The *8th*™ Programming Language

Manual version 20.03

Ch. 1 Installation

1.1 Requirements

8th is known to run on the following systems:

- Microsoft Windows XP/7/8/10
- macOS 10.7 and later (only 64-bit; 32-bit is no longer supported)
- Linux - Ubuntu 14.04 and later, and derivatives or similar systems based on **`libc6`**
- Raspbian (Raspberry Pi version of Debian Linux; also works for similar embedded Linux systems)
- Android 4.4 (API level 19, ARM devices) and later
- iOS 9.0 and later

The “Raspberry Pi” version is tested on the Pi Zero W, Pi 2, Pi 3, and Rock64. It *may* also run on other ARM-Linux based boards. The Linux version *may* run on distributions other than Ubuntu derivatives; however, those *are not specifically supported*.

Those versions running on Intel (or compatible) hardware, *require* a CPU which has SSE2 instructions. That covers almost all Intel computers currently running. *8th* will *not* run on older hardware.

Similarly, those versions running on ARM hardware require at least "ARMv6", and will not run on older hardware.

1.2 Installing 8th

8th is distributed in an all-inclusive "zip" file, which contains versions of *8th* for all the platforms supported by the version you downloaded. Your versions of *8th* are *licensed specifically to you* and may not be shared.

In order to use *8th*, you will need to unpack the zip file into any folder which is accessible to you.

There are a number of folders included in the zip:

folder	description
`bin`	*8th* executables for all the supported platforms
`docs`	The manual and tutorials
`libs`	*8th* support libraries (*8th* code)
`samples`	Sample code to supplement the tutorials and manual
`keys`	The encryption keys for this specific version of *8th.*

Pro+ Note that the encryption keys are *unique* to each specific version of *8th* you download, and they are important for the purpose of building encrypted, deployable applications.

Note also that regardless of the platform on which you *develop*, you can *produce* applications for any platform supported by *8th*.

Once you have unpacked the zip file, you can run *8th* directly. For example, if you installed it in **`/opt/8th`** on a 64-bit Linux, then you should be able to type **`/opt/8th/bin/lin64/8th`** with whatever parameters you like, and it should work correctly. Do something similar on macOS or Windows.

For your convenience, you will want to update your system's **`PATH`** variable so that your *8th* binary can be found by simply typing **`8th`** .

1.3 Starting and stopping 8th

Please note: *8th* itself is a *command-line program* — it isn't meant to be started by clicking on a desktop icon. You can *create* regular GUI programs with it, but the *8th* compiler itself is a CLI (command-line interface) program.

If you are running on Windows, and using an MSys or Cygwin shell, then you *may* also need to use the freely available **`winpty`** program in order for your console mode programs to work properly. Recent versions of Windows 10 don't seem to require that.

The *8th* command-line looks like this:

```
8th [options] [[-f] file...] [-e 'code'...]
```

Where the **`[-f] file`** option means "interpret the contents of the file", and **`-e code`** means interpret the specific *8th* code given. Both options may be given more than once on the command-line, and the effect is cumulative. In other words:

```
8th -f first.8th -f second.8th
```

will interpret the contents of the file **`first.8th`** and then interpret the contents of the file **`second.8th`** . Note that if you want to just run one *8th* file, you do not need to say **`8th -f first.8th`** , you can instead just type **`8th first.8th`** .

Here are all the CLI options *8th* understands:

option	description
-e strl	Interpret the string **`str`** as *8th* code
-ee strl	Interpret the string **`str`** as *8th* code and quit afterwards
-f nm	Interpret the file called **`nm`**
-g	Enable **`SED`** debug/checking
-h	Display the help
-H N	Set the console history size to **`N`** lines (default: 100)
-k N	Set the data- and r-stack sizes to **`N`** items (default: 128K)
-l	More debugging information
-r N	Set the callback data-stack size to **`N`** items (default: 256)
-s N	Set recursion stack size to **`N`** bytes (default: 0, meaning system-default)
-v	Print the *8th* version and quit
-vv	Same as -v but with additional information for debugging purposes
-x	Turn off stack type and bounds checking

option	description
-z	Turn on data-item counting
-? nm	Print help for the item 'nm'
--	Signals the end of 8th options

In general you should not need to use the **`-k`** , **`-r`** , or **`-s`** options; they are provided in case you find them useful, for example if 128K items on the stack at one time is too few (to be honest, you should change your algorithm if that's the case...).

If you find you are using command-line switches often, you can simply use a "shell script" (on macOS or Linux), or a "batch file" (on Windows; also called a "command file") to start *8th* with the options you prefer.

To quit *8th*, do any of the following:

- Type **`bye`** and hit **`ENTER`** . That will tell *8th* to quit normally
- Type **`1 die`** and hit **`ENTER`** . That will tell *8th* to quit abnormally and return the status-code 1 to the operating system
- Press the key **`Ctrl+C`** twice in rapid succession, or **`Ctrl+D`** once. Either tells *8th* to quit immediately
- In a running application, invoking **`throw`** will cause an exception. If you're running from the *8th* console, you will be returned there in most cases; if you're running a file or a packaged application, a message will be printed and the app terminated.

1.4 Running your programs

8th can run your programs in several modes:

mode	description
interactive	just start *8th* and type your code in the console
script	put your code in a text file and run it using **`8th mycode.8th`** (for example). See the tip below…
app	using the **`build`** script you can convert your code into an packaged application, ready to deploy. Pro+ You can create encrypted packages to help deter hackers and other nefarious parties

In the latter two scenarios, your code must be in *plain-text files* (e.g. not a word-processing format). Any supporting files should be placed in the *same folder* as your code, or in a sub-folder of it. You can access those other files in *8th* by means of the word **`app:asset`** .

An "app" is a standalone program which runs on its own like any other program for the target platform, and does not need to be run by *8th* via the command-line. The contents may be encrypted to help deter hackers, as mentioned above. The details of producing standalone applications may be found in the section on using the **`build`** tool.

Tip: if you're running on macOS, Linux, or RPI systems, you can make your script run like a regular system command by:

1. Making the file executable: **`chmod +x scriptname`**
2. Make the first line of your code: **`#! /usr/bin/env 8th`**
3. Ensure a relevant *8th* executable is on your **`PATH`**

This will also work with packaged **`appdata`** , so if you used **`bin/build`** to create an executable you could use the **`appdata`** it created in the same manner by doing:

```
cat - path/to/appdata > newappdata
#! /usr/bin/env 8th
chmod +x newappdata
```

1.5 Reporting bugs or other issues

When reporting a bug, please give as much detail as possible in the description. That will make it easier for us to understand the issue, reproduce, and address it. If you want us to contact you regarding the issue, please say so and tell us how. It is recommended you include your email in the bug report, for this purpose.

It is *strongly preferred* that you report bugs online in the bug-tracker database That is because:

- The developers will get an email about the issue
- The issue will be tracked so it doesn't get lost or forgotten
- The issue can be prioritized relative to other issues

You may also post a bug report on the forum , but that is likely to result in the bug getting lost in the shuffle.

If you *must* include confidential or proprietary information in your bug report, please send that information to us via email . See the **`README.txt`** file for Ron's PGP public key, in case you want to encrypt messages to him.

1.5.1 Proper reporting

You probably want your bug to be fixed. For that to happen, we need the following information:

- The output from **`8th -vv`** on the system where the bug occurred
- A precise description of what the symptoms of the bug are: what you saw, vs. what you expected
- A concise bit of code which demonstrates the bug
- A set of instructions on how exactly you got the bug to appear

Please make sure that when you log a bug, you are as *clear as possible* as to how it should be reproduced. If you're reporting *anonymously* on the bug-tracking database, please include your email address in the report so we can ask you for more details if needed.

1.5.2 What's a bug?

A "bug" is one of:

- A **`Program crashed`** message, or a program crash even without a message
- Behavior of a word which is not in accord with the documentation
- Missing or incorrect documentation

Any of those should be reported in the bug database; or, as a last resort, on the forum.

A "bug" is *not* an **`Exception...`** message, unless you followed the documented SED of a word and the exception message resulted.

A "problem" is any difficulty you're having which isn't an actual bug. All problems can and should be posted on the forum, so that others can also chip-in to help solve your issue.

1.6 Updating 8th

When a new release of 8th is available, you can get it by going to https://8th-dev.com/refresh.php and entering your customer id or the email you registered with. This is one reason it is important to use a permanent email address when registering, rather than a "throwaway" one.

Update your *8th* version from within the *8th* console by invoking **`app:current`** .

Once you've got the new release, simply unpack the ZIP file in the same location as your current version unless you want to also keep the older version of *8th*.

If you want to keep multiple versions of *8th*, then you *must* unpack the ZIP in a different location from your current version!

To be kept appraised of new releases, you can:

- Look at the canonical 8th version page , or
- Check the "Announcements" section of the 8th forum , or
- Check our Twitter feed

1.7 Differences between the 8th SKUs

There are several distinct versions (SKUs) of *8th*, which differ in their built-in functionality. All versions have built-in encryption, SQLite database support, big-math and many other features; however, some features such as Bluetooth support or encrypted executables are only available in some SKUs.

The current list of SKUs and their features is on our website . You can upgrade to a more capable SKU at any time, and your license will be extended by a year from its current expiration date (or the current date, if that is later). If you have difficulties using the upgrade page, or if you have special needs, please contact us and we will make every effort to help you within two business days.

Please note that all SKUs come with *one year of free updates* to the software. Once that year is over, if you wish to continue receiving updates you will need to renew your update service by visiting the *8th* update page and extending your service. Note further that if more than 90 days have elapsed since the end of your update year, you will have to purchase a full license again (rather than pay the substantially lower update fee).

Updates are *not required* for you to continue using *8th* or for your applications to continue to run. However, if you don't update *8th* you will miss out on bug fixes and new features.

Ch. 2 Introduction to 8th

8th is a secure cross-platform development language which lets you concentrate on your application's logic instead of worrying about differences between platforms.

It lets you write your code once, and simultaneously produce applications for multiple target platforms.

Its built-in encryption helps protect your application from hackers and intellectual property leaks.

This manual covers all the features of the *8th* language and programming environment. Its purpose is to explain in-depth all aspects of the product. If you note something missing or incorrect, please bring it to our attention (post a bug report!).

2.1 Typographic conventions

The following conventions are used in this manual:

Code samples appear indented from the body text, in a monospace bold font:

```
: sample
  "Code is monospace!" . ;
```

In stack-effect diagrams (*SED*, for short), a stack item in means that kind of item is read from the input-stream (either the *8th* source code in a file, or from the keyboard, or standard-input if input is redirected).

SEDs have the format: **`\ n --`** to indicate the data-stack on entry has one item, a number, and consumes the item. If the r-stack is affected, it is documented identically, but using curly-braces to indicate the r-stack, e.g. **`\ { -- m }`**

This symbol indicates items to which you should pay special attention!

In addition, the following markup is used to indicate that a feature is only available in certain SKUs:

symbol	meaning
Hobby+	Available in Hobby, Professional, Embedded, and Enterprise
Pro+	Available in Professional, Embedded, and Enterprise
Emb	Available in Embedded
Ent	Available in Enterprise

2.2 Glossary

This manual uses terms which are sometimes different from what you may be used to from other programming environments. To make it clear what is meant, we present a short list of terms as used by *8th*:

term	definition

term	definition
asset	anything packaged along with the code (fonts, graphics, other code, etc)
container	a type of data item which contains other data items
factor	a unit of code which can be extracted to its own word and invoked
invoke	"execute", "run", or "call" a word
item	any of the data types known to *8th*
namespace	a "vocabulary" of (usually) related words
refcnt	the "reference-count" of a data item
scalar	a type of data item used for its value; for example a number
SED	"stack-effect diagram" — short diagram of how a word affects the stack
task	the same as a "thread" or "co-routine" in other languages
utility	a type of data item which is neither a scalar nor a container
whitespace	the ASCII characters 9–13 and 32
word	the same as a "function" or "procedure" in other languages

2.3 Some historical background

8th is based upon a much earlier language called Forth, which was initially designed in the early 1970s for controlling telescopes. Forth quickly found its niche in embedded systems because of its small size, low resource requirements, ease of porting to new hardware, and flexibility.

Despite its many advantages Forth has remained a niche language, partly because of the lack of true standardization between versions. That lack led directly to the cynical observation, "if you've seen one Forth, you've seen one Forth".

8th's immediate ancestor is Reva Forth, also written by Ron. Though *8th* shares no source code with Reva, it was influenced by many of its ideas. Throughout this manual and the accompanying documentation, we refer to *8th* as well as other implementations of the Forth language as "Forths". Whether or not you are already familiar with Forths, you may benefit from working through the *8th* tutorials, located in the tutorials sub-folder of the samples folder.

8th came about because Ron was looking for a development tool to help him write an application which he wanted to deploy on a variety of popular platforms. He searched far and wide for something appropriate. After trying a number of products, he found all of them lacking for his particular needs. So he started writing his own solution, basing it on ideas from his previous Reva Forth and from his decades of experience in the software field. The result is *8th*.

Though there is a Forth standard (actually, there are several), *8th* does not adhere to it in any particulars, choosing instead to be inspired by Forth's concepts while being more accessible to a wider audience. Most design decisions were made in the interest of keeping applications secure while providing freedom to accomplish normal programming tasks in a cross-platform and reasonable manner.

2.4 Unique features of the 8th language

As a developer, you are probably familiar with a number of programming languages. Most of the ones in common

use today are similar enough that one rarely has difficulty picking up the essentials. You may be intrigued, then, that *8th* is different enough from what you are probably used to that you will need to pay close attention as you learn it. Take heart from the fact that it is not a difficult language. Here are some of the concepts which set *8th* apart from most other languages:

2.4.1 Words

The smallest unit of execution in most languages is a "function" or "procedure". In Forths the equivalent is called a “word”. That is because Forths try to interpret any whitespace-delimited group of characters in the input. If that group of characters is a recognized "word", a Forth will execute it. What that means will become clear in the next few sections.

2.4.2 Interpreter or Compiler?

The most popular programming languages are either compiled (like C/C++) or interpreted (like JavaScript or PHP). Some languages (such as Java) are compiled in a two-phase process; first interpreting the source into an intermediate format which is then compiled "on-the-fly" at runtime (this is known as "JIT", or "just-in-time" compilation).

8th operates in two modes: "interpretation" and "compilation". When interpreting (by parsing words one by one and looking them up), it immediately executes the found code. When compiling, it produces native-code directly by compiling a call to the word. The precise details of *8th*’s syntax and how its interpreter works may be found in the syntax reference.

Unlike most current languages, *8th* does not perform any optimizations on your code, except for tail-call elimination. The reason is twofold. First, Ron’s experience is that optimizers often cause incorrect code; correctness and predictability of the application suffer as a consequence. Second, the best optimizer is between the ears of the programmer. The most significant performance gains are made by choosing an appropriate algorithm, rather than relying on a compiler to choose an optimum instruction sequence. Of course some disagree…

2.4.3 Stacks

Like all other Forths, *8th* is a *stack-based language*. This means that parameters to words as well as results from them are put onto a *stack*. In this manner, the output of one word is immediately available as input to the next one. This encourages what is often called a "concatenative" programming style, because words are "chained together".

This concatenative style can serve to make code much more readable, since the "noise" of naming parameters is eliminated. For example, a hypothetical dishwasher controller might look like:

```
fill-water rinse-dishes drain-water dry-dishes
```

On the other hand, because the parameters are not named, code can also become *less readable*! Therefore it is important to make liberal use of comments, especially those regarding a word’s SED. It is also very highly recommended to restrict the number of stack items a word uses to three or fewer, to make code easier to

understand. *8th* has several words dedicated to manipulating items on the stack. A full description can be found in the chapter on stacks.

2.4.4 Item types

In *8th*, all builtin data-types "know" what they are, and words can (and most do) check to ensure they are operating on the type of data they expect.

For example, **`123`** is a numeric value just as in other Forths. However, it is not *just* a value on the stack. Rather, it is an item of the namespace **`n`** , and other words can determine that it is in fact a number and not, for example, a string by using a code snippet like:

```
>kind ns:n n:= if …
```

The various builtin types known by *8th* are listed in the chapter on data types, and detailed information about them is there and in subsequent chapters.

2.4.5 Reference counting

8th assumes you are not interested in the drudgery of keeping track of memory allocations and de-allocations. Not only that — it does not provide you any way to directly allocate memory!

Most of the time you are not interested in the reference-counting mechanism either, you just care that it works. But in case you want more information, it can be found in the reference-counting section in the data-types chapter.

2.4.6 Exceptions

Rather than let you do something illegal, *8th* will throw an exception. There are a number of different exceptions which *8th* knows about, and you can throw your own if you like.

The default handler (called, unsurprisingly, **`handler`**) causes *8th* to display a message and quit if an exception occurs. If you prefer to handle exceptions in a different way, you can override the default handler word using **`w:is`** . See the section about words for more information.

You may also use a task-specific handler, using **`t:handler`** .

Exceptions are thrown if you underflow the stack, if you pass an incorrect data type to (most) words and for many other situations. You can also invoke **`throw`** yourself and cause an exception.

8th treats exceptions as *fatal errors*, so the default behavior of quitting is best.

Note: in interactive mode, e.g. when you start *8th* and just start typing code, a thrown exception *does not quit the interpreter* unless it's a stack over- or under-flow. The reason for this is to make it easier for you to see what happened and take corrective steps as you interactively work through your code. However, the exception is still a "fatal error"; and *8th* may not be able to continue without crashing. This does not happen when running from a file or an application, since in those situations *8th* will quit (unless you overrode the handler word, in which case *caveat programmer*).

Many words indicate an error condition by returning the value **null** which can be checked using the phrase

```
null? if (null processing goes here…) then
```

2.4.7 Getting help

This manual, the tutorial and the sample code provided with *8th* should be your first source of help if you have difficulties. If you cannot figure something out, or if you just want to discuss the matter, you should join in the discussions on the forum .

If you are typing code in the console and want some help, there are two helpful words at your disposal: **help** and **apropos** . The first lists all words whose names match the text given after it, along with their documentation. The second does the same, but also matches any help text which contains the text you type. A further set of helpful words is **words** and **words/** . The first lists all the words *8th* knows about; the second lists all words whose names contain the text given after it.

2.4.8 About the name

As mentioned before, Ron had written a Forth-based language called "Reva Forth". The word "reva" in Hebrew (רבע) means "one-fourth" and Reva is also "one Forth". Since we were working on a second-generation of Reva, so to speak, we multiplied 4 by 2 giving 8. So instead of one-fourth we have one-eighth.

Aren't you glad you asked?

2.4.9 Quick introduction for users of "mainstream" languages

If you're coming from C or Java or most more common languages, you may find *8th* a bit puzzling. To help set you on the right path, here are some of the primary differences between *8th* and "the mainstream", as well as some helpful hints:

- As a consequence of the way the interpreter looks up items, you must declare a var or a word prior to its first use. Failure to do so will result in the exception **can't find ...**
- A var is a *named container* for other items. It is *not* the name of the item referred to! So **var x** may hold an array, but it is wrong to try to access **x** as if it were itself an array, and doing so will cause an exception to be thrown
- You cannot declare a var inside (e.g. local to) a word, don't try it! You can, however, use **w:@** and **w:!** to access *word-local variables*
- Try to write your own words so that they can be *chained together* with other words. For example: the "file words" do some operation on a file and leave the file item on the stack (and perhaps other information) for the next word to work on
- Keep your words short. Comment them. Be sure, especially, to comment the stack-effect, and...
- ... *test* each word you write (preferably as you write it or shortly thereafter), ensuring it adheres to its documented SED. This will help you write bug-free code. Re-test if you change the SED or the code
- Consult the **help** and **apropos** words for details on the SED action, and side-effects of any word you aren't sure of
- In *8th*, an exception *is a fatal error*, and will cause the application to quit (this is the default behavior). Don't

expect to "catch" one and handle it effectively

- There is no compile/link cycle. Instead, *8th* is an engine which first interprets your code and if necessary compiles it at runtime. When running on a device, your code is native code for the platform, not bytecode running in a VM

Ch. 3 8th's syntax

As in other Forths, *8th* plucks whitespace-delimited words from its input and tries to interpret them, one at a time. However, there are some special lead-in characters which make *8th* interpret the characters which follow in a different manner, and the sequence of events in the interpreter is important.

8th's syntax is quite minimal: it is completely described by the few rules listed below. As mentioned, there are two "modes" as in other Forths: "interpret mode" and "compile mode".

mode	description
interpret	*8th*'s initial mode. The text you enter is interpreted immediately according to the rules listed below
compile	Initiated by the **:** or **(** words, terminated by **;** or **)** respectively. In compile mode, the words you enter are compiled into the word being created, to be executed when it is invoked

8th has *no "reserved words"*. This means you can override *any* of its built-in words. Please do be careful if you do so, since the old word is then no longer easily seen by the interpreter, which may have *Unusual Consequences*™. Also bear in mind that with great power, comes great responsibility. Just because you *can* do something, does not mean you *should*.

3.1 Interpreter rules

Here is what happens inside the *8th* interpreter:

1. Parsing starts by picking up characters one-by-one from the input (which may be a file, an "eval" string, redirected standard input, or the keyboard), and collecting them into a word. Any whitespace stops the collection process
2. The parsed word is looked up using the equivalent of **find** , and is *case-sensitive*:
    1. If **only** was invoked, it only looks inside that namespace. Otherwise:
    2. Look in the namespace of the "current item". In interpret mode, that's the item on TOS (top of stack) if there is one. In compile mode, it's the last item compiled
    3. If not found, look in each namespace in the "with list". By default, that list contains just the **user** namespace
    4. If not found, see if it's a "fully qualified" name, e.g. a name with a namespace-name, colon and word — such as **n:+** — and then look in the specified namespace
    5. If not found, look in the namespace specified by the last **ns:**
    6. If the word is still not found, look in the **G** , or "global" namespace, with is thus *always* checked if the word was not found beforehand
3. If not found, examine the first character of the word to see if it is a special item type (see the section on "Special characters" below)
4. If none of the above succeeds, try to interpret as a number using the current **base**
5. If that fails, try parsing it as a **complex**
6. If that fails, try parsing a **date**
7. When all else has failed, invoke all the "last gasp" words installed using **G:+hook** , with a string containing the offending character sequence. The default behavior is to throw the exception **Unknown: …**

Note how the interpretation rules allow you to override *any* word, since *8th* first checks for existing words. You can, for example, override **8** to print "eight":

```
: 8 "eight" . ;
```

Since that's an incredibly bad idea, it's fortunate that you can undo the damage by telling *8th* to forget your newly created word:

```
"8" w:forget
```

When an item has been successfully parsed, the interpreter pushes it on the stack (in interpret-mode), or compiles it into the current word (in compile-mode). This behavior is the usual case, but "immediate" words, and likewise use of **p:** , **i:** , and **l:** modify this. Details are provided in the chapter on words.

3.2 Strings

When interpreting a string, *8th* follows the same conventions used in the "C" language. First, a string is any sequence of characters delimited by double-quotes (**"**). Second, if a back-slash character (****) is encountered, the following characters are interpreted specially:

"	double-quote, ASCII 34
a	alarm, ASCII 7
b	backspace, ASCII 8
f	formfeed, ASCII 12
n	newline, ASCII 10
r	carriage-return, ASCII 13
t	tab, ASCII 9
v	vertical tab, ASCII 11
x	the next two characters are hex digits (e.g. \x20 is the space character)
u	the next four characters are Unicode hex digits (e.g. \u201c is the typographic ")
U	the next eight characters are Unicode hex digits (e.g. \U0000201c is the typographic ")

Any other character following a backslash is inserted literally. Finally, the text in a string is terminated with **ASCII NUL** (a single byte with the value 0). Even though strings keep track of their own length, it proves useful to be able to pass them to system routines without performing conversions.

Note that strings are sequences of UTF-8 encoded characters, so they may contain any valid Unicode character (even if your font doesn't display it properly).

Windows users, take note! A file name like **"C:\Program Files"** will not give the results you desire. Instead, use **"C:\\\\Program Files"** , or **"C:/Program Files"** .

Note: *8th* is not tolerant of malformed UTF-8. So if, for example, you have a buffer containing text in the CP-1255 encoding and then convert the buffer to a string, it is very likely that an exception will be thrown complaining about invalid UTF-8. *8th* can convert between character encodings using the **conv** word, but that requires the external **libiconv** library in order to work. *8th* can also convert between UTF-8 and UCS-2 using **ucs2>** and

`>ucs2` .

3.3 Special characters

There are characters which have a special meaning when encountered during interpretation of the input. That is to say, when encountered as the first character of a new word parsed from the input, they cause *8th* to interpret the remaining characters differently. The special characters and their meanings are:

"	string, terminated by a matching "
/	regex, terminated by a matching /
B	"bint" number
F	"bfloat" number
X	buffer, in hex format
[	array, following modified JSON syntax
{	map, also following modified JSON syntax

Note: the special characters listed above are processed only if a matching word was not found in the dictionary.

3.4 Numbers

Numbers are interpreted using special rules. If the lead-in character is:

+	make number positive
-	make number negative
0X	or…
0x	or…
$	interpret number as hexadecimal (e.g. base 16), regardless of current base
%	same, but binary (base 2)
&	same, but octal value (base 8)
#	interpret number as decimal (base 10), regardless of current base
'	The following single character is interpreted as an ASCII character value
B	If the numeric base is 10, interpret the rest as a "big int"
F	If the numeric base is 10, interpret the rest as a "big float"
e	or
E	If base is decimal, number is floating-point and following is the exponent
.	Anywhere in the input, means number should use floating-point
i	At the end of the input, means number is **complex** (e.g.: 3+4i)

3.5 Regular Expressions

The regular-expression syntax used in *8th* is that of PCRE , with all its features and limitations. When entering a new regex, one may either use the slash notation, e.g.

```
/(cat)|(dog)/
```

or one may choose to construct a regex from a string, using **r:new** :

```
"(cat)|(dog)" r:new
```

In either case, a new regex item is created. However, the second version may be used to create a regex from *any* (appropriate) string, and the parsing rules for strings then also apply. Using slash-notation, the regular string parsing rules do not apply.

3.6 Scoping

8th is unlike most other languages you may be familiar with, in that it has very primitive "scoping rules". For example, in the C++ language a variable declared inside curly-brackets is only visible to code that is also inside those brackets. *8th* does not work that way.

In *8th*, everything which can be looked-up with **w:find** must be inside a namespace. That means that named words and vars may be found in some namespace. If you take no other steps, any words and variables you create will be put in the "user" namespace, **ns:user** . That is, if you create new word **foo** , it will be fully-distinguished as **user:foo** . You can create new namespaces and use them to distinguish various aspects of your application.

Note: you *cannot* declare a var inside a word. vars are *always* global in scope.

However, you can easily change the scope of your words to another namespace. First, you can prefix the word with the namespace-name and colon, which will create it in the given namespace. Note that the namespace *must already exist* for *8th* to be able to create the new word there:

```
: m:xxx 123 ;
```

That example created a (useless) word called "xxx" in the m (map) namespace. You can also use the **ns** word to let *8th* know that you want to create new words in that namespace, e.g.:

```
ns:m ns
: xxx 123 ;
```

This has the exact same effect as the previous example, but is much nicer if you have many words you wish to put into a particular namespace. You can create a namespace of your own in which you put all your application's words (or some subset of them) by simply doing:

```
ns: mycode
```

This will create a new namespace "mycode" if it doesn't already exist, and informs *8th* that new words should be created there. It also makes mycode one of the namespaces searched for words. However, it does not affect the general search order. For that you need to use **with:** and **;with** .

3.6.1 Word-local Scoping

"word-local variables" are named-variables, whose scope is limited to the word in which they are declared, and to any non-local-containing words invoked from it.

In order to start a local variable scope, a word must be declared to have locals, using **locals:** like so:

```
locals:
: word-with-locals
  ... ;
```

Inside that word, and any non-local words invoked by it, you may access an effectively unlimited number of named local variables by using the **w:@** and **w:!** words:

```
locals:
: foobar
  1000 "baz" w:! ;
: bar
  120 "baz"
  w:! foobar ;

locals:
: foo
  100 "baz" w:! …
  bar …
  "baz" w:@ ;
```

In this rather useless example, foo and foobar are declared to begin a word-local-scope, while bar is a regular word. When foo is invoked, it sets the local variable named baz to the value 100. At some point it invokes bar, which sets baz to the value 120, and then invokes foobar, which sets baz to 1000. However, foobar is declared to start a new scope, so its "baz" is not the same as the "baz" declared by foo. As a result, when foo ends, it pushes the value 120 on TOS which was set by the subordinate bar.

This demonstrates the scoping rules for word-local variables as implemented by *8th*. You may see this in action in the sample **misc/locals.8th** .

3.6.2 Task-local variables

8th also implements task-local variables. Those are named-variables whose scope is limited to the task in which they are declared. They are comparable to "USER" variables in other Forths, or thread-local variables in other languages. They are accessed using the words **t:@** and **t:!** .

Each task has its own task-local namespace, therefore one may use the same variable name in different tasks. This means, for instance, that using the same code with the same variable name, one may parallelize a calculation without worrying about locking — since the same name in different tasks will access different actual variables. Of course, this only applies if the **t:@** and **t:!** words are used, rather than the usual **@** and **!** (e.g. **G:@** and **G:!**). See the sample **misc/tasklocal.8th** .

3.7 Namespaces

A "namespace" is a logical grouping of words, with a dictionary for looking up their actions. For example, the **n** namespace groups together all words which act on numbers, the **f** namespace words act on files, etc. Generally speaking, namespaces contain words which operate on certain types of data or have similar actions.

A namespace can also have associated pools of items, if items from that namespace are ever allocated. The pools

are allocated for namespaces on a per-task basis. For example, numbers are allocated from the **n** namespace pool, files from the **f** namespace pool, and so on.

There is a default namespace: **G** so named because it is "general". It is the namespace which is *always* searched after any other namespace. We'll explain more below.

To see all the words which belong to a namespace, invoke **help/** . For instance, all the array words are found by typing **help/ a:** .

3.7.1 Purpose of namespaces

Using namespaces allows us to accomplish a few things. First, we can keep our dictionaries clean. That means that we don't stick every word we create into one gigantic dictionary. That makes the name lookup faster as well, though perhaps not by much.

That also implies that it helps keep us from creating words with conflicting names, thereby reducing the likelihood we'll accidentally overwrite an existing word. It also implies that we can create different words with the same name, in separate namespaces. For example, we have **G:@** , **a:@** , **m:@** etc. — all words named **@** , but in different namespaces.

Namespaces also let us control access to words. We can, for instance, use **only** to make a particular namespace the only one the interpreter can search, thereby making it safe to allow access to the interpreter from untrusted code (e.g. user inputted code).

3.7.2 Proper use of namespaces

In interpret mode — that is, when *8th* is processing input from the console or from a file outside of a word-definition — it tries to deduce the namespace from the item on TOS.

For example, if you type: **1 2 +** in the console, *8th* correctly chooses **n:+** because **2** is a number, from the **n** namespace's pool of items. In this case, you don't have to specify **n:+** , because *8th* can correctly figure out what to do.

However, if you were to type **[1,2,3] (. drop) each** then *8th* will complain that it can't find **each** . That is because the item on TOS is a word, and there is no **w:each** . In this case, you must specify **a:each** so *8th* can do what you were expecting it to do.

Similarly, in compile mode — when *8th* is compiling a new word-definition — the way it finds words is different, and it is therefore often important to specify the namespace of a word, since it may not be able to tell which of several words to use.

3.7.3 Namespaces and the search order

The order in which *8th* looks for words is:

1. the namespace of the item on TOS, then
2. the namespaces, in order, in the **with list** (see **with:**), then
3. the global namespace **G** , then

4. in the specific namespace given by the word (e.g. n in `n:+`), then
5. finally the current namespace (which is `user` by default).

What that means is that you can modify the order *8th* searches for words. That can reduce the typing and clutter in your code, but may lead to unexpected results at times. So use with caution.

Note: it is *almost never* necessary to use the prefix `G:` to tell *8th* that a word is in the general namespace. That is because it will always look there for a word, eventually. The only time you must use the `G:` prefix is when it is possible that a namespace being searched has the word you are using, in compile mode. For example:

```
ns: a
var v
: x v @ ;
x
```

This will throw the exception, `expected Number, but got Variable` . That is because while compiling, *8th* looked in the `a` namespace and found `a:@` , and compiled `a:@` into the word `x` instead of `G:@` . However, you put the var v on the stack before the `@` , expecting that you would be using `G:@` which operates on vars. So in this scenario, you are required to explicitly say `G:@` . However, these scenarios are not very common, and it is preferable to avoid using the `G:` prefix so your code is more legible.

The built-in namespaces usually have short names of one or two characters, to save typing and compress the source. So for example, the namespace `s` contains the string words, and an item which has a namespace (numeric) identifier of `ns:s` is a string. The word `>kind` may be used to determine what namespace an item belongs to. It places the numeric value of the item on TOS, where you may then compare it with the value of the namespace identifier. For example:

```
: isnumber? \ x -- true|false
  >kind ns:n n:= ;
```

This example converts TOS to the numeric value of its namespace using `>kind` , and then compares that value with the numeric value of `ns:n` , the namespace identifier of numbers. It then does a numeric compare (note: `n:=`).

You may find the notation `n:=` a bit confusing, but it's there for a good reason: many other namespaces also have a word named `=` (for example, strings may be compared with an equals sign), but the only real connection between the various equals is that they do a comparison of their respective data types. Because it is convenient to use a similar symbol or word for similar actions, *8th* lets you have both a string `=` and a numeric `=` . Since they are nevertheless completely different, they are in separate namespaces.

If you find yourself using `n:` (say) very often, you may wish to use the word `with:` , which lets *8th* know it should check for the word you typed in those namespaces first. If you do use `with:` , you should pair it with `;with` when you are done, to remove the namespaces from the impromptu search order. To check on what has been put in the `with-list` , type `.with` . The above example could also have been written:

```
with: n
: isnumber?
  >kind ns:n = ;
```

There are three special namespaces: `ns` , `I` , and `#p` . The `ns` namespace contains the *names of all the namespaces*, and invoking any of the names it contains puts the numeric identifier of that namespace on TOS, as seen above. The `I` namespace contains internal factors of *8th* words which are useful in various places *within*

8th, but are not sufficiently useful to be documented. They are put in a separate namespace to reduce clutter in G. The `#p` namespace contains the names of words declared between `private` and `public` (or end of file). They are visible *only* during the compilation of a file (or library) and will become invisible after the compilation, meaning that searching for them using `w:find` will not succeed outside of their file or library.

3.8 JSON Rules

8th uses a modified form of the industry-standard JSON syntax for defining data items. While it understands standard JSON just fine, it makes a few additions to the standard syntax which are geared to making it work more conveniently with *8th*.

Comments. When declaring an array or map, you may insert an *8th* backslash comment in between elements (or between a key and its value, for an map). So this is legal syntax:

```
{
  "key" : \ this comment is perfectly legal (but not in standard JSON)!
  "some value"
}
```

Remember: in *8th*, backslash comments run to the end of the line. Also note that this syntax modification *only* works with the single-backslash comment, it does not work with `--` or any other comment words.

Expressions. In an array or map declaration you may use the backtick character ` to bracket an expression to evaluate. That expression will be evaluated when the JSON is, and its value will be inserted where you expect the item to appear. This *must* leave a valid *8th* item on TOS to be inserted into the map; thus the resultant array or map may not be convertible to standard JSON again! Additionally, the backtick *cannot* be used to evaluate a value for the *key* of a map, only for the key's *value*.

Strings in *8th*'s version of JSON may be spread over multiple lines, and the line-breaks are then implicit. So:

```
"key" : "A long
string" ,
```

is valid under *8th*'s version of JSON, but it doesn't adhere to the strict standard and may not be understood by other tools.

Code snippet. You can also put an anonymous word in the item you're declaring:

```
[ ( 123 321 n:+ ) ]
```

This is different from the backtick-expression because it is compiled code, and the word is held in the item you declared (array or map). You can then use `w:exec` to execute the word, or pass it on to some other word which requires it (`a:each` for example).

Alternatively, use the tick word `'` to find a word and insert it.

Complex numbers. A value such as `1+2i` will be interpreted as a complex number just as if it were entered in the REPL.

Bare key strings. Key strings in maps *without* enclosing double-quotes are permitted. In such a case, the key is understood to be from the first non-whitespace value until the colon (:) which separates the key from the value:

```
{
  key: "some value",
  complex: 1+2i
}
```

Note that when a map is converted to string (e.g. for printing or otherwise), keys are *always* enclosed in double-quotes.

@ expressions. In an array or map, a leading “@” symbol dereferences the variable appearing immediately after. For example:

```
123 var, x
{
    key: @x
}
```

The value of “key” will be 123 in this case. This is particularly useful when creating maps or arrays which have values which are constants. Note that the interpolation of the “@” occurs *at interpretation time* and so the value does *not* change when the variable value changes!

Buffers. A value beginning with 'X' is treated as a hex buffer, e.g.: `[ X1234 ]` is an **array** with a single value, a 2-byte buffer with the bytes 0x12 and 0x34.

Regex. A value beginning with '/' is parsed as a regular expression, e.g: `[ /abc/ ]` .

Here’s an example with all the modifications to standard JSON which *8th* knows:

```
[
  123,              \ a number followed by comment
  B123,             \ a bint
  F123,             \ a bfloat
  ' myword ,        \ a ticked word
  ( myword ) ,      \ an embedded anonymous word
  ` 200 300 n:+ `, \ an evaluated expression
  1+2i,             \ a complex number
  X1234,            \ a 2 byte buffer
  /abc/,            \ a regular expression
  @x                \ an “@” interpolation
]
{
  abc: "123"        \ “bare” key name
}
```

3.9 Working more effectively with JSON

Since *8th* uses JSON extensively, it is worthwhile discussing some "best-practices".

First of all, it is best to format your JSON so that it is legible. The parser doesn’t care if your JSON is illegible, but you will. So use indentation judiciously and keep opening and closing brackets at the same indentation level (or keep the ending bracket at the same indentation level as the item which caused the indentation). For example:

```
{"foo":"abc","bar":[1,2,3],"blah":{"blarg":1000}}
```

is legal JSON, but difficult to read. It would be better to format it like so:

```
{
  "foo" : "abc",
  "bar" : [1, 2, 3],
  "blah" : {
    "blarg" : 1000
  }
}
```

While JSON data structures can be created in your *8th* code, you may also keep them in separate files to be loaded at runtime. For example, the above JSON could be stored in a file **`data/foo.json`** (relative to your source files). In that case you could load the JSON using assets:

```
"data/foo.json" app:asset
```

Having loaded it in this way, the JSON text will be held in a buffer which you must then convert to its respective data structure using either **`json>`** or **`eval`** . The word **`json>`** converts only standard JSON. That is, it will not convert the *8th* additions to the JSON standard. If your file contains “enhanced JSON” then you *must* use **`json-8th>`** to convert it, or **`eval`** if you trust the source.

If you choose to store your JSON in some other location (for example: in some global location on disk), then you can use **`f:slurp`** to load the JSON file directly into a buffer; however, you must use the complete (relative or absolute) path to the file.

To convert a data structure into JSON text for storage or transmission, you can use the **`>json`** word, which can convert to enhanced JSON if you have included non-standard elements in the data structure (words, for instance).

Pro+ You can use the **`b:>mpack`** and **`b:mpack>`** words to convert *any 8th* data item for storage or transmission.

Ch. 4 The data stack

In common with all Forths, *8th* utilizes the “data stack” to pass data to receive results from words. Because of its importance, programmers must become familiar with how to use it properly. This chapter will describe in detail the words *8th* provides to manipulate the stack. When we say “the stack” we are referring to the default data stack.

By default, the data- and r- stacks allow 128K items. Memory for the stacks is allocated “on-demand” by the underlying OS, so the actual memory footprint will normally be whatever a page-size is on the OS or some multiple of that. If you need more than 128K items at a time, you must use the **`-k`** or **`-r`** command-line options to *8th*, or use the **`stack-size`** word before you start your main program.

If you really find that 128K items on the stack is too few, you are *almost certainly* abusing the stack and should really reconsider what you’re doing! Consider moving your data to an array, or to an auxiliary stack created using **`st:new`** .

4.1 Stack basics

Conceptually, the stack is similar to a stack of pancakes: the last pancake put on is the first one taken off. This is known as a LIFO data structure — last in, first out. The *8th* stack is exactly that. The act of putting something on the stack is called “pushing” and taking something off it is called “popping”. If you push items on the stack, the next action can pop items off and push new ones on. Of course, a word is not required to push or pop anything at all, but it almost always will.

```
ok\> 123 .s
1 n: 00007fb2e9844a00 1
123
```

In this example, the number **`123`** was pushed onto the top of the stack (TOS) simply by typing it in. The *8th* interpreter recognized a number, and allocated one and gave it the value 123, and then placed it on TOS for further use. The **`.s`** word prints the top ten items on the stack (words beginning with “.” are commonly used to indicate “print something”).

When writing words, it is *very* strongly recommended that you comment the word’s stack usage. This is called a “stack-effect diagram” (also called SED) , and is traditionally written like so: **`\ in1 in2 -- out1`**

In this case, the two parameters pushed on the stack were **`in1`** and **`in2`** , where **`in2`** is on TOS. The word is documented to consume those two and leave **`out1`** on TOS. When you’ve properly documented your words’ SEDs, you’ll be able to come back later and more easily understand what the word was intended to do. In *8th*’s documentation, a stack item in means that kind of item is read from the input-stream (either the *8th* source code in a file or from the keyboard or standard-input if input is redirected).

4.2 Common stack words

The most commonly used words for stack manipulation are:

word	SED	description
dup	x -- x x	duplicate TOS
drop	x y -- x	remove the item on TOS
swap	x y -- y x	exchange the item on TOS with the second item
over	x y -- x y x	duplicate the second item and put it on TOS
nip	x y -- y	remove the second item
tuck	x y -- y x y	duplicate TOS and put it in the third position
rot	x y z -- y z x	rotate the top three items, making the third TOS
-rot	x y z -- z x y	rotate the top three items, making the second TOS
2dup	x y -- x y x y	duplicate the top two items on the stack
2drop	x y --	drop the top two items on the stack
2over	x y a b -- x y a b x y	duplicate the third and fourth items on the stack
pick	n -- m	pick up the "n"th item on the stack

`0 pick` has the same effect as **`dup`** , and **`1 pick`** has the same effect as **`over`** . Try **`apropos stack`** to find other possibilities!

Note that “duplicate” *does not* mean the same thing as “clone”; rather, it makes the precise same item available in another position on the stack, increasing its **`refcount`** .

4.3 Using the stack

8th words take their parameters from the stack, and push any results back onto it. Thus, the SED mentioned above is important documentation for any word you write, as well as for the words built-in to *8th*.

Because a word’s parameters are *not named* as they are in most other languages, but simply reside on the stack, it is recommended to avoid using very large parameter lists. Generally speaking, if you have more than three or four parameters to a word, you should look at refactoring your code to break the parameter list into something more manageable.

In particular, use of the words **`pick`** or **`roll`** probably indicates your parameter list is too big, and you should give thought to reorganizing your code. Of course, **`pick`** and **`roll`** are provided because sometimes you *do* need long parameter lists.

4.4 Extra stacks

The words **`>r`** **`r>`** and **`r@`** provide access to an additional stack, which is intended to be used to store temporary values. If you are familiar with other Forths, you need to note that this is not the “return stack”. *8th does not* provide access to the actual return-stack, for security reasons.

Note: the word **`r@`** is *not* the same as the word **`r:@`** ! The latter accesses matches contained within a regex after **`r:match`** was invoked..

Use these words to stash a value away temporarily. The r-stack is as big as the data-stack, by default 128K items deep. But you will usually only need to use one or two items at a time on it. If you do find yourself needing to

access arbitrary locations on the r-stack, you can use **rpick** .

You can also create any number of other stacks of any size using **st:new** and the other stack words. You can push, pop, peek and pick just like you can with the data-stack and r-stack; however, you do not have access to the full complement of stack-words which operate on the default data-stack.

4.5 Controlling your stack with SED:

The word **SED:** was introduced to assist *8th*'s users with some of the more difficult issues facing new (and not so new) *8th* programmers. It does three things:

- Documents a word's SED
- Checks that the stack has the correct number of items on entry and exit from the word
- Checks that the parameters on the stack as well as the results returned match the SED documentation

How does it work? Simply use the **SED:** word instead of the backslash after your word's declaration, and adhere to the simple formatting rules:

- Separate the input and output sections with a double-hyphen
- Use the namespace identifier of the expected items instead of symbolic names
- Use an asterisk to indicate "don't care" values

For example:

```
: myword SED: a s - a
  \ rest of definition...
```

This indicates that **myword** expects an array and a string on entry, and leaves an array on exit (perhaps the same one, but that's not checked). It also indicates that the stack depth will be one less on exit than it was on entry.

By default, **SED:** will simply act as a comment, just like **** does. However, if you start *8th* with the command-line parameter **-g** it will have **SED:** checking enabled from the beginning.

You can also enable SED checking only for some portions of your code by putting **needs debug/sed** in your code file, and then invoking **true SED-CHECK** before words you wish to check, and **false SED-CHECK** afterwards to disable SED checking from that point on.

Active SED checking comes at a price: it's considerably slower than not having it activated. So it should only be used while developing new code, in order to help guarantee correctness. It can also be activated when you encounter bugs in your code, to help track down stack issues.

Ch. 5 Data types

8th has many built-in data types — some are almost self-explanatory, while others are less familiar. This chapter will acquaint you with the types and what they are used for in general. Further chapters will expand on specific data types. All data types used in *8th* are self-contained, and occupy only one cell on the stack (or in a container).

Some of *8th*'s data types come into being by declaration, and others can be created using some form of the word **`new`** . If a namespace has its own **`new`** , that will be indicated. All the types which are available in *8th* as of this version are listed in the file **`docs/words.pdf`** .

5.1 Scalars, containers, and utilities

In *8th*'s parlance, a scalar is an item which has a value used for calculation or display. For example, a number or a string. A container is an item which is used to hold (or "contain") other items. And there are still other items, such as files and regexes which are utilities.

The scalar types hold a value, which is almost always immutable. That means that when you modify such an item, you will get a new item of that type, with the value you expect.

Containers, on the other hand, are designed to be modified. That is because they are typically used to store other items to be used, and their utility is in their ability to be reused.

Utilities are neither scalars nor containers, are most often stateful, and are not themselves changed so much as they cause change in other parts of the system.

The following table lists all the data-types which are built-in to *8th*, as well as information about them. Containers are type **`C`** , scalars are type **`S`** , and utilities are type **`U`** . The "new" column indicates that there is a word new which creates the given type:

type	ns	CSU	new	description
Array	a	C	✓	A container allowing fast random access by numeric index
Boolean	T	S		true and false values
Buffer	b	S	✓	A chunk of memory, byte-wise accessible
Complex	c	S	✓	Complex number math
Crypto	cr	U		Cryptographic manipulation
Database	db	U		SQLite, MySQL, and ODBC databases
Date	d	S	✓	Dates and times
File	f	U		Disk files, pipes, etc.
Font	font	U	✓	Text fonts
GUI	g	U	✓	Graphics items
Global	G	S	✓	null
Graph	gr	C	✓	A container describing relationships between items

type	ns	CSU	new	description
Hardware	hw	U		Hardware access
Heap	h	C	✓	A container allowing serial access in sorted order
Image	img	U	✓	A graphical image (PNG, etc.)
Map	m	C	✓	A container allowing fast access by key
Matrix	mat	S	✓	A mathematical matrix (either numbers or complex)
Network	net	U		Sockets and internet access
Number	n	S		Math numbers
Pointer	ptr	U		A container which holds another item for passing through the FFI
Queue	q	C	✓	A container allowing serial FIFO access
Regex	r	U	✓	PCRE regular-expressions support
SQL	sql	U		SQL queries (used with the DB items)
Serial	sio	U		Hobby+ Access to the serial ports
Sound	snd	U	✓	Hobby+ Sound playing and recording
Stack	st	C	✓	A container allowing serial LIFO access
String	s	S	✓	A sequence of UTF-8 characters
Task	t	U		Same as a "thread"; allows multithreaded programming
Variable	v	C		A container holding exactly one item at a time
Word	w	U		The equivalent of a "function" or "routine" in other languages
XML	xml	S		An encapsulation of XML data

5.2 Reference-counting and pools

Each item in *8th* comes from a pool of similar items. As new items are needed, *8th* looks for ones which have been put on the released list in the pool. If there are any such items, they will be re-used. If there are no such items, a new item of the requested type will be created. Each task has its own set of pools.

Each item has a refcount, which determines whether or not it is available, and how many “holds” have been placed on it. The refcount is incremented every time another item holds a particular item (for example, if it is duplicated on the stack or is put in a var or other container). The refcount is decremented whenever a hold on the item is released (for example, if it is popped off the stack, or another item is put into the var, etc.).

When the reference count of an item is about to transition back to zero, *8th* performs whatever cleanup is necessary for that item (e.g. closing files, releasing memory) so that the soon-to-be-available item will be ready to be re-used and not leak memory or other resources. It is then put on the released list of its pool.

There may be usage-patterns which create a lot of released items. For example, if you create a large array of numbers for some reason, and then release it. Or if you push items to a task which doesn't create new items of that type. After that release, all those numbers end up on their pool's released list. If you are concerned about the excess memory usage, you can cause it to be reclaimed by invoking **`pool-clear`** or **`pool-clear-all`** in the task where the excess items were released.

5.3 Mutability

Scalars in *8th* are usually immutable. That means that, for example, if you have two numbers and add them together, the result will be a new number which is the sum of the two, rather than a modification of either of the original numbers.

Containers are mutable. That means that, for example, if you have an array and push some other item into it, the original array is changed (it now has one more item in it).

There are a few exceptions to these rules, and they are documented in the help. At present these words modify the originals:

`a:+ a:op= b:append b:clear b:fill` (if original has **`b:writeable`** set) **`b:move img:copy img:scroll s:append s:clear`**

Note that if you do not want a container to be modified, you should invoke **`const`** so that you get a *new container* with the same (but cloned) contents. Alternatively, use **`clone-shallow`** so you clone the container itself but not its contents.

5.4 A note about data conversion

Implicit data conversions are a major source of subtle bugs. Anyone who has used Perl or JavaScript for any length of time has probably been bitten by this. Therefore, *8th never*[1] converts data from one type to another, relying instead on the programmer to tell it when a conversion is necessary.

This is also why *8th* words often return the value **`null`** to indicate error conditions, because **`null`** is not usually a valid value, and it is unique in the system.

---

[1] or really, almost never. The sole exception is number-to-boolean, in which any non-zero number is considered the same as **`true`** .

Ch. 6 Flow control

8th has a number of words which control program flow; some are familiar to users of other languages and some are unique to *8th*.

6.1 First things first!

When *8th* runs your code from a file, it interprets and compiles or runs it as it goes along, as necessary. When it is done reading your code, it looks for a word **`app:main`** , and invokes it if it is found. Otherwise, it will wait for your input if your file did not invoke some process-starting word. If the file did end with a word invocation, that word is invoked as you would expect.

6.2 Conditionals

The standard Forth conditional words **`if... else... then`** are implemented in *8th*. Unlike Reva, there are no specialty versions of **`if`** . Note: the conditional words **`if else then`** may only be used in compile mode. *8th* will complain if you try to run them outside a word definition. At run-time, the word **`if`** looks at TOS, and if it evaluates **`true`** — that is if it is **`true`** or a non-zero number — it continues to the word following the **`if`** . Otherwise, it will skip to the enclosing **`then`** or the enclosed **`else`** . For example:

```
: test if
    "yes!" .
  else
    "no, sorry" .
  then ;
```

In this case, **`true test`** will print **`yes!`** while **`false test`** will **`print no, sorry`** . You can nest conditionals:

```
: test
  if
    some-condition if
      "yes!" .
    then
  else
    "no, sorry" .
  then ;
```

Such nesting may be as deep as you need; but if you find yourself writing code with more than a few nested **`if`** statements, you should seriously rethink your code's design!

Indentation or other text formatting is entirely optional! However, aligning your if/else/then will make your code easier to read, debug, and maintain.

Another set of conditional words is **`case`** , **`caseof`** , **`when`** , and **`when!`** . They operate differently (and more elegantly) than nested **`if… then`** or the **`switch`** statement in C and the like.

The `caseof` word accepts a container, either an array or a map, and a value which is either a number (if an array was given) or a string (if a map was given). It then looks up the value in the container; and if that item exists, it is either executed (if it is a word) or put on TOS. For example:

```
[ ' first , ' second , ' third ] 1 caseof
```

The `caseof` will take `1` as an index into the array, and find the word `second` and execute it. If the `array` had contained anything other than words, those items would be put on TOS. Note that the word `'` inside JSON *requires* whitespace after the name of the word it parses! `case` is similar, but the parameters are reversed from `caseof` , and it expects the values to be words to invoke; if the key is not present, nothing happens.

The `when` word takes an array consisting of pairs of words. It iterates over the array, evaluating the first word of each pair. As soon as it finds a word which returns `true` , it evaluates the second word of that pair and stops searching. The `when!` word is the same, except that it does not stop searching. In other words, it will iterate the entire array , executing the second pair of words whenever the first returns `true` .

```
[ ' test1 , ' action1 , ' test2 , ' action2 , ' test3 , ' action3 ]
when
```

Assuming `test2` is the first one which returns `true` , `when` will execute `action2` and stop.

Another `if...then` construct is `#if` , `#else` , `#then` — which are similar to the "C pre-processor" constructs, and allow you to compile code for a particular platform, for instance:

```
os 1 = #if
  \ Windows ...
#else
  \ Normal OSes ...
#then`
```

6.3 Repetition

There are several ways to repeat yourself in *8th*. The more familiar words are `repeat` , `again` , and `while` . Just like `if else then` , they may only be used inside a word definition. The phrase `repeat… again` is an infinite loop, which will repeatedly do whatever is between the `repeat` and `again` . The phrase `repeat… while` will conditionally repeat until TOS evaluates to `false` .

Unlike standard Forths, the *8th* version of `while` does not consume TOS. If you want a "consuming while" you can use `while!` instead.

```
: ra \ infinite loop
  repeat
    "Hi" . cr
  again ;
```

This will repeatedly print `Hi` endlessly, because when `again` is encountered it jumps back to the previous `repeat` . In order to leave you need to invoke `break` , `;;` or similar.

```
: rw   \ repeat while a condition is true
  100 \ give an initial value
  repeat
    dup . space
```

```
    n:1- \ the item on TOS is not removed...
  while drop ;
```

This repeats 100 times and prints the numbers in descending order, because TOS starts at 100 and the **while** peeks at TOS and returns to the **repeat** if TOS doesn't evaluate to **false** . So until TOS is 0 (which evaluates as false), it repeats. The **while** doesn't remove the item from TOS when it falls through, either; thus the **drop** is required to keep the stack balanced.

You can do a counted repetition in one of three ways: **times** , **loop** , or **loop-** . The **times** word takes a word to execute and a repetition count. It executes the word as many times as indicated:

```
: a "hi" . cr ;
' a 10 times
```

That will print **hi** on a line of its own, ten times. Note that we use the tick word (`) to take the value of the word a rather than invoking it immediately.

The **loop** and **loop-** words are identical, except for the direction of the looping. **loop** counts up while **loop-** counts down. Just like **times** , they take a word to execute; but they also take a low and high value, which are the beginning and ending values (inclusive) for which to execute the loop. For example:

```
: a . cr ;
' a 10 13 loop
```

This will print out **10 11 12 13** . **loop-** would print them in reverse order. The loop index is passed on TOS to the word being invoked, which must consume that number unless it is intended to leave the items on the stack. One way to get an array of sequential numbers is to use loop :

```
' noop 10 19 loop 10 a:close
```

That gives you an array of 10 sequential numbers, from 10 through 19. Be careful if you use this kind of trick, since you can overflow the stack — there are better ways to accomplish the same idea. For example:

```
' noop 10 19 a:generate
```

Besides those methods, you may also iterate a known number of times by invoking **a:each** on an array , or **m:each** on a map. Those will iterate the contents of the containers they were given, allowing you to do something for each item contained.

```
[1,20,300] ( . space drop ) a:each
```

That will print **1 20 300** . The **drop** is there because you aren't interested in the index value, and you do want to keep the stack clean. The array remains on the stack after **a:each** (and likewise, the map remains after **m:each**).

6.4 Breaking up is easy to do

The various repetition words can be stopped by using **break** , which signals *8th* to terminate the loop at the next repetition (not immediately, unlike C). You can test for "break" having been invoked in your own loops using **break?** . The **break** word will also terminate iterations in **s:eachline** , **f:eachline** , **a:each** and **m:each** as well as **db:exec-cb** and **repeat... again** .

Ch. 7 Words, the interpreter and compilation

Previously we said, “A word is the equivalent of a ‘function’ or ‘procedure’ in other languages. It is the smallest unit of execution”. So how do you create a new one?

The word `:` (ASCII 58, the colon character) tells *8th* it should create a new word whose name will be the following sequence of non-whitespace characters. For example:

```
: plus1
  1 n:+ ;
```

This creates a new word called `plus1` . The initial `:` is the word creator; it looks at the next bit of text and creates a word with that name. Knowing that white-space is treated as insignificant, scan for the next words: `1` and `n:+` are compiled into `plus1` . The final `;` (ASCII 59, the semi-colon character) tells *8th* to terminate the new word and resume interpretation mode. At this point, executing `2 plus1` will result in the number `3` on TOS. This is because the `2` pushed that number to TOS, and `plus1` was then invoked, which itself pushes `1` and then invokes `n:+` to perform a numeric addition. The result is, as expected, `3` on TOS. You are *encouraged* to use meaningful names for the words you create, especially since you can use any UTF-8 sequence whatsoever as long as it doesn’t contain white-space!

As mentioned in the chapter on syntax, *8th* interprets words one at a time and either executes them or compiles them into new words, depending on the state *8th* is in at the time.

7.1 Named versus anonymous words

Words created using `:` have a name: the whitespace-delimited run of characters after the colon. But sometimes you don’t need or want to name a word, you just want the action itself to be available. Such a word may be useful as a callback or in an iterator. You can create such *anonymous words* using the `(` and `)` words, like this: `( 1 n:+ )`

Note that an anonymous word still takes space in your application! It does not get cleaned-up and removed as you might be used to with JavaScript or some other languages. Moreover, `w:find` cannot find it because it has no name, so if you lose your reference to it, it is permanently “lost” even though the code still exists. If you don’t need to access it by name you will have saved a little bit of space (and will also have avoided using up useful names for other words).

7.2 Deferred words

A *deferred word* is one whose code can be modified at run-time. That is, the word is declared using `defer:` , and its action is assigned later using the word `w:is` . For example:

```
defer: some-word
: some-action ... ;
```

Then, later on: `' some-action w:is some-word` …

There are two main uses for deferred words. First, you may need to use the word before you can define it. That is, you are compiling word **`A`** which uses word **`B`** , but you cannot yet define word **`B`** . The deferred word facility allows you to make a “forward declaration” of **`B`** so that *8th* can compile **`A`** , and you can then fill in **`B`** ’s code later on.

A second reason to use a deferred word is that you may want the ability to change the effect a word has at run-time. For example, you may wish to redirect the normal output words to write to a string instead of to the screen. Since the words **`putc`** and **`puts`** are deferred, you can reassign them to do whatever you like. Of course, you will want to be careful if you do this!

The **`help`** facility will inform you if a word is deferred.

The assignment of an action to a deferred word can be undone, using **`w:undo`** . Each deferred word can have one level of undo. That is, invoking **`w:undo`** twice will remove all assigned actions, leaving the word inert (more specifically, **`G:noop`** is assigned to it).

7.3 Word attributes

A word may be *immediate*, which means that it is executed immediately when interpreted, even in compilation mode. For example, **`if`** is immediate because it compiles the *action* of the **`if`** into the word currently being compiled. Most words aren’t immediate, which means that when you put them between **`:`** and **`;`** they get compiled in. In interpret mode, all words are immediately executed.

If you want a non-immediate word to behave as if it were immediate for the moment, you can invoke **`i:`** before it, which means “treat the next word as if it were immediate”.

Likewise, **`p:`** postpones the action of the otherwise immediate word which follows it. In addition, you can flag the word you are creating as an immediate word itself, by terminating it with **`i;`** rather than **`;`** — this is the rough equivalent of using **`IMMEDIATE`** in ANS Forths.

In addition to **`p:`** and **`i:`** , there is also **`l:`** , which makes the next word “late-bound”. That means that the *name* of the word is looked up at run-time instead of interpret-time, so that if it isn’t known when *8th* first sees it, it will still compile. Use this with care, since it is much slower (since the lookup happens every time it is invoked), and can give seemingly random results (because the word found at run-time may not be the one you expect).

7.4 Recursion

The term recursion is used to mean when a word invokes itself. This is possible in two ways: first, by invoking the *name* of the word being defined. This, however, is deprecated, meaning that it may not be supported in future versions of *8th*. The second method is to invoke **`recurse`** , which will work even inside anonymous words.

Ch. 8 Numbers and math

The syntax section described how *8th* knows what a number is. *8th* gives you a great deal of flexibility in how you enter numbers, and it keeps you from having to worry about what kind of number you are working with. It tries to just “do the right thing”. numbers can be entered in most any numeric base you like, but *8th* provides shortcuts for certain bases.

A simple prefix character (as detailed in the syntax section) can tell *8th* you intend hexadecimal or octal numbers. For instance, assuming the current base is 10, the following all represent the same number , decimal 16:

```
16 #16 0x10 $10 &20 %10000
```

You can change the base currently being used at any time by invoking the word **`base`** . Any base may be specified, but only bases up to 36 are actually useful. Each task has its own idea of the current base.

Numbers automatically use a representation which is at *least as big* as required. So if you type in **`100`** , a native integer will be used to hold that value. If you type in **`100.1`** , a native floating-point number will be used instead. This is internal to *8th*, which converts between representations as needed; you will only rarely need to be aware of this. You can tell the internal representation used by invoking **`.s`** — any number which is a regular float with have a **`f`** indicator, a big-float will have **`F`** and a big-integer will have **`B`** .

See the section on “Numeric trade-offs” for important information about when you should specifically use the various kinds of number.

Knowing the internal representation becomes important to you if you are trying to compare numbers, and you think that the following should be true :

```
3.14159 100000 * 314159 =
```

This *would be* exactly equal if you were doing the math by hand, but *8th* is doing the math, and it has to rely on the computer’s processor. The floating-point number you end up with at the end of the multiplication is not precisely the same as the integer number, so the comparison may fail. This is just something you need to keep in mind if you use floating-point numbers in calculations. An appropriate solution is to check for equality within some tolerance using **`n:~=`** , for example.

8.1 Big numbers

Sometimes you need to calculate with values which are bigger than the native capacity of even 64-bit machines to handle (for reference, the upper limit on a 64-bit CPU is 9,223,372,036,854,775,807, which is big enough to handle the US GDP in cents, for the moment…).

8th has no problems handling really large integers. You could, for example, calculate **`30!`** (30 factorial, or 30 times 29 times 28…) which is an eye-watering 265,252,859,812,191,058,636,308,480,000,000. But you’ll notice that is bigger than the upper limit on integers.

When *8th* determines that a math calculation will exceed the native CPU’s capacity, it converts the numbers (internally) to a big number format, which is mostly unlimited precision (not for integers: they are limited to just

a ridiculously large number, about 2^{8400} on 64-bit systems, smaller on 32-bit; big-floating point numbers can be as large as your machine's memory can handle). This does come at a price: big integers are slower to work with than native ones, and they require more system resources to process. Big floats are even slower. However, we feel the convenience outweighs the penalties, since you the user don't have to be concerned with the internal format of the numbers you are using. Most of the time, anyway...

8th can also use big floating-point to represent high-precision floating point values. The default uses regular floating point unless the number exceeds the capacity of a regular float, or if you use the word **`bfloat`** to convert it to a big-float. That's also useful if you need to calculate beyond the capacity of the big-integer. Along those lines, you can force a number to use a different representation using one of these words:

word	description
int	convert to a native integer. This will also truncate a floating-point number
bint	convert to a big integer. Also truncates a FP number
float	convert to a native floating-point number
bfloat	convert to a big floating-point number

8.2 Complex numbers

The built-in complex numbers (namespace **`c`**) are native double only. That is, they do not utilize *8th*'s flexible "number" type. That makes them very fast, but it also means they do not automatically overflow to larger data types. If that is not important to you, you should use them (because they are 4 or more times faster than the library version).

If you *do* want high-precision complex math, then you need to use the library version of complex numbers in the **`math/complex`** library.

The built-in and library versions of complex numbers are *not interoperable*! So if you're going to use the library, do it before you use any complex words.

A complex can be created using **`c:new`** , being passed either an array with two numbers in it, or two numbers on the stack. For example, **`1 2 c:new`** will create the complex number **`1+2i`** . It is also possible to enter a complex number in the interpreter (and also in JSON), e.g. **`1+2i`** will create the same complex as **`1 2 c:new`** . There must be no spaces in the text representing the value, and it must end with a lowercase "i", in order to be properly recognized.

8.3 Matrices

Matrix math is supported internally, using native double or complex (two doubles) data types, just like the complex data type.

A matrix is a fixed-size numeric container, having two or more dimensions. You create one like this:

```
[1,2,3,4] [2,2] mat:new
```

This snippet creates a new 2x2 matrix, and sets its initial values to those in the array . So row 0 consists of the numbers **`1`** and **`2`** , and row 1 consists of **`3`** and **`4`** . Column 0 consists of **`1`** and **`3`** .

The type of a matrix is determined by the first element in the initializing data. So if you want it to contain complex numbers, you need to make sure the first element in the initializing array is a complex . All the members of the created matrix are of the same underlying data type, which may not be changed once the matrix is created.

The dimensions given to **`mat:new`** are in the order of columns, then rows (and analogously for higher dimensions). The words provided have special-cases for two-dimensional matrices, since those are the most commonly used.

A note of caution: *8th* does not yet implement “sparse matrices”, which means that you can quickly run out of memory if you use high dimensions with even modest matrix sizes. For example, a matrix of just 10 dimensions with only 4 entries per dimension will require over 40 megabytes on a 64-bit system! So think over whether you really want a super-high-dimension matrix or not, or create your own sparse matrix data type.

8.4 Manipulating numbers

8th has a number of words which can be used for manipulating numbers. Besides the usual arithmetic ones, there are some specialties such as the scaling word **`*/`** which performs a multiply-and-divide at once, providing more accuracy than separate multiply, then divide. Or **`/mod`** which returns the quotient and remainder at the same time. A full list appears in **`docs/words.pdf`** .

8.5 Limitations

As mentioned, big integers do actually have an upper limit. That limit is reached by calculating **`985!`** (985 factorial; that is, 985 times 984 times 983…) which is a *really* big number (the limit’s **`1004!`** on 64-bit systems). The reason that big integers are limited in this way is because it makes them relatively efficient to use, since *8th* doesn’t have to continually check to see if it needs to resize the number and allocate more memory. It also caps the memory usage for a big integer. Big floating-point numbers are not limited in this way, but they are therefore not nearly as fast or efficient to use, being several times slower than big integers.

The decision to use very large, though limited, big integers is a trade-off between runtime efficiency and user-convenience. We think that in most cases, you will not find the upper limit of the big integer too constraining; if you do, you can use big floats instead or perhaps use the **`math/gmp`** library, or roll your own big-math words.

In normal usage, *8th* will promote big integers to big floats when there is a danger of exceeding the limits of big integers. However, in order for the accuracy of calculations to keep pace with the enlarging numbers, you must invoke **`n#`** with the number of digits of accuracy you require. Otherwise, only 32 digits will be retained in big float mode. This is in order to save memory at runtime.

8.6 Numeric trade-offs

As mentioned above, *8th* tries to do the right thing when it comes to numbers, so you can generally ignore its internal numeric representation. However, it may be the case that you get unexpected results from your calculations. For example:

```
0.1 0.2 + 0.3 = .
```

will print **`false`** when you would naively expect **`true`** . That is because *8th* uses the native double floating-point

representation of these numbers, and powers of 10 are not precisely representable.

So too:

```
25 ## 0.1 0.2 + . cr
```

prints **`0.30000000000000000444000000`** which, indeed, is not exactly **`0.3`** . What can you do?

If you find that you are running up against inaccuracies due to native floating-point being insufficiently precise (which is a real and common issue), you can use big float math explicitly. Either invoke **`bfloat`** on a non-big-float number to convert it into one, or use the leading-F syntax as mentioned above:

```
F0.1 F0.2 + F0.3 = . cr
```

This returns **`true`** , because the big-float is as precise as you've allowed it to be (the default is 32, set using **`n#`**).

Note that math between dissimilar types will always return a value of the bigger type (or perhaps an even bigger type if the result would still overflow). In this context, an "int" (backed by a 64-bit integer) is smaller than a "float" (backed by a 64-bit IEEE double), both are smaller than a "bint" (a big-integer representing whole numbers up to a certain limit) which is smaller than a "float", which can represent pretty much any rational value up to the limits of memory.

Ch. 9 Text and strings

8th has many words which ease working with text. Unlike most Forths and unlike C, a string in *8th* is:

- dynamic: automatically allocates space for added text
- single: a string contains its own length; you do not need to pass the length separately
- UTF-8 encoded: may contain any character from any of the spoken languages on Earth
- C-syntax: if you know C or C++ or Java, etc., you already know how to declare a string
- NUL-terminated: as in C, a string ends with a terminating `ASCII NUL` . But since the length is also maintained, *8th*'s strings are more efficient to work with than C's version

9.1 What is a string?

At the simplest conceptual level, a string is a sequence of characters. As mentioned above, *any* Unicode character may be part of a string . To create a new string , you simply declare it as you would in C: `"cat\n"` .

Typing that sequence will put the four characters `c` , `a` , `t` , and `ASCII 10` into the newly-formed string . The syntax chapter has much more detail on the actual characters allowed in a string (unlike C, the NUL character *may appear inside a string*).

9.2 Manipulating a string

String words operate on sequences of characters rather than sequences of bytes. This is an important distinction, because a string contains UTF-8 encoded characters, each of which may require multiple bytes to express. If one were to modify an arbitrary byte in a string, an invalid UTF-8 character sequence might result.

Unlike C strings, an *8th* string is immutable. If you add to it or remove from it and modify individual characters, a new string is created. To concatenate two strings you use the `s:+` word:

```
"cats and" " dogs" s:+
```

This results in the string `cats and dogs` . Remove characters from the string using `s:-` :

```
"cats" 1 2 s:-
```

This leaves you with the string `cs` . There are quite a few string manipulation words. A few examples:

word	SED	description
s:/	s x – a	Split the string on "x"
s:=	s1 s2 - f	Compares two strings for textual equality (see also `s:cmp`)
s:lc	s - s1	Convert string to all lowercase (`s:uc` converts to uppercase)

Splitting the string with `s:/` is quite flexible. The "x" could be a number, to split at a location in the string, or a string or regex to split on matches in the string.

9.3 Multilingual support (I18N and L10N)

8th supports easy localization of text using the **s:lang** and **s:intl** words. The manner in which they are used is straightforward.

First you need to create an asset directory called **lang** , and you further need to create a separate asset for each language you wish to support. For example, if you want to have English and Spanish in your application, you would (at least) create an asset **lang/es** .

That language asset contains the text to use for long and short day-names and month-names, as well as a simple JSON map whose keys are the original (e.g. "default") text, and whose values are the translated text. For example:

```
[ "Ene", "Feb"… ] short-months !
…
{
  "hi" : "¡Hola, mundo!",
  "bye" : "Hasta la vista...",
  …
}
```

To utilize this asset, two steps are required. First, you must tell *8th* to use the Spanish language asset: **"es" s:lang** . Second, you need to tell *8th* that you want to translate a string: **"hi" s:intl** . This latter phrase will produce the string **¡Hola, mundo!** .

You can support as many languages as you wish, and as many strings as you like. We hope you'll agree that the clear JSON syntax makes the translator's work easier!

The arrays of strings for the localized names of weekdays and months must be loaded by the asset, and the vars to load into are called **short-days** , **short-months** , **long-days** , and **long-months.** If you switch back to English, you should reset those to the same named item suffixed by **-en** . For example, **short-days-en** .

9.4 Search, replace and parameterized substitutions

8th lets you search and replace in strings in several ways, and you are encouraged to look in the comprehensive word-list for all the details. However, a few notes are in order:

First, searching and replacing can be done with either a string or a regex . The regular-expression syntax is that of PCRE, and sub-matches are supported. That is to say, one may search using a regular-expression such as **/(c\S+) and (d\S+/)** against the string **cat and dog** using **s:search** , and it will say it found the expression at position 0 (the start of the text). Of course one can also search for a literal string .

Using the same regex and string but with **r:match** instead, one gets the result **3** , meaning there are three matches. Match 0 is the entire matched expression, and other matches correspond to capturing parentheses. In this example, giving the regex and saying **1 r:@** will give the result **cat** , just like in Perl or other similar tools.

Substitution is done using **s:replace** (to replace just once) and **s:replace!** (to replace all occurrences). The pattern may be either a string or a regex, but the replacement must be a string.

8th also has something called templated substitution, using the word **s:tsub** . This is a very powerful substitution mechanism which allows you to replace parameters in the template by position or by name. For example:

```
"Hi there, %name%!" { "name" : "Mary" } s:tsub
```

This will produce the string **`Hi there, Mary`** . While this specific example could also trivially be accomplished using **`s:+`** , templated substitution can do much more. Localized sentences often have different word order; the **`s:tsub`** approach to building localized strings is flexible enough to handle that and many other similar problems.

In addition, *8th* can do "printf" style substitutions. For example:

```
123 "Joe" "%s owes me $%d" s:strfmt
```

will result in the string **`Joe owes me $123`** . You can either put the substitutions on the stack or in an array, and there are quite a few formatting options. See the sample **`strings/strfmt.8th`** for more details.

9.5 Strings vs. Buffers

8th treats strings and buffers similarly in many respects. In particular, it is possible to ask for the "n'th" character of a string, or "n'th" byte of a buffer. Though the syntax is similar, there are big differences between the two.

As mentioned above, strings in *8th* are encoded using UTF-8, which is a variable-length encoding designed to allow every Unicode character to be represented, while requiring only one byte to encode Latin-1 (e.g. most European language characters). This has important ramifications.

First is that accessing an arbitrary character of a string requires traversing the entire string up to that character. It is not possible to know where a particular character will begin until it has been encountered. Thus, due to the use of UTF-8, the words **`s:@`** and **`s:!`** are relatively slow — especially so as the length of the string grows. An optimization is in place to make random-character access fast if and only if all characters in the string have the same length in terms of their UTF-8 encoding.

Secondly, arbitrary data should not be stuck into a string. Since it will be interpreted as UTF-8, unpleasant side-effects will probably occur.

Buffers do not suffer from these issues, since a buffer is nothing more than a container for a specific number of bytes. Accessing any particular byte is extremely fast. However, a buffer makes no assumptions as to its contents' meaning, so one may not assume the "n'th" byte is the "n'th" character (unless it can be assumed a Latin-1 or similar encoding was used on the data).

Besides all the above, buffers are fixed in size, while strings are dynamic. Both types accept the **`set-wipe`** word, which tells *8th* that the data in this item is sensitive and should be zeroed out before releasing it back to its pool. This is important when hardening an application for security reasons. It is the only way to change the actual contents of a specific string.

9.6 Markdown

Parsing Markdown ("MD") formatted text is useful in many applications, and *8th* now includes a capable MD parser implemented via two words: **`xml:md-init`** and **`xml:md-parse`** . There are two libraries available to make MD processing easier: **`md/2html`** and **`md/2console`** .

9.6.1 Initializing an MD parser

The word **`xml:md-init`** receives a map with the following keys:

name	kind	description
enter_block	w	Invoked when the parser enters a block, for example a paragraph
enter_span	w	Invoked when the parser enters an inline span, for example “emphasized text”
leave_block	w	Invoked when the parser leaves a block
leave_span	w	Invoked when the parser leaves an inline span
opts	a	Array of strings which are parser options; see opts below
text	w	Invoked when the parser has a run of text to output

The parser options are all strings, and may be any of:

option	description
atx	Do not require space in ATX headers (###header)
collapse	collapse non-trivial whitespace into single ' '
email	Recognize e-mails as autolinks even without '<', '>' and 'mailto:'
github	Same as setting links, tables, strike, and task
latex	Enable $ and $$ containing LaTeX equations
links	Same as setting email, url, and www
nohtml	Same as setting nohtmlspans and nohtmlblocks
nohtmlblocks	Disable raw HTML blocks
nohtmlspans	Disable raw HTML (inline)
noindent	Disable indented code blocks. (Only fenced code works)
strike	Enable strikethrough extension
tables	Enable tables extension
task	Enable task list extension
url	Recognize URLs as autolinks even without '<', '>'
wiki	Enable wiki links extension
www	Enable WWW autolinks (even without any scheme prefix, if they begin with 'www'

9.6.2 Using an MD parser

The word **`xml:md-parse`** takes the MD parser created as above, a user-specific data-item (whatever you think is useful, or **`null`**), and a string containing MD to parse. The parser scans the text provided and invokes your callback words when appropriate. The SED for your words is always **`m -- f`** where the map received always contains a key **`tag`** which determines what is being parsed, **`user`** which is the data you gave the parser (passed through without modification), and other keys depending on the value of the tag and the kind of callback:

Text callback:

key	kind	description
`text`	s	the actual text

key	kind	description
type	s	What kind of text it is. One of the strings:
br		a line-break
code		text in a code block, or inlined code
entity		an "entity" like ** ** or **Ӓ** or **ካ**
html		raw HTML
math		inside a LaTeX equation
norm		normal text
nul		a NULL character (s/b replaced with character **\uFFFD**)
sbr		a soft line-break

Span callback:

tag	kind	description
CODE	s	**<code>**
DEL	s	**<del>**
EM	s	**<em>**
LATEXDISPLAY	s	LaTeX display math
LATEX	s	LaTeX math
STRONG	s	**<strong>**
WIKI	s	Wiki links (only if the "wiki" option was given to the parser)
A	s	Link. Contains additional keys as follows:
	href	The URL for the link
	title	Title text for the link
IMG	s	Image link. Contains additional keys as follows:
	src	The source URL for the image
	title	Title text for the image

Block callback:

tag	kind	description
DOC	s	<body>
HR	s	<hr>
HTML	s	block of raw HTML
P	s	<p>
QUOTE	s	<blockquote>
TABLE	s	<table>
TBODY	s	<tbody>
THEAD	s	<thead>
TR	s	<tr>
CODE	s	<code>, additional keys:

tag	kind	description
	fence	A number indicating "fence" character, or 0 if indented block
	info	Information string
	lang	Contents of "lang="
`H`	s	<h...>, additional keys:
	level	Number indicating header level, 1-6
`LI`	s	, additional keys:
	mark	Number if 'task': one of 'x', 'X', or ' '
	ofs	Number if 'task': offset of mark between '[' and ']'
	task	Number: 1 if "task list"
`OL`	s	, additional keys:
	mark	Number: character delimiter, e.g. '.' or ')'
	start	Number: start index of ordered list
	tight	Number: non-zero if "tight" list, 0 if loose
`TD` or `TH`	s	<td> or <th>, additional keys:
	align	String, one of: 'left', 'right', or 'center'
`UL`	s	, additional keys:
	mark	Number: bullet character, e.g. '-', '+', '*'
	tight	Number: non-zero if "tight" list, 0 if loose

9.7 Character encoding

As mentioned before, *8th* encodes strings using UTF-8. However, the real world contains text in a wide variety of encodings, and you may need to read or write in an encoding other than UTF-8. *8th* provides the word **`b:conv`** to perform these conversions. It is in the buffer namespace because strings are always UTF-8 encoded.

Linux and Raspberry Pi users take note: this functionality is only available if you have installed the **`libiconv`** library. You must download, build and install it before **`b:conv`** will work! If you wish to distribute an application for Linux or RPI, note that this is a *runtime requirement*.

Convert a buffer from one named encoding to another using **`b:conv`** , which will return a buffer with the converted text, or **`null`** followed by a numeric error code. The error code will be one of:

1. The **`libiconv`** library is not installed (Linux or RPI only)
2. The character encoding given was not recognized
3. The text could not be converted between the given character encodings

The encodings available differ between operating systems, which is a bit of a complication for you the programmer. The complete list of encodings by platform is in the **`docs/encodings.txt`** file.

Ch. 10 Date, time, and calendars

10.1 Dates and times

Dates in 8th contain date, time, and timezone information, with millisecond resolution. The word **`d:new`** generates a new date initialized with the current date and time as of its invocation.

If you want to initialize a date with a specific date and time, you can invoke **`d:parse`** on a string containing one of the ISO-8601 formats . For example: **`"2019-10-12" d:parse`**

It is also possible to initialize a date by manually constructing one using **`d:join`** .

When entering dates in the console, it is possible to type e.g. **`2019-10-12`** and the interpreter will parse that as a date, assuming that there is not already a word with that name.

For the purpose of timing short durations, the word **`d:msec`** provides the current time in milliseconds since 01 Jan 1970, and **`d:ticks`** provides a high-accuracy timer count whose exact meaning is OS-dependent.

10.2 Calendar manipulations

8th's date libraries contain various calendar manipulations. The Gregorian, Hebrew, and Islamic calendars are specifically supported. So too are generic date manipulations.

Pro+ “daylight savings time” query is also available.

Ch. 11 Containers

8th has several built-in container namespaces. “Containers” are items which contain other items. All the containers in *8th* can contain any kind of item:

kind	description
var	a single-item at-a-time container
array	fast random access by number index
map	fast random access by string key
stack	LIFO serial access via push and pop
queue	FIFO serial access via push and pop
heap	sorted serial access via push and pop
graph	data (nodes) organized by relation (edge: weight and direction)

11.1 Variables

A var (“variable” in other languages) is a single-item container. That is to say, it can contain only one thing at a time, though that “thing” can be *any 8th* data type. You declare a var using either **`var`** or **`var,`** — the difference being that the first initializes the var to the number 0, while the latter initializes the var to whatever was on TOS at the time of the declaration. For example:

```
"A string" var, astr
```

This creates a new var named **`astr`** , and initializes its contents to the string **`A string`** . To use the value inside the var , you must use the word **`@`** :

```
astr @ . cr
```

That will print the value currently held in **`astr`** . Change the value it holds using **`!`** :

```
1024 astr !
```

After this, **`astr`** holds the number 1024. So while the name **`astr`** is a poor choice, you hopefully get the idea.

Please note that the name which you gave the var *does not refer to the contents* of the var! So the following code will throw an exception complaining, **`Expected Array but got Variable`** :

```
[] var, an-array
an-array 100 a:push
```

What you probably intended in this case was:

```
[] var, an-array
an-array @ 100 a:push
```

The first example throws an exception because you are using an array accessor but the item called **`an-array`** is

actually a var! Remember to always dereference the var before using its contents.

The word **constant** provides a way to have a var which doesn't change:

```
123 constant OneTwoThree
OneTwoThree . cr
```

In this case, unlike the var, the value in the constant is put on TOS by invoking the constant's name. Since there's no option to change the value held by the constant, there is no reason to require **@** .

11.2 Arrays

An array is a container which can hold any number, kind, and mix of items (limited only by the memory available), and whose items are accessed by numeric index (starting at 0 for the first spot). Declare an array using JSON, or by invoking **ns:a new** or **a:new** .

```
[1,2,3] var, a1
a:new var, a2
```

After this, **a1** contains an array with three elements, all numbers, while **a2** contains an array with no elements.

Array elements are accessed with **a:@** and **a:!** , as well as with a number of other more specialized words. For example one may easily iterate over an array:

```
[ "one","two","three" ]
( "Item " . swap . " is " . . cr )
a:each
```

This will print **Item 0 is one** , etc. for each item in the array.

8th's arrays are not "sparse", so if you put an item at index 0 and another at index 10,000, *8th* will comply — but the resultant array will have 9,999 empty spots in it and will take up a lot more memory than you might have expected.

11.3 Maps

A map is a container which can hold any number, kind, and mix of items (again, subject to available memory), and whose items are accessed by a key which is usually a string . You declare an map using JSON or using **ns:m new** or **m:new**

```
{ "one" : 1, "two" : 2 } var, m1
ns:m new var, m2
```

After this, **m1** will contain a map which has two elements, and **m2** will contain an empty map. A map's key may also be any data type, though to take advantage of that you must use the accessor words rather than JSON syntax.

When using a data item as a key in a map, you must ensure that the key-item remains intact *for the lifetime of the map*, because when you reference the key (using e.g. **m:keys** or **m:each** or **m:@** or **m:!**) *8th* will assume the reference is still valid. It does not keep track of that internally in the map, for efficiency's sake; so *caveat programmer*!

When using a number as a key with `m:!` , it is converted to a string as if `>s` were invoked on it. So the above restriction does not pertain. But when using a number with `m:@` , *8th* returns the n'th item in the map according to the order in the map's internal structure. This may be useful in some cases, and it's the prior behavior, so it's not been changed, though it is deprecated. If you want to extract the item with the key given by the number 2, use the string `2` .

In analogy to arrays, maps are accessed using `m:@` and `m:!` , as well as with more specialized words (such as `m:each`).

11.4 Stacks, Queues and Heaps

You're already familiar with "the stack". The stack type is simply an independent stack which can be used in much the same way as the regular data-stack. By default, a stack will throw an exception if you push too much onto it or pop from it when it's empty. You can change that behavior by using the `st:throwing` word to disable that behavior.

A queue is more or less the same as a `stack` , except that it forces access to the items placed in it to be first in, first out. It also has most of the same words as the stack. You can make a queue behave like a circular buffer using `q:overwrite` .

Both stacks and queues are of fixed size, which is established when they are created.

A heap is different in that it does not have a fixed size, and in that access depends on the items pushed into it. You provide a word to `h:new` which is then used to determine the order of the items pushed. They are then accessed in order based on the word you used to initialize the heap.

11.5 Graphs

A graph is unlike the other container types in that it specifies a relationship between each pair of contained items. The items are called "nodes" of the graph, and the relationships are called "edges". An edge may connect any node to any other, and it may also have a "weight" and a "direction". The default edge has neither weight nor direction.

Graphs are created by passing a map to the `gr:new` word , and the keys permitted in the map are:

key	description
nodes	if present, an array of items which are the initial set of nodes in the graph
edges	if present, an array with an entry for each node, which is itself an array of edges. Each edge is an array of numbers
weight	if present, a word which accepts two nodes and returns the "weight" the edge between them should have
directed	if present, then if true indicates that the graph is a directed one

An empty map or the value `null` passed to `gr:new` will result in an unweighted, undirected, and initially empty graph.

Once you've populated the graph with data, you can traverse it using `gr:traverse` , which allows either depth-first or width-first traversal. After the traversal is complete, you can iterate the nodes with `gr:each` and determine which nodes have not been visited by testing with `G:mark?` .

The `G:>s` word will convert a graph into a map which can be used to recreate the graph.

11.6 Listeners

You can register one or more listeners on some kinds of containers: var, array, and map. A listener is simply a word which is invoked when the container it is listening to has been changed.

A listener is added to a container by invoking **`container listenercode +listener`** . By default, a listener is invoked immediately, on the same task as the code which modified the container. If instead you invoke **`+listener`** like: **`container listenercode true +listener`** then the listener is invoked asynchronously, on the main task.

Ch. 12 Files, databases, sockets, etc.

There are plenty of words to help you perform all sorts of I/O. First we'll mention the simplest: **.** and **putc** which let you write to the console (or to a string or other item if you've reassigned the low-level words).

12.1 Files

Regular files are handled by the various words in the **f:** namespace. These include the usual **f:open** , **f:create** and **f:close** as you might expect. They also have the ability to easily write an entire string or buffer to the file using **f:write** . If you want to write only a specific number of bytes you can do that with **f:writen** . Similarly, you can read directly into a string or buffer (though you need to specify how much to read).

Two special words are very useful for file processing: **f:slurp** and **f:eachline** . The first "slurps" an entire file into a buffer which is actually memory-mapped to the underlying file, allowing you to process it quickly in memory. The second lets you process a text file line-by-line. For example:

```
"data-file" f:open
' process-line f:eachline
f:close
```

This snippet opens (the existing) file **data-file** and passes each of its lines one-by-one to the word **process-line** (which you've defined somewhere else). It also shows the concatenative nature of *8th*, where the output of one word is passed to the next in line.

If you want, you can write the inverse of **f:slurp** , which takes a string or buffer and a file-name, and spits the item into a file:

```
: f:spit \ item fname --
  f:create
  swap f:write drop
  f:close ;
```

A special set of file words deals with ZIP files. You can create them, iterate their directories and extract their contents.

12.2 Databases

All versions of *8th* include a built-in version of SQLite . You can create and use high-speed local encrypted or non-encrypted SQL databases using the **db:** namespace words.

Pro+ Additionally, Pro+ versions also support remote (or local) MySQL databases, provided you have installed the "MySQL C Connector". They also support ODBC connected databases, so almost any database is available to *8th*. ODBC support is native on Windows; but Linux, Raspberry Pi and macOS platforms need to have **unixodbc** or **iODBC** installed. Mobile platforms do not currently have ODBC support.

```
"my-database" db:open
```

```
' process-one-row
"SELECT * FROM mytable WHERE id=1"
db:exec-cb db:close
```

Note the similarity of operation between the database and file words. You can do parameterized queries as well as simple ones.

To open a MySQL database, you provide `db:open` with a map which describes the specific settings needed. For example:

```
{
  "kind" : "mysql",
  "host" : "db4free.net",
  "db"   : "eighthdev",
  "user" : "user8th",
  "pwd"  : "password"
} db:open
```

To open an ODBC database, you follow the same steps as for the MySQL example above, but change the `kind` from `mysql` to `odbc` . In addition, you need to add a `dsn` key which is a string containing the DSN connection string for your particular ODBC database connection.

8th lets you easily create completely encrypted SQLite databases using the `db:key` and `db:rekey` words. The first is used immediately after `db:open` , and sets the encryption key to use thereafter. It will not work if the database was not already encrypted with that same key.

The second is used to either change the encryption key of the database, or to assign one when the database had not previously been encrypted.

The encryption key must be either the result of `cr:randkey` (in which case you will need to save it somewhere safe), or the result of `cr:genkey` . The actual encryption of the database uses AES-256-GCM. The *entire* database is encrypted, including all metadata, making it impossible for an attacker to glean any information from it or modify it. Needless to say, the encryption key is not stored in the database, and if you lose or cannot recreate it, you will not be able to access the data!

Encrypted database support is *solely* for local SQLite databases. There is currently no support for encrypted MySQL or ODBC databases in *8th*. Of course you can encrypted the individual fields using the `cr:` words, prior to storing them in a non-encrypted database; however, SQL queries on those fields will not be useful.

12.2.1 User-defined functions in SQLite

SQLite doesn’t implement functions in its dialect of SQL, but it provides something far more powerful: a hook to allow SQL statements to call your own functions.

8th offers the `db:add-func` word, which takes a map with various parameters, and lets you add a new function in a specific (open) SQLite database. The parameters are documented in the help.

Here is some more information to help you work correctly with user-defined SQLite functions. There are three kinds of user-defined functions you can add: scalar, which returns a value for each row based on that row’s values; aggregate, which operates on an entire selection of rows and returns a value based on the whole set; and window, which may act as either a scalar or aggregate, on a window within a selection. See the SQLite documentation for more details.

To add a new SQLite function, you invoke **`db:add-func`** , passing it a map of options and the database to which you wish to add the function. The map's keys may be:

key	description
final	required for aggregate and window-aggregate functions, invoked to get the final value of the function
func	the *8th* word implementing a scalar function (don't use for other function types)
inverse	required for window functions: invoked to perform the inverse of "step"
name	required:the name for the new SQL function
nparams	the number of parameters for the SQL function. Defaults to -1, which means "any number of parameters"
step	required for aggregate and window-aggregate functions, invoked for each row to process
value	required for window functions: invoked to get the current value of the function
window	if true , it's a "window function"

The **`func`** word receives an array which contains the parameters passed to the SQL function. It should process those parameters as appropriate, and leave a value on TOS which will be the result of the SQL function. SQLite understands a very limited number of types, so what you leave on TOS is interpreted as follows:

string	SQLITE TEXT
buffer	SQLITE BLOB
null	SQLITE NULL
number	If floating-point, SQLITE FLOAT. If integer, SQLITE INTEGER. "Big" numbers are not understood, so if you have a big number to return, convert it first to a string

All other types are interpreted as an error, which causes the SQL statement to fail with an error code. So don't try to pass a map unless you first convert it to JSON text.

The step and inverse words are similar to **`func`** , but they are accumulators. So underneath the parameters array they get the current accumulated value, and they must leave a new accumulated value on TOS.

The final and value words receive the accumulated value and return the final value, or the current value, based on the accumulated value (usually it would be the same as the accumulated value).

All the *8th* words are invoked in the same task as the SQL statement, e.g. the **`db:exec`** or similar. The stack is cleaned up automatically after the invocation of any of the words, so *do not* rely on leaving a trail of bread-crumbs on the stack.

12.3 Sockets and network I/O

Sockets are fundamentally the same as files, but the words which deal with sockets have been placed in the **`net`** namespace. This helps clarify for example whether **`bind`** is the database or the network version. If you are familiar with the typical "Unix sockets" functionality, the *8th* implementation is mostly just a thin layer over that, so it should be familiar.

In addition to the low-level socket words, there are some high-level ones to help make your use of internet APIs easier. **`net:get`** and **`net:post`** perform HTTP GET and POST calls, respectively. They may be used as building-blocks for other operations, for example, the **`libs/net`** utility words for JSON-RPC or SOAP. Besides **`get`** and

post , there are also **delete** , **put** , and **head** to help you interact with RESTful services. These higher-level words are not built-in, but rather reside in the various "net" libraries, e.g. **net/http** for HTTP GET and POST.

All the words mentioned in the previous paragraph accept a map with information for the call. Note that the **libs/net** words may require additional fields. The fields which may be used by these **net** words are:

key	description	default
bufsize	Set the size of the buffer used to read	65536 bytes
cookies	An array of strings which are cookies to be sent to the server	
data	The data payload (for post/put; required for them)	
expire	Check that the certificate has not expired	**true**
getheaders	If **true** , retrieve the headers from the call as a map	**false**
headers	A map containing key,value pairs of additional headers	
overwrite	If **true** , put will write over the current item	**false**
pincert	A buffer with the PEM encoded cert the server must present in order to validate the connection	
proxy-port	If present, this is a port number for the HTTP proxy-server	
proxy-server	This is a host name which is an HTTP proxy	
readcb	Invoked for each chunk of data. Gets the net item as well as the number of bytes received so far	
redirs	Maximum redirects to process. 0 means you need to manually handle redirections	5
rootcert	A buffer containing the PEM encoded root certificate which the server cert chain must match	
sni	Do SNI request	**true**
sniname	A string which gives the hostname to use for SNI. Implies a "sni" **true**	
tlsdowngrade	If **true** and **tlsver** is 13, will downgrade the TLS to 1.2 if the 1.3 connection fails	**true**
tlsver	A number indicating the TLS version to use. Acceptable values are 10,11,12, and 13	12
to	If present, a number of seconds before the connection will time-out	15
url	The URL of the service to connect to (required)	
vchain	whether or not to check the certificate's chain of validity	**false**
verify	If **false** , do not verify the SSL connection	**true**
vhost	Verify the hostname of the certificate in the SSL connection	**true**
vhostname	A string which sets the hostname for "vhost" to check. Implies "vhost" **true**	

The keys **checkexpire** , **pincert** , **rootcert** , **sni** , **sniname** , **vchain** , **verify** , **vhost** and **vhostname** are only applicable to TLS (e.g. "https") connections.

The **get** , **delete** , and **head** words may take a string instead of a map , due to their simpler nature. All the words are executed synchronously, and return a **true** and perhaps data on success, or **false** and an error code on failure. If an error code is returned it will be either an HTTP code or a negative number, and you should check **t:err?** for more information.

Because of the *synchronous* nature of the calls and because network I/O can take a long time, you should run the query in separate task, and use the synchronization primitives to handle results.

The **net** words are proxy-aware, but you need to tell them what proxy to use. Do this using **net:proxy!** , which takes a map with proxy parameters **proxy-server** and **proxy-port** .

12.3.1 Socket options

When creating a socket using **net:socket** , one may use a map of various options. The keys permitted are:

key	kind	description	default
domain	number	net:INET4 or net:INET6	INET4
proto	number	net:PROTO_TCP or net:PROTO_UDP	TCP
sockopts	map	values to pass to net:setsockopt	
type	number	net:DGRAM or net:STREAM	STREAM

The keys currently understood for the **sockopts** map are:

key	kind	description
broadcast	number	0 or 1 (SO BROADCAST)
debug	number	0 or 1 (SO DEBUG)
dontroute	number	0 or 1 (SO DONTROUTE)
keepalive	number	0 or 1 (SO KEEPALIVE)
level	number	defaults to SOL SOCKET
linger	map	key: “on” (0 or 1) and “time” (a number) (SO LINGER)
oobinline	number	0 or 1 (SO OOBINLINE)
rcvbuf	number	the size of the receive buffer, (SO RCVBUF)
rcvlowat	number	minimum number of bytes to process for receive (SO RCVLOWAT)
rcvtimeo	number	number of seconds to wait before timeout on receive (SO RCVTIMEO)
reuseaddr	number	0 or 1 (SO REUSEADDR)
sndbuf	number	the size of the send buffer, (SO SNDBUF)
sndlowat	number	minimum number of bytes to process for send (SO SNDLOWAT)
sndtimeo	number	number of seconds to wait before timeout on send (SO SNDTIMEO)

A map with the **sockopts** settings may be used on an existing **net** using **net:setsockopt** .

12.4 Serial I/O

Hobby+ Support for serial I/O is present in Hobbyist, Professional, Embedded, and Enterprise versions of *8th*.

The words in the **sio** namespace control serial I/O. The **sio:open** word is passed a string which is the name of the serial-port to open. This is an OS-specific value: for example, **COM1** on Windows or **/dev/ttyS0** on Linux. It is possible to query the system for valid names using **sio:enum** . That will return an array of names which are valid.

So in order to successfully use **sio:open** you must pass it a valid port name; however, that’s not enough. That port must also be configured to be used, and on Linux at least, you must have read-write access to its corresponding **/dev** file. If the name given to **sio:open** does not meet those conditions, the return value will be **null** ; otherwise, it will be a **sio** which is then passed to the remaining serial I/O words.

Before one can use the **sio:read** and **sio:write** words, the serial port must be configured to use the correct baud-rate and other settings. This is done using **sio:opts!** , which takes a map whose keys represent the values to be modified. You can read the current values with **sio:opts@** , which returns a map with all the values which can be set.

Note that not all settings are applicable to all OS platforms, due to differences in the low-level handling of serial I/O on various platforms.

The most common settings to modify are:

setting	kind	description
baud	number	between 50 and 230400 (on macOS) or 4000000 (other platforms)
bytesize	number	one of 5,6,7 or 8
parity	boolean	if **true** , then "paritybits" is used
paritybits	number	one of 0 (none), 1 (odd), 2 (even), 3 (mark), 4 (space). 3 and 4 are invalid on Linux
stopbits	number	one of 0 (one), 1 (1.5), 2 (two). Note that 1.5 is only valid on Windows

12.5 Bluetooth

Pro+ Support for Bluetooth Classic and BLE is present in the Professional, Embedded, and Enterprise versions of *8th*. You are urged to consult the sample code in **apps/bt/bt.8th** , **hw/ble.8th** , and **hw/bluetooth.8th** .

On Linux and RPI you must either run as “root”, or set appropriate system permissions in order for BLE to work.

At present, BT and BLE only work (completely) properly on Android. We are continuing to improve the cross-platform availability of this important feature!

In order to use BT or BLE functionality, you first need to let *8th* know you want to do that:

```
needs gui/bluetooth
```

Alternatively, use **requires** :

```
requires bluetooth
```

This section is still mostly empty, relying on the sample code as documentation. We will be filling it in in future versions...

12.6 Data persistence

Since you can read and write files and databases and sockets, you may wonder about the best way to persist data (and to transfer it).

If your data is simply a buffer, then it’s simple enough to handle. But more commonly you will have structured data: a map or array or some other data item.

Here are the methods available to you:

method	description	comment
>s / **eval**	convert to a (probably) JSON string	not valid for all types
>json / **json>**	convert your data to JSON string	good for standard JSON, not round-trip for all types
pack / **unpack**	convert to and from binary buffer	machine/OS specific, may be difficult to get right
b:>mpack / **b:mpack>**	convert to and from MessagePack binary format	Pro+ for all types

Note that you cannot persist all data types, simply because the support has either not yet been written, or doesn’t make sense. So while you may want to persist a stack, you’ll have to do it manually; and it makes no sense to try to persist a font.

Ch. 13 The 8th Console

"Console" is another word for the "terminal" or "command shell". *8th* provides a number of words which let you do I/O with the console. Unsurprisingly, these words are in the **`con`** namespace.

If you are running on Windows, and using an MSys or Cygwin shell, then you *may* need to use the freely available **`winpty`** program in order for your console mode programs to work.

You may set text attributes using color-pairs, such as **`red onWhite`** . By default, *8th* starts with a "normal" color defined as **`black bright onWhite`** , but you can change that by changing **`con:normalAttr`** in your scripts, or by setting the environment variable **`EIGHTHCOLOR`** to something else, such as **`bright white onRed`** . No, don't use that color scheme...

Note that **`red`** by itself will not work, the **`onWhite`** is required! You may set or get the current text position using **`gotoxy`** and **`getxy`** . You can also move about the screen with **`up`** , **`down`** , **`right`** and **`left`** . Look in the word list of the **`con`** namespace for the complete list of capabilities.

If you want to grab keys one at a time you can use **`con:key`** , and you can query their availability using **`con:key?`** . The most interesting word, perhaps, is **`con:accept`** . It lets you input up to a given amount of text while taking advantage of the console editing keys. It has a sibling, **`con:accept-pwd`** , which does not display the entered text and which marks the returned text as requiring wipe on release.

13.1 Colors and text attributes

8th includes basic console functionality to begin with. You can use the **`accept`** words to get input, move the cursor around, clear the screen, and print text.

If you want to set colors or text attributes, you need to tell *8th* to load more console support:

```
needs console/loaded
```

You can, alternatively, use requires:

```
requires console
```

That will enable the "foreground" colors (in the **`con:`** namespace): black, red, green, blue, magenta, cyan, white, and yellow. The corresponding "background" colors are prefixed with "on", e.g. "onBlack".

If you wish to have a "bright" color, invoke **`bright`** before the color:

```
with: con
bright blue onWhite
bright red bright onBlue
```

The attributes *8th* knows about are: normal, bold, dim, italic, uscore, blink, fast-blink, reverse, conceal, strike, frame, encircle, and overline.

Note that not all of these are available on all terminals! In particular, 'fast-blink', 'strike', 'frame', 'encircle', and

'overline' are not well supported. Unless you know a terminal supports the attribute, you should stick with the commonly supported ones.

An attribute is turned off using **`end`**

```
with:con
italic "Hi there" . end italic
```

However, for some reason I cannot fathom, **`end bold`** is not supported on most terminals (though the others seem to work). Use **`normal`** to disable bold, though that also disables *all* the attributes!

13.2 Editing keys

The *8th* console gives you some editing capabilities which are similar to what you may be used to from shells like **`bash`** . Here is the exhaustive list of editing keys and their function:

key	action
BKSP	Delete character to the left
Ctrl+A	Move to beginning of line
Ctrl+C	Cancel the input Ctrl+E Move to end of line
Ctrl+D	Quit 8th
Ctrl+K	Show help for the word immediately before the cursor
Ctrl+L	Clear the screen
Ctrl+Left	Move one word left
Ctrl+Right	Move one word right
Ctrl+V	Paste from the system clipboard
Down	Next item in history
END	Move to end of line
ENTER	Accept the input
ESC	Cancel the current input, restore the original
HOME	Move to beginning of line
Left	Move left one character
Right	Move right one character
SHIFT+TAB	Insert a literal TAB character
TAB	Complete the named item immediately before the cursor
Up	Previous item in history

One last thing: if you start *8th* in the console, a thrown exception will not quit *8th*, unlike the behavior when *8th* is executing a file or an application. This is intended to make it easier to deal with mistyped JSON (for example), which would cause an exception and dump you at the OS prompt.

However, pressing **`Ctrl+C`** twice in rapid succession will quit *8th*, with the same effect as typing **`bye`** . So will typing **`Ctrl+D`** once. Likewise, if too many exceptions are thrown in a short time, *8th* will quit.

13.3 TAB completion

While in the console, pressing the **TAB** key will cause *8th* to attempt to perform word completion. It does this by taking the text you entered so far (on the current line), and taking the last space-delimited part. For example, if you entered `123 n:` and pressed **TAB** , the completion code would take the `n:` and attempt to complete it.

It uses the `words-like` word to get a list of all named items which match the prefix you typed. It then filters that list so only items whose prefix matches what you typed are in the list. If there is only one item in the list, the completion is that item. If there are no items in the list, your original prefix remains. If there are multiple items, then those items are listed and your original prefix remains.

13.4 History

By default, the console remembers up to 100 lines worth of your commands. You can access previous history items using the up and down arrows, and once accessed you can edit them. By default, *8th* does not save your history, but you can change that behavior by using the word `con:save-history` , which will save your history by appending it to the named file.

You can change the number of lines the history tracks by using the `-H` command-line option when starting *8th*.

You may likewise restore the history to some previously saved (or manually edited) set, by using `con:load-history` to read in a named file with one history item per line, and a flag which indicates whether to overwrite or append to the current history.

13.5 The prompt

The ubiquitous `ok>` prompt which *8th* presents in the console is actually more complex than it appears. Firstly, you as the user may change the prompt shown, by assigning a different value to the deferred word `prompt` . Before you run off and do that, however, you should know what the default prompt shows.

First of all, the `ok>` prompt is the normal state of affairs. It shows when *8th* is awaiting new input to interpret in the REPL. If the prompt shows anything other than `ok>` , it is indicating a state of incompletion.

If the prompt includes the `"` character, it means a string was being entered but has not yet been completed. If it includes the `{` character, it means a map was not completely defined. Similarly, if a `[` is shown, then an array was not completely defined. Finally, if a `+` is included in the prompt, a word was being defined but not completed.

These indicators may be expected, for example, if you are entering a long bit of text at the console and are entering it on multiple lines. They may also indicate an error. For example, if you typed `".` instead of `" .` to terminate a string and print it.

Ch. 14 Cryptography

8th has excellent built-in facilities for encryption, based upon the well-regarded LibTomCrypt library as well as some other sources.

In general, particular crypto settings are task-specific. That means that setting the hash or cipher to use inside a particular task (including the main task) will affect *only* that task. The default settings are **`aes`** for the cipher, **`blake`** for the hash and **`GCM`** for the cipher mode.

14.1 Hashing

The default hashing algorithm used in *8th* is BLAKE2s , which was one of the SHA-3 finalists. It is very fast and secure. However, you may need to use other hashes, so *8th* lets you easily choose from a number of other hash algorithms. Just use code like: **`"sha1" cr:hash!`** .

The valid values which can be passed to the word **`cr:hash!`** vary from time to time as more are added. The currently supported strings which may be used are: **`blake blake2b md5 rmd128 rmd160 rmd256 rmd320 sha1 sha256 sha384 sha512 sha3-256 sha3-384 sha3-512 tiger whirlpool`** . All of the hash functions may also be used with HMAC. After having set the hash, the chosen hash function remains in force until changed.

The word **`cr:hash`** commences the computation of a hash, and likewise **`cr:hmac`** commences an HMAC-hash. Further data to be hashed are passed to **`cr:hash+`** , and finally either **`cr:hash>s`** or **`cr:hash>b`** are invoked to finalize the hash and produce a result (a readable string in the first case, or a buffer with the hash data in the second).

14.2 Random data

8th has several words providing random data. They are:

word	description
rand	A cryptographically strong but slow PRNG using the Fortuna PRNG
rand-jit	A very slow CPU-jitter PRNG based on the " jitterentropy " library
rand-jsf	The fastest PRNG, based on Bob Jenkin's small PRNG
rand-native	A PRNG using the OS-specific entropy provider
rand-pcg	A fast and strong PRNG using the PCG PRNG
random	A deferred word which is initially set to **`rand-pcg`**

When 8th starts up, it initializes the entropy for the crypto routines with a combination of the OS-specific entropy provider, and the "jitter" entropy provider, if it is available on the specific hardware being used.

Only **`rand`** is guaranteed to be cryptographically strong, and should be used if your application requires that. Using **`random`** and setting it to the PRNG desired at runtime is a convenience for the programmer.

The word **cr:randbuf** returns a buffer with bytes randomly generated using rand. If you don't need cryptographically secure randomness, then **cr:randbuf-pcg** will be much faster.

A random seed for **rand-pcg** and **rand-jsf** is generated on startup. If you want repeatable sequences you need to initialize the PCG PRNG using **rand-pcg-seed** . Currently, the JSF PRNG does not have a seed word.

Note: the PRNGs (with the exception of Fortuna) are task-local, meaning that each task has its own seed and PRNG state. Thus setting the PCG seed in one task will not affect **rand-pcg** invoked from another task.

14.3 Passwords and key-generation

There are several methods for producing an encryption key in *8th*. The simplest is **cr:randkey** , which simply produces a buffer of appropriate size for the current cipher, filled with random data. One could just as easily use **cr:randbuf** which takes the number of bytes and returns a buffer with that many random bytes, though the key returned by **cr:randkey** is also set to auto-wipe as a security feature.

If you want to take a user-provided password and convert it to a key, you can use **cr:genkey** , which implements the PBKDF2 algorithm. You provide it the user's key, a salt string and the number of iterations, and it will return a 32-byte buffer to use as a key. To input the password, you can use **con:accept-pwd** in a console-based application, or use an **edit** control in a GUI, setting **password-char** to whatever you like (* is typical).

14.4 Encryption

14.4.1 Public key encryption (PK)

8th currently supports the RSA public-key (PK) encryption and decryption scheme.

RSA PK encryption is done using **cr:rsa_encrypt** , which takes an RSA public key and data to encrypt, returning an encrypted buffer. RSA PK decryption reverses that process using **cr:rsa_decrypt** which takes the RSA private key corresponding to the public key used to encrypt and the encrypted buffer. It returns a decrypted buffer. The SHA256 hash function is used during the RSA encryption or decryption.

RSA public and private keys are generated using **cr:rsagenkey** , which takes the size of the key (1024, 2048 or 4096 bits) and returns a pair of keys to be used with the RSA encryption words.

At present there is no facility for importing RSA keys from third-party systems.

It is also possible to sign using **cr:rsa_sign** , which takes the hash of a message and a private RSA key and produces a buffer which is the signature. Then one may verify that signature using **cr:rsa_verify** , which takes the hash of the message, the public RSA key and the signature, and produces a **true** or **false** response.

14.4.2 Symmetric encryption

There are several ciphers available for symmetric encryption: **AES** (also known as Rijndael), **Twofish** , **Camellia** , **Blowfish** , **CAST5** , **DES** , **3DES** , **ChaCha20** , **ChaCha20Poly1305** . The default cipher is **AES** , with a key size of 256 bits.

The encryption modes available are: **`GCM`** , **`CTR`** , **`CBC`** , **`ECB`** , **`CFB`** , and **`OFB`** . The default mode is **`GCM`** , and the others are selected using the appropriate invocation, e.g. **`cr:CTR`** . The mode is irrelevant for the **`ChaCha20`** ciphers, since they are stream rather than block-based.

For the block ciphers, **`GCM`** is the recommended mode, since it authenticates the data have not been corrupted. Thus, a valid decryption in **`GCM`** mode means the data were not tampered with or corrupted. This same benefit applies to the **`ChaCha20Poly1305`** encryption method.

`GCM` encryption is accomplished with **`cr:>encrypt`** which takes data and a key and returns a crypto for further processing. That may be the addition of more data to encrypt using **`cr:encrypt+`** or the finalization of the encryption using **`cr:encrypt>`** . As mentioned, the default cipher is **`AES`** 256-bit, **`GCM`** mode. You may also use **`Twofish`** or **`Camellia`** with the **`GCM`** encryption words.

Decryption is done with the corresponding **`cr:>decrypt`** , **`cr:decrypt+`** and **`cr:decrypt>`** . If decryption fails, TOS will contain **`null`** to indicate a problem; otherwise it will contain the decrypted data.

14.4.3 Diffie-Hellman and ECC

There are a number of words available for Diffie-Helman or ECC key-exchange, signatures and verification. Simply use the cr:**`dh-genkey`** or cr:**`ecc-genkey`** etc. words. If the value **`32`** is passed to **`dh-genkey`** , a **`Curve25519`** key-pair is generated. If the value **`64`** is passed, then a **`Ed25519`** key-pair is generated, which is suitable for passing to cr:**`dh-sign`** or cr:**`>edbox`** , etc.

14.4.4 "Boxing" words

A number of convenience words have been added to make it much easier and safer for normal users to take advantage of the strong cryptography features in *8th*. We'll list the most important high-level boxing words — so named because they put everything in a box which the user needn't worry about:

boxing word	description
`cr:>aes256gcm`	Given an item and a key, returns a box which is encrypted with AES-256-GCM. The box contains the generated GCM tag, the random IV which was used for GCM, as well as a box header which (along with the IV) serves as the "AAD" for GCM. The result is that if any bit of the box is changed, the decryption will fail. The box is decrypted using **`cr:aes256gcm>`** , which will return a buffer if successful, or **`null`** if the decryption failed
`cr:>cpe`	Given an item, a key, and an Ed25519 private key, encrypts the item using ChaCha20Poly1305 and creates an encrypted box using **`cr:>cp`** . Then it signs that using the Ed25519 key and creates a signed box. The box can be decrypted and verified using **`cr:cpe>`** , which takes the box, a key and the Ed25519 public key
`cr:rsabox`	Takes an item, and an RSA private key, and creates a box with a header and the signature for (item,key). The signature is verified using **`cr:rsabox>`** , which takes the box and the RSA public key

There are more high-level encryption words available, you are encouraged to view the word-list.

14.4.5 Sharing secrets

It is possible to share secret keys using either Diffie-Helman or ECC keys. The appropriate words are **`cr:dh-secret`** and **`cr:ecc-secret`** . Note that with the DH keys, values from 96-512 will generate NIST keys, which are widely considered to be suspect. Therefore, you should use either 32 - for Curve25519 keys, or 64 - for

Ed25519 keys.

You may also share secrets using Shamir's Secret Sharing System, which is implemented using the words **`cr:shard`** and **`cr:unshard`** . In this process, you "shard" the secret into Y pieces, of which any X must be used to recreate the secret.

Ch. 15 Hardware query and control

The hardware interfacing abilities of *8th* are relatively simple. The three major areas handled by *8th* are general queries, camera control and sensors.

15.1 General queries

There are a number of words whose purpose is to determine the physical nature of the device *8th* is running on, for example the amount of installed RAM. They are all in the **`hw`** namespace, and amply described in the words documentation. The device's operating system is given by **`G:os`** .

15.2 Camera

To use a camera, first query the hardware using **`hw:camera?`** which returns **`null`** if no cameras are present, or an array of maps, one per camera.

If there are cameras present, you may request the use of one with **`hw:camera`** , passing it one of the maps returned from the **`hw:camera?`** query. If the camera is available, a **`hw`** is returned which is used in subsequent camera invocations. Otherwise, null is returned.

If a valid **`hw`** was returned, you may then request a picture to be taken using **`hw:camera-img`** which will invoke the callback word you gave it when a picture has been taken. This process takes varying amounts of time depending on the specific hardware in use. The callback is given an img which contains the latest picture taken, and must return **`true`** to continue taking pictures, or **`false`** to stop taking them.

15.2.1 Raspberry Pi

In order to use the camera on a Raspberry Pi, you need first of all to enable the camera with the **`raspi-config`** utility, and select "Enable Camera". Then you need to load the appropriate kernel module:

```
sudo modprobe bcm2835-v4l2
```

(or whatever is appropriate on your specific hardware) in order for *8th* to talk to the camera.

15.3 Sensors

8th can read the following kinds of sensors:

name	description
accel	The accelerometer, which measures linear acceleration
compass	The compass, which measures magnetic fields
gps	The GPS or other location service

name	description
gyro	The gyroscope, which measures rotational acceleration

In order to use any of them, the steps are the same:

1. Ask for the sensor, passing a string (e.g. "accel") to **`hw:sensor`**
2. If that returned a hw (and not **`null`**), start the sensor using **`hw:start`**
3. Periodically ask for data using **`hw:poll`**
4. When done, relinquish the sensor using **`hw:close`**

The string to pass to **`hw:sensor`** is any of the ones on the left-side of the above table. If the sensor does not exist or is unavailable, **`null`** will be returned.

The data returned by **`hw:poll`** is a map whose keys are specific to the kind of sensor and are listed in the documentation for **`hw:poll`** .

It is your responsibility to poll the hardware, and the polling should be done in a task so as not to block the main GUI or REPL tasks.

15.4 GPIO

Hobby+ On platforms which support GPIO, you may access that hardware using the words **`hw:gpio@`** , **`hw:gpio!`** , **`hw:gpio-cfg`** and **`hw:gpio-set`** . Currently only Linux, Android, and Raspberry Pi support GPIO access.

These words may require root access. So you need to have done **`sudo -s`** or the equivalent, or set up file access permissions properly, in order to take advantage of the GPIO words.

Furthermore, the physical pin layout corresponding to the GPIO registers *may vary between devices*, and so you *must* know what those values are, in order to safely use these words!

15.5 I2C

Hobby+ On platforms which support I2C communications with peripheral devices, you may use **`hw:i2c`** , **`hw:i2c@`** and **`hw:i2c!`** to perform that communication. Currently only Linux and Raspberry Pi support I2C.

Just as with GPIO, root access may be required. And just as with GPIO, it is important to know the details of the hardware device with which you are communicating, since improper access may destroy the peripheral or otherwise cause damage. In addition, you must run **`raspi-config`** on an RPI to enable the I2C interface, and also run **`modprobe i2c-dev`** prior to using I2C functionality.

NOTE: *8th* and Aaron High-Tech, Ltd. are *not liable for*, and take *absolutely no responsibility for* any damage or financial loss caused by use of these low-level hardware accessors!

Please be careful to check and double-check any hardware connections and the corresponding pin numbers or registers before you use the GPIO or I2C words.

Ch. 16 FFI: Foreign Function Interface

The FFI, or “Foreign Function Interface” is how an *8th* program communicates with third-party libraries — whether built-in to the operating system, or from a vendor or other party.

Because *8th*’s built-in data types do not and cannot map directly onto those used by external libraries (usually based on C types), the FFI must hook up some “plumbing” to make the data flow correctly between the *8th* and the external code, and back again.

Fortunately for you, doing this is reasonably straightforward.

16.1 Declaring and invoking FFI routines

In order to access an external routine, *8th must* know first of all what library that routine is in. To do that, declare the library like so:

```
"user32.dll" lib u32
```

This declaration creates a new word called **`u32`** , which when invoked will put the identifier of the external library named **`user32.dll`** on TOS. It also makes that library the one which will be used in subsequent FFI function declarations. The identifier will be **`null`** if the library was not located or could not be loaded for any reason. That fact may be used to perhaps choose a different library at runtime.

Windows users already have **`k32`** declared (**`kernel32.dll`**), while Linux, RPI, and macOS users already have **`libc`** declared (**`libc.so`** or **`libc.dylib`**).

Note: the name passed to **`lib`** may be an OS-specific one, as in the above example: **`user32.dll`** . It may *also* be just the base name, **`user32`** . In this latter case, the library will be searched for as follows:

1. Using the name given: **`user32`** , then
2. With the OS-specific suffix: **`user32.dll`** (or .so or .dylib), then
3. With the "lib" prefix: **`libuser32.dll`**

This allows you to write code which uses a common shared-library across platforms without worrying about the OS naming details.

Functions within that library are then declared as follows:

```
u32 drop
"NNN" "SetClipboardData" func: setClipData
```

The first line tells *8th* that subsequent FFI declarations will use the library loaded by **`u32`** . Each declaration consists of a parameter list, the name of the routine as exported by the library, and the name of the new word which *8th* will use to access that routine.

16.2 Parameters

The parameter list mentioned in the previous section is simply a string, where each letter indicates the type of the item passed or received. *8th* takes care of translating between its internal data representations and those of the external library, based upon this list.

The first character in the parameter list is the return value. That may be one of:

char	description
c	"complex-float", on platforms which have complex types
C	"complex-double", on platforms which have complex types
D	"double", an 8-byte IEEE floating-point value
F	"float", a 4-byte IEEE floating-point value
L	number, the system default signed long integer type (4 byte for 32-bit, 8 byte for 64-bit)
N	number, the system default signed integer type (4 byte for 32-bit, 8 byte for 64-bit)
P	"pointer", the return value is a pointer (will be returned as a ptr)
T	boolean; (any number value other than 0 is `true`
U	number, the system default unsigned integer type (4 byte for 32-bit, 8 byte for 64-bit)
V	"void", or "no return value"
X	number, the system default unsigned long integer type (4 byte for 32-bit, 8 byte for 64-bit)
Z	string

Any other value will result in an **`out of bounds`** exception being thrown.

The rest of the parameter list is the type of each parameter, as expected by the receiving external function. Conversions from *8th* types to these will be performed at run-time. Valid types and possible modifiers are:

char	description
+	The numeric type is the size of a C "long int"
-	The numeric type is the size of a C "short int"
=	The numeric type is the size of a C "int"
&	The numeric type is the size of a C "void *"
?	The numeric type "" is as big as a "size_t"
1	The numeric type is 1 byte long…
2	… 2 bytes
4	… 4 bytes
8	… 8 bytes
b	Buffer without the size (C "void *")
D	Double floating-point (C "double")
F	Floating-point (C "float")
L	System-default long number (32 or 64-bit integer, C "long" type)
N	System-default number (32 or 64-bit integer, C "int" type)
P	Pointer (C "void *", etc): requires a ptr!
S	String with count ("int" string length, then C "char *")
T	Boolean (true, false, or number)

char	description
U	Unsigned integer
W	Word (C function pointer, e.g. "void (*)()")
X	Unsigned-long integer
Z	String (C "char *")

Note that the **B** and **S** both accept either buffer or string values, and work as expected. The string length is the number of bytes of string or buffer data, *not* the number of characters.

If the function requires a pointer to a standard type (for example, a pointer to an **int**) then you must use the **ptr** data type to safely encapsulate that call. See **samples/ffi** for proper usage.

If a numeric modifier is given, it must appear *before* the item it modifies. For example, **8N** means the item is a number which is 8-bytes long. Similarly, **?B** means to treat the buffer as a pointer to the contents, followed by a length which is as big as a **size_t** .

The parameter list describes what the external function expects. At run-time, the FFI verifies it can convert between the item on the stack and its corresponding parameter type. If it cannot, an **out of bounds** exception is thrown. If you incorrectly describe the FFI signature of the external function, the likelihood is that *8th* will crash. If **null** is given as a parameter, it will be considered the equivalent of a NULL pointer.

Note: you may also pass an array of items to be sent over the FFI, rather than just push them on the stack. This is particularly useful if you want to keep pointers around for subsequent use after the FFI call.

16.2.1 "vararg" — C-style variable-argument lists

8th can also handle foreign functions with variable argument lists. The way this works is simple: declare the **func:** as usual, but make the parameter list include only the fixed parameters. Then at runtime, you *must* use an array to create your parameter list, with the last element being an array with the variable parameters. That array will have as its *first* element a string which is a type-list for the variable parameters, so that *8th* can map them correctly. See the **ffi/ffi.8th** sample for the practical details.

16.3 Dealing with arbitrary data ("structs", etc.)

Because the *8th* data types do not map directly to external types, you may need to do further remapping. Specifically, if an external routine returns a C struct, you will probably have to split it apart in order to get at the data you need. This is easily done using the **pack** and **unpack** words, something like this:

```
\ Routine returns struct with four 32-bit int values
"iiii" unpack
```

After this, there will be a number on TOS with the value **16** (the number of *bytes* processed), and an array under it containing four integer numbers, corresponding to the format string passed to **unpack** . The format could also have been specified as **4i** .

Similarly, if you need to pass a “struct” from *8th* to an FFI routine, you will need to create a buffer with appropriate data, using **pack** to convert an array with the struct’s fields. You then tell the FFI that the function

took a pointer, and *8th*'s FFI will convert the buffer appropriately.

The format string for **`pack`** and **`unpack`** has the syntax: **`{[0-9]*x}+`** . That means that each element may have a count, which is an integer saying how many times to repeat the element, followed by **`x`** which is the element specifier. This (count, element) group may be repeated as many times as necessary to complete the layout, and there must be at least one such group (if **`count`** is omitted it defaults to 1).

Valid element specifiers, their meanings and the types they become with **`unpack`** are:

char	description	type
&	"pointer"	number
*	use entire item size (string or buffer)	
+	long int	number
-	short int	number
=	int	number
b	byte	buf
B	byte	number
c	char	string
d	8-byte IEEE float	number
f	4-byte IEEE float	number
h	reverse hex-dump	number
I	4-byte BE integer	number
i	4-byte integer	number
L	8-byte BE integer	number
l	8-byte integer	number
p	pointer to buffer or string	X
P	pointer to number,buffer,string	ptr
s	size bytes (for next specifier; count is number of size bytes)	
W	2-byte BE integer	number
w	2-byte integer	number
x	ignore byte	
X	word pointer (should be an X from w:cb)	

If the **`s`** or **`S`** size-byte specifiers are used, the preceding count, if any, is the number of bytes in that size (default is 1). If a **`b`** or **`c`** specifier is used, then the number format becomes **`x:y`** , where **`x`** is the repeat count, and **`y`** is the number of bytes. If **`y`** is **`*`** , then the entire buffer or string is used. In that case, it is recommended to make sure the item is the correct size desired.

Note that if you wish to create a buffer "big enough" to hold arbitrary data from an FFI call, you can invoke **`pack`** with **`null`** instead of an array of items; the returned buffer will be as big as the format string specified.

16.4 Creating "callback functions"

Some external libraries will call back to your code. *8th* has additional functionality to make this possible.

Creating a callback is rather simple: take a word to be called-back, and a string containing a description of the parameters the callback will receive as well as the return-type, and then invoke **`w:cb`** . The resultant **`X`** is what needs to be passed to the FFI when invoking the external function which will, in turn, call-back to your *8th* word.

See the sample in **`ffi/ffi.8th`** for a working example of this. At present, the callback functionality in *8th* is limited:

- Does not yet work on ARM systems (RPI,Android, and iOS)
- Only accepts "N" (int) and "Z" (char *) parameters
- Only returns "N" (int) or "V" (void) values

Note too, that the callback is not run in the normal *8th* context. That is, it is possibly running in a different thread (but not a task!). So if you need to modify *8th*'s global state, you should take care to use locking to prevent unexpected results.

16.5 Custom libraries

One use of the FFI interface is to utilize code you've written in C in order to do some processing which might be otherwise cumbersome to do in *8th* alone. An example might be image processing.

When designing your own custom libraries to work with *8th*, take into account the fact that *8th* can parse JSON efficiently. Thus, if you wish to transfer a struct from C to *8th*, it may be worthwhile to convert that to a JSON representation first, in your C code, and then return the JSON string.

16.6 Java interface (Android only)

It is possible to access arbitrary Java code from within *8th*, using the Java FFI words **`jclass`** , **`jmethod`** and **`jcall`** . These follow the JNI conventions, so you should be familiar with those before trying to use them. As an example, to call the **`Thread.sleep`** method, you could do this:

```
\ First get the class on the stack:
"java/lang/Thread" jclass
\ And now make a method item:
"sleep" "(J)V" jmethod
\ And finally, invoke the method:
[ 200 ] swap jcall
```

This is particularly useful if you want to enhance your Android application using any APIs which *8th* doesn't expose. Simply write a Java class which performs whatever you need done, modify the manifest file accordingly if necessary, and include an appropriate Java invocation in your code to run the Java code. Take advantage of the easy JSON interfaces in both Android and *8th* to ease passing complex results back to *8th*.

16.7 Danger!

Passing data across the FFI must be done with care. You have *no guarantee* that the external routine will behave nicely, so data returned to you *should be checked* to ensure you have been given something reasonable. Certainly you should not pass a returned string to **`eval`** , as that allows the external library direct access to your application's internals (unless you've restricted the interpreter using **`only`**)!

As a rule you will also want to check that the library you desire to load was in fact loaded. A wise precaution would be to also check that the version of the library is what you expected (if the external library provides a routine to give that information). If an FFI word is invoked and either the library is not loaded, or the function cannot be found, *8th* will throw an exception.

Furthermore, you must be careful when defining the string used to declare the parameters for the FFI function. An incorrect parameter declaration can cause *8th* to crash as mentioned above.

Ch. 17 Graphical User Interface: GUI

17.1 Overview

As of version 20.01, GUI support in *8th* is provided by the "Nuklear" library, instead of "JUCE". If you are new to *8th*, you won't care; but if you've upgraded from a prior version, you'll want to pay very close attention and review the "Updating to 20.01" section

The GUI interfaces are provided by in the **`nk`** namespace, which is a thin layer on top of Nuklear. The former **`g`** namespace has been removed. So if you're upgrading a GUI application, you have no choice but to pay attention!

This manual refers to Nuklear as "NK" hereafter.

17.2 What is Nuklear?

The Nuklear library is an "immediate-mode UI". The formerly used JUCE library is a traditional "stateful UI". The difference is that a stateful UI creates UI "objects" which maintain their own state based on system inputs, and allow the programmer to interact with them only through specialized APIs. An immediate-mode UI requires the programmer to manage (almost all) the UI state, and exposes all of the internals of the UI.

Formerly, you created a "GUI item" by describing it using JSON. That item was instantiated by the JUCE library and its internal functioning was entirely opaque. Now, you create any GUI item on-the-fly, within a "render loop".

The sample code in **`samples/nk/`** amply demonstrate what is meant.

It takes a bit to get used to it, but it fits a lot better with *8th*'s architecture.

17.3 Glossary

The following terms are used throughout all our documentation. Please note:

pt or **point**	An *array* of [x,y] values
rect	An *array* of [x,y,w,h] values
screen window	An OS-specific outermost window. Created by **nk:win** to house your UI
screen	Like you expect, the physical screen (actually: logical screen)
widget	A UI element such as a combo-box or button
window	A NK window created with **nk:begin**. This is required before you can create your UI

17.4 Sample code

You are *urged* to study the samples in **`samples/nk/`** , they provide explanations of the major features, and can be used as templates for your own GUI applications.

17.5 Initialization

Your GUI application will probably invoke **`needs nk/gui`**. This pulls in a number of support libraries you'll find useful, such as the enumeration definitions. In either case, **nk:init** is invoked to ensure the NK subsystem is ready to operate.

Within your **`app:main`**, you'll create at least one screen window which is where your application will create its UI. You might perform any other necessary initialization (such as loading fonts or images), and then you'll invoke **nk:render-loop**, passing it your rendering *word* as well as an event-loop timeout in milliseconds.

Your rendering *word* will be invoked by **nk:render-loop**, and it is there that you create and process your UI.

Note: instead of UI items which maintain state on your behalf, *you* control all the UI state directly. So logic such as when and what to display is handled directly in your code.

17.6 Various

You can get the current screen size with **nk:screen-size**

17.7 UI Components

This is an exhaustive overview of all the UI components provided by the *8th* NK layer.

17.7.1 Screen Window

Before you can do any UI work, you need an OS-specific window, called a "screen window", to contain your UI. The relevant *words* are:

word	SED	description
nk:close-this!	nk --	Closes the the specified screen window
nk:close-this?	nk -- nk T	Same as **nk:close?** for the specified screen window
nk:close?	-- T	Tells whether the current screen window should be closed
nk:screen-win-close	nk --	Flags the specified screen window as needing to close
nk:setwin	s -- T	Makes the named screen window the current one for rendering
nk:win	m --	Creates a screen window
nk:win?	s -- T	Returns **true** if there's a valid screen window

The *map* given to **nk:win** may have the following keys:

key	type	description	default
alpha	n	Opacity of the window, in range [0,1.0]	1.0
bg	clr	Background color to paint the window between frames	0xFF808080

key	type	description	default
decorated	T	Does the window have a title bar etc.	true
display	n	The physical display to use	0
font	s	Default font for items drawn	system
fontheight	n	Default height of font	13
fonts	m	Map of (id,filename) of all fonts this window or its children will use	[system]
high	n	Height of the window in pixels	screen
name	s	REQUIRED: The unique name by which this window is known to 8th	
onenter	w	When window entered (**true**) or left, SED `nk T --`	
onfocus	w	When window got (**true**) or lost focus, SED `nk T --`	
onminmax	w	When window maximized (1), restored (0), or minimized (-1) SED `nk n --`	
onmove	w	When window moved, SED `nk x y --`	
onshow	w	When window shown (**true**) or hidden, SED `nk T --`	
onsize	w	When window resized, SED `nk w h --`	
resizable	T	Does the window have a resize widget	true
title	s	The titlebar title	appname
topmost	T	Is the window 'always on top'	false
unicode-ranges	a	Ranges of Unicode glyphs to include [low,high],...	[0x0020, 0x00FF]
visible	T	Is the window visible initially	true
wide	n	Width of the window in pixels	screen
x	n	Position of left in pixels	centered
y	n	Position of top in pixels	centered

An exception will be thrown if:

- No valid fonts were given
- No name is given, or the name already exists
- An OS drawing context cannot be created
- The window could not be created for some reason
- The NK subsystem has not been (or could not be) initialized

The x, y, wide, and high keys are in **pixels**, unless they are in the range (0,1]. In that case they represent a fraction of the screen size. If wide or high is omitted, the default is the screen width or height. If x or y is omitted, the default is to center within the "display" screen.

Note: These screen window coordinates are in the *logical space* of all physical displays, so (0,0) will not necessarily be in what you think of as the default window. You can find out all the information 8th has on your displays by invoking **nk:display-info**, which returns an *array* of *maps* for each physical display on your system.

Once a screen window has been created, you can create a (NK) window inside it, or use the drawing primitives.

17.7.2 Window

"Windows" are the NK construct which is the main persistent UI state. You create a window with **nk:begin** and terminate its definition with **nk:end**. The "begin... end" must be within your render loop, which means you are "creating" your windows all the time.

You **must** pair **nk:begin** and **nk:end**, and you **must** invoke **nk:begin** prior to creating other UI.

The *map* given to **nk:begin** may contain any of the following. A default with "win" means the value from the enclosing screen window:

key	type	description	default
bg	clr	Color to paint this window's background	win
bounds	rect	Initial size and placement of window	win
flags	n	Some combination of the nk_panel flags	0
font	font	The font to use for this window's elements	win
name	s	A unique identifier for the window	
padding	pt	Window padding [x,y] in pixels	[0,0]
style	m	A "style" to use for this window. See "Styles" below	
title	s	A title for the window, and its identifier if **name** isn't given	anon

Throws if **nk:win** was not invoked first.

Relevant flag values: WINDOW_BACKGROUND WINDOW_BORDER WINDOW_CLOSABLE WINDOW_MINIMIZABLE WINDOW_MOVABLE WINDOW_NO_INPUT WINDOW_NO_SCROLLBAR WINDOW_SCALABLE WINDOW_SCALE_LEFT WINDOW_SCROLL_AUTO_HIDE WINDOW_TITLE

Relevant *words*: nk:begin nk:end nk:win-bounds nk:win-bounds! nk:win-close nk:win-closed? nk:win-collapse nk:win-collapsed? nk:win-content-bounds nk:win-focus nk:win-focused? nk:win-hidden? nk:win-hovered? nk:win-scroll-ofs nk:win-scroll-ofs! nk:win-show

17.7.3 Fonts

Currently, fonts are created solely from existing TTF (TrueType) or OTF (OpenType) font files. They may be specified in a few ways. Assuming the font file is "font.ttf" and we want a 20 pixel high font:

"font.ttf:20"	A *string* which includes the full font file name, a colon, and the font size
"*font.ttf:20"	A *string* which starts with an asterisk means "load from asset"
"font1.ttf:20;font2.ttf:20"]	A *string* separated by semicolons means try each font string until a good one is found

The font file may be loaded into a buffer first, and passed to **font:new** for example.

You may use **font:system** to get a system-specific font in a given size, without having to worry about the name or location of the font files.

More to come...

17.7.4 Layout

“Layouting” in general describes placing widgets inside a window with a position and size. There are several APIs for performing layout, each with different trade offs between control and ease of use.

You start layouts with one of the following:

nk:layout-row-dynamic SED: `h c --` Lay out rows with **c** columns of widgets, and row-height **h**. Putting more widgets than **c** starts a new row with the same layout.

nk:layout-row-static SED: `h w c --` Like layout-row-dynamic, but each widget gets the same width **w**, and the row size doesn't grow with the window.

nk:layout-row-begin SED: `fmt h c --` Begin a series of rows using **nk:layout-row-push**, until **nk:layout-row-end**. Unlike the two previous, it does not automatically repeat.

nk:layout-row SED: `fmt h [ratio] --` If the layout of each row is the same, you can use this one to lay out rows in terms of [ratio], which is an *array* of *numbers*. If the values are less than 1, they are a percentage of the window width. Otherwise they are pixel values. Automatically repeats.

nk:layout-row-template-begin SED: `n --` Start a row template, for a row-height of **n** pixels. Once the template has been established and **nk:row-layout-template-end** invoked, subsequent widgets are layed out according to the established template. You create each widget's template within the row by invoking one of **nk:layout-push-dynamic**, **nk:layout-push-static**, or **nk:layout-push-variable**. See the help for what those specifically do.

nk:layout-space-begin SED: `fmt h c --` Allows direct placement of widgets within the window. Coordinates begin at the end of the last row; so you generally would use this for an entire window. Pair with **nk:layout-space-end** to terminate the layout, and **nk:layout-space-push** to position the next widget either as a pixel location or a ratio of the space.

17.7.5 Group

Groups are basically windows inside windows. They allow you to subdivide space within a window, to layout widgets as a group. Almost all more complex widget layout requirements can be solved using groups and basic layout fuctionality. Groups, just like windows, are identified by an unique name and internally keep track of scrollbar offsets by default.

Relevant words: **nk:group-begin** and **nk:group-end**, which must be paired.

17.7.6 Tree

Trees represent two different concepts. First, the concept of a collapsable UI section that can be in either a hidden or a visibile state. They allow the UI user to selectively minimize the current set of visible UI.

The second concept is tree widgets for visual UI representation of trees.

Trees can be nested for tree representations and multiple nested collapsable UI sections. All trees are started by invoking **nk:tree-state-push** and ended with **nk:tree-pop**. Or more conveniently one may use **nk:tree-push**, which saves the state in the *var* given or in the window's map under the key *string* given.

Note: the tree-pop *must only be invoked* if the tree-push *words* returned **true**.

17.7.7 Widget

The **nk:widget*** *words* operate on the current layout slot. **nk:widget** creates space for a new widget inside the current layout. If these words are used inside custom widget creation code, they operate with the previous layout slot, not the custom widget.

Take a look at the **`nk/widget*`** samples for ideas on how to implement your own widgets.

17.7.8 Label

Labels are simply read-only text. The *words* available to create them are: nk:label nk:label-colored nk:label-wrap nk:label-wrap-colored

17.7.9 Button

Buttons are clickable UI elements which initiate some kind of action. There are a few *words* to create them: nk:button nk:button-color nk:button-label nk:button-symbol nk:button-symbol-label

The workhorse is **nk:button**, which takes a *map* of options; see the help for details on what the options are.

The important bit to remember is that you *must* provide a *word* for the button to invoke. If you've used **nk:button** to create the button, then the *map* you gave it is also passed down to the *word* when it is invoked, allowing you to pass arbitrary information to the button handler word.

17.7.10 Checkbox, Radio button (Option)

These are essentially the same, in that they're handled by the same underlying code in NK. However, you create them differently and they look different.

Checkboxes are created using nk:checkbox.

Radio-buttons are called 'options' in NK, and are created with nk:option

17.7.11 Selectable

Creates an item which is like a toggle switch: create with nk:selectable. You can display an image, a predefined NK symbol, or nothing along with the text.

17.7.12 Slider

This is a slide-control which allows you to select a range of integer or float values. Implemented in nk:slider, but

ease-of-use wrappers are in the **nk/sliders** library (nk:slider-int and nk:slider-float).

17.7.13 Progress bar

A “progress-bar” which goes from [0..max]. Created with nk:progress.

17.7.14 Color picker

A color-picker widget, created using nk:color-picker. You give it a color to start with and it returns the chosen color (which may be the same as given).

17.7.15 Properties

A “property control” widget, which is similar to a “slider” except that you can directly enter a value in the associated edit control. Basic implementation in nk:(property), and ease-of-use wrappers in the **nk/property** library (nk:prop-int, nk:prop-float).

17.7.16 Text edit

A fairly versatile text-editing widget created with nk:edit-string. The ‘filter’ provide can be a *word*, in which case you can decide what characters to allow. Built-in filters are any of PLUGIN_FILTER_ASII, PLUGIN_FILTER_FLOAT, PLUGIN_FILTER_DECIMAL, PLUGIN_FILTER_HEX, PLUGIN_FILTER_OCT, and PLUGIN_FILTER_BINARY.

17.7.17 Chart

Serveral kinds of chart: line and bar, created using nk:(chart-begin) and nk:(chart-begin-colored) until nk:(chart-end). More commonly using nk:chart-begin and nk:chart-end (loaded in the **nk/loaded** library).

17.7.18 Popup

A “popup-window” which may be a menu or any other sort of window overlayed on the current window. Start with nk:popup-begin and end with nk:popup-end. Within the popup you define layout or draw on it as you would a regular window.

17.7.19 Combo box

A “combo-box” created with either nk:combo or nk:combo-cb. You determine the size of the drop-down and the current selection; the new selection (which may be the same) is returned to you.

17.7.20 Contextual

A “contextual-menu”, typically initiated with a right-click. Start with nk:contextual-begin and end with nk:contextual-end. Other words of import: nk:contextual-close nk:contextual-item-text nk:contextual-item-image-text nk:contextual-item-symbol-text.

17.7.21 Tooltip

Show a “tooltip” using nk:tooltip.

17.7.22 Menu

Full-fledged menu widget. Start with nk:menu-begin until nk:menu-end. In between, use nk:menu-item-label, nk:menu-item-symbol, or nk:menu-item-image. Also nk:menu-close to force-close a menu.

17.7.23 Image

Images are handled via the **`img:`** namespace. They are created using img:new, which can load a PNG, BMP, TGA, GIF, PIC or JPEG image. Output formats (with img:>file) can be PNG, BMP, TGA, or JPEG.

More later...

17.7.24 List

A “list control” created with nk:list-new, which keeps list state, and rendered between nk:list-begin and terminated nk:list-end. See the 'database/foodlist.8th' sample for how it's used.

17.7.25 Drawing

There are a variety of drawing primitives you can use. Currently supported are (all in the **`nk`** namepace):

fill-rect fill-rect-color fill-circle fill-arc fill-triangle fill-poly draw-image draw-text draw-text-high stroke-line stroke-arc stroke-curve stroke-rect stroke-circle stroke-try stroke-polyline stroke-polygon

Note: the ability to perform arbitrary affine transforms is currently lacking, but will be added in the next release or two. Likewise, the ability to get drawing data (paths) from fonts is not yet implemented.

17.7.26 Input

You can check for various input events using the following **`nk`** words:

clicked? hovered? down? released? key-pressed? key-released? key-down? text?

17.7.27 Style

Are you unhappy with the default styling? Is it too dark and depressing?

Well don't worry! You can easily change the styles used, either on a whole-window basis or for specific widgets.

The simplest word is nk:style-from-table, which takes an array of numbers which are color values, and uses it. The values, in order, correspond to:

```
COLOR_TEXT, COLOR_WINDOW COLOR_HEADER, COLOR_BORDER, COLOR_BUTTON,
COLOR_BUTTON_HOVER, COLOR_BUTTON_ACTIVE, COLOR_TOGGLE, COLOR_TOGGLE_HOVER,
COLOR_TOGGLE_CURSOR, COLOR_SELECT, COLOR_SELECT_ACTIVE, COLOR_SLIDER,
COLOR_SLIDER_CURSOR, COLOR_SLIDER_CURSOR_HOVER, COLOR_SLIDER_CURSOR_ACTIVE,
COLOR_PROPERTY, COLOR_EDIT, COLOR_EDIT_CURSOR, COLOR_COMBO, COLOR_CHART,
COLOR_CHART_COLOR, COLOR_CHART_COLOR_HIGHLIGHT, COLOR_SCROLLBAR,
COLOR_SCROLLBAR_CURSOR, COLOR_SCROLLBAR_CURSOR_HOVER,
COLOR_SCROLLBAR_CURSOR_ACTIVE, COLOR_TAB_HEADER
```

The most complex word is nk:make-style, which takes a map describing all the style parameters, and is used in conjunction with nk:use-style. The map may contain any or all of these keys (all items default to the style values of the enclosing screen window if missing):

key	type	description
button	m	A map of button-style entries
chart	m	A map of chart-style entries
checkbox	m	A map of toggle-style entries for checkbox items
combo	m	A map of combo-style entries
contextual-button	m	A map of button-style entries for contextual items
edit	m	A map of edit-style entries
font	fnt	The named font (or fnt) to use
menu-button	m	A map of button-style entries for menu items
option	m	A map of toggle-style entries for radio-button (option) items
progress	m	A map of progress-style entries
property	m	A map of property-style entries
scrollh	m	A map of scrollbar-style entries
scrollv	m	A map of scrollbar-style entries
selectable	m	A map of selectable-style entries
slider	m	A map of slider-style entries
tab	m	A map of tree-tab-style entries
text	m	A map of text-style entries
window	m	A map of window-style entries

Button style

key	type	description
bg	clr	Normal background color

key	type	description
bg-active	clr	Active background color
bg-border	clr	Border background color
bg-hover	clr	Hovering background color
border	n	Border width
draw	w	Word to invoke for custom drawing
draw-end	w	Word to invoke after custom drawing
img-padding	pt	Padding around the image
padding	pt	Padding around the text
rounding	n	Radius of corners
touch-padding	pt	Padding around the displayed portion for clicking
txt	clr	Text normal color
txt-active	clr	Text active color
txt-align	n	Text alignment flag
txt-bg	clr	Text background color
txt-hover	clr	Text hover color

Chart style

key	type	description
bg	clr	Normal background color
border	n	border width
border-color	clr	Normal border color
color	clr	item color
padding	pt	padding around charts
rounding	n	radius of corners
selected-color	clr	Selected color

Combo style

key	type	description
bg	clr	Normal background color
bg-active	clr	Active background color
bg-hover	clr	Hovering background color
border	n	border width
border-color	clr	Border color
button	m	button style
button-padding	pt	padding of button
button-sym-active	n	active button symbol
button-sym-hover	n	hover button symbol

key	type	description
button-sym-normal	n	normal button symbol
content-padding	pt	padding of widget
rounding	n	radius of corners
spacing	pt	spacing between symbol and text
sym	clr	symbol color
sym-active	clr	symbol active color
sym-hover	clr	symbol hovering color
text	clr	label text color
text-active	clr	label active text color
text-hover	clr	label hovering text color

Edit style

key	type	description
bg	clr	Normal background color
bg-active	clr	Active background color
bg-hover	clr	Hovering background color
border	n	border width
border-color	clr	Border color
cursor-hover	clr	hovering cursor color
cursor-normal	clr	normal cursor color
cursor-size	n	size of the cursor
cursor-text-hover	clr	hovering text cursor color
cursor-text-normal	clr	normal text cursor color
padding	pt	padding around widget
rounding	n	radius of corners
row-padding	n	padding between rows of text
scrollbar	m	scrollbar style
scrollbar-size	pt	size of the scrollbar
selected-hover	clr	hovering selected color
selected-normal	clr	normal selected color
selected-text-hover	clr	hovering selected text color
selected-text-normal	clr	normal selected text color
text-active	clr	active text color
text-hover	clr	hovering text color
text-normal	clr	normal text color

Progress style

key	type	description

key	type	description
bg	clr	Normal background color
bg-active	clr	Active background color
bg-hover	clr	Hovering background color
border	n	Border width
border-color	clr	border color
cursor-active	clr	Color of active cursor
cursor-border	n	Cursor border width
cursor-hover	clr	Color of hovering cursor
cursor-normal	clr	Color of normal cursor
cursor-rounding	n	Cursor corner radius
dec-button	m	Dec button style
draw	w	Word to invoke for custom drawing
draw-end	w	Word to invoke after custom drawing
edit	m	Edit control style
inc-button	m	Inc button style
padding	pt	Extra padding
rounding	n	Radius of corners

Property style

key	type	description
bg	clr	Normal background color
bg-active	clr	Active background color
bg-hover	clr	Hovering background color
border	n	Border width
border-color	clr	border color
dec-button	m	Dec button style
draw	w	Word to invoke for custom drawing
draw-end	w	Word to invoke after custom drawing
edit	m	Edit control style
inc-button	m	Inc button style
padding	pt	Extra padding
rounding	n	Radius of corners
sym-left	n	Left-arrow symbol
sym-right	n	Right-arrow symbol
txt	clr	text color
txt-active	clr	Active text color
txt-hover	clr	Hovering text color

Scrollbar style

key	type	description
bg	clr	Normal background color
bg-active	clr	Active background color
bg-hover	clr	Hovering background color
border	n	border width
border-color	clr	Border color
border-cursor	n	Border around the cursor
cursor-active	clr	active cursor color
cursor-border-color	clr	Cursor border color
cursor-hover	clr	hovering cursor color
cursor-normal	clr	normal cursor color
dec-button	m	style of decrement button
dec-symbol	n	symbol for decrement button
draw	w	Word to invoke for custom drawing
draw-end	w	Word to invoke after custom drawing
inc-button	m	style of increment button
inc-symbol	n	symbol for increment button
padding	pt	padding of widget
rounding	n	radius of corners
rounding-cursor	n	Radius of corners around the cursor

Selectable style

key	type	description
bg	clr	normal active background color
bg-hover	clr	hovering active background color
bg-pressed	clr	pressed active background color
bgi	clr	normal inactive background color
bgi-hover	clr	hovering inactive background color
bgi-pressed	clr	pressed inactive background color
draw	w	Word to invoke for custom drawing
draw-end	w	Word to invoke after custom drawing
image-padding	pt	image padding
padding	pt	widget padding
touch-padding	pt	touch padding
txt	clr	normal active text color
txt-align	n	text alignment
txt-bg	clr	text background color
txt-hover	clr	hovering active text color

key	type	description
txt-pressed	clr	pressed active text color
txti	clr	normal inactive text color
txti-hover	clr	hovering inactive text color
txti-pressed	clr	pressed inactive text color

Slider style

key	type	description
bar-active	clr	active background bar color
bar-filled	clr	filled background bar color
bar-height	n	height of bar
bar-hover	clr	hovering background bar color
bar-normal	clr	background bar color
bg	clr	Normal background color
bg-active	clr	Active background color
bg-hover	clr	Hovering background color
border	n	border width
border-color	clr	border color
cursor-active	clr	active cursor color
cursor-hover	clr	hovering cursor color
cursor-normal	clr	normal cursor color
cursor-size	pt	cursor size
dec-button	m	dec button style
dec-symbol	n	dec button symbol
draw	w	Word to invoke for custom drawing
draw-end	w	Word to invoke after custom drawing
inc-button	m	inc button style
inc-symbol	n	inc button symbol
padding	pt	padding around widget
rounding	n	radius of corners
show-buttons	n	whether or not to show buttons
spacing	pt	spacing inside widget

Tab style

key	type	description
bg	clr	Normal background color
border-color	clr	Border color
text	clr	Text color
border	n	border width

key	type	description
padding	pt	padding of widget
spacing	pt	spacing from indicator to text
indent	n	how far to indent each tab
rounding	n	radius of corners
maximize-button	m	style of maximize button
minimize-button	m	style of minimize button
node-maximize-button	m	style of node maximize button
node-minimize-button	m	style of node minimize button
sym-minimize	n	symbol for minimize button
sym-maximize	n	symbol for maximize button

Text style

key	type	description
color	clr	The color to draw the text in
padding	pt	The padding [x,y] to apply around the text

Toggle style

key	type	description
bg	clr	Normal background color
bg-active	clr	Active background color
bg-border	clr	Border background color
bg-hover	clr	Hovering background color
border	n	Border width
draw	w	Word to invoke for custom drawing
draw-end	w	Word to invoke after custom drawing
padding	pt	Padding around the text
spacing	n	Spacing between the selector and the text
touch-padding	pt	Padding around the displayed portion for clicking
txt	clr	Text normal color
txt-active	clr	Text active color
txt-align	n	Text alignment flag
txt-bg	clr	Text background color
txt-hover	clr	Text hover color

Window style

key	type	description

key	type	description
bg	clr	Background
border	n	Width of border
border-color	clr	Border color
combo-border	n	Width of combo border
combo-border-color	clr	Combo border color
combo-padding	pt	Padding around contents for combos
contextual-border	n	Width of contextual border
contextual-border-color	clr	Contextual border color
contextual-padding	pt	Padding around contents for contextuals
fixed-bg	clr	Fixed background
group-border	n	Width of group border
group-border-color	clr	Group border color
group-padding	pt	Padding around contents for groups
header	m	Window header style
menu-border	n	Width of menu border
menu-border-color	clr	Menu border color
menu-padding	pt	Padding around contents for menus
min-row-height-padding	n	Minimum padding for a row
min-size	pt	Minimum window size
padding	pt	Padding around contents
popup-border	n	Width of popup border
popup-padding	pt	Padding around contents for popups
rounding	n	Radius of corners
scaler	clr	"Scaler" (resize widget) color
scrollbar-size	pt	Size of the scrollbars
spacing	pt	Extra spacing
tooltip-border	n	Width of tooltip border
tooltip-border-color	clr	Tooltip border color
tooltip-padding	pt	Padding around contents for tooltips

Window header style

key	type	description
align	n	Alignment of text
bg	clr	Background
bg-active	clr	Active background
bg-hover	clr	Hovering background
close-button	m	Style for close button
close-symbol	n	Symbol for close button

key	type	description
label-active	clr	Color of active header text
label-hover	clr	Color of hovering header text
label-normal	clr	Color of header text
label-padding	pt	Padding around text
maximize-symbol	n	Symbol for maximize button
minimize-button	m	Style for min/maximize button
minimize-symbol	n	Symbol for minimize button
padding	pt	Padding around header
spacing	pt	Extra spacing for header

17.8 Updating to 20.01

The **`g`** namespace no longer exists, it has been replaced by **`nk`** . The reason for the replacement is that the new Nuklear library works in such a different manner, that virtually none of the **`g`** functionality made sense.

This means that any code which formerly had **`requires gui`** for example, will need to be reworked. I apologize, but it was necessary.

GUI items are no longer defined by maps, generally speaking. Please review the samples in **`samples/nk`** to get a clear understanding of how the new **`nk`** GUI works.

Sound has also been reworked, though it mostly works the same as before. Missing features currently are: seeking to a specific time location in a sound, and recording sound. Also, the variety of sound formats supported is more restricted.

Video/camera is not currently supported. That will be remedied in the next release (or perhaps the one after).

Ch. 18 Tasks and parallelism

18.1 Introduction to tasks

A task in *8th* is effectively the same as a thread or co-routine in other languages. You create one using either **`t:task`** or **`t:task-n`** — the first simply invokes the word it is given on a separate task, while the second transfers the top **`N`** items from the current stack to that of the new task.

```
: the-task-word ... does something ... ;
' the-task-word t:task
```

A task will run as long as the word you gave as its starting point continues to run. When that word terminates, the task is complete; if there are no references to it, it cleans-up after itself and disappears. If you do hold a reference to it, you can retrieve the last value on its TOS using **`t:result`** (which will be **`null`** if there was no result).

Using this facility, you can place long-running tasks in the background so they don't interfere with your foreground GUI (or console). You can also split tasks into smaller pieces which can be run independently of each other, and perhaps gain a speedup. If you have a multi-core system, you will be more likely to experience a speedup than if you don't. However, placing "blocking" or long-running tasks in a background task will make your user's subjective experience better because the application will be more responsive.

18.2 Creating a task

The word **`t:task`** is used to create a new task, as shown above. It can be invoked either with a single word on TOS, or with a map containing options. In either case, there must be a word which the task will invoke. Once that word exits, the task is finished and its exit code may be retrieved using **`t:result`** as mentioned in the previous section.

If starting the task using a map, a number of options may be set:

name	kind	description	default
affinity	n,a	The CPU or CPUs to which this task's "affinity" should be set	
auto	T	If **`true`** , the task will auto-GC when the xt has finished	**`false`**
name	s	Sets the initial name of the task	xt's name
num	n	Number of items to transfer from the creating task's stack. The new task will start with those items on its stack	0
qsize	n	the size of the task's queue	**`t:def-queue`**
stack	n	the data-stack size for the task	**`t:def-stack`**
xt	w	required: the word invoked as the task's xt	

18.3 Task stacks

Each task receives its own set of data- and r-stacks. This gives you the freedom to do whatever you need to on the task's stack without being concerned you might mess up the main stack.

The default size of a task stack is set by `t:def-stack` , and originally is 128K items. This default value You may also specify a particular task's stack size by passing a map to the `t:task` or `t:task-n` words as listed above.

There is usually no need to make the stack size smaller, because the memory for it is only reserved and not committed until used. So even though the default stack has room for 128K items, it will only be committed in 4K sized chunks (on most OSes).

18.4 Multitasking and locking

By default, *8th* does not lock in most situations! That means that if you have an array which you access *simultaneously* from more than one task, you may end up with bugs which are difficult to diagnose — up to, and including, random crashes. *8th* doesn't lock by default because it doesn't need to. Items are allocated from task-specific pools, and those pools are only accessed from within a particular tasks's context. This greatly increases *8th*'s speed.

However, it is sometimes necessary to lock when you're running multiple tasks in an application, to prevent data corruption. The specific case where it's necessary is when multiple tasks can access the same item simultaneously:

```
0 var, counter
: task-word
  counter lock
  1 n:+!
  counter unlock drop ;

' task-word t:task drop
' task-word t:task drop
```

This shows how `task-word` may safely increment the global variable `counter` even though it is running in two different tasks. By using the `lock` and `unlock` words judiciously, you will avoid data corruption problems.

Be careful that you don't create a deadlock situation when locking, where one task has locked an item but did not unlock it before another one needed it.

The `queue` data type *does* do locking, since it is generally intended to be used simultaneously from various tasks. You do not use locking on that data type!

Note also that only *container types* generally require locking, since they are mutable. So in the above example, the var itself is locked, but the item contained within it is not.

18.5 Using task queues

A task also includes a queue. That queue will have a size determined by `t:def-queue` (the default is 8 items), or by a map key qsize given to t:task.

To push an item onto another task's queue, use `t:push` . You need to have the task as created by `t:task` in order to do that. You can also push onto the main task's queue by using t:main as the identifier. *8th* always creates one task, the REPL; and a gui app also has a GUI task.

If your task consumes data from its queue but doesn't produce new data, you will build up a large free-list for the pools of those items which are not created. So for example, if you push maps to another task, and that task doesn't create new ones (which will be allocated from the pool's free list) then there will be a memory usage increase (not exactly a leak, but in effect the same as one).

Inside the recipient task, you use `t:pop` to retrieve any items which have been pushed to it. You can also determine how many items are in that queue using `t:qlen` .

18.6 Being a good citizen

It is good practice to make your task do its work in short bursts. The precise amount of work depends on a great many things — but if your task never sleeps, you will lose overall system performance and your users will not be happy.

So you will probably want to work in a loop where you do a `0.1 sleep` (or whatever is appropriate to your application) between bursts of processing. A typical scenario is that the task waits using **-1 sleep**, and is notified by another task that it has data to process.

Alternatively it can use `-1 t:q-wait` in which case it will awaken as soon as an item has been pushed to its queue.

18.7 Parallelism

If your processing can be profitably broken into chunks which can be worked on independently, then you can leverage tasks to do your bidding. For example:

```
0 100 2 ' process t:task-n drop
100 100 2 ' process t:task-n drop
200 100 2 ' process t:task-n drop
```

This sample partitions the processing into three chunks: one which works from 0 to 99, one from 100 to 199 and one from 200 to 299.

To see a fully-worked example of parallel processing, look at the sample `misc/parallel.8th` .

18.8 Task best practices

There are several points to be careful of when using multi-tasking. In no particular order:

- In a GUI app, do not access the GUI from a task until the GUI has been created. Failure to ensure that will cause sporadic crashes. Use the `init` GUI event to kick off your GUI handling tasks, since it signals that the GUI is ready.
- If you are using global variables, make sure access to them is protected by a `lock… unlock` pair, in particular if the variable may be modified by one or another of the tasks. Failure to do so will result in incorrect results, and possibly crashes.
- Avoid accessing global variables from multiple tasks! While you can do that if you properly lock access, the act of locking slows your program down, and it make your program more fragile. It is better to take

advantage of task-specific variables, and to structure your code so you don't need to access a single global variable.

- Leverage **`t:task-n`** to pass values directly to the task from inception, rather than using global variables.
- Use task-local variables in preference to global-variables.
- Despite the above warning to lock global variables, if different tasks access different portions of a container or img (for example), then locking may not be necessary. Not locking is always faster than locking (but caveat coder)!

Ch. 19 Exceptions and error handling

8th distinguishes between "errors" and "exceptions". An "error" is a condition which is expected to possibly occur, and which the programmer should handle. An "exception" is an exceptional condition which the programmer cannot handle safely. Thus, an exception is treated as a "fatal error" in all cases.

This is in contrast to languages like Java and C++, where a block of code may be surrounded by a **`try... catch`** construct. *8th* does not use that construct, since exceptions indicate an error condition which by definition *cannot* be handled safely.

That said, one may intercept thrown exceptions by installing a new word for handler, like so:

```
: my-handler
  … figure out if we should handle this
  true ;
' my-handler w:is G:handler
```

The **`true`** return value means your handler word decided *8th* should continue. The default behavior is to quit the application after having displayed a message. Execution will continue (if **`true`** was returned) from the place where the exception was thrown, so your code needs to know how to repair the stack or take other corrective measures as needed.

Not all exceptions are created equal! If you get a stack underflow or overflow, you cannot continue processing because the state of the stack is indeterminate at that point.

Your own code can choose to **`throw`** a string (which is what *8th* always does) or to **`throw`** some other data type which may convey more information which your handler code can then act upon. It is recommended that you throw exceptions as a way to indicate to the end-user what the particular fatal error condition is.

In console mode, exceptions thrown will return to the console rather than quitting *8th*. This is so in order that you may interactively debug the issue or re-enter mistyped text. Be aware: the stack may contain garbage on it because of the **`throw`** . So it's a good idea to invoke **`.s`** after an exception in the console, so you can be certain it contains what you intend.

You may also define a *task-specific* exception handler using **`t:handler`** . That will be invoked in the task which encountered the exception, allowing you to preferentially terminate just the specific task, or handle and continue, or the default handler behavior.

19.1 User-defined exceptions

As mentioned above, you may choose to throw a simple string, which will then be displayed. Any other type you throw will be converted to a string for display.

If you choose to **`throw`** a number, then you should be careful to use one greater or equal to 1,000 — anything between 0 and 1,000 is reserved for *8th*'s internal use.

19.2 Built-in exceptions

8th has a number of built-in exceptions it can throw. Here is a list of all of them. The "%" character indicates a place where a **`sprintf`** insertion will be made at runtime:

exception text	description
the namespace name '%s' is invalid	Attempt to create a namespace with a ':' character in the name
eval! does not understand the type	eval! could not evaluate the value it was given
%s is not a file	The "file" given on the command-line was not a valid file
the file given is not valid input	You gave *8th* a file to interpret and it wasn't able to understand what format it was
recursive JSON	Self-referencing JSON was encountered
invalid JSON %s	An invalid JSON array, map, or string was parsed
can't parse regex	Tried to parse a regular-expression beginning with "/" and failed
unable to parse a word header	The header for a word cannot be parsed from the input
probable missing ; or) in %s	You forgot to terminate your word definition properly
number too big	Conversion of a big integer will overflow the native type
out of bounds access	Trying to access beyond the valid bounds of a buffer,etc.
expected a camera	You gave something other than a camera hw
can't clone a %s	That item type is one of the kinds which cannot be cloned
can't use G:new for ns(%u)	The namespace identifier does not allow G:new
may only be used in compile mode	The word may not be used in interpret mode, only in compile mode
can't get crypto provider	Can't initialize the Windows crypto provider
unknown %s: %s	You requested an unknown cipher or hash type
mismatched flow-control: %s	A flow-control word was not correctly matched
deprecated: %s	The listed word has been deprecated, and you must not use it any longer
cannot use libev timers or watches on this platform	the OS doesn't support events properly
propval! can't handle a %s	The propval GUI doesn't know what to do with the type it was given
ffi for %s expected %c type but got %s	The **FFI** word expected one kind of parameter but got another
ffi for %s invalid retval %s	The **FFI** doesn't know what the given return type is
ffi for %s doesn't handle %s	The **FFI** word doesn't handle the parameter type given
fp overflow	Converting a bfloat or bint to a float overflowed the float
gui: duplicate name [%s]	The GUI def specified more than one item with the same name
preferences gui requires preficon	A preference GUI item must have a 'preficon' specified
toolbar requires a valid img	The image you specified for the toolbar could not be loaded
unknown gui type [%s]	The GUI 'kind' you specified isn't known
expected hex digits	The string escapes \x or \u expect two (or four) hexadecimal digits and didn't get them

exception text	description
%s requires matrix of odd size (%d invalid size)	the size of the matrix you supplied for the filter is invalid
invalid matrix given in %s	The matrix you supplied for the filter is invalid
'%s' is unknown	The interpreter couldn't figure out what that text is
%s may only be used in interpret mode	The specified word cannot be used inside a word definition
invalid bounds: %s	The "bounds" string was incorrectly formatted
invalid regex %s at %d	The given regular-expression is invalid, beginning at that offset in the expression
jni %s expects %d parameters, %d provided	Android: the JNI call requires a certain number of parameters, which is not what was given
jni can't find %s with parameters %s	Android: JNI was unable to find a function with the given signature
unknown word in JSON	When parsing the JSON, a word was encountered which was not found in the namespace search
needed system libraries not found: %s	A required system library was not found (Linux/RPI/AGL)
map size beyond system limits	The map had to expand its hash-table beyond the limits allowed by the system
math exception	A hardware math exception happened (not recoverable)
this matrix expects '%s' type	The matrix required a number and you provided a complex or vice versa
bad memory access or corruption	A SEGV or similar occurred (not recoverable). Typically because of invalid stack access. This should not normally occur in proper code. If your code is otherwise correct, report this as a bug
mysql bind requires parameters in an array	db:bind for MySQL databases requires the parameters be in an array
don't know how to send item of ns%s to MySQL	db:bind does not know how to send the type indicated to MySQL
db:open mysql: no support	Cannot open a MySQL database, because the required support isn’t loadable
don't know how to convert MySQL type %d	db:exec does not know what to do with the returned MySQL type shown
net:server invalid %s	The SSL certificate or key of the net:server is invalid
no locals available	You attempted to use a word-local variable but didn’t specify locals: before the word
no GUI is available	No GUI is available, probably because X-Windows is not available
only Android	The Java interface words are only available on Android
you’re out of memory	The system cannot allocate any more memory
queue empty	The queue is empty
queue full	The queue is full
REPL task died	The main task died unexpectedly
%s requires %s	Variety of situations; details are given in the message
%s requires a string or a buffer	Variety of situations; details are given in the message
%s needs root privileges	This functionality requires "root" access
use the specific 'new' word for the ns	You are using G:new for a type which has a different method of creation
couldn't load image for splash	The image file for the "splash" GUI was not found or could not be loaded
%sstack overflow	You attempted to access above the top of the stack

exception text	description
%sstack underflow	You attempted to access below the bottom of the stack
format spec %s requires a %s	The s:strfmt specification given requires the type given
too few format params were given	The s:strfmt word requires more parameters than were given
s:new requires a number, string, or buffer	Just as it says
new task requires a word to execute	You didn't provide a word for the new task to invoke
debug breakpoint not allowed	Don't try to use a debugger on me
unhandled exception %08x	Windows: an system exception we don't know how to handle happened
varargs array must have a fmt string and at least one arg	Just as it says
couldn't initialize WINSOCK	Windows: the WINSOCK library could not be initialized

19.3 Signals

The word **`app:signal`** is invoked when certain signals are received by 8th. On all platforms, SIGINT, SIGTERM, and SIGABRT are trapped. On non-Windows platforms, SIGHUP, SIGUSR1, and SIGUSR2 are also trapped.

When **`app:signal`** is invoked, the name of the signal received is on TOS, e.g. **`INT`** or **`USR1`** . The handler must not assume it is in any particular task, and must do as little as possible. The default handler just prints the name of the signal received and quits the program.

You can trap additional signals using **`app:trap`** , but you must use a signal number rather than a symbolic name, and signal numbers are OS-specific. Signals you trap in this manner are also routed to **`app:signal`** , and the "name" on TOS will be the number of the signal (as a string).

You can also send a signal to your application using **`app:raise`** . Only signals the system recognizes may be set.

19.4 Error handling

Each word handles errors in its own manner, most frequently returning **`null`** as an indication that an error occurred. You must pay attention to the documented behavior of each word in order to properly handle errors.

Many will also provide more error information by setting the return value of **`t:err?`** . That returns a map which has a numeric error code as well as an error message in most cases.

In all cases it is up to the programmer to check for error results if there is a possibility that an error may occur and require handling. A typical trope is:

```
do-something null? if   \ failure…
  drop …
else                    \ all's well
  process-something
then
```

Ch. 20 Debugging

Unlike many other languages and development environments, *8th* does not have a dedicated debugger. Instead, it gives you tools to help you find problems in your code interactively.

20.1 Categories of problems

There are several kinds of problems you might encounter while writing *8th* code. They are, in decreasing order of severity:

- **crashes** : A system-level crash, e.g. "SIGSEGV" or "Access error" or something like that. You should not normally see this, but if you do it is a serious bug which needs to be reported to us. Please use the bug reporting application to let us know about this kind of problem. However: if the crash is subsequent to a thrown exception in interactive mode, it is not a bug in *8th*
- **security** : If you have found a way to subvert *8th*'s security model (e.g. modify an encrypted application so it runs without complaint), that is also a serious bug we would like reported to us
- **refcount** : This is yet another serious bug: if the refcnt of an item is 0 (or is some largish number) then unless you were using **`-ref`** or **`+ref`** , you should report this as a bug
- **exception** : If you get an "Exception" message (either in the console or in a message box), this is an error being reported by *8th*. In the *8th* model, an exception is a condition which is not tolerable. However, it usually indicates an error in your code rather than a problem in *8th*
- **incorrect** : The code compiled without complaint, but it does not give correct results
- **inconsistent** : The code does not behave the same way from one run to the next

20.2 Debugging techniques

While developing your application, you have a number of tools at your disposal for debugging problems. Since the most common cause of problems is losing control of the stack, your first line of defense is to keep your words short and comment the SED! Shorter words are easier to understand, and comments which are accurate are extremely useful.

Whether your words are short or not, you absolutely *must* ensure the SED is adhered to. Therefore: your next most important tool is the phrase **`.s cr`** , which you can make a little more fancy by putting it in a word of its own:

```
: XX log .s cr ;
```

This **`XX`** word would then be used like so:

```
: some-word...
  "part 1" XX
  …
  "part 2" XX
  … ;
```

This would print **`part 1`** and (up to) the top ten items of the stack, then do something, then print **`part 2`** and (up to) the top ten items of the stack at that point. This gives you an annotated real-time stack-dump, which can help you quickly pinpoint problems related to improper stack usage.

Most of the time, the **`.s cr`** phrase is enough to track down and eliminate stack problems. The following hints may also help:

- Pay attention to the word's SED! Make sure you are giving words the correct stack picture to use. A frequent problem is putting too many or too few items, or the wrong kinds of items on the stack. Check the documentation!
- If you are using an iterator like **`a:each`** , make sure you consume the correct number of items from the stack. Again, check the documentation!
- Factor your words into smaller pieces, and document their stack-effect with a SED. Then make sure you adhere to the documented effect for each factored word

Another potential source of problems is using an "unqualified word"; that is, saying **`+`** instead of **`n:+`** (for instance). In fact, you will generally see an exception in such a case, telling you (for example) that *8th* expected a string but got a number. In interpret-mode, *8th* will usually pick the correct word (though not always), while in compile-mode it will more often error. So avoid ambiguity and use qualified words (with the sole exception of "G:" as mentioned earlier).

In addition to the techniques mentioned in this chapter, *8th* provides some rudimentary debugging words in the **`dbg`** namespace and in the **`debug`** libraries. They are documented with the rest of the built-in words and in the libraries chapter.

Ch. 21 How do I?

This chapter contains some tips and tricks to help you figure out how to accomplish some typically useful tasks with *8th*.

21.1 Start 8th?

Check:

- You are using the full pathname to the `8th` (or `8th.exe`) binary
- Linux/macOS/RPI: make sure that binary is executable (`chmod +x`). It is, by default, but you might have munged it.
- Did you put the binary in your `PATH` (if not using the full path)?

Windows users: You need to run *8th* from the console. You cannot just "click on it", because it's a command-line application.

21.2 Get an updated version of 8th?

There are a few URLs to help you:

- Re-download your existing *8th*: https://8th-dev.com/refresh.php
- Upgrade *8th* to a different SKU: https://8th-dev.com/upgrade.php
- Acquire an additional year of updates to *8th*: https://8th-dev.com/update.php

21.3 Open a file

Are you sure you gave the correct path to the file?

Windows users: did you recall that a backslash `\` must be **doubled** inside a string?

21.4 Convert a 'character' to a 'string'?

In *8th*, a character is a Unicode value, which means it's a number. A string is a UTF-8 sequence of bytes representing characters.

When you use `'` to create a character, for example `'a` , it puts a number on TOS (in this particular case, 97). To convert that to a string, you need to append the character to an existing string. So:

```
: char>s \ n--s
  "" swap s:+ ;
```

21.5 Print a single character (like EMIT in ANS Forth)

Use **`putc`** .

21.6 Convert text to a number?

Use **`\>n`** .

21.7 Convert anything to a string?

Use **`\>s`** .

Ch. 22 Standalone Applications

A “standalone application” is one which may be run on its own, like any other native application on the target platform. *8th* provides a simple method for producing standalone applications from a single set of source code. This functionality is available to all versions of *8th*.

Pro+ However, producing encrypted standalone applications is only available to users of the Professional, Embedded, or Enterprise versions.

22.1 Application life-cycle

An application is loaded by the *8th* engine and then verified and decrypted (if it was encrypted) and the plain-text code is then interpreted. After that, assuming the code doesn’t invoke some other word as its last step, the *8th* engine invokes **`app:main`** . So a typical application will have code resembling this, at the end of the code:

```
: app:main
  initialize
  run-stuff
  shut-down ;
```

Android and iOS applications may handle the OS suspend and resume by hooking their own code instead of the default (do-nothing) **`app:suspended`** and **`app:resumed`** . Those will get invoked at the usual OS-specific times.

You can use **`onexit`** to add words to be executed on program shutdown. The usual **`bye`** or **`die`** words will cause the **`onexit`** chain to be invoked.

22.2 Setting up your application

Before you do anything else, please make sure that you have set up *8th* as discussed in the chapter on installation. Once you have done that you can use the **`build`** tool.

In order to produce a standalone application, **`build`** needs your code and associated resources to be in a folder of their own. It also needs a project description file, which is just a JSON map with information helpful to **`build`** . If you have not created such a file, the **`build`** tool will create a template version of it in the folder you named on the **`build`** command-line; it is up to you to ensure the contents are correct.

For example, if your application is named **`test.8th`** , you might put it in a folder called **`test`** , along with the application icon **`test.png`** and other support files. Included in that folder you could also place **`test.proj`** , the project description file. You can either copy the sample **`ttt.proj`** in the **`samples/games/tictactoe`** folder, or use the **`build`** tool to create one for you.

22.3 Building the application

To create the packaged application, run the build tool which is located in the bin folder. A typical command to run it would be:

```
8th bin/build demo/tictactoe/
```

Once invoked, the **build** tool presents a GUI which allows you to enter specific information about your application, and save the settings in the project description file. Note for macOS users: due to macOS limitations, you must run *8th* from within the usual app folder; otherwise, you will not be able to enter text!

In the build GUI, you can select the platforms you wish to produce output for. You can select the program icon and permissions for iOS or Android applications (note: selecting the icon and/or permissions is currently inactive; it will be activated in a future release).

When you are ready to produce the final executables, simply press the "Generate" button, and within a few seconds your application will be packaged for all the platforms you have selected. The output will appear in the out subdirectory of the project's folder.

Note for CLI users: if you prefer to use a command-line interface, or if you would like to do so for unattended builds, you can pass the **-g** flag before the directory name, like so:

```
8th bin/build -g demo/tictactoe/
```

That will take all its information about what to produce from the JSON map, so make sure you've set the values there as you would like them to be.

22.4 Android

The build process *8th* creates a binary which will run on your Android device, but it does not do the Android-specific signing and packaging. That is handled by the **bin/makeapk.8th** script. In order to create a proper APK file which can be installed on your Android device, please follow these steps:

1. Prerequisite: the Android SDK. You must have installed the Android SDK from Google, and you must have the Android SDK tools available so you can access them from the command-line. The location of the SDK must be in either the **ANDROID_SDK** or the **ANDROID_HOME** environment variable.
2. Prerequisite: Java JDK. The location of the JDK must be in the **JAVAHOME** environment variable.
3. Using the build tool, click "Generate" for your application, after having chosen "Android" as an OS target.
4. Edit the **out/android/AndroidManifest.xml** file in your project's folder, ensuring you have set only those permissions your application requires. Also change the "package" and other parameters to those applicable to your application
5. You probably also want to copy your application's logo files over the ones in the various **out/android/res/drawable...** folders
6. Copy any external libraries (**.so** files) into the **lib** folder
7. Copy any compiled Java classes you added into the **lib** folder.
8. Copy any resource files you need into the **res** folder
9. Run **makeapk** : **8th bin/makeapk.8th my/project/folder**

The **makeapk** script accepts various command-line parameters which change its behavior from the defaults. Type **8th bin/makeapk.8th -h** to see a listing of the options.

You can deploy to your Android device by using the command-line utility **adb** from the Android SDK or any other method you desire.

22.5 iOS

8th does not sign and package your code in the iOS required manner. The steps are also similar to those for Android, but much more finicky:

1. Prerequisite: A computer running macOS. Sorry, you cannot package or sign on anything but a Mac.
2. Prerequisite: An iOS Developer account. You must have the correct credentials from Apple, or you will be unable to create an application for an iOS device.
3. Prerequisite: XCode. You need the Apple XCode tool in order to properly sign and deploy your application
4. Prerequisite: Provisioning profile (create one in your iOS developer account online and download the file)
5. Unzip the ios.app.zip file from *8th* to a temporary area. For example, **`myapp`** . After that, you should have a folder **`myapp/IOS.app`** .
6. Edit the **`Info.plist`** so it has the permissions you require
7. Copy your application logo.png over the **`IOS.app/logo.png`**
8. Copy your provisioning profile into **`IOS.app/embedded.mobileprovision`**
9. Copy the XCode iPhoneOS.sdk **`ResourceRules.plist`** into **`IOS.app/ResourceRules.plist`** (unless you have a different set of rules)
10. Create an appropriate entitlements file and save it outside the IOS.app folder
11. Using the build tool, click "Generate" for your application. It does not matter which OS you choose (as long as you have chosen one)
12. Copy the build-generated file **`appdata`** over the file **`myapp/IOS.app/assets/appdata`**
13. Copy the appropriate *8th* binary over **`IOS.app/8th`** . Note: if you wish to submit to the App Store, you will need to use **`lipo`** to create a fat-binary of *8th* (otherwise you can choose to just use one of 32 or 64 bit iOS binaries). Do that with **`lipo bin/ios*/8th -create -output fat8th`** , and copy the **`fat8th`** binary over **`IOS.app/8th`**
14. Run the *8th*-provided bash script **`bin/floatsign.sh`** to create the IPA file

The floatsign.sh script is invoked like this:

```
bash bin/floatsign.sh myapp/ios.app "iphone developer" \
  -p my.mobileprovision \
  -e entitlement.file \
  IOS.ipa
```

Here, “iPhone Developer” is the signing key you wish to use, **`my.mobileprovision`** is the provisioning profile to use and **`entitlement.file`** is the entitlements file. The script will sign the **`IOS.app`** and embed the provisioning profile, leaving you with a signed and ready to deploy IPA in **`IOS.ipa`** . Getting that onto your device is left to you (**`fruitstrap`** is a nice command-line method).

Thanks to Daniel Pfeiffer for the **`floatsign.sh`** script.

Submitting to the Apple App Store requires you sign with a distribution key and comply with numerous other details which are prone to change and therefore left to you to work through when you so desire.

22.6 macOS

The story here is much simpler. To get a properly functioning GUI application on macOS, your application needs to be packaged correctly. Fortunately, it’s not difficult (sorry, Apple: the **`osx.app.zip`** file will stay with that name):

1. Unzip the **osx.app.zip** file from *8th* to a temporary area. For example, **myapp** . After that, you should have a folder **myapp/OSX.app** .
2. Convert your application **logo.png** to the **ICNS** format (there are online and command-line tools to do that)
3. Copy the converted **ICNS** file over the file **Icon.icns** in the **OSX.app** Resources folder.
4. Copy the **build** generated file (given in the 'App name' field) for macOS (either 32 or 64 bit) over the file *8th* in the **MacOS** folder
5. Copy the build-generated file **appdata** over the file **myapp/OSX.app/assets/appdata**
6. Edit the **Info.plist** as necessary
7. Rename the **OSX.app** folder to correspond to the **appname** of your application

At this point you should be able to run the application by simply clicking on its icon in the Finder application. You can also sign it if you wish, prior to distributing it using the **codesign** tool from XCode.

Ch. 23 Effective 8th

For most programmers, *8th* will be a bit unfamiliar. The paradigm shift can be difficult at times, but once you are comfortable with it, we believe *8th* to be a more powerful and productive language than many. In this chapter we will try to help you become a more effective *8th* programmer. See also the "Best Practices" document **`bestpractice.pdf`** .

23.1 Iterative refinement of code

If you come from other programming languages, you may be used to the “waterfall model”, where the coding happens after a great deal of thought went into the design, and once the design is established the coders write code for a few weeks and then test the code. This is a very bad approach for writing *8th* programs. Why? Because small problems add up, and it can be very difficult to track down issues in large bodies of code.

Therefore, the recommended approach is to write a word and then test it immediately. Since you have access to a REPL, you can interactively test words as you write them. Or if you are working on a GUI based application, you can test via the application. In either case, the key is to iteratively refine your code. How?

Start with the main code in pseudocode:

```
: app:main
  initialize
  main-code
  clean-up
  bye ;
```

Establish the application’s high-level flow. Now implement each of the words you started with as a place-holder:

```
: initialize "initialize" log ;
: main-code "main-code" log ;
: clean-up "clean-up" log ;
```

Verify that when you run your app, you see the appropriate log output. Then proceed to “fill in the blanks” for each word. As you write each one, document its SED, and then immediately test it to ensure that the code you wrote does indeed have the stack-effect as well as any side effects it is supposed to have.

The **`log`** word is asynchronous by default. That means that its output occurs some time after it was invoked. If you are in a console-only application and you quit out before the log is written, you will see no output. If that’s the case, you can invoke **`false log-async`** to tell the logger to print immediately.

Tell the logger to print the current time as of the invocation of **`log`** , by invoking **`true log-time`** .

You will find that spending the time to test while coding will pay off many-fold in reduced time debugging and lower blood-pressure.

23.2 Factoring code

By the term factoring, we mean "break your code into smaller pieces". Ideally those pieces will themselves be useful in their own right. A well-factored *8th* program will consist of many small words instead of a few large ones.

For example, let's say your task is to write a word which returns the sum of the squares of two numbers. The word will get two numbers on the stack, and return a single number. So your first effort may look like this:

```
: sum-of-squares \ a b -- a2+b2
  dup n:*        \ a b2
  swap dup n:*   \ b2 a2
  n:+ ;
```

Nice and simple; it works, and is documented sufficiently well. Factoring really involves scanning the code and looking for repetitive phrases. In this example, we notice **`dup n:*`** is repeated, so we factor it out into its own word:

```
: square \ a -- a2
  dup n:* ;

: sum-of-squares    \ a b -- a2+b2
  square            \ a b2
  swap square       \ b2 a2
  n:+ ;
```

In this specific example, factoring out **`square`** may seem to give little benefit. But it serves more than one purpose. First, the factor **`square`** is useful in its own right, and is so simple that it is easy to see that it works. Second, by using **`square`** instead of **`dup n:*`** it is clear at a glance what we are trying to do inside the **`sum-of-squares`** word. Finally, by extracting that factor we have made it much less likely we will have an error caused by dropping a word (accidentally deleting **`dup`** for example).

Don't be concerned about making too many words. The heavy cost of insufficiently factoring the code is much greater than the very small cost of adding more words. The benefit of more easily maintainable and more robust code, usually outweighs any other consideration.

As an added benefit, factoring makes it easier to verify your code. In our example, you can simply type in some test cases in the console:

```
10 square . cr
-2 square . cr
```

If you don't see **`100`** and **`4`** , you'll know something is amiss with the code. You can (and should) also verify that in fact the stack depth is the same before and after you invoked square. A common source of bugs, as we mentioned in the previous chapter, is losing control of the stack.

Avoiding complexity is helped by proper factoring as well. By breaking your code into smaller factors, you help reduce the size of words, and make it easier for you to grasp at a glance what the code does.

Write short words! They should be relatively short (3-7 lines of phrases of 3-5 words). This makes it much easier for you to debug them and ensure they do what you want.

Interactively debug your words as you write them. Do not wait until you've built a colossal program to debug the components. Given *8th*'s interactive nature, it is very easy to simply invoke a word and see if it does what you intend. Better yet, write a test-suite for your words which loads and tests them.

23.3 Use the stack

One of the most difficult habits to break for programmers coming from more well-known languages, is the reliance on variables. *8th* is built around a stack, and data is passed back and forth on it — you get the stack for free, it makes sense to use it. In the example we gave before of sum of squares, someone with experience in C-type languages may very well write something like this:

```
var sq1
var sq2
: sum-of-squares \ a b -- a2+b2
  \ b2 -> sq1
  \ a2 -> sq2
  dup n:* sq1 !
  dup n:* sq2 !
  sq1 @ sq2 @ n:+ ; \ get sq1 and sq2 and add them
```

This code *does* work, but is inferior to the example we gave earlier in a couple ways. First, moving data from the stack to a var and back again takes extra time and extra code. Second, the vars take up space. Third, the code is less clear because of all the noise of moving data back and forth. And finally, the code is larger than it should be by quite a bit.

Obviously you may use variables in *8th*, since they are part of the language! And there are indeed occasions where you *must* use them: for example, if you have global state you need to keep track of. However, the words you write should ideally get everything they need from the stack and put their results back on the stack as far as possible.

As mentioned, one reason for this is that moving data back and forth to variables is expensive. However, another reason is that using the stack makes your words re-entrant, while if you use variables your words will not be. This may be important, particularly in GUI applications where callbacks may occur simultaneously (or nearly so).

If you find yourself using variables to store intermediate results, you probably need to factor your code a bit more. Even if the factors don't make sense as standalone words, they may vastly simplify the stack-picture in your code.

In order to use the stack effectively, you should restrict your words to using no more than three items at a time from the stack, and attempt to factor to reduce stack juggling. In particular, if you find you must use **`pick`** very much, you probably need to factor the code some more.

23.4 Faster code

If your goal is to produce the fastest code possible, you should consider the following:

- Pick the fastest algorithm which matches your constraints
- Avoid store and fetch from variables (or other containers)
- Juggle the stack less
- Use fewer words. Yes, this will make your code less readable and violates the principle of factoring. Each word invoked takes time.
- Utilize the built-in data types rather than creating your own parallel versions
- Consider breaking your code into tasks which can be run in parallel, particularly if you are running on a multi-core machine

- Consider using the FFI to offload CPU intensive work to an optimized library

Ch. 24 Libraries

8th includes some code as externally-loadable libraries, accessible using the word **`needs`** . So for example, to utilize the code in the *8th* library **`net/soap`** , include a line like this in your code: **`needs net/soap`**

After that, the words in that library will be available to be used in your *8th* program. Note: the library name should *not* be enclosed in quotes.

24.1 8th libraries, include files, and your code

When you use the word **`needs`** , *8th* looks for a source-code *8th* library of the name you give it. So **`needs net/soap`** will look for the library **`soap`** in the **`net`** subdirectory of the *8th* **`libs`** directory. If the file you specify is not found there, *8th* will also attempt to find the file in the directory specified in the **`EIGHTHLIB`** environment variable, if there is one. This lets you add your own libraries to *8th* without modifying the distribution files.

If instead of **`needs`** you use **`f:include`** , *8th* looks for the file in the path you give it, perhaps relative to the invoking file. It also will look for that file in the **`incs`** subdirectory relative to your main file.

`needs` and **`f:include`** provide similar, but not identical, functionality.

When you use the **`bin/build`** utility to package an executable, it packages your code as well as any *8th* libraries required to make it run, as determined by **`needs`** . All that code gets put into assets in the packaged executable, and is available no matter what platform you are running the package on. In order for your files loaded with **`f:include`** to be similarly packaged, you must have them located in the **`incs`** asset folder (that is, the **`incs`** subdirectory relative to your main source file).

24.2 Private and Public words

8th lets you create “private” words in a library. That is, words which are not findable once that library has been loaded. You declare that the following words being created are private by simply invoking **`private`** (usually on a line by itself, though that’s not required).

Once **`private`** has been invoked, the following words are accessed normally until either **`public`** is invoked, or the end of the library file has been reached. Once **`public`** has been invoked, the private words can be accessed in the special namespace**`#p`** :

```
private
: mum "is the word" . cr ;

public
: say-mum #p:mum ;
```

Once the library file has been loaded completely, the private words are no longer accessible in the **`#p`** namespace, though of course their code remains.