the name wrong. Two scripts *can* agree to share one file, and even then neither of them names it: see **Sharing one file between two scripts** at the end.

BYOK.data.read() — needs `data.read`

The whole file as one piece of text. When it cannot give you that, it hands back `nil` **and a reason**:

```
local text, why = BYOK.data.read()
```

why	What happened
"missing"	There is no file yet.
"toolarge"	Over 8 KB. Your file is there, untouched.
"unreadable"	It exists but could not be opened.

Only "missing" means it is safe to write. The other two mean your file is still on the card, and saving over it would destroy something you cannot get back:

```

local text, why = BYOK.data.read()
if not text and why == "missing" then
    return -- something is there; leave it alone
end

```

Any script that saves what it has learned needs that check. Without it a file that has merely grown too big reads as a first run, the script starts again from nothing, and the next save replaces everything that was in it.

Ignore the second value if your script only ever reads.

BYOK.data.write(text) — needs `data.write`

Replaces the file entirely. Returns `true` if it was written. There is no way to add to the end — read it, change it, write it back.

At most 8 KB, on the way in and on the way out. **Check what write gives back:** a refusal is otherwise silent, and a script that keeps adding to a list will one day cross the line and stop saving without saying so.

Refusing rather than shortening is the same rule as a long selection: half a file treated as though it were all of it is worse than none.

Nothing decides what goes in it but you. It is plain bytes as far as the BYOK is concerned — a list, some settings, notes to yourself. Whatever you write, your script has to be the thing that understands it.

That makes it the simplest way to let someone change a script without editing the script. Ship a `.dat` alongside, read it if it is there, fall back to something built in if it is not:

```

local function load_scenes()
    local text, why = BYOK.data.read()
    if not text then
        if why == "missing" then return nil end -- do not overwrite what is there
        return built_in
    end

    local scenes = {}
    for line in text:gmatch("[^\r\n]+") do
        if not line:match("^%s*$") and not line:match("%s*#") then
            local name, body = line:match("^(.+)|(.*)$")
            if name then scenes[#scenes + 1] = { name = name, text = body } end
        end
    end

    if #scenes == 0 then return built_in end
    return scenes
end

```

Now the file is editable on a computer in any text editor, and the script still works for someone who never opens it.

A file that parses to nothing falls back too. An empty list is a mistake rather than an instruction.

Falling back is not always the kind thing. Where the file IS the point of the script — a word list, a set of rules — quietly using something else when the writer's own edit failed to parse is worse than stopping and saying so. They will believe their change took effect. A spell checker should carry no built-in word list for exactly this reason: better to stop and say the list could not be read than to correct against a different one.

Renaming a script leaves its data behind. `scene.lua` renamed to `slug.lua` will look for `slug.dat` and find nothing. Rename the `.dat` with it. A script that borrows another's file, below, does not have this problem — it is named after the owner rather than after itself.

Sharing one file between two scripts

Sometimes the script that SETS something and the script that USES it cannot be the same script. A word-sprint watcher counts while you type, but a watcher cannot ask you anything in the middle of a sentence — so the target gets set by a one-shot you run on purpose. Two scripts, one setting between them.

Two header lines arrange that, and it takes **both**:

```
--- @shares          -- in sprint.lua: my data file may be used
--- @uses            sprint    -- in sprintgoal.lua: I want sprint's
```

Now both of them read and write `sprint.dat`.

Either line on its own does nothing. A script saying `@uses sprint` when `sprint.lua` has not said `@shares` does not load at all — it will not appear in the script list, and the log says why. Consent runs both ways, so no script can reach into another's file by guessing its name.

You still never name a file. You name a **script**, and only one that has agreed. There is no path to write, no folder to point at, and nothing you can misspell into somewhere else: a name matching no installed script is refused rather than quietly creating something.

Only one of them should write. `data.write` replaces the whole file, so two scripts both saving will take turns destroying each other's work. What works is one writer and any number of readers — which is usually what you wanted anyway. The one-shot writes the setting; the watcher reads it.

A script cannot do both. `@shares` and `@uses` together is refused. A script has one data file, and those two lines ask for different ones.

What each may do is still @caps. `@shares` says the file may be addressed by name; it grants nothing on its own. The setter needs `data.write` and the watcher needs `data.read`, exactly as they would for a file of their own.

If the owner is not on the card, nothing that borrows from it loads. That is the intent rather than an accident. A borrower with nothing to borrow from would otherwise read an empty file and

behave as though you had never set anything — which looks like your setting being ignored, and sends you to the wrong script to find out why.

BYOK.ui — asking and telling

`BYOK.ui.alert(title, body)` — needs `ui.alert`

A box with a title, one line of message, and OK. Both lines are about 26 characters; anything longer is cut.

`BYOK.ui.list(choices)` — needs `ui.list`

Puts a list on screen and waits. Returns the number of the choice, counting from one, or `nil` if the writer backed out.

```
local dividers = {"* * *", "- - -", "~ ~ ~"}

local choice = BYOK.ui.list(dividers)
if not choice then return end

BYOK.doc.insert(dividers[choice])
```

Your script stops at that line and carries on from it with the answer in hand. Nothing else is needed — no waiting, no second function.

It gives back a **number, not the text**, so it lines up with the table you already have. That matters when the list you show is not the text you insert:

```
local names = {}
for i, scene in ipairs(scenes) do names[i] = scene.name end

local choice = BYOK.ui.list(names)
if choice then BYOK.doc.insert(scenes[choice].text) end
```

Backing out is a real answer, and the commonest one. Either the return arrow at the foot of the list or the escape key gives you `nil`. Handle it first and say nothing — the writer knows what they did.

At most 32 choices. More than that is refused outright rather than showing the first 32, because a menu quietly missing its last entries is worse than one that never appeared.

A watcher may only open a list from a TAB. Tab is you asking for something to happen, so a menu is a reasonable answer to it. Finishing a word is not asking, and a list after every word would leave you no way to type at all.

So from `WORD` or `LINE` you get back `nil` and a reason, the same shape as `data.read`:

Called from	choice	why
a one-shot, or a TAB watcher	the number chosen	nil
either, when the writer backs out	nil	nil
a WORD or LINE watcher	nil	"notab"

Nothing appears on screen and your watcher keeps running. That is deliberate: the mistake belongs to whoever wrote the script, and the person who would see a dialog is the one typing — who did nothing wrong, cannot fix it, and would lose a watcher they had chosen to switch on. The reason goes to the serial log instead, where the person who can fix it is looking.

A Tab watcher that offers a list still owes the range a replacement on every path, including the one where you back out. See the completion example among the complete scripts at the end.

BYOK.ui.ask(prompt) — needs **ui.ask**

Puts your prompt on screen above an empty line and waits. Returns what was typed, or **nil** if the writer backed out.

```
local heading = BYOK.ui.ask("Heading")
if not heading then return end

BYOK.doc.insert("\n" .. heading:upper() .. "\n\n")
```

Like a list, your script stops at that line and carries on from it with the answer in hand.

There is no way to fill the line in advance, and there will not be. It asks; it does not edit. So whatever comes back is something the writer chose to type, and a script cannot put words in their mouth by offering them pre-written. The cost is real and worth stating: a script that renames something makes them type the whole new name rather than change a letter of the old one.

An empty answer cannot happen. The box will not accept a blank line — pressing enter in an empty box does nothing, and neither does OK. So **if answer then** is the whole test, and you never need a second check for empty text.

Backing out is escape, or Cancel. Both give you **nil**, and anything already typed is discarded.

At most 40 characters, and **the line does not scroll** — what fits on screen is what you can type. That is a decision, not a gap: it is one line, for a short answer to a short question. If you need a paragraph from the writer, let them type it on the page and take it with a **LINE** watcher.

A watcher may only ask from a TAB — the same rule as a list, and it matters more here. A list you can escape; a box that takes typing takes **your keyboard**. A watcher allowed to open one after every word could swallow the next thing you wrote. From **WORD** or **LINE** you get **nil** and **"notab"**, nothing appears on screen, and the watcher keeps running.

BYOK.screen — a window to draw in

Needs `screen.draw`, one-shots only. A script can take the whole screen for a chart, a diagram or a game, wait for keys, and give it back.

It has a section of its own — see **Drawing on the screen**.

BYOK.sys — everything else

Writing to the log — needs no capability

```
print( ... )
```

Lua's own `print`, and the first thing to reach for when something is not behaving. Everything you pass is written to `Scripts/scripts.log` on the card, separated by tabs, with numbers and tables turned into text for you:

```
print("got this far")
print("word was", word, "at", from)
```

```
BYOK.sys.log(message)
```

The same destination, one piece of text, and nothing converted for you. `print` is easier and there is no reason to prefer this one.

`warn( ... )` also goes there, marked as a warning.

None of these need anything in your @caps line, including in a script with no @caps line at all. See **When something goes wrong** for what the file looks like, how to read it, and what else the BYOK puts in it.

How long it has been on — needs no capability

```
BYOK.sys.uptime()
```

The number of thousandths of a second since the BYOK was switched on.

```
local started = BYOK.sys.uptime()
-- something slow
print("that took", BYOK.sys.uptime() - started, "ms")
```

Subtract two readings. The number on its own means nothing. The difference between two of them is a real length of time, and that is what lets a script pace itself, time something it is unsure about, or count a writing session.

It is not the time of day, and there is no way to ask for that. The BYOK has no clock it can be told the truth about, so a script that wants to write today's date cannot have it. Better to know that now than to find a snippet has been stamping the wrong year into your work.

It starts again from zero every time you switch on, and the BYOK switches itself off after fifteen minutes of nothing happening — worth remembering if you are counting anything longer than that.

How much memory it is using — needs no capability

`BYOK.sys.memory()`

How many bytes your script is using, and the most it may use. See **How much a script can hold** for what counts and how to use less.

`BYOK.sys.registerEvent(name)` — watchers only, inside `init`

Says what your watcher wants to be told about: `"WORD"`, `"LINE"` or `"TAB"`. What each one gives you, and how much text it can carry, is in the first section. Call it once per event you want; a name that is not one of those three is refused, so a typo cannot leave you with a watcher that quietly never fires.

The rest of Lua

You have Lua's strings, tables, maths and coroutines, and `utf8` for working with accented letters. There is no `io`, `os` or `require`, so a script cannot read or write files or pull in a library: everything it needs is in its own file. `load`, `loadfile` and `dofile` are there as in any Lua. If you want to count words, `( "%S+" )` is a word counter and there is no reason for the BYOK to provide one.

If you are borrowing older Lua

The BYOK runs **Lua 5.5**. Much of the Lua you will find in books and online was written for 5.1, and three differences catch nearly everyone:

- **A for loop's own variable cannot be changed inside the loop.** `for w in text:gmatch("%S+") do w = w:lower() end` is refused: the script will not start, and the error names the line. Copy it first — `local word = w:lower()` — and change the copy.
- **Some names have gone.** `unpack` is now `table.unpack`. `table.getn(t)` is `#t`. `math.pow(a, b)` is `a ^ b`. `loadstring` and `setfenv` no longer exist.
- **Strings count bytes, not letters.** An accented letter is two bytes, so `#"café"` is 5. `utf8.len` counts letters, and `s:sub(1, utf8.offset(s, -1) - 1)` takes the last letter off whole rather than leaving half of it.

What a script cannot do

Positions. Nothing gives you a number saying where you are, and nothing takes one to send you somewhere. `find` moves you without either. The document as `chunks()` reads it and the document as the editor holds it are counted separately, so a number would mean two things.

Whole-word matching. `find` matches letters wherever they appear, and cannot be told to want a word. See `doc.find` above for what that costs and what to do instead.

Putting something ON the clipboard. A script can read what you copied; it cannot copy for you.

Stopping a key. A watcher is told what happened, never asked what should happen. See the first section.

If a script seems impossible, one of those is usually why.

Drawing on the screen

A script can take the whole screen and draw on it — a chart of something about your writing, a diagram, a game while you think. Needs `screen.draw`, and one-shots only: a watcher runs while you type, and a window appearing over your page because you finished a word is not something to offer.

Nothing here touches your document. That is why a drawing script can say `@requires none` and run on a BYOK with no file open at all.

Opening one

```
function main()
  if not BYOK.screen.open() then return end
  local s = BYOK.screen

  s.clear()
  s.text(4, 12, "Hello")
end
```

`open()` gives back `false` if one is already open — you get one, and asking for a second is refused rather than quietly replacing the first. It also gives back `false` if the window could not be put up at all, which you handle the same way: don't draw.

When it gives back `true`, the window is **ready to measure and draw on**. `width()`, `height()` and `lineHeight()` are real on the very next line.

The window stays after `main` finishes. `s.close()` puts it away, and **escape always closes it** whatever your script thinks — that is the writer's way out and you cannot take it from them.

While your window is up, the page will not take changes

This is the rule that catches people, including the person who wrote this guide.

Your window sits exactly where the document does, and it has the screen. For as long as that is true, **`doc.insert` and `doc.replaceRange` are refused** — they hand back `false` and nothing happens. Not delayed: refused.

So a script that draws and also edits has to do them in that order, with the window gone before the first change:

```
-- 1. Gather, while you have the screen
local answers = {}
while s.isOpen() do
    ...
end

-- 2. The window has gone. NOW edit.
for _, a in ipairs(answers) do
    BYOK.doc.insert(a)
end
```

Your script keeps running after the window goes, which is what makes that possible. The moment `key()` or `sleep()` hands back because the window closed — or `close()` returns — the page has the screen again and your edits are taken.

Check what `insert` gives back and you will find out at once if you have this the wrong way round. Ignore it and your script will report work it did not do.

If your script stops with an error, the window goes with it. A script that has failed cannot tidy up after itself, and a half-drawn picture with an error message over it is no use to anybody.

Where things go

`width()` and `height()` are the size of your window. Use them rather than typing numbers: they are why a script still lays out correctly if the screen ever changes.

Corners are `0, 0` at the top left. Everything is measured from your window's own corner, and anything you draw outside it is simply dropped — you cannot mark the rest of the screen by accident.

Coordinates must be whole numbers, and Lua hands you a fraction the moment you divide. That catches nearly every chart, because a bar's length is a proportion of something:

```
local width = math.floor(barMax * fraction)  -- not barMax * fraction
```

A fraction is refused rather than rounded, and the message names the argument — *error decoding argument #3*. Refusing is the more useful answer: it points at the line where the arithmetic went fractional, which is where the mistake is, rather than quietly drawing something a pixel off and leaving you to notice.

// does the same job as `math.floor` when both sides are already whole: `(s.width() - w) // 2` centres something.

The drawing calls

Call	What it does
<code>s.clear()</code>	Empties the window
<code>s.line(x1, y1, x2, y2)</code>	A straight line
<code>s.rect(x1, y1, x2, y2)</code>	An empty box
<code>s.fill(x1, y1, x2, y2, pattern)</code>	A solid or shaded box
<code>s.circle(x, y, r, filled)</code>	A circle, hollow or solid
<code>s.pixel(x, y, on)</code>	One dot
<code>s.invert(x1, y1, x2, y2)</code>	Flips black and white in a box
<code>s.text(x, y, str)</code>	Writes text
<code>s.textWidth(str)</code>	How wide that text will be
<code>s.lineHeight()</code>	How tall a line of it is
<code>s.wrap(text, width)</code>	Where each line starts and ends, wrapped as the page wraps
<code>s.textFit(str, width)</code>	How much of the text fits in a width

`pattern` is a name, not a number: "solid", "dark", "grey", "light", "clear". On a black-and-white screen those are how two bars next to each other stay telling apart. "clear" is how you rub something out.

Text sits ON the line you give it

`text(x, y, ...)` treats `y` as the **bottom** of the letters. They grow upward from it.

So text against the bottom edge is `height() - 1`, not `height() - lineHeight()`. If you think of a line as sitting so many pixels down from the top, add `lineHeight()`:

```
local function textAt(s, x, top, str)
    s.text(x, top + s.lineHeight(), str)
end
```

Getting this backwards puts your text a line's height away from where you meant, which usually looks like the call being broken rather than the number being wrong.

Measure rather than guess. `textWidth` counts the font actually being used, so right-aligning is `width() - s.textWidth(str) - 4` and it stays right whatever the font turns out to be.

Fitting text into a width. Two calls do the measuring for you, in the font on screen:

```
local starts, ends = s.wrap(paragraph, s.width() - 4)
for i = 1, #starts do
    s.text(2, i * s.lineHeight(), paragraph:sub(starts[i], ends[i]))
end
```

`wrap` breaks text into lines the way your page does: between words, keeping each space with the word before it, and cutting a word only when it is too long for a whole line. It gives back where each line starts and ends rather than copies of the text, so a long paragraph costs very little. A line break in the text always starts a new line, and empty text is still one line, an empty one.

`s.textFit(str, width)` says how many bytes of `str` fit in `width` pixels, whole letters only. Add `true` to count from the end instead — that is how you show the end of a line that has grown too long.

Both need your window open, because they measure what is on it.

The same font limits apply as anywhere else: the BYOK's fonts cover the ordinary keyboard and little else, so an em dash or a curly quote draws as a filled diamond. Stick to plain characters in anything you draw.

Fonts

Your window starts in the font the writer chose, whatever that is — including one they installed on the card. A script that never mentions fonts therefore looks like the rest of their device rather than announcing itself.

`s.fonts()` gives back the names you may ask for. `s.setFont(name)` takes one, and gives back the name actually in use.

```
for _, name in ipairs(s.fonts()) do
    BYOK.sys.log(name)    -- "default", "arial small", ... "mono large"
end
```

Every name in that list works. There is nothing in it you cannot select.

A name is a family, a size, or both, in any order:

```
s.setFont("large")        -- bigger, same family
s.setFont("mono")        -- same size, different family
s.setFont("mono large")  -- both
s.setFont("default")     -- back to the writer's font
```

Families are `arial`, `times` and `mono`; sizes are `small`, `medium` and `large`.

A word it does not recognise changes nothing. `s.setFont("garamond")` leaves you on whatever you already had, and hands that name back. Being left where you were is easier to predict than being moved somewhere you did not ask for — and it means a typo costs you a font you wanted, not a layout you had.

Not every pairing exists. Times has no small. Asking for one you cannot have gives you the nearest size in the family you asked for — the typeface being the part you chose deliberately. That is why `setFont` hands a name back: compare it if you care, ignore it if you do not.

```
local got = s.setFont("times small") -- "times medium"
```

Changing font changes `lineHeight()`. What you have already drawn stays where it is, but everything you measure afterwards is in the new size. Set the font first, then lay out.

Waiting, and reading keys

`s.key(ms)` waits for a keypress for up to `ms` milliseconds. It gives back a name — "up", "down", "left", "right", "enter", "tab", "backspace", "delete", "home", "end", "pgup", "pgdn" — or the character itself for anything printable, so a space comes back as " ". If nobody pressed anything in time, it gives back `nil`.

Escape is not among them. It closes the window instead, which is what `isOpen()` is for.

Keys held with Ctrl or Shift come back as a second value, with the extra keys spelled out:

```
local key, name = s.key(500)
-- name: "ctrl+left", "shift+end", "ctrl+shift+right", "ctrl+backspace", "ctrl+c" ...
```

For a key pressed on its own the two are the same, so `name` is the one to use everywhere. The combinations there are: Ctrl or Shift with the arrows, Home and End; Ctrl with Delete, Backspace, the space bar and any letter; and Ctrl+Shift with Left and Right. Ctrl+Shift+Up and Down arrive as Ctrl+Up and Down, and Ctrl+H, I, J and M arrive as Backspace, Tab and Enter — that is how the keyboard sends them.

Check `name`, not the first value, for any of these. The first value was never meant for them: a Ctrl+letter gives back `nil` there, the same as no key at all.

That timeout is the whole trick to anything that moves:

```
while true do
  local k = s.key(140)
  if not s.isOpen() then break end -- escape was pressed
  if k = "left" then ... end
  if k = "right" then ... end
  -- runs every 140ms whether or not a key came
end
```

`nil` is not a failure. It is the beat — it is what moves the ball when the player is holding still.

`s.sleep(ms)` waits without watching for keys, and gives back `false` if the window closed while it waited. Use it when you want a pause of a known length — drawing a chart bar by bar so it can be watched:

```
for i, bar in ipairs(bars) do
  s.fill(10, i * 12, 10 + bar.width, i * 12 + 8, "dark")
  if not s.sleep(120) then return end
end
```

Keys deliberately do not cut a `sleep` short. An animation should not run at a speed that depends on whether anyone happened to touch the keyboard.

`s.isOpen()` is `false` once the writer has pressed escape. Check it after anything that waits, and stop when it says to.

Waiting needs a window. `sleep()` is the only pause a script has, and it does nothing once the window has gone. So there is no way to slow a loop down after you have closed it — if something needs pacing, do the pacing while you still have the screen.

Draw what changed, not everything

This is the one thing that decides whether something feels good or feels broken.

Your window keeps what you drew. Nothing repaints it behind your back. So a moving picture should erase the bit that moved and draw the bit that is new — not clear the window and rebuild it:

```
-- Once, before the loop
s.clear()
s.rect(0, 0, w, h)

-- Each frame: two small changes, not a rebuild
s.fill(oldX, oldY, oldX + 3, oldY + 3, "clear")
s.fill(newX, newY, newX + 3, newY + 3, "solid")
```

Clearing the whole window every frame makes the screen visibly blink. On a chart that is merely ugly; on anything that moves it is unplayable.

A window that scrolls

A conversation, a log, a list longer than the screen: keep every line in a table, remember how far back the reader is, and draw only the lines that fit. There is no scroll call. Redrawing that part of the window is the scroll.

```

local lines = {} -- every line so far, each already short enough to fit
local scroll = 0 -- how many lines back from the newest you are reading

local function rowsThatFit(s)
    return (s.height() - s.lineHeight() - 4) // s.lineHeight()
end

local function drawLines(s)
    local lh = s.lineHeight()
    local bottom = s.height() - lh - 4 -- room for a typing line below
    s.fill(0, 0, s.width() - 1, bottom, "clear")
    local last = #lines - scroll
    for i = math.max(1, last - rowsThatFit(s) + 1), last do
        s.text(2, bottom - (last - i) * lh, lines[i]) -- the newest sits lowest
    end
end

local function scrollBy(s, n)
    local most = math.max(0, #lines - rowsThatFit(s))
    scroll = math.min(math.max(scroll + n, 0), most)
    drawLines(s)
end

-- and in your key loop
local k = s.key(500)
if k = "up" then scrollBy(s, 1) end
if k = "down" then scrollBy(s, -1) end

```

Three things make it feel right:

- **Count up from the newest line, not down from the first.** A short conversation then sits just above the line being typed, the way a chat does, instead of hanging at the top of an empty window.
- **Anything new goes back to the bottom.** Set `scroll` to 0 when you add a line. Someone who has read back and then replies wants to see their reply.
- **Cut long lines before you store them.** Break each message into screen-wide pieces with `textWidth` as it arrives, so every entry in the table is exactly one line on screen. Scrolling is then only counting. `s.wrap` does the breaking for you.

The same idea works sideways for a line being typed. Once it is wider than the window, draw the end of it, because the end is the part being written. `s.textFit(line, width, true)` says how much of the end fits.

For a full example, see `jane.lua`: a chat with Miss Austen about your writing, with a typing line, a blinking cursor, and small arrows when there is more above or below. It is worth reading alongside this section.

When something covers your window

The BYOK may need to tell you something while your script is running — that the battery is low, most often. Its message appears over your window, and when it goes it takes that part of your picture with it.

`s.damaged()` is `true` once after that has happened, and `false` again when you have asked. Nothing else can put your picture back, because nothing else knows what was on it:

```
if s.damaged() then
    redrawEverything()
end
```

Check it wherever you already wait. Anything meant to stay on screen for a while should — a chart nobody redraws will sit there with a hole in it.

Stopping

Escape closes the window. If a script has stopped listening — a loop with a mistake in it, most likely — **hold Ctrl and press the full stop key**. That stops the script wherever it is and gives you the screen back. It works for any script, not only one that has drawn something.

You should not need it often. It is there so that a script you are still working on can never take the machine away from you.

Fonts a script cannot choose

Fonts from the card, by name. You get the writer's own font through `"default"`, and the built-in families by name, but a script cannot ask for one of their installed fonts specifically. Reading one in means holding a chunk of memory for as long as the window is open, so a script's window uses the built-in fonts or the writer's own.

When something goes wrong

A script that fails says so on screen, with the file and line it failed at.

```
Script error
scenebreak.lua line 9
This script needs a
capability it lacks
```

Your document is never touched by a script that fails. Nothing half-happens.

The log file

The screen holds two short lines. Most of what you want to know does not fit in two short lines, so it goes to a file instead:

```
Scripts/scripts.log
```

Take the card out and open it on a computer. Every line looks like this:

```
19120 scenebreak starting
19300 scenebreak scenebreak.lua:9: attempt to index a nil value (local 't')
```

The number is how long the BYOK had been switched on, in thousandths of a second. The name is the script the line came from. **The file starts empty every time the BYOK starts**, so what you are reading is this sitting and nothing older. It stops at about 32 KB and says so when it does.

print writes to it. This is the thing worth remembering:

```
print("got this far")
print("word was", word, "at", from)
```

Anything you pass gets written, separated by tabs, numbers and tables included. It needs nothing in your `@caps` line — neither does `BYOK.sys.log`, which does the same with one piece of text.

The whole error goes there too, not the shortened one. The dialog has room for a fragment; the file has the sentence:

```
dialog: scenebreak.lua line 9 | attempt to index a nil / value
file: scenebreak.lua:9: attempt to index a nil value (local 't')
```

The part naming *which* thing was missing is usually the part that does not fit on the screen.

The messages you are most likely to see

It says	It means
This script needs a capability it lacks	The <code>@caps</code> line does not ask for something the script calls. Usually a spelling.
This script needs a function called <code>main()</code>	A one-shot with no <code>main</code> . Check the spelling and that it is not <code>local</code> .
This script needs a function called <code>listen()</code>	A watcher with no <code>listen</code> .
<code>init()</code> must say which events to listen for	A watcher whose <code>init</code> never called <code>registerEvent</code> .
That function does not exist	A call the BYOK does not have. Check it against the list of what you can call.
The script used too much memory	Usually building one enormous string. Work in pieces.
The script ran for too long	A loop that does not end.

Anything else is Lua's own wording, shortened to fit. The log has it in full.

A watcher that fails switches itself off

A watcher runs on every word you type. If it failed each time, you would get a dialog for every word and no way back except restarting.

So the first failure is the last: you see one message, the watcher switches off, and the list shows it off. Fix the script, copy it across, restart, and switch it on again.

When a script does nothing at all

Not every refusal is an error, because most of them are not the script's fault.

- **A script you have just copied across is not there yet.** Scripts are looked for once, at startup. Restart the BYOK and it will appear — and the same goes for a change you made to a script already on the card. If your edit seems to have had no effect, this is why.
- **A script that needs a document, when none is open,** does nothing. Same if a menu or a dialog is in front — it acts on the document you are looking at, and if you are not looking at one there is nothing to act on. The script list is the exception: it is a way of running scripts, so opening it does not count as leaving your document, and it does not lose your selection either.
- **A letter with nothing bound to it** does nothing.
- **A watcher whose event is already taken** by another watcher that edits will not switch on. Only one watcher may rewrite text on a given event, because the second would be working from positions the first has already moved. The refusal names nothing you need to fix in your script — switch the other one off instead. Watchers on *different* events never collide.
- **A fifth watcher** will not switch on. Four is the limit.
- **A watcher that missed an event.** Rare, and the log says so. Either events arrived faster than your `listen` could deal with them, or another script changed the document in between — an event describes a stretch of text, and once something else has moved that text the description is no longer true, so it is thrown away rather than acted on. If a watcher seems to skip things, read the log before rereading your script.

None of these beep or complain. Nothing started, so nothing is left half-done.

When it worked but looks wrong

If text appears as a filled diamond, the character is in your file correctly and the screen simply has no shape for it. See the note on fonts under `doc.insert`.

Complete scripts, line by line

Copy any of these into a new file in the `Scripts` folder and change it — that is a faster start than an empty page.

Copy it, then restart the BYOK. Scripts are looked for once, when it starts up. A script you have just copied across will not appear in the list until then, and neither will a change you made to one

already there.

A scene break

```
--- @name          Scene break
--- @description    Types a centred scene break at the cursor
--- @type           oneshot
--- @key            b
--- @requires       document
--- @caps           doc.edit

function main()
  BYOK.doc.insert("\n\n* * *\n\n")
end
```

Six header lines and one line of work. `Ctrl+T b` puts a blank line, three spaced asterisks, and another blank line wherever you are. `Ctrl+Z` takes the whole thing back in one press — the BYOK treats a script's edit as one action, not as nine separate characters.

`\n` is a new line. `"\n\n"` is therefore a blank line between paragraphs.

A snippet expander

```
--- @name          Expand snippets
--- @description    Replaces an abbreviation as you type
--- @type           watcher
--- @requires       document
--- @caps           doc.edit

local snippets = {
  CHR = "CHARACTER:",
  INT = "INTERNAL",
  EXT = "EXTERNAL",
}

function init()
  BYOK.sys.registerEvent("WORD")
end

function listen(event, word, from, to)
  local text = snippets[word]
  if text then BYOK.doc.replaceRange(from, to, text) end
end
```

Three parts.

The table lists what turns into what. It is built once, when you switch the watcher on, and kept — which is the real difference between a watcher and a one-shot. Add your own lines to it; that is the only part you need to change.

None of them ends in a space. The space you typed to finish the word is what sent the event, it is not part of what gets replaced, and it is still there afterwards — so a snippet with a space of its own would give you two.

`init` asks to be told when a word is finished. Without this nothing ever happens.

`listen` runs each time you finish a word. It is given the word and where it was. If the word is in the table, it is replaced; if not, the script does nothing at all, which is the usual case and costs nothing.

Look at what it does not ask for. No `doc.read`. A watcher is HANDED the word by the event, so it never has to go and look — and `registerEvent` needs no capability at all. `doc.edit` is the whole of it, because changing the text is the only thing this does.

`from` and `to` matter. By the time `listen` runs you have typed the space after the word, and possibly more — so the script replaces *where the word was*, not wherever you have got to.

The first argument, `event`, says which event this is. This watcher only asked for one, so it could be ignored — but check it anyway. The day you add a second event, a `listen` that assumed there was only ever one becomes a bug you have to find rather than one you were told about.

The same idea, on demand

Change two lines and it waits for Tab instead:

```
function init()
  BYOK.sys.registerEvent("TAB")
end
```

`from` and `to` now cover the abbreviation *and* the spaces Tab left behind, so the same single `replaceRange` clears both.

The spaces are the one thing to think about. A `WORD` expander inherits the space you typed and must not add one of its own; a Tab expander inherits nothing, so it wants the trailing space back: `CHR = "CHARACTER: " .`

And whatever else it does, **a Tab watcher must always replace the range** — even when it has no expansion to offer, and even when you change your mind. Do nothing and those spaces stay in the prose, where they are invisible and nobody will spot them for a hundred pages:

```
function listen(event, word, from, to)
  local text = snippets[word:upper()]
  BYOK.doc.replaceRange(from, to, text or word)  -- 'or word' puts it back
end
```

Which you want depends on the abbreviation. Automatic suits marks you always mean — `CHR` is never a word. Tab suits anything that is also ordinary English, where expanding it every time you typed it would be maddening.

Tab can offer you a choice

Because Tab is you asking for something, a Tab watcher may put a list up — the only event that may. This is `pick.lua`: type `list`, press Tab, choose a divider.

```
--- @name      Pick a divider
--- @description Type list and press Tab to choose one
--- @type      watcher
--- @requires   document
--- @caps       doc.edit, ui.list

local dividers = { " * * * ", "- - -", "~ ~ ~", " * * * * " }

function init()
  BYOK.sys.registerEvent("TAB")
end

function listen(event, word, from, to)
  if word:lower() == "list" then
    BYOK.doc.replaceRange(from, to, word) -- not ours: give the word back
    return
  end

  local choice = BYOK.ui.list(dividers)
  BYOK.doc.replaceRange(from, to, choice and dividers[choice] or word)
end
```

Both endings replace the range, and that is the whole lesson of the script. The last line reads "the divider they chose, or the word back if they chose nothing" — so escaping the list leaves the line exactly as it was, rather than with five invisible spaces on the end.

The word you Tab after is how the watcher knows the list is wanted. Anything else gets handed straight back, which is what lets this sit alongside your ordinary typing without ever surprising you.

One that asks first

```
--- @name      New scene
--- @description Choose a scene type and drop in its heading
--- @type      oneshot
--- @key       s
--- @requires  document
--- @caps      doc.edit, ui.list

local scenes = {
  { name = "Interior, day",   text = "INT. LOCATION - DAY\n\n" },
  { name = "Interior, night", text = "INT. LOCATION - NIGHT\n\n" },
  { name = "Exterior, day",   text = "EXT. LOCATION - DAY\n\n" },
  { name = "Flashback",      text = "INT. LOCATION - DAY (FLASHBACK)\n\n" },
}

function main()
  local names = {}
  for i, scene in ipairs(scenes) do
    names[i] = scene.name
  end

  local choice = BYOK.ui.list(names)
  if not choice then return end

  BYOK.doc.insert(scenes[choice].text)
end
```

Ctrl+T s, pick a kind of scene, and its heading appears where you were. Type over LOCATION.

Each entry keeps its name and its text together. Adding a scene type is one line, and the two halves cannot drift apart — which they would if the names and the templates were separate lists that had to stay in the same order.

The list is given just the names, built in the loop above. This is why `BYOK.ui.list` takes a table rather than a run of arguments: the list you offer is usually made from something rather than typed out.

The answer is a number, not the text, which is what lets `scenes[choice].text` work. Had it handed back the name you would have to search the table for it again.

if not choice then return end comes first. Backing out is the commonest answer of all, and a script that forgets to check it goes on to use nothing as though it were something.

Worth going one step further once it works: read the scene list from `scene.dat` and fall back to the table above when there is no file. Add `data.read` to the `@caps` line, and see **BYOK.data** for how. Changing the scenes then means editing a plain text file rather than the script.

One that asks you to type

A list is for answers you already know. When you cannot know them — a name, a title, a word that has never been written down — ask instead. This is `ask.lua`.

```

--- @name      Insert a heading
--- @description  Asks for a heading and writes it on its own line
--- @type      oneshot
--- @key       i
--- @requires   document
--- @caps      ui.ask, doc.edit

function main()
    local heading = BYOK.ui.ask("Heading")
    if not heading then return end

    BYOK.doc.insert("\n" .. heading:upper() .. "\n\n")
end

```

Ctrl+T `i`, type a heading, press Enter, and it lands in capitals on a line of its own.

The same first line as the scene picker, for the same reason: `nil` means they backed out, it is the commonest answer, and it is checked before anything else happens.

One check, not two. The box refuses to accept a blank line, so there is no such thing as an empty answer and `if not heading` catches everything. A `heading ≅ ""` alongside it would be guarding against something that cannot occur.

Nothing is waiting in the box when it opens, and nothing can be. There is no way for a script to fill it in advance. That is a deliberate limit rather than a missing feature: it means every answer is one the writer typed, and no script can offer them words and have those words come back looking like their own. The price is that a script which changes something makes them type the whole of the new version.

One that watches whole lines

`WORD` and `TAB` hand you a word. `LINE` hands you the whole paragraph you just finished, every time you press Enter. This is `tidyline.lua`, and it takes out the spaces you left hanging on the end:

```

--- @name      Tidy line ends
--- @description  Drops spaces left hanging at the end of a line
--- @type      watcher
--- @requires   document
--- @caps      doc.edit

function init()
    BYOK.sys.registerEvent("LINE")
end

function listen(event, text, from, to)
    local tidy = text:gsub("%s+$", "")
    if tidy ≅ text then
        BYOK.doc.replaceRange(from, to, tidy)
    end
end

```

Trailing spaces are the right size of problem for a watcher. Nobody ever means them, you cannot see them, and fixing one changes nothing you would have wanted kept. Anything you might have written on purpose does not belong in something that runs while you type.

if tidy $\cong$ text is not a tidiness of its own. A watcher that rewrites every line you finish puts an undo step behind every Enter, so the writer pressing `Ctrl+Z` gets a change they never made instead of the one they were reaching for. Touch the document only when there is something to change.

Remember `LINE` carries a paragraph, and a paragraph past about a thousand bytes does not arrive at all — see the size column in the first section.

Reading the whole document

Everything so far has acted on one spot. `chunks()` walks the lot, front to back. This is `count.lua`:

```
--- @name          Count paragraphs
--- @description    Counts paragraphs and words in the current document
--- @type           oneshot
--- @key            p
--- @requires       document
--- @caps           doc.read, ui.alert

function main()
    local paragraphs, words = 1, 0
    for chunk in BYOK.doc.chunks() do
        for _ in chunk:gmatch("\n%s*\n") do paragraphs = paragraphs + 1 end
        for _ in chunk:gmatch("%S+") do words = words + 1 end
    end

    BYOK.ui.alert("This document", paragraphs .. " paras, " .. words .. " words")
end
```

It adds up as it goes and keeps no text. That is the shape to copy. A script that glued the chunks together into one string to search afterwards would be holding the whole document in memory, which is the one thing chunking exists to avoid — and on a long manuscript it would run out.

Counting per chunk and summing is safe because no piece of the document arrives twice. Each chunk ends at the end of a line and the next one starts where it left off, so a word is never split and never repeated.

There is no word-counting call, and there should not be. `( "%S+" )` is a word counter — a run of anything that is not a space. Lua already does this, so the BYOK does not.

A window that hands something back

The last shape, and the one that catches people. This is `jot.lua`: a blank screen to write a note on, away from the page, which lands at your cursor when you press escape.

```

--- @name      Jot
--- @description  Type a line on a blank screen, escape puts it in the page
--- @type      oneshot
--- @key       j
--- @requires  document
--- @caps      doc.edit, screen.draw

local s = BYOK.screen

-- Redraw one row: rub it out, write it again.
local function showLine(text)
    local y = s.lineHeight() * 3
    s.fill(0, y - s.lineHeight(), s.width(), y, "clear")
    s.text(4, y, text)
end

function main()
    if not s.open() then return end

    s.clear()
    s.text(4, s.lineHeight(), "jot -- escape when done")

    -- 1. GATHER. Nothing here touches the document.
    local text = ""
    while s.isOpen() do
        local k = s.key(400)
        if not s.isOpen() then break end
        if k == "backspace" then
            text = text:sub(1, -2)
        elseif k == nil and #k == 1 then
            text = text .. k
        end
        showLine(text)      -- redraws one row, not the window
    end

    -- 2. The window has gone. NOW the page will take it.
    if text == "" then
        if not BYOK.doc.insert(text) then
            BYOK.sys.log("jot: the editor refused the insert")
        end
    end
end
end

```

The order is the whole point. While your window is up, the page will not accept changes — `doc.insert` hands back `false` for as long as you hold the screen. So this collects into an ordinary Lua string and writes nothing until escape has closed the window.

Doing it the other way round, inserting as each letter arrives, looks perfectly reasonable and **does nothing at all**. Every call is refused, and if you do not check what `insert` gives back, nothing tells you.

A printable key comes back as the character itself, so `#k == 1` takes letters, digits and spaces and leaves the named keys alone — `"up"`, `"enter"`, `"backspace"` are all longer than one byte.

It redraws one row, not the window. Clearing everything on each keystroke makes the screen blink once per letter, which is unusable to type into. That is all `showLine` does: `fill` with `"clear"` rubs the row out, then `text` writes it again. It is also where you would call `damaged()`, for when a notification has covered your window and the whole thing needs drawing afresh.

More examples

More scripts, including longer ones than fit in a guide, are on the BYOK website.

Switching a watcher on

`Ctrl+T`, wait a moment for the list to appear, move to the script, press the space bar for its details, space again to change On to Off or back, then Enter.