LUKE WREN

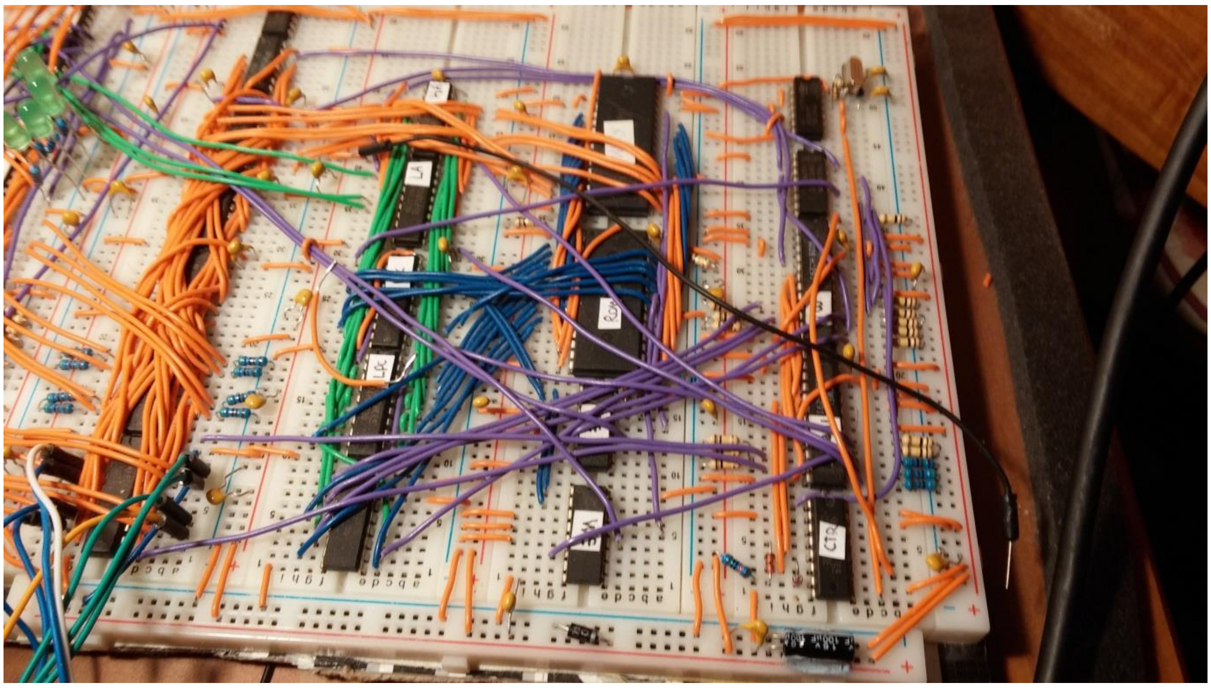
SPOON: PROGRAMMING MANUAL

Writing Programs in Spoon | 2014

Table of Contents

Introduction

Spoon is a high level programming language, which runs on a unique processor architecture, known as a single instruction computer.



To write programs in Spoon, you type statements, which are instructions to the computer: add these two numbers together, look this up in a table, jump to this other part of the code.

A compiler then translates these high-level instructions into machine code: the binary code that the computer runs in order to carry out its task.

Each individual instruction is very simple, but the Fork machine – the custom built computer shown above – can carry out 800,000 instructions per second (state of the art processors can carry out billions), and enough simple instructions in sequence can be combined to accomplish difficult tasks.

This book is divided into three sections. The first, very short, describes the architecture of the Fork machine: how it works at a low level. The idea of a Von Neumann architecture is explained, and the memory map and instruction format are described (these terms are explored in this section).

The second section is a tutorial: writing your first program in Spoon, learning about the different types, libraries and functions, and learning to write programs for the Fork OS operating system.

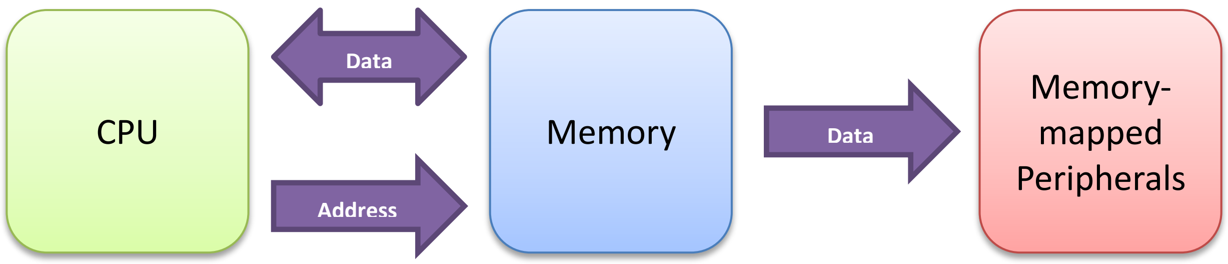
The third and final part is a reference, for more advanced programmers. It describes the language syntax and semantics in detail, lists the builtin functions, and describes the language's semantics.

Section 1: Architecture

Machine Architecture

There are two main competing types of machine-level computer architecture: the Von Neumann architecture, and the Harvard architecture. The key difference lies in where the instructions (parts of a program) and the data are stored: in a Von Neumann architecture, these two types of data both live in the same place – memory. This is an attractively simple model, but there are drawbacks: mainly, the sharing of the same data bus by both instructions and data creates a performance bottleneck. There are also security issues, with problems such as buffer overruns overwriting executable code. The main competing model, the Harvard architecture, separates instructions and data into separate stores. This brings performance and security benefits, at the cost of complexity.

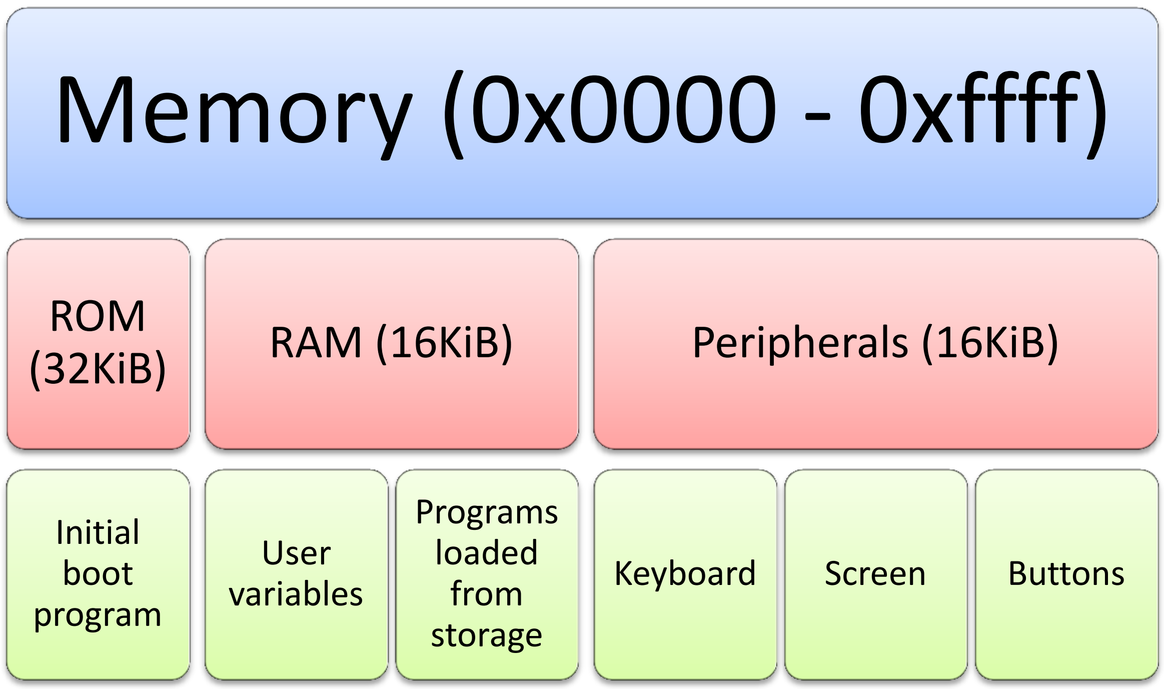
For simplicity, the Fork machine is based on a Von Neumann architecture.



The machine is 8-bit: this means that any calculations taking place inside the computer use binary numbers 8 bits in size (bytes). Larger numbers are supported only by treating them as collections of bytes.

Addresses are 16 bits long: this means that the computer can refer to any of 65536 (2^{16}) locations in memory.

The memory is split into various address ranges, corresponding to different devices on the circuit board, as shown:



ROM is read-only memory: it cannot be changed by the running program. It is used for the storage of the bootstrap program, and any code (drivers etc) used by the operating system.

RAM (random access memory) can be freely edited by programs. Any variables declared and used by the programmer are allocated to a position in RAM, which is reserved for that variable when the program is compiled.

The final 16KiB of memory is mapped to various (up to 8) I/O registers, used for input and output to/from peripherals. Any user interaction, including the screen and the keyboard, is used by reading and writing specific values to these registers.

Instruction Set Architecture

The instruction set architecture (ISA) describes the compiler's view of the computer: in short, the list of instructions, how they are formatted and decoded, and what they do.

This machine is fairly unique in having only one instruction. It seems odd that such a simple instruction set can lead to a functional computer, but this is indeed the case!

The specific instruction chosen is Nor and Fork Conditionally, or NFC. Each instruction consists of four addresses (8 bytes total), known as A, B, C and D. Execution consists of the following steps:

1. X ← A NOR B
2. A ← X
3. If X ≠ 0:
    a. Jump to C
    b. Else jump to D

The NOR of the first two indicated bytes is calculated, and then stored In the first location. Depending on the result of the NOR, the computer then conditionally branches (forks) to one of the second two addresses.

Thinking carefully we see that this single instruction provides all we need to do some computation:

- Reading from memory
    - Locations A and B are read in step 1.
- Manipulating data
    - NOR can be used to compose any logical operation, which can be combined for things like arithmetic.
- Writing to memory
    - A is written to in step 2.
- Branching
    - We can branch both conditionally and unconditionally (if both skip addresses are the same).

For example, in hexadecimal:

```
bfff 0018 0008 0008
bfff c001 0010 0010
c000 bfff 0000 0000
ff
```

The first instruction: NOR the location bfff (the last byte of RAM) with 0018 (the constant ff) at the end of the program, and store the result in bfff. Since the result of NOR with ones is always zeroes, this has the effect of clearing this location. It then jumps unconditionally to 0008, the location of the next instruction.

The second instruction NORs the byte in memory (now all zeroes) with the debug input register (c001), which has the effect of setting bfff to the inverse of the debug input switches. It then jumps to the next instruction.

The third instruction writes the inverse of bfff (i.e. the original value of debug input) to the debug output. This works because output ports are always read as zero. It then jumps back to the top of the program, closing the loop.

This is a very simple example; programs compiled from Spoon with the Spoon compiler can be thousands of instructions in length.

Section 2: Programming in Spoon: Tutorial

Spoon is a relatively powerful language, tailored specifically for the Fork machine. It builds on the ideas and syntax of C, with a few simplifications (and limitations).

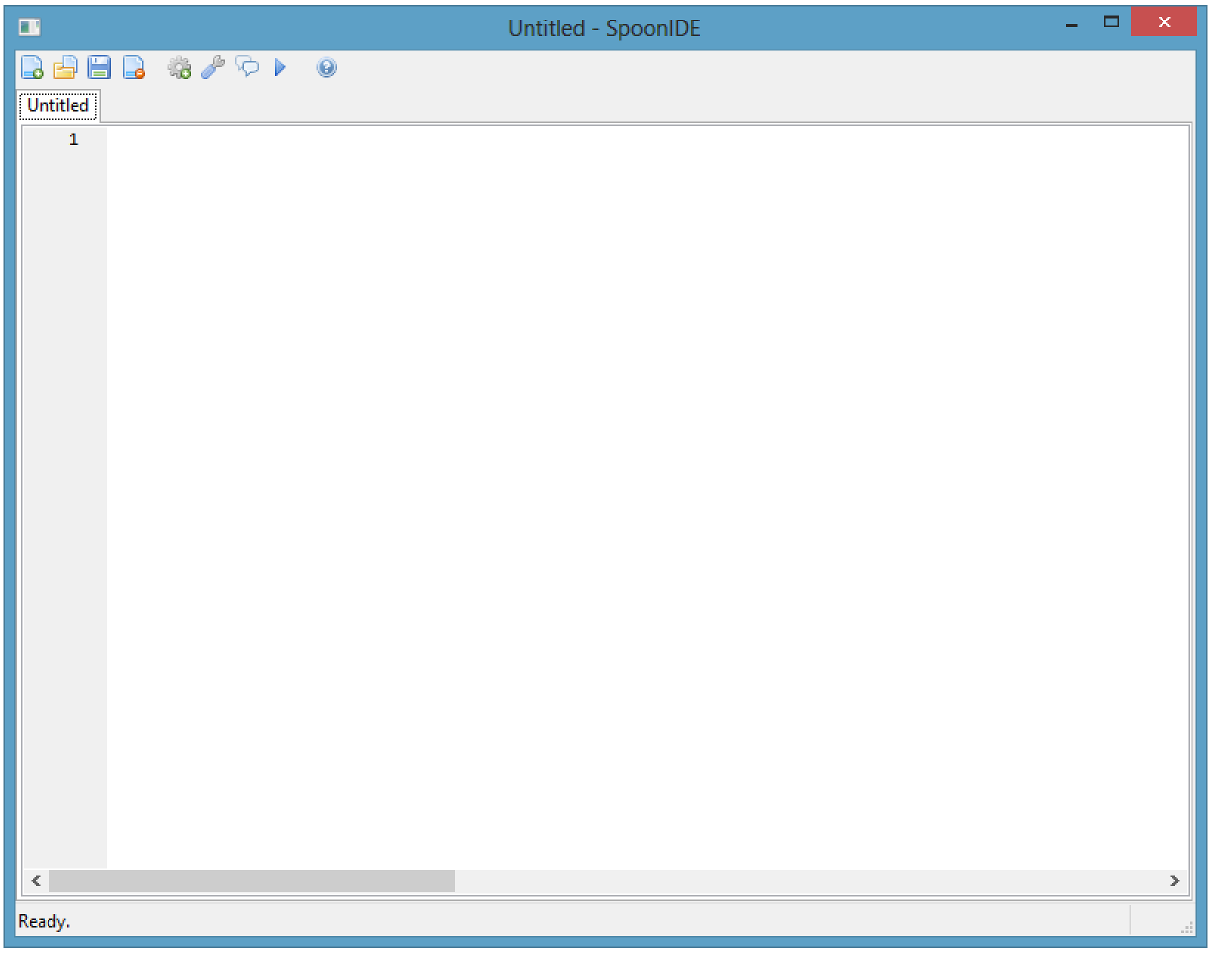
This is a step by step guide of writing your first program in spoon; understanding: statements, control flow, variables, pointers, libraries; using the integrated development environment; and writing programs compatible with the Fork OS operating system, allowing you to use the screen, keyboard and flash storage devices.

We'll start slow, and then maybe get a little faster as things move on.

Your First Spoon Program

Double click on SpoonIDE to open the integrated development environment:

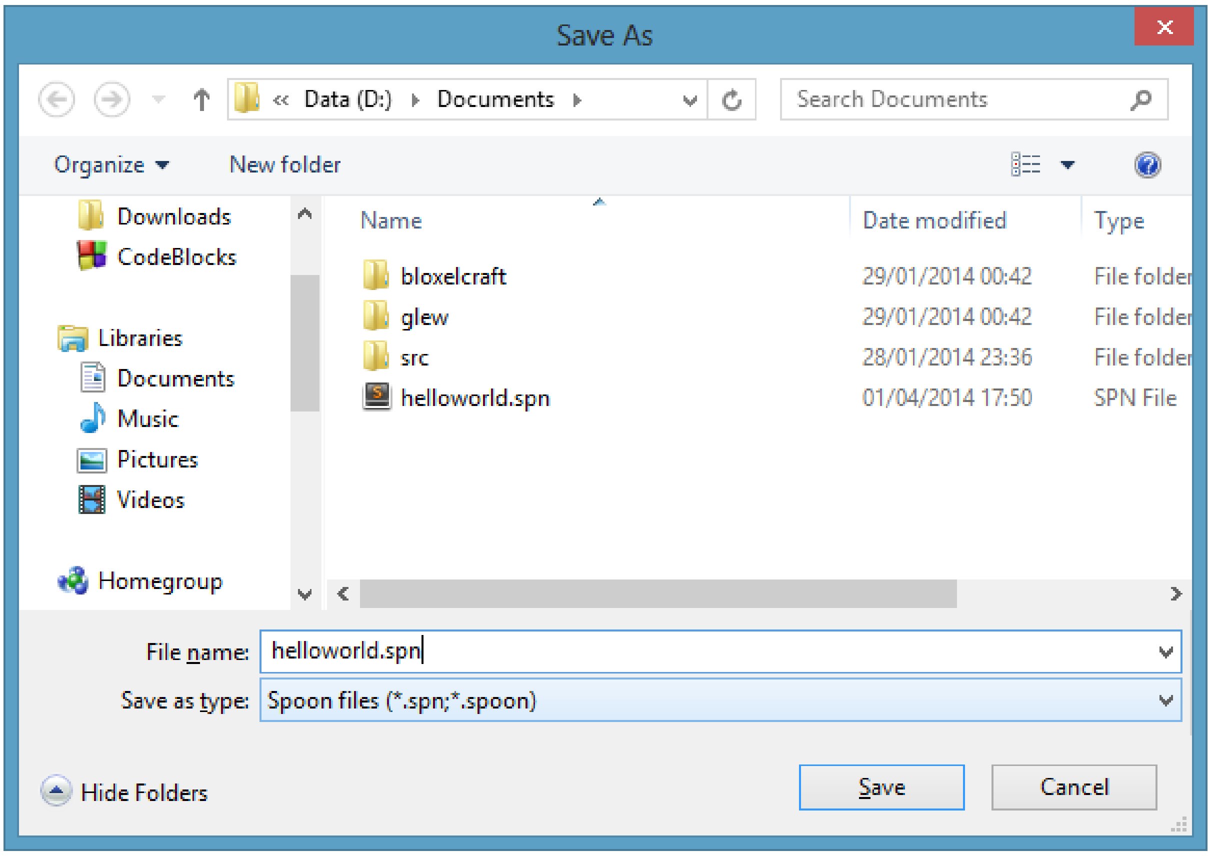
You should see a screen that looks like this.



This is the IDE, or integrated development environment. It's a text editor, where we'll type all of our code. It also makes it much easier to use the tools needed to program in Spoon, such as the compiler and the emulator.

There is a row of buttons at the top of the screen: the first four are to create, open, save and close files. The next four are to build (compile) the current files, set options for compilation, view messages from the compiler about the last build, and run the current file in the emulator.

Click the save button now, and save our new file as "helloworld.spn".



Name	Date modified	Type
bloxelcraft	29/01/2014 00:42	File folder
glew	29/01/2014 00:42	File folder
src	28/01/2014 23:36	File folder
helloworld.spn	01/04/2014 17:50	SPN File

File name: helloworld.spn
Save as type: Spoon files (*.spn;*.spoon)

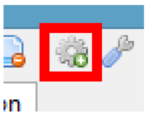
Press save to save the file. Remember the folder you saved it in – you will need it later!

Before we set out, we need an idea of what our program is going to do. Let's start off with something simple: printing text onto a screen.

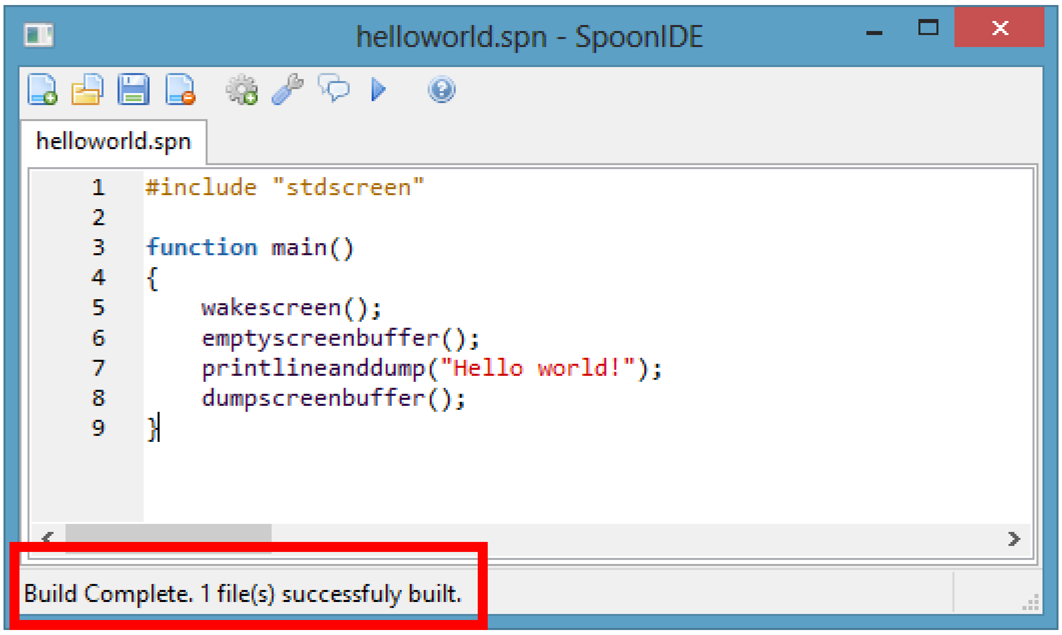
Carefully, type the following into the main textbox in the middle of the screen:

```
#include "stdscreen"

function main()
{
    wakescreen();
    emptyscreenbuffer();
    printlineanddump("Hello world!");
}
```

Once you're certain your code is correct, press the "build" button (the gear).



A message should appear at the bottom of the window, informing you that the file has been successfully built:



```
#include "stdscreen"

function main()
{
    wakescreen();
    emptyscreenbuffer();
    printlineanddump("Hello world!");
    dumpscreenbuffer();
}
```

Build Complete. 1 file(s) successfuly built.

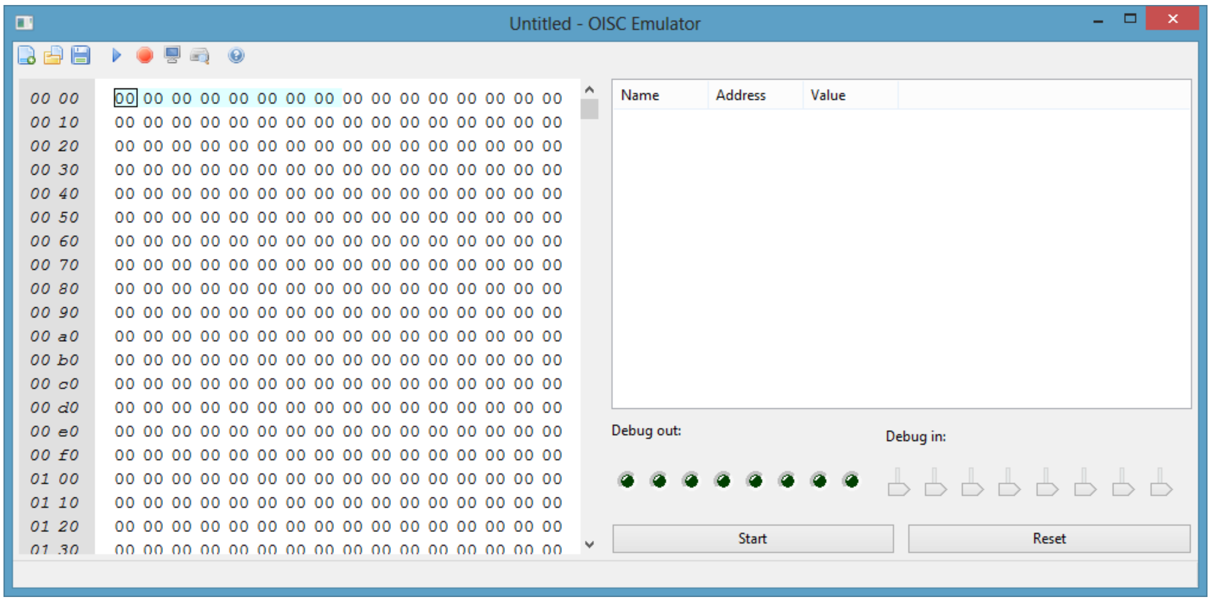
If you see an error message, double-check your code.

The program has now been successfully compiled: this means the compiler has read through the source code (what you just typed in) and turned it into a binary program which the computer can run. We'll see what the program does in a moment.

Getting Things Running

We have a binary file – the compiled program, ready to go. It's not really doing much though, just sort of sitting there.

Open the emulator (emulator.exe):

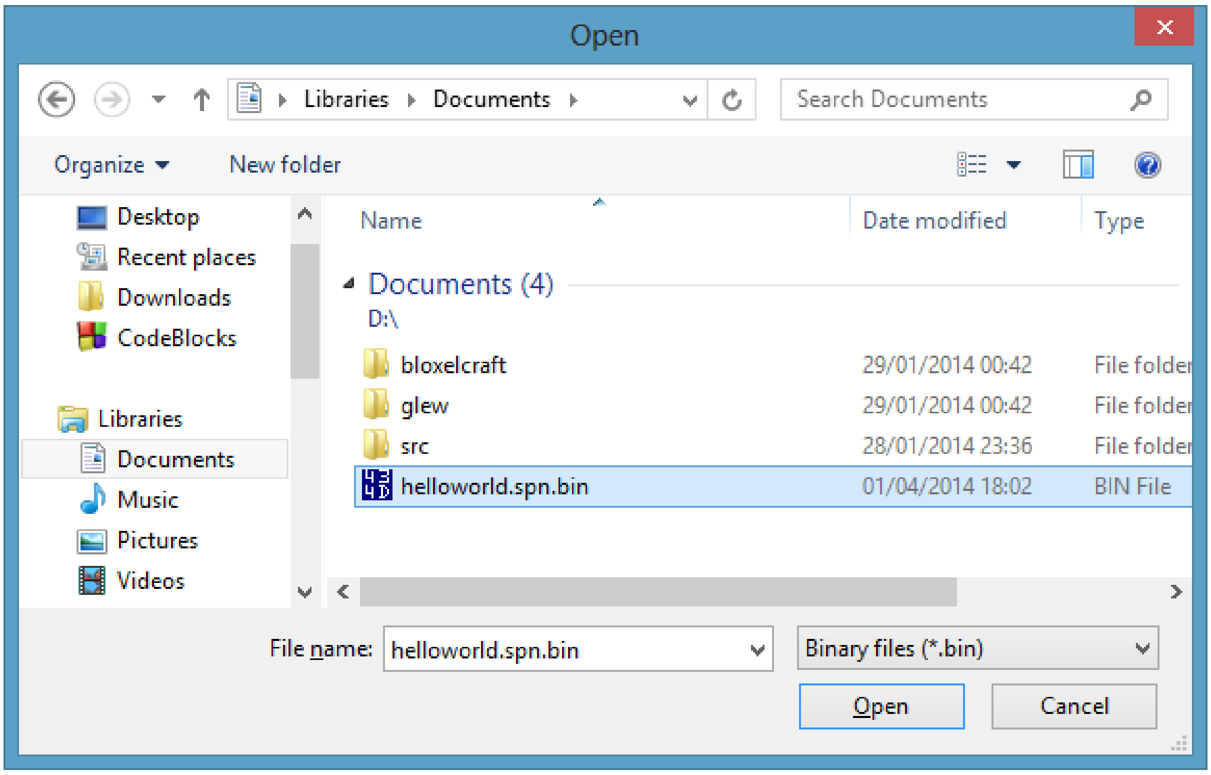


```
00 00   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 10   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 20   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 30   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 40   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 50   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 60   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 70   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 80   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 90   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 a0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 b0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 c0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 d0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 e0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 f0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01 00   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01 10   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01 20   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01 30   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

An emulator is a piece of software that provides the functionality of one machine on another. This piece of software allows us to run compiled Spoon programs on a regular desktop computer, including using hardware such as the screen and the flash storage.

There's a lot to take in here, but the most important part is the large grid of numbers on the left: this displays the contents of ROM and RAM, in hexadecimal. The numbers in the grey section on the left are the address of that line in memory, and the numbers in white are the memory contents. The pale blue highlight shows the location of the instruction pointer (the instruction the machine is currently processing).

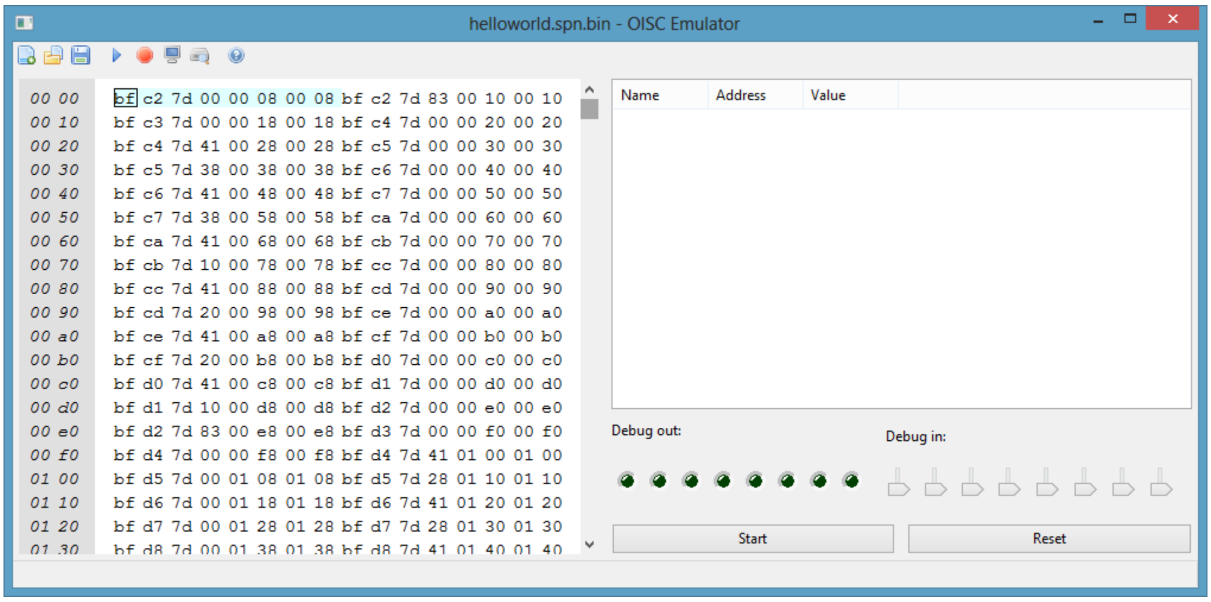
Press the "open" button (the yellow folder), and navigate to the folder where you saved our source file (I told you to remember it!). Find the file with the name "helloworld.spn.bin" (or "(yourname).bin" if you chose a different name) and open it.



Name	Date modified	Type
Documents (4)		
D:\		
bloxelcraft	29/01/2014 00:42	File folder
glew	29/01/2014 00:42	File folder
src	28/01/2014 23:36	File folder
helloworld.spn.bin	01/04/2014 18:02	BIN File

File name: helloworld.spn.bin

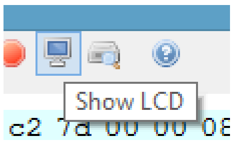
If we look at the left, we can see that the emulator has loaded the machine code into the virtual ROM:



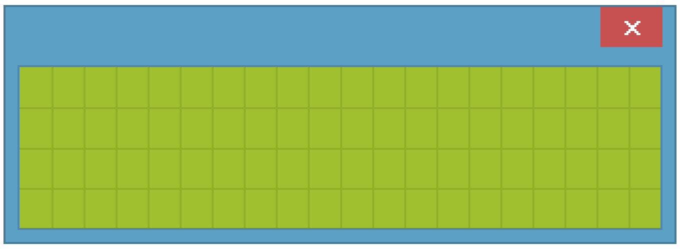
```
00 00   bf c2 7d 00 00 08 00 08 bf c2 7d 83 00 10 00 10
00 10   bf c3 7d 00 00 18 00 18 bf c4 7d 00 00 20 00 20
00 20   bf c4 7d 41 00 28 00 28 bf c5 7d 00 00 30 00 30
00 30   bf c5 7d 38 00 38 00 38 bf c6 7d 00 00 40 00 40
00 40   bf c6 7d 41 00 48 00 48 bf c7 7d 00 00 50 00 50
00 50   bf c7 7d 38 00 58 00 58 bf ca 7d 00 00 60 00 60
00 60   bf ca 7d 41 00 68 00 68 bf cb 7d 00 00 70 00 70
00 70   bf cb 7d 10 00 78 00 78 bf cc 7d 00 00 80 00 80
00 80   bf cc 7d 41 00 88 00 88 bf cd 7d 00 00 90 00 90
00 90   bf cd 7d 20 00 98 00 98 bf ce 7d 00 00 a0 00 a0
00 a0   bf ce 7d 41 00 a8 00 a8 bf cf 7d 00 00 b0 00 b0
00 b0   bf cf 7d 20 00 b8 00 b8 bf d0 7d 00 00 c0 00 c0
00 c0   bf d0 7d 41 00 c8 00 c8 bf d1 7d 00 00 d0 00 d0
00 d0   bf d1 7d 10 00 d8 00 d8 bf d2 7d 00 00 e0 00 e0
00 e0   bf d2 7d 83 00 e8 00 e8 bf d3 7d 00 00 f0 00 f0
00 f0   bf d4 7d 00 00 f8 00 f8 bf d4 7d 41 01 00 01 00
01 00   bf d5 7d 00 01 08 01 08 bf d5 7d 28 01 10 01 10
01 10   bf d6 7d 00 01 18 01 18 bf d6 7d 41 01 20 01 20
01 20   bf d7 7d 00 01 28 01 28 bf d7 7d 28 01 30 01 30
01 30   bf d8 7d 00 01 38 01 38 bf d8 7d 41 01 40 01 40
```

Have a scroll through it, and see if you can understand any of the instructions.

When you're done, press the screen icon at the top:

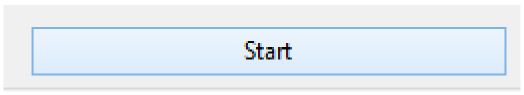


A green rectangular window should appear:

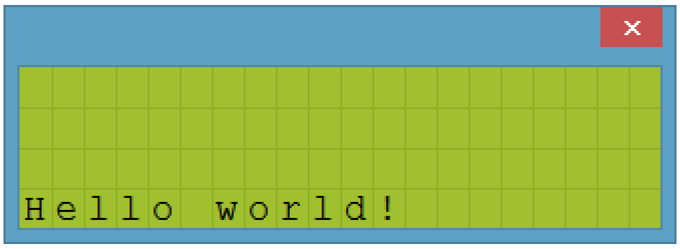


This represents the LCD (liquid crystal display) screen that the computer uses to communicate with the user.

Now… wait for it… reach down to the bottom right of the screen, and press the “Start” button.

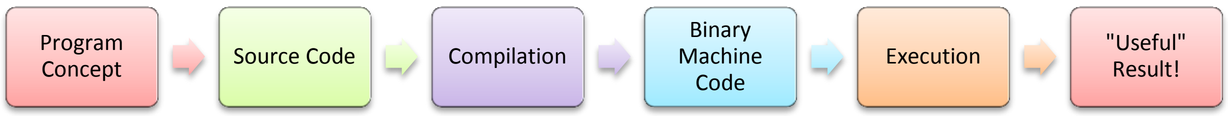


And pow! Like magic!



What Just Happened?

Let’s look at the steps that just happened in our (the programmer’s) head and our computer.



We started out with an idea of what we wanted our software to do – to print some text onto the screen. (We will set loftier goals later!)

We then defined this idea a little more strictly, with source code. Source code is not just any collection of symbols you want to write down: it has a certain grammar to it, like English or German, and like these languages, it is used to communicate: in this case, to communicate our intentions to the compiler, in a way that it understands.

When we pressed the gear button, the IDE (our text editor) sent a message to the compiler, telling it to read our source code file. It reads the file in, line by line and letter by letter, interpreting it in a very specific way and translating it into machine code. Think of it as a very pedantic translator working for you: if it finds any mistakes in your source code, it will report them back to you with a gleeful error message, but it beats writing in machine code.

There is a very real need for this pedantry: unlike human languages, computer languages need to be completely ambiguous (consider a computer trying to find the meaning of an English headline such as “Man chases sheep in car”.)

The compiler spat out the end result: the binary machine code file (*.bin). This is raw instructions, to be interpreted by the computer’s CPU – look back to Section 1 if you want a reminder on how this works.

Then, we loaded the file in the emulator. This takes the place of the CPU: the real computer is made up of thousands of tiny logic gates etched onto small pieces of silicon, which conspire together to carry out 800,000 instructions together. In this case, the CPU has been replaced with a piece of software that reads in the instructions and interprets them in the same way as the real computer.

After all this, we get some text on the screen. Phew!

What Did All That Code Mean Anyway?

In writing our first program, I got you to type the following code into the IDE:

```
#include "stdscreen"

function main()
{
    wakescreen();
    emptyscreenbuffer();
    printlineanddump("Hello world!");
}
```

I kind of glossed over this – apologies, we needed to get things moving! I promise I’m about to make up for it.

Each line of code is an instruction to the compiler, either that it should do something, or that something should happen at this point in the program. Let’s look at the first line:

```
#include "stdscreen"
```

This is called an “include statement”: its purpose is to import previously written code into the current program. A collection of such code is called a library – we are importing the `stdscreen` library into our program, which includes all of the functions (subroutines) for communicating with the display. Libraries can include other libraries: the `stdscreen` header file, for example, also includes `stddefs`, which tells the compiler things like where the input/output registers are in memory.

The next line of code is the start of a function definition:

```
function main()
```

A function is a section of code that can be used repeatedly by other parts of the program – a function called `wash_the_dishes()` might call upon other smaller functions called `run_tap()` and `add_soap()`, for example. We are defining a special function, called `main`: the computer will start execution of our program by jumping to this function. It's known as the entry point.

An opening brace ("`{`") signifies the beginning of the function's body – what follows is a series of instructions, telling the compiler what happens when this function is called upon.

```
wakescreen();
```

This is a function from `stdscreen`: when called, it begins communications with the LCD screen, switching it on and telling it to be ready for further instruction. The compiler finds the code for this function from the `stdscreen` file we included, and tags the compiled machine code for it onto the end of our program, after the main function, so that the main function can call it when needed.

```
emptyscreenbuffer();
```

When we included the stdscreen header, it contained a note for the compiler, telling it to set aside an area of memory big enough for all of the characters that will fit onto the display. A piece of memory set aside like this is called a variable – a program uses variables to "remember" things (keep track of them) while it's going along. The note to the compiler was called a variable declaration (i.e. a variable was declared: specifically, this variable was an *array* of bytes. Check out page 29).

What the `emptyscreenbuffer()` function does is to fill in this space in memory with blank space characters. When the computer is first turned on, there's no guarantee what sort of junk is in memory, and all that junk will end up on the screen if you don't clear it out first.

```
printlineanddump("Hello world!");
```

Now that things are set up, we write the text. This function does two things:

- It takes our string of text (which the compiler has helpfully tucked away in ROM at the end of the program for us) and copies it into the screen buffer. The string is passed as a "parameter" to the function: the function has its own special variable, which the main function sets to the location of this text in memory so that the function knows where to look.
- It sends the entire screen buffer to the screen, character by character. This happens very fast, so looks more or less instant.

A closing brace ("`}`") signifies the end of the main function, and of our program. At this point the compiler inserts a "jump to self" instruction, bringing the CPU to a halt by putting it into a loop.

So, About Those Variables

I briefly mentioned variables, but although I said what they are, I neglected to mention what they're *for.*

Imagine you're a waiter, taking orders at a restaurant. Except you've had a bad day, and you've forgotten you pen and notepad – darn, not again! The first person says they would like a beef and horseradish casserole, followed by a raspberry jelly in the shape of a salmon. The second would like a castle made entirely of marmite and toast. The third…

You go back to the kitchen and shout the orders through the hatch, hoping that someone heard, and then go and find the next hopeful looking customer, forgetting about the ones you just served while their meals are being cooked.

What happened there? The customers (the user) gave you some data, and you set aside some space in your head to store this data. You possibly did some sort of calculation (totting up a running bill), and then shouted the data into the kitchen (output!). Once you were finished with this task, you then forgot about the old data and used the same storage space for your next set of data, from another user.

This is more or less what a computer does, except they were *designed* for it, while your brain evolved to hunt woolly mammoths. While an average person can keep track of around seven objects at a time, a computer can remember thousands, or millions. Each time it needs to remember something new, it sets aside another piece of memory for it. When it's finished with that data, the memory space is free for use again. For the programmer's convenience, each variable can be referred to with a name (`number_of_customers`, `likes_bread`) to help them remember what each variable contains. The computer neither knows nor cares about these names – they are agreed upon only by the programmer and the compiler.

Let's have a look at how this works.

```
#include "stdscreen"

function main()
{
    wakescreen();
    emptyscreenbuffer();
    var int a, b;
    a = 5;
    b = add(a, 1);
    printlineanddump(int2hexstr(a));
    printlineanddump(int2hexstr(b));
}
```

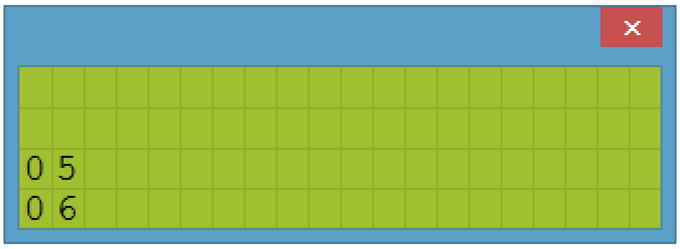
Most of this we are already familiar with – have a look at this line though:

```
var int a, b;
```

This is a variable declaration: we have asked the compiler to set aside room for two integer numbers, which we will call a and b.

```
a = 5;
b = add(a, 1);
```

We then assign `a` the value of five: at this point in the program, the machine code will write a value of five (in binary) into the memory location referred to as `a`. When we assign `b` its value, it first reads a to yield the value five, adds it to one, and stores the result six into the space referred to by `b`. The next two lines print out the variables `a` and `b` (as the numbers they represent, not directly by ASCII value). Let's check that this works:



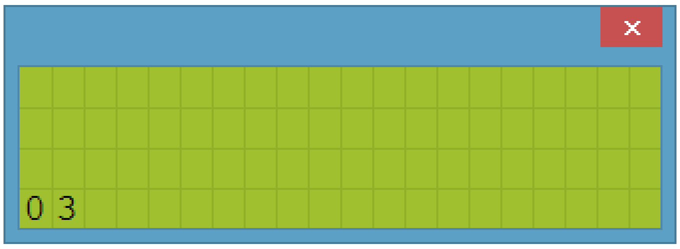
```
05
06
```

Neat!

These variables, a and b, are referred to as local: they are only "visible" within the main function, not outside of it. In some cases, it can make sense to use so-called *global* variables: variables that any function can use. This is often a bad idea, because it's difficult to keep track of where they are used and the code quickly gets messy, but let's look at a simple example:

```
#include "stdscreen"

var int number_of_cakes_eaten;

function eat_cake()
{
    number_of_cakes_eaten = increment(number_of_cakes_eaten);
}

function main()
{
    wakescreen();
    emptyscreenbuffer();
    number_of_cakes_eaten = 0;
    eat_cake();
    eat_cake();
    eat_cake();
    printlineanddump(int2hexstr(number_of_cakes_eaten));
}
```

We have a global variable – the number of cakes we have been eaten. A function (procedure) called `eat_cake()` is called each time a cake is eaten, increasing the cake count by 1. We eat three cakes and then print out the count:



```
03
```

Three cakes! Suppose we were at a party, and lots of people were eating cakes. We could just call the `eat_cake()` procedure lots of times, but what if we could tell the function how many cakes have just been eaten, and let it add it to the running total? It turns out there is a way to do this, called a function parameter. Let's look at the amended function:

```
function eat_cakes(int cakes)
{
    number_of_cakes_eaten = add(number_of_cakes_eaten, cakes);
}
```

`cakes` is a *parameter* to the function: when we call the function, we supply the appropriate value for cakes, and the function behaves accordingly. This works in a very similar way to a variable: the value is put into a specially reserved location in memory that only the function can see.

For example,

```
eat_cakes(5);
```

Would add 5 cakes to the total number of cakes eaten.

Let's make one last change: we want the function to tell us how many cakes have been eaten so far each time we call it.

```
#include "stdscreen"

var int number_of_cakes_eaten;

function int eat_cakes(int cakes)
{
    number_of_cakes_eaten = add(number_of_cakes_eaten, cakes);
    eat_cakes = number_of_cakes_eaten;
}

function main()
{
    wakescreen();
    emptyscreenbuffer();
    number_of_cakes_eaten = 0;
    printlineanddump(int2hexstr(eat_cakes(5)));
    printlineanddump(int2hexstr(eat_cakes(4)));
}
```

The first change we can see is in the opening line of the function definition:

```
function int eat_cakes(int cakes)
```

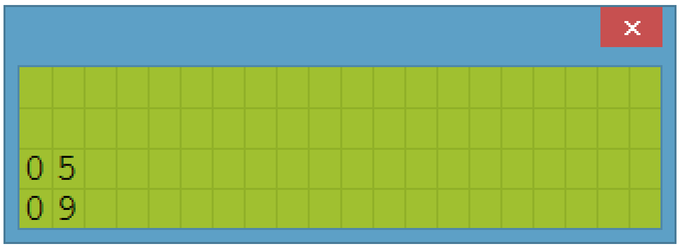
Notice the keyword `int` after `function`: this is called the return type. It tells us that the function has a result, and that the type of that result is an integer. Previously we've left this out, and the compiler has assumed that the type is `void`.

On this line:

```
eat_cakes = number_of_cakes_eaten;
```

We say that the result of the function is equal to the number of cakes that have been eaten.

So we eat 5 cakes, print out how many cakes have been eaten, eat another 4 and print out again.



```
05
09
```

Looks good!

Loops and Branches, and Why You Need Them

Sometimes, we might want to repeat a section of code – if we want to wait until a button is pressed, for example, we can repeatedly check until it is pressed. We might also want to make decisions – is this number too big? Will this file fit onto the hard drive?

If we want to repeat a section of code, we use a loop – so called because the flow of control loops back on itself, repeating the earlier section. For example, waiting for a button to be pressed:

```
while(button_not_pressed())
{
    do_nothing();
}
carry_on();
```

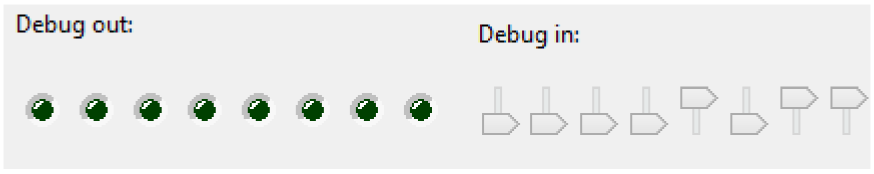
This will repeatedly check whether the button is pressed, and *while* it is it will continue to call the function `do_nothing()`. As soon as this condition turns out to be false (i.e. the button is pressed) it will exit the loop on the next go round, and continue with execution.

Branch is the term for when control flow takes two separate paths – a fork in the road. This will be dependent upon some condition defined by the code. For example, this program:

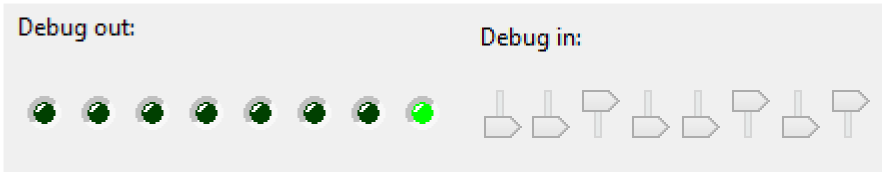
```
#include "stddefs"

const int minimum_needed = 35;

function main()
{
    while (true)
    {
        if (lessthan(read(debugin), minimum_needed))
            output(val(0), debugout);
        else
            output(val(1), debugout);
    }
}
```

`while (true)` means that the rest of a code is in a loop that will run forever, because the loop condition will never be false. What follows inside is a branch – in this case, an `if` statement. If the number read in from the debug input port is less than 35, it will set the debug output port to 0. If the input is greater than or equal to 35, it will follow the `else` branch and set the output to one, turning on an LED.

If we test this with a value of 11 (binary 1011):



But with 37 (100101):



These sorts of decision underpin everything that modern computers do.

More Uses For Loops

In mathematics, there is a sequence of numbers known as the Fibonacci sequence:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

It goes on forever, so we need to be satisfied with only printing the first few numbers.

The recipe for making the Fibonacci sequence is that each number is found by adding the two preceding numbers: 2 is 1+1, 3 is 1+2, 5 is 2+3... so on. How would we write a program find the nth term of this sequence? That is, if we give it the number 3, it will give us the 3rd number (2), or if we give it the number 8, it will give us the 8th number (21).

In essence, what we need to do is to add a pair of numbers together n times. Can we do this with loops? What we need is some sort of variable to store n, and a loop that repeats until n is zero, decrementing each time.

Try the following function:

```
function int fibs(int n)
{
    var int a, b, temp;
    a = 0;
    b = 1;
    // loop n times:
    while (n)
    {
        // calculate the next number in the sequence
        temp = add(a, b);
        a = b;
        b = temp;
        n = decrement(n);
    }
    fibs = a;
}
```

The while loop executes until `n` is non-true, i.e. zero. For each count, it adds the two current numbers together to find the next one, stores this result in `temp`, and then shifts the values down into `a` and `b`. The result is the Fibonacci sequence. To consider why `temp` is needed, think about what would happen if you assigned the result to `b` before you tried to move the value of `b` to `a`!

Notice the comments, on the lines starting with double slashes. A sprinkling of these makes your code much more understandable.

Writing Programs with Fork OS (Let's Get Forking!)

At this point, we've seen most of the basic features the language has to offer (for a more complete reference, see section 3). However, all of our programs so far have been limited by the fact they've been running on bare hardware: no software support for the keyboard or flash storage, no use of the filesystem for storing user data, and none of the handy helper functions (other than those in the libraries).

What is an operating system?

An operating system has two main purposes: to obscure the complexity of the hardware, and to add functionality for the programmer that the hardware does not naturally have. Fork OS is an operating system written in Spoon for the Fork machine – it provides drivers for the keyboard, screen and flash storage (allowing you to use them directly from your code), a file system for storing user documents on programs, facilities for loading programs off of disk and a small suite of utility programs for managing and editing files, on the machine itself. Much of this functionality has been factored out into the libraries.

The Spoon compiler is already set up to compile programs for Fork OS. First though, a brief description of how these programs are actually loaded:

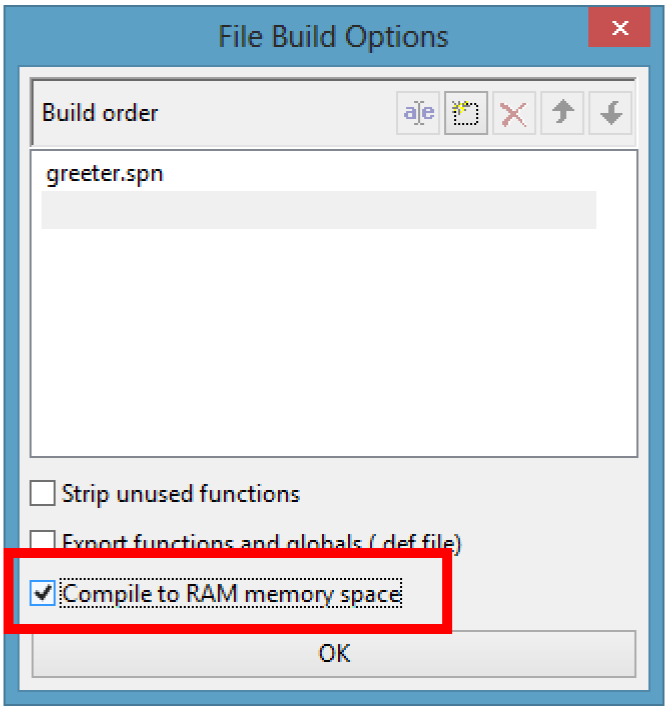
- The boot code lives in ROM, in the first 32KiB of the memory map. This includes all of the library code, drivers, the filesystem code, and code for actually loading the programs.
- The next 16KiB following ROM is RAM (starting at 0x8000 -> 0xbfff). The bootcode is allocated memory starting at the end of RAM, such that the maximum possible contiguous space is available at the start.

- The runprogram() function is called with a filename as an argument. It looks this name up in the file allocation table (FAT). If it doesn't find it, the computer is reset. Otherwise:
- Details are retrieved from the FAT of the program's position and size on disk.
- The precompiled binary file is compiled verbatim into memory, starting at offset 0x8000. It was previously compiled to base all of its addresses from this offset.
- The program's variables are allocated from the end of the program's own code, growing towards the end of RAM.
- Execution jumps to 0x8000 and the program takes over. It is the program's duty to return control to the shell program when it exits.

So, two things:

- The program needs to be compiled to fit into the RAM memory space (0x8000 onwards) rather than the ROM (0x0000 to 0x7bff)
- It needs to know the locations of all the functions and any global state variables provided by the operating system.

This means you need to do two things in order to compile programs for Fork OS; you will need to go to the build options for your current file, and select "compile to RAM memory space":


File Build Options

Build order

greeter.spn

[ ] Strip unused functions

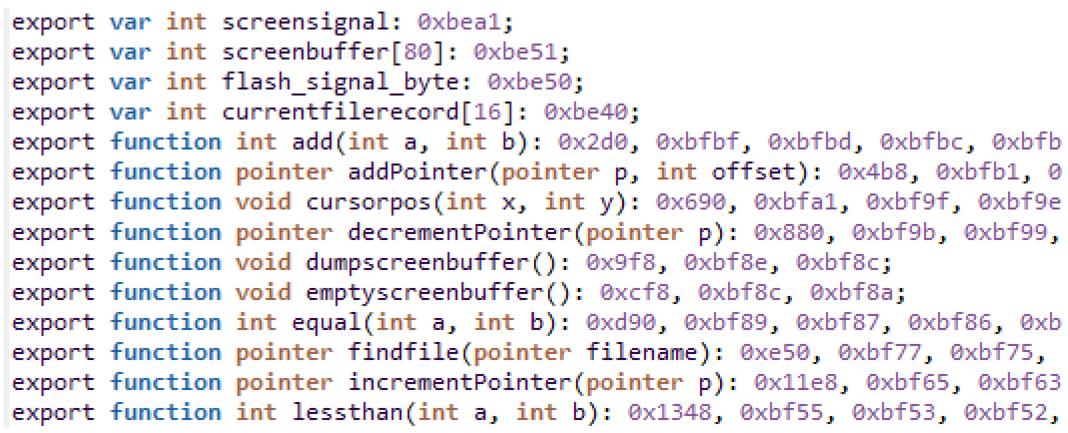
[ ] Export functions and globals (.def file)

[✓] Compile to RAM memory space

You will also need this file:

```
#include "forkos.spn.bin.def"
```

Which is automatically generated by the compiler, and contains the following definitions:



```
export var int screensignal: 0xbea1;
export var int screenbuffer[80]: 0xbe51;
export var int flash_signal_byte: 0xbe50;
export var int currentfilerecord[16]: 0xbe40;
export function int add(int a, int b): 0x2d0, 0xbfbf, 0xbfbd, 0xbfbc, 0xbfb
export function pointer addPointer(pointer p, int offset): 0x4b8, 0xbfb1, 0
export function void cursorpos(int x, int y): 0x690, 0xbfa1, 0xbf9f, 0xbf9e
export function pointer decrementPointer(pointer p): 0x880, 0xbf9b, 0xbf99,
export function void dumpscreenbuffer(): 0x9f8, 0xbf8e, 0xbf8c;
export function void emptyscreenbuffer(): 0xcf8, 0xbf8c, 0xbf8a;
export function int equal(int a, int b): 0xd90, 0xbf89, 0xbf87, 0xbf86, 0xb
export function pointer findfile(pointer filename): 0xe50, 0xbf77, 0xbf75,
export function pointer incrementPointer(pointer p): 0x11e8, 0xbf65, 0xbf63
export function int lessthan(int a, int b): 0x1348, 0xbf55, 0xbf53, 0xbf52,
```

…which allows your program to make calls to all of these ROM-resident functions. You can find this file in the forkos folder in the compiler's directory; make sure a copy is in the same folder as your project so the compiler will find it.

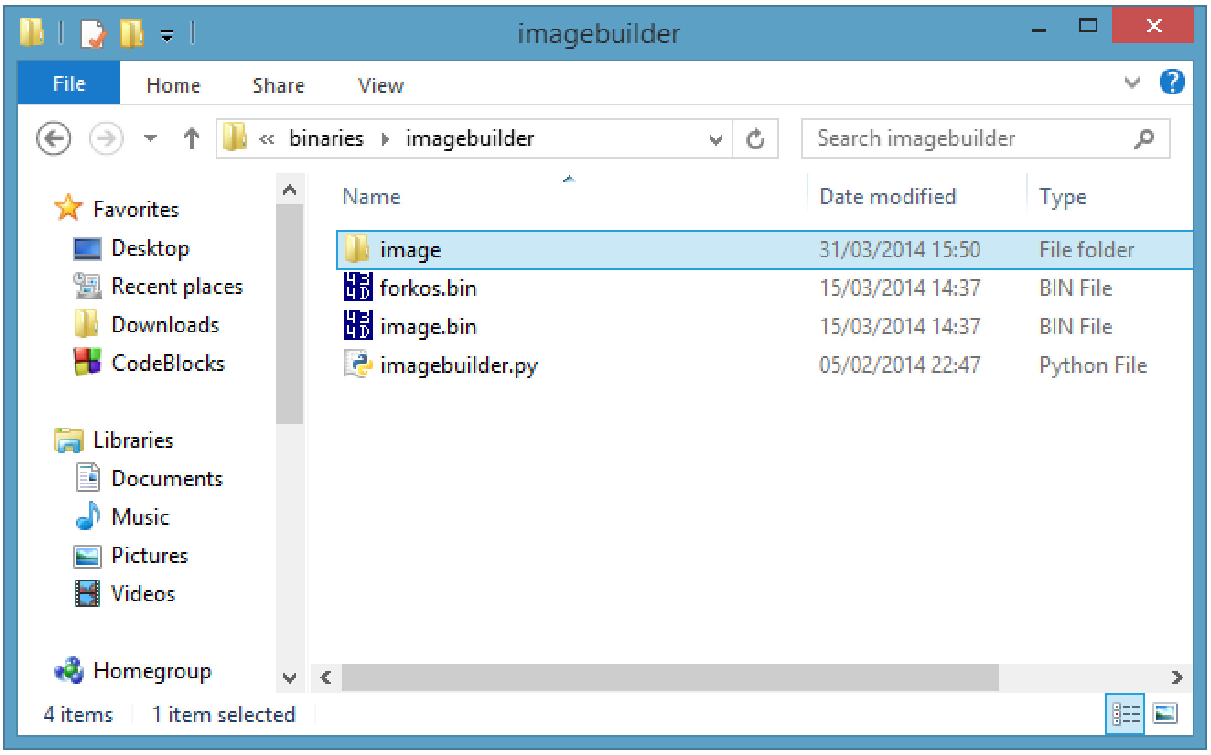
Let's try a simple example:

```
#include "stddefs"
#include "forkos.spn.bin.def"

function main()
{
    var char name[21];
    name[20] = 0;
    var pointer tableloc = findfile("name");
    if (equal(first(tableloc), 0xff))
        createfile("name", 1);
    readflashnbytes(currentfilerecord[0], pair(currentfilerecord[1],
currentfilerecord[2]), name, 20);
    while (true)
    {
        printlineanddump("Last visitor was:");
        printlineanddump(name);
        printlineanddump("What is your name?");
        strcpy(name, getstring());
        printlineanddump("Hello,");
        printlineanddump(name);
        erasefilesectors(0, 1);
        programflashnbytes(currentfilerecord[0],
pair(currentfilerecord[1], currentfilerecord[2]), name, 20);
   }
}
```

This will open the file "name" (or create it if it doesn't exist), and load from it the name of the last person to use the program. It will print who the last user was, ask you your name, say hello and store your name in the file for future use, before running through the loop again. Let's take it for a spin.

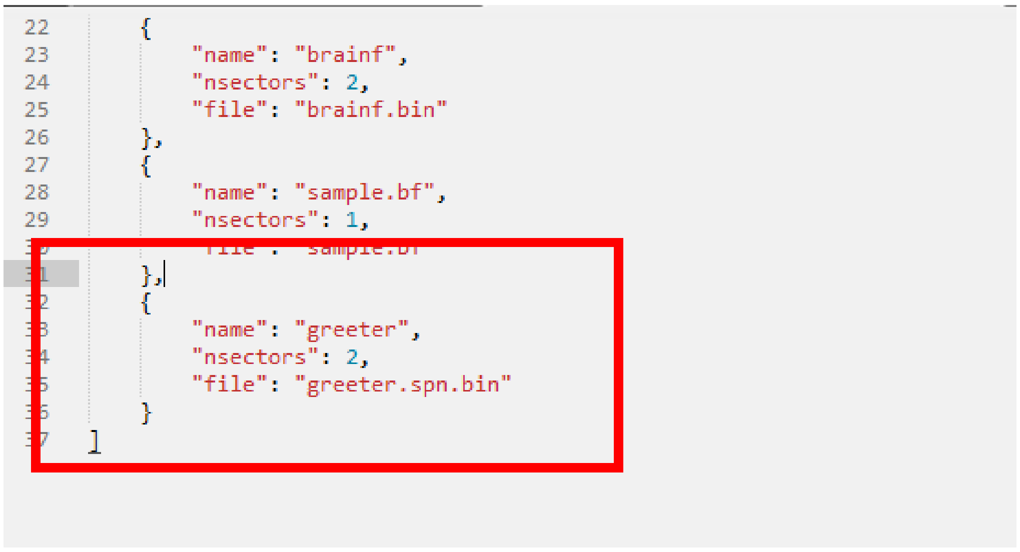
Notice how we don't need to call any of the setup code for the screen or other peripherals ; the operating system has already done this.

Set the compile to RAM option and press build. Once you have the binary file, copy it into the “image” directory:



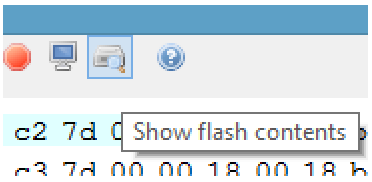
Name	Date modified	Type
image	31/03/2014 15:50	File folder
forkos.bin	15/03/2014 14:37	BIN File
image.bin	15/03/2014 14:37	BIN File
imagebuilder.py	05/02/2014 22:47	Python File

and add the following entry to “index.json”:

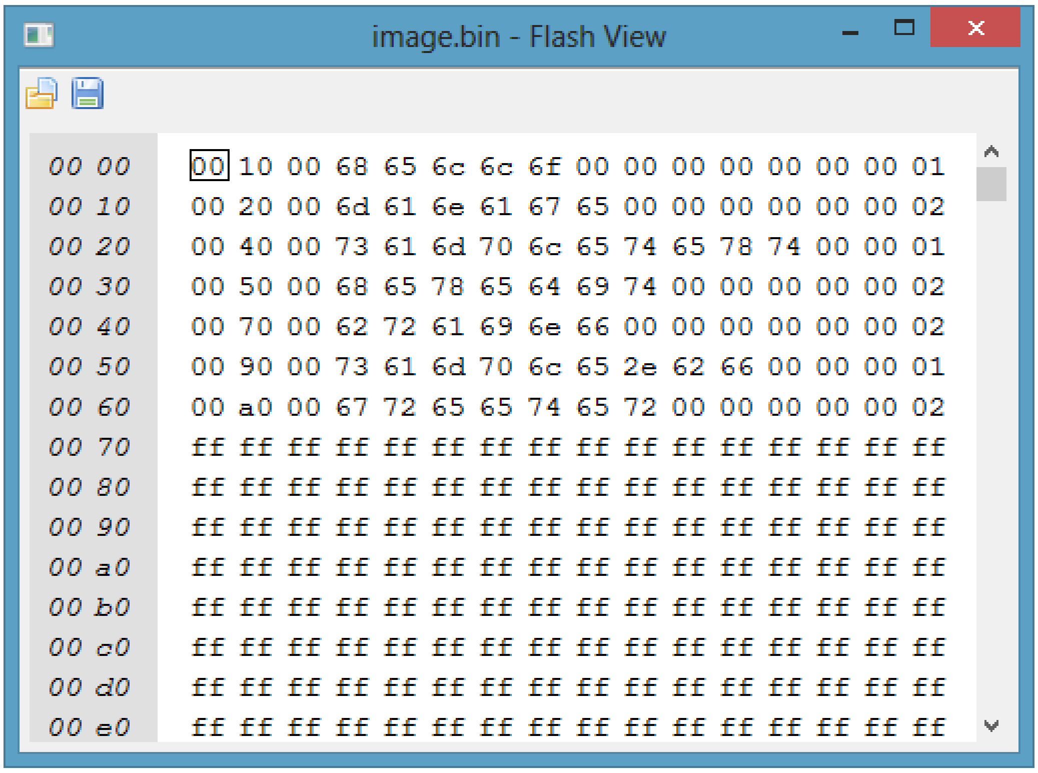


```
    {
        "name": "brainf",
        "nsectors": 2,
        "file": "brainf.bin"
    },
    {
        "name": "sample.bf",
        "nsectors": 1,
        "file": "sample.bf"
    },
    {
        "name": "greeter",
        "nsectors": 2,
        "file": "greeter.spn.bin"
    }
]
```

Drag the image folder onto the imagebuilder script (make sure you have Python 3 installed) and open the emulator. Open forkos.bin in the emulator and then click on the following button to open the flash view:

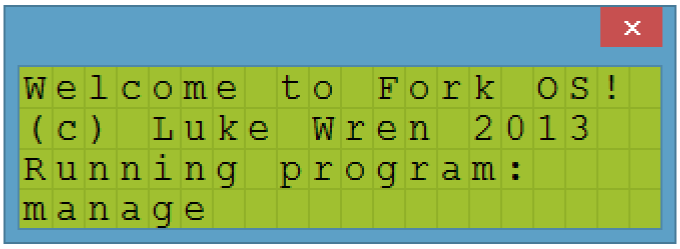


In the new window, press open and load “image.bin”. This is the filesystem disk image, containing all of the files and programs.



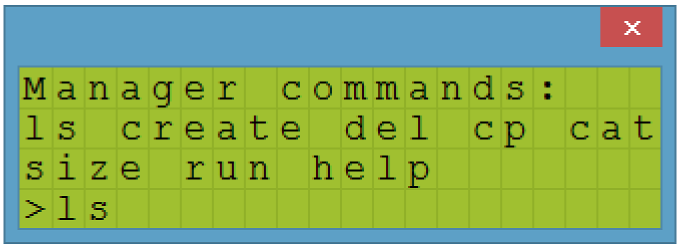
```
00 00   00 10 00 68 65 6c 6c 6f 00 00 00 00 00 00 00 01
00 10   00 20 00 6d 61 6e 61 67 65 00 00 00 00 00 00 02
00 20   00 40 00 73 61 6d 70 6c 65 74 65 78 74 00 00 01
00 30   00 50 00 68 65 78 65 64 69 74 00 00 00 00 00 02
00 40   00 70 00 62 72 61 69 6e 66 00 00 00 00 00 00 02
00 50   00 90 00 73 61 6d 70 6c 65 2e 62 66 00 00 00 01
00 60   00 a0 00 67 72 65 65 74 65 72 00 00 00 00 00 02
00 70   ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
00 80   ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
00 90   ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
00 a0   ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
00 b0   ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
00 c0   ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
00 d0   ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
00 e0   ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
```

Open the LCD and then, on the main window, press start; Fork OS will boot and begin running the shell program.

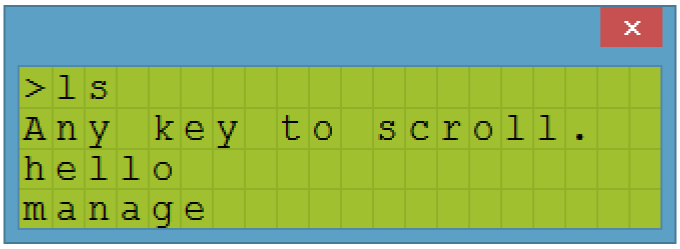


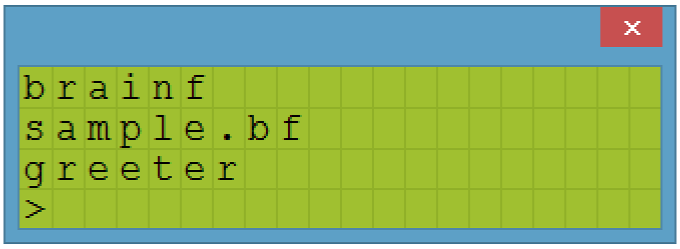
```
Welcome to Fork OS!
(c) Luke Wren 2013
Running program:
manage
```

The loading should be finished by now. Type “ls” to list files and scroll down to check that our program is on disk.

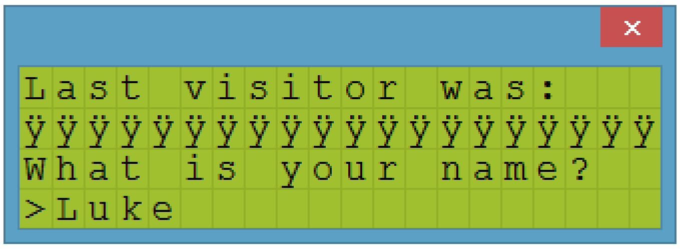


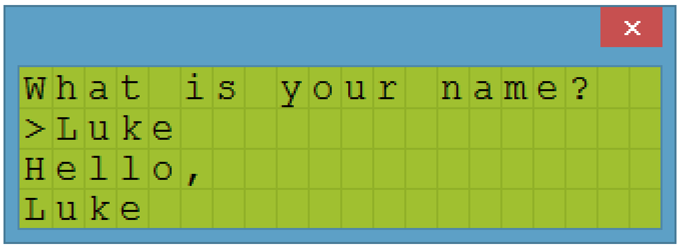
```
Manager commands:
ls create del cp cat
size run help
>ls
```

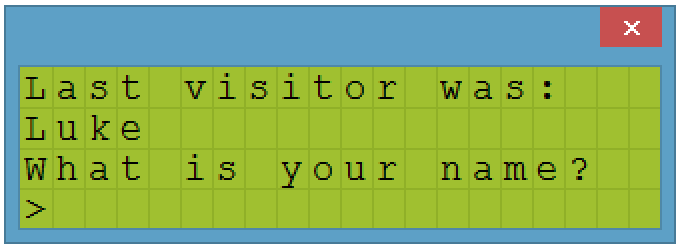

```
>ls
Any key to scroll.
hello
manage
```


```
brainf
sample.bf
greeter
>
```

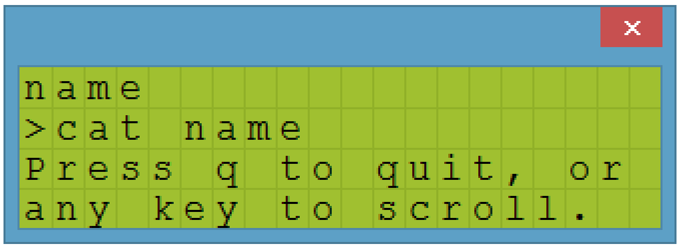
We can see it at the bottom, so type “run greeter” to run it:

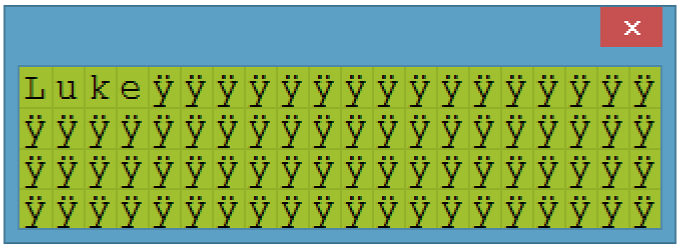

```
Last visitor was:
ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
What is your name?
>Luke
```


```
What is your name?
>Luke
Hello,
Luke
```


```
Last visitor was:
Luke
What is your name?
>
```

If we reboot the computer and type “cat name”:


```
name
>cat name
Press q to quit, or
any key to scroll.
```



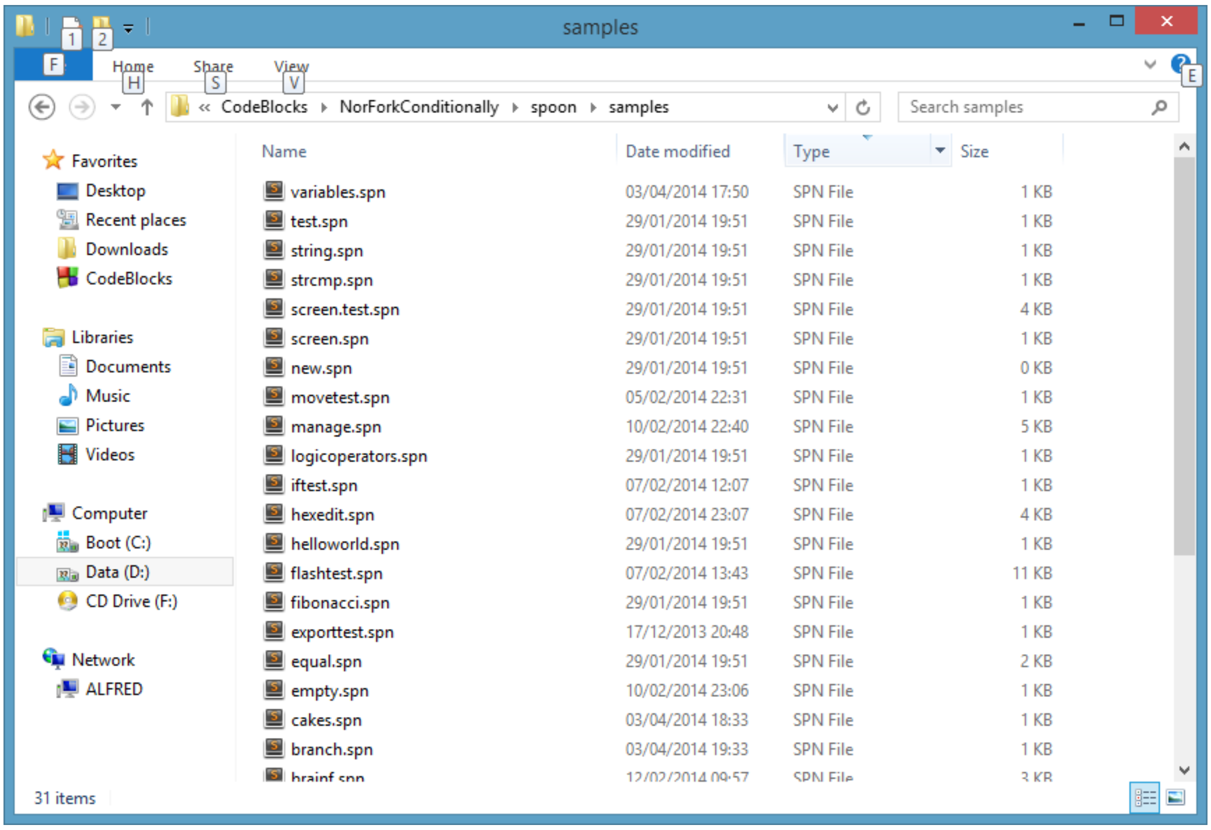
```
Lukeÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
```

We can see that the last person's name is stored on disk for the next user.

What Now?

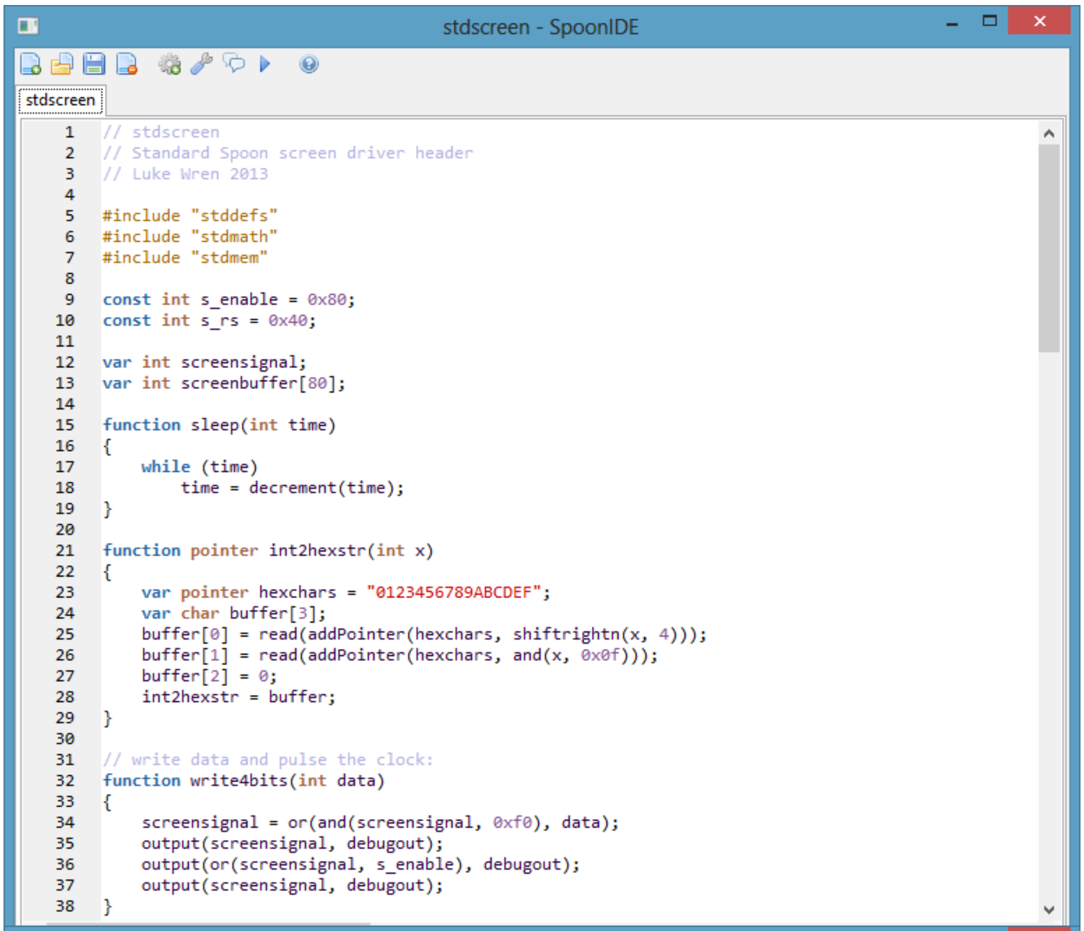
Now that you can write programs in Spoon and use Fork OS, what's the next step?

Try browsing some of the sample programs:

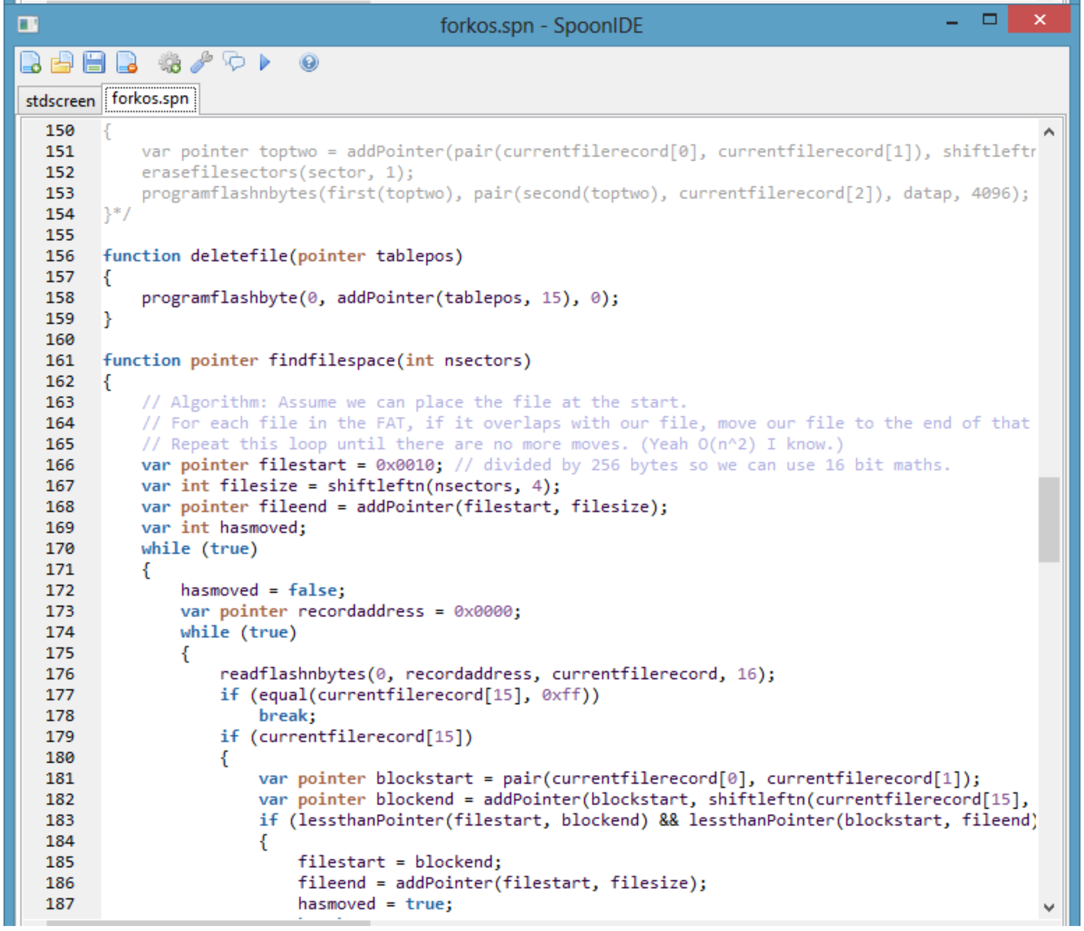


Name	Date modified	Type	Size
variables.spn	03/04/2014 17:50	SPN File	1 KB
test.spn	29/01/2014 19:51	SPN File	1 KB
string.spn	29/01/2014 19:51	SPN File	1 KB
strcmp.spn	29/01/2014 19:51	SPN File	1 KB
screen.test.spn	29/01/2014 19:51	SPN File	4 KB
screen.spn	29/01/2014 19:51	SPN File	1 KB
new.spn	29/01/2014 19:51	SPN File	0 KB
movetest.spn	05/02/2014 22:31	SPN File	1 KB
manage.spn	10/02/2014 22:40	SPN File	5 KB
logicoperators.spn	29/01/2014 19:51	SPN File	1 KB
iftest.spn	07/02/2014 12:07	SPN File	1 KB
hexedit.spn	07/02/2014 23:07	SPN File	4 KB
helloworld.spn	29/01/2014 19:51	SPN File	1 KB
flashtest.spn	07/02/2014 13:43	SPN File	11 KB
fibonacci.spn	29/01/2014 19:51	SPN File	1 KB
exporttest.spn	17/12/2013 20:48	SPN File	1 KB
equal.spn	29/01/2014 19:51	SPN File	2 KB
empty.spn	10/02/2014 23:06	SPN File	1 KB
cakes.spn	03/04/2014 18:33	SPN File	1 KB
branch.spn	03/04/2014 19:33	SPN File	1 KB
brainf.spn	12/02/2014 09:57	SPN File	3 KB

Have a look through the operating system and library code:



```
// stdscreen
// Standard Spoon screen driver header
// Luke Wren 2013

#include "stddefs"
#include "stdmath"
#include "stdmem"

const int s_enable = 0x80;
const int s_rs = 0x40;

var int screensignal;
var int screenbuffer[80];

function sleep(int time)
{
    while (time)
        time = decrement(time);
}

function pointer int2hexstr(int x)
{
    var pointer hexchars = "0123456789ABCDEF";
    var char buffer[3];
    buffer[0] = read(addPointer(hexchars, shiftrightn(x, 4)));
    buffer[1] = read(addPointer(hexchars, and(x, 0x0f)));
    buffer[2] = 0;
    int2hexstr = buffer;
}

// write data and pulse the clock:
function write4bits(int data)
{
    screensignal = or(and(screensignal, 0xf0), data);
    output(screensignal, debugout);
    output(or(screensignal, s_enable), debugout);
    output(screensignal, debugout);
}
```



```
{
    var pointer toptwo = addPointer(pair(currentfilerecord[0], currentfilerecord[1]), shiftleftr
    erasefilesectors(sector, 1);
    programflashnbytes(first(toptwo), pair(second(toptwo), currentfilerecord[2]), datap, 4096);
}*/

function deletefile(pointer tablepos)
{
    programflashbyte(0, addPointer(tablepos, 15), 0);
}

function pointer findfilespace(int nsectors)
{
    // Algorithm: Assume we can place the file at the start.
    // For each file in the FAT, if it overlaps with our file, move our file to the end of that
    // Repeat this loop until there are no more moves. (Yeah O(n^2) I know.)
    var pointer filestart = 0x0010; // divided by 256 bytes so we can use 16 bit maths.
    var int filesize = shiftleftn(nsectors, 4);
    var pointer fileend = addPointer(filestart, filesize);
    var int hasmoved;
    while (true)
    {
        hasmoved = false;
        var pointer recordaddress = 0x0000;
        while (true)
        {
            readflashnbytes(0, recordaddress, currentfilerecord, 16);
            if (equal(currentfilerecord[15], 0xff))
                break;
            if (currentfilerecord[15])
            {
                var pointer blockstart = pair(currentfilerecord[0], currentfilerecord[1]);
                var pointer blockend = addPointer(blockstart, shiftleftn(currentfilerecord[15],
                if (lessthanPointer(filestart, blockend) && lessthanPointer(blockstart, fileend)
                {
                    filestart = blockend;
                    fileend = addPointer(filestart, filesize);
                    hasmoved = true;
```

And for any details about the language, there's section 3.

Section 3: Language Reference

This section is for looking up any details of the language – whether it be the built-in functions, the details of the syntax, or the grubby gubbins of the semantics – and also for deeper learning. Each aspect of the language is examined in turn, usually with a healthy spoonful of examples. Enjoy!

Syntax

The syntax of the language is defined by the following BNF/regular expressions:

```
<program> ::= {<constdef>|<funcdef>|<macrodef>|<vardecl>}

<constdef> ::= `const` <type> <name> `=` <expression> `;`

<funcdef> ::= `function` [<type>] <name> `(` <type> <name> {`,`
<type> <name>} `)` <block>

<type> ::= `int`|`pointer`|`void`

<macrodef> ::= `macro` <name> `(` <name> {`,` <name>} `)` <block>

<block> ::=  <statement>
         |   `{` {<vardecl>} {<statement>} `}`

<vardecl> ::= <type> <varname> {`,` <varname>} `;`

<varname> ::= <name> [`[` <number> `]`] [`=` <expression>]

<statement> ::=  <name> `(` <expression> {`,` <expression>} `)` `;`
             |   <name> `=` <expression> `;`
             |   `if` `(` <expression> `)` <block> [`else` <block>]
             |   `while` `(` <expression> `)` <block>
             |   <name> `:`
             |   `goto` <name> `;`
             |   (`break` | `continue` | `return`) `;`



<expression> ::= <value> [(`&&` | `||`) <expression>]

<value> ::= `!`<value>
         |  `(`<expression>`)`
         |  <name>
         |  <name> `(`<expression> {`,` <expression>}`)`
         |  <name> `[`<number>`]`
         |  <string>
         |  `{`<number> {`,` <number>} `}`
         |  <number>

<name> ::= [a-zA-Z_][a-zA-Z0-9_]*

<number> ::= (0x[0-9a-fA-F]+)|([0-9]+)
          |  `'`<char>`'`
```

```
<string> ::= \"[^\"]*\"
```

Where non-terminals are in `<angle-brackets>`, terminals are in `` `backquotes` ``, `[x]` represents optional `x` and `{x}` represents optional multiple `x` (a Kleene closure).

Identifiers and operators are all case sensitive.

A program is a set of declarations: of functions, of macros, of variables and of constants. The only stipulation is that there must exist a function called `main`, which the linker will use as the program entry point. `main` takes no arguments and returns `void`.

Types

There are 3 main types: `void`, `int` and `pointer`. You can also use the name `char` instead of `int`, or `int16` instead of `pointer`, but they are logically and internally the same.

`void` is a pseudo-type, and represents the result of a function that does not return anything. You can define a variable of type void, but trying to assign anything but the result of a void function to it will generate a type error. (Why you would want to do this is beyond me, but it is logically consistent). It has a size of 0 bytes.

`int` has a size of one byte (8 bits). It is the basic unit of arithmetic, and can also represent a single ASCII character. The nickname `char` is a completely equivalent type, but is a little more natural when writing text processing code (a la C).

`pointer` is 2 bytes long, or 16 bits. It refers to a single 8-bit word of machine memory, or can be used to represent a large integer, such as the size of a file in bytes (hence the pseudonym `int16`).

The only compound type is the array. If you need to represent a complex data structure (like C's `struct`), use an array.

Variable Declarations and Scope

Variable declarations can take place anywhere inside a block, but scoping is at a block-level – this has the effect that all variable declarations are processed upon entering a block.

Multiple variables of the same type can be declared at once, and each variable can be initialised with any valid expression:

```
var int x = 5, y = f(x);
```

The initialisations are guaranteed to be performed left-to-right, so f will receive an argument of 5 in the above example.

While the declarations are processed as though they were at the top of a block, the initializations are compiled in the position they are written, so there is no loss of sequencing.

Variables are lexically scoped, as per Algol, C, Haskell and Common Lisp. For example:

```
function main()
{
    var int a, b, c;
    a = 5;
    if (true)
    {
        var int a;
        a = increment(a);
    }
    output(a, debugout);
}
```

Will result in an output of 5, because the new `a` defined inside the `if` block “shadows” the old one, so the original `a` is unchanged.

Arrays

The language allows you to define arrays:

```
var int arr[5];
```

An area of size n * sizeof(type) bytes is allocated by the linker and guaranteed not to be touched by other variables for the duration of the current block, e.g. an array of 5 pointers would be 10 bytes. You can assign into arrays if you have a constant offset:

```
arr[3] = f(x);
```

The following, however, is invalid if x is a variable:

```
arr[x] = f(x);
```

As addition of pointers is a library function rather than a language builtin. (It is not possible to generate the above code statically). This can be written as the following:

```
write(addPointer(arr, x), f(x));
```

Using addPointer() from the stdmath library.

Array initialisers are not supported:

```
var int numbers[5] = {1, 2, 3, 4, 5};
```

Because this list syntax is actually a special case of a string literal, so is of type `pointer` (consider that the table must be stored in ROM).

The following:

```
var pointer numbers = {1, 2, 3, 4, 5};
```

And

```
var int numbers[5];
memcpy(numbers, {1, 2, 3, 4, 5}, 5);
```

Are both perfectly valid (using memcpy from the stdmem library).

Strings

Strings are constant arrays in ROM. When evaluated in expressions, they result in pointers. Attempting to write to these pointers is valid code, but will not result in any change (it’s called read only memory for a good reason).

These:

```
var pointer str = "Hello!";
```

And

```
var pointer str = {'H', 'e', 'l', 'l', 'o', 33, 0};
```
Are the same (ASCII, null-terminated).

String literals are immutable (as a consequence of the laws of physics as well as language semantics) – in order to perform mutating string operations you must first copy the string into a memory buffer using `strcpy()` or similar.

```
var char buffer[64];
var pointer p = buffer;
p = strcpy(p, "I'm going to ");
p = strcpy(p, "concatenate some strings!");
println(buffer);
```

Functions

Functions are defined with the following syntax:

```
function int f(int x)
{
   f = increment(shiftleft(x));
}
```

Note that "BASIC-style" returns are used (assign to function name). The `return` statement does exist, but it takes no arguments and simply exits the function.

The return type is optional –if omitted, the function returns void.

Arguments are passed by value, so the following:

```
function int sumto(n)
{
     sumto = 0;
     while (n)
     {
        sumto = add(sumto, n);
        n = decrement(n);
     }
}
```
Does not modify the original expression for n.

Recursion is not supported, as there is no machine stack – however, any recursive algorithm can be easily converted into an iterative one through the use of a stack, and stacks are easy to implement with pointers. Indirection to the rescue.

Memory allocated in one function is guaranteed to not be touched by another function (unless other functions do pointer arithmetic and don't use bounds checking). This means all variables are the equivalent of `static` variables in C. See the following:

```
function int howManyTimesHaveIBeenCalled(int initialise)
{
     if (initialize)
```

```
        howManyTimesHaveIBeenCalled = 0;
    else
        howManyTimesHaveIBeenCalled = 
increment(HowManyTimesHaveIBeenCalled);
}
```

(This `static`ness includes the function variable). After having been called initially with an argument of true, the above function will return 1, then 2, then 3… etc.

Operators and Operator Precedence

There is no operator precedence. The only operators which are syntactically separate, as opposed to builtins using the function call syntax, are `&&` and `||`, which are really control flow operators; they operate based on short-circuit behaviour, as in C, Python, Lua and other languages:

- `&&`: evaluate the first argument. If it is true, evaluate the second argument and return it. If the first argument is false, return false immediately.
- `||`: evaluate the first argument. If it is true, return true immediately. If it is false, evaluate the second argument and return it.

These are useful if you want to make a decision based on a function of a pointer, but are not sure whether the pointer is valid(non-null):

```
if (pobject && some_function_of(pobject))
    do_something_to(pobject);
```

The function of the object will only be evaluated if the pointer is non-null.

These operators have equal precedence, and are left-associative. Precedence can be changed by the use of brackets to group expressions (see the logic operator test program).

Control Flow

Besides && and ||, the available control flow keywords are `if`, `else`, `while`, `break`, `continue`, `return` and `goto`. These are almost entirely “borrowed” from C.

`if` and `else` work as expected. The `else` clause is optional. If the result of the `if` expression is non-zero, then the `if` branch is taken. If false, the `else` branch is taken, or the `if` branch is simply skipped if the `else` clause is omitted. The only quirk is that, if the expression results in a multi-byte value (i.e. a pointer), the decision is made based on the first (most significant) byte only. Use the `first()` and `second()` builtins if this is not the behaviour you’re after.

As a consequence of the fact that a block can be either a statement or multiple statements between braces, we can “chain up” `if`s and `else`s to make an `else if`:

```
if (a)
    func_a();
else if (b)
    func_b();
else
    func_c();
```

There is in fact no `else if` keyword. If the above were written out fully, it would look like this:

```
if (a)
```

```
{
    func_a();
}
else
{
    if (b)
    {
        func_b();
    }
    else
    {
        func_c();
    }
}
```

while loops run the loop body for as long as the result of the expression is non-zero (with the same caveat as the if expression). If the expression is zero upon the first evaluation, the loop body is not entered at all. The continue statement jumps back up to the test expression (e.g. when we want to move on the the next item in a list) and the break statement exits the current loop.

The return statement exits a function (jumps to the epilogue) but, as noted in the function specification, it does not take an argument, as value returns are achieved by assigning to the function variable.

goto jumps to an address or label:

```
loop:
    x = increment(x);
    goto loop;
```

gotos are not very useful on their own, and in fact it is generally best to avoid them (see: Djikstra, Goto Considered Harmful). However, in conjunction with macros and the existing control flow operators, it is possible to implement entirely new control flow operators from within the language, as per TAGBODYs and GOs in Lisp.

Labels follow lexical scope, in the same way as variables.

Macros

Macros allow frequently-used blocks of code to be defined, and then used in-line.

For example:

```
macro output(src, dest)
{
    var int temp;
    nfc(temp, val(0xff));
    nfc(temp, src);
    nfc(dest, temp);
}
```

This macro from stddefs is used to output a value to an output port: this is a frequent operation, and code becomes much more descriptive as a result of the macro's use.

Each “call” to a macro is expanded inline. This means they can take up more space than functions because of repetition, but the performance is usually higher. Furthermore, each time a macro argument is referred to within the macro body, this is also an inline substitution: this may have unintended side effects, as functions passed as macro arguments can be called multiple times.

Builtin Functions

To make the language useful, a number of functions are built-in. These are usually sufficiently small to be generated in-place (and will be, for reasons of speed), and so do not use the regular calling convention.

Name (Signature)	Description
`int and(int, int)`	Returns the bitwise and of two integers. E.g. and(3, 5) = 1.
`int andnot(int, int)`	Returns the bitwise and of the first integer and the complement of the second. This takes only three machine instructions (the same as a byte copy).
`int decrement(int)`	Returns the value of the argument – 1. Implemented as a table lookup.
`int first(pointer)`	Returns the first (big endian => more significant) byte of a pointer.
`void nfc(ref, ref)`	Compiles down to one single NFC machine instruction. This one is a special case – arguments are evaluated as follows: • Functions are called and the return address ends up in the NFC instruction. • Variables’ addresses end up in the NFC instruction. • Literal numbers are used directly as addresses. In other words, everything is as standard unless you pass a number or constant in directly, e.g. nfc(debugout,debugin), in which case it will use that constant as an address. To perform NFCs with a value and not an address, use the val(int) operator.
`void nfc4(ref, ref, ref, ref)`	Same as nfc(ref, ref) but arguments for all 4 fields are supplied, as opposed to the compiler stitching in the next instruction automatically. Used in combination with macros and labels, you can create your own control structures.
`int not(int)`	Returns the bitwise complement of the argument, e.g. not(0) = 0xff.
`int or(int, int)`	Returns the bitwise or of its arguments.
`pointer pair(int, int)`	Returns the pointer value that results from concatenating its two arguments. The first argument is the more significant, i.e. first in memory (big endian).
`int read(pointer)`	Returns the result of reading from the passed pointer (dereferences the pointer). If a literal number is passed in as the address this will compile down to *0* instructions. (This is still semantically significant, e.g. compare the meaning of “x = debugin;” and “x = read(debugin);”: these are both single instructions but one assigns 0xc001 to x whilst the other reads a value from the debug input port).
`int second(pointer)`	Returns the second (less significant) byte of a pointer.
`int shiftleft(int)`	Shifts its argument to the left by one bit, e.g. shiftleft(10) = 20. Implemented as a table lookup.

`int shiftright(int)`	Shifts its argument to the right by one bit, e.g. shiftright(10) = 5. Implemented as a table lookup.
`pointer val(int)`	Takes an 8 bit integer as its input and returns an address that corresponds to that integer, e.g. "nfc(debugout, val(0xff));" will clear the debug output port.
`void write(pointer, int)`	Writes a byte to a given location (lval pointer dereferencing).
`int xor(int, int)`	Returns the logical xor of the two arguments.

Other functions such as addition, copying chunks of memory, finding the length of strings etc. will not be built into the compiler, but instead be part of the standard library. A complete listing of the standard library can be found in Appendix 2.

Headers and Libraries

Rather than writing everything from scratch every time you create a new program, it is convenient to be able to reuse common code and functionality. These collections of code are known as libraries.

The header/library model used by spoon is very simple: libraries are included with a `#include` statement, as such:

```
#include "stddefs"
#include "forkos/forkos.bin.def"
```

Each include statement is processed at parse-time, and the included file is loaded and parsed, and its definitions added to the parser's output. Headers can contain either function definitions or merely declarations, so long as each function is defined exactly once in total in the main file and all included files.

Header guards are not used or necessary; the compiler will ignore an include statement for a file that has already been included.

The compiler searches for the included file first in the directory of the main file, and then in the compiler executable's directory. If it is found in neither directory then the user is informed of the problem and the compiler exits.