

stoc

Generated by Doxygen 1.8.13



# Contents

<b>1</b>	<b><a href="#">stoc</a></b>	<b>1</b>
<b>2</b>	<b><a href="#">Data Structure Index</a></b>	<b>5</b>
2.1	<a href="#">Data Structures</a> . . . . .	5
<b>3</b>	<b><a href="#">File Index</a></b>	<b>7</b>
3.1	<a href="#">File List</a> . . . . .	7
<b>4</b>	<b><a href="#">Data Structure Documentation</a></b>	<b>9</b>
4.1	<a href="#">decl_t Struct Reference</a> . . . . .	9
4.2	<a href="#">instruction_t Struct Reference</a> . . . . .	9
4.3	<a href="#">iterator_t Struct Reference</a> . . . . .	10
4.3.1	<a href="#">Detailed Description</a> . . . . .	10
4.4	<a href="#">pick_t Struct Reference</a> . . . . .	10
4.4.1	<a href="#">Detailed Description</a> . . . . .	11
4.5	<a href="#">rewrite_t Struct Reference</a> . . . . .	11
4.5.1	<a href="#">Detailed Description</a> . . . . .	12
4.6	<a href="#">stoc_t Struct Reference</a> . . . . .	12

<b>5</b>	<b>File Documentation</b>	<b>13</b>
5.1	asm.h File Reference	13
5.1.1	Detailed Description	14
5.1.2	Function Documentation	14
5.1.2.1	readfile()	14
5.2	emulator.h File Reference	14
5.2.1	Detailed Description	15
5.2.2	Function Documentation	15
5.2.2.1	mem_read()	15
5.2.2.2	mem_write()	15
5.2.2.3	step()	16
5.3	main.h File Reference	16
5.3.1	Detailed Description	17
5.3.2	Function Documentation	17
5.3.2.1	hexdump()	17
5.4	optimization.h File Reference	17
5.4.1	Detailed Description	18
5.4.2	Function Documentation	18
5.4.2.1	compare()	18
5.5	pick.h File Reference	18
5.5.1	Detailed Description	19
5.6	search.h File Reference	20
5.6.1	Detailed Description	20
5.6.2	Function Documentation	20
5.6.2.1	deadcodeelim()	21
5.6.2.2	stoc_exh()	21
5.6.2.3	stoc_opt()	21
	<b>Index</b>	<b>23</b>

# Chapter 1

## stoc

Stochastic superoptimiser targetting the 6502

We've got a few different search strategies implemented actually, and these exercise the emulator, equivalence tester and everything. Some assembly language files in `examples/` contain goofy code sequences that contain obvious inefficiencies. They are just there to demonstrate stoc.

To build the system, type `make`. For each architecture, (currently a few varieties of 6502) `make` will generate the appropriate source code and compile an executable named `stoc-$arch`.

### Supported architectures

So far, we've got a few varieties of 6502. These are:

- *stoc-6502* which is a generic NMOS 6502, including the `jmp` indirect bug, but does not use any of the illegal opcodes
- *stoc-6510*, another NMOS 6502, and has some of the same illegal opcodes that the Commodore 64 guys use
- *stoc-65c02*, targets the later CMOS chips with extra opcodes like `phx` and so on
- *stoc-2a03*, basically the same as *stoc-6502* but has no decimal mode. Dead-code elimination here will remove instructions `sed` and `cld`.

The above list is essentially what's provided by the *fake6502* submodule. If you are interested in adding other architectures, I would suggest that the easiest way would be to graft in another emulator. At build-time, a particular emulator is linked in, and this is what determines which architecture the binary supports. A separate program is built for each supported architecture.

### Theory of operation

The basic idea with this is to generate better programs than traditional compilers can, by copying a working program and making many small random successive changes to it. If the copy is found to be equivalent (or close enough), then it might get written to the standard output. Otherwise, another attempt is made, until an improvement is found.

There are a few different ways we can introduce mutations into the program, and these have names such as Dead Code Elimination, or `.dce`, Stochastic Optimisation, or `.opt`, etc. They are described in more detail below.

How can we see if two programs are equivalent? We can spam them with random numbers, and then check whether they produce the same output. Earlier versions of stoc worked in this way, but there was a slight chance that the random numbers didn't exercise the entire program. This could lead to a buggy program being output. To mitigate this risk, I've introduced the concept of testcases. A testcase is partially derived from the RNG and partially derived from the reference program. A testcase specifies what output the program should yield given a specific input. stoc will remember a number of these, and test each putative program against them all. I would guess that 99% of the garbage produced by the search algorithm is caught by the first few testcases.

## Dead Code Elimination

This search strategy looks for a more optimal rewrite by selecting random instructions for deletion; up to five at a time (this is to give pairs of instructions, such as a `pha` and corresponding `pla`, a chance to get deleted together). If the program proves to be equivalent without the selected instructions, then the instructions are deleted and the same procedure is done again. Use this procedure by using the `.dce` action.

```
$ ./stoc-2a03 examples/add_two_constants.stoc .dis .dce
; starting at $2000
; 5 instructions
; 7 bytes
; 5 clockticks
    clc
    lda #$07
    sed
    clc
    adc #$05

; starting at $2000
; 3 instructions
; 5 bytes
; 18 clockticks
    lda #$07
    clc
    adc #$05
```

It might be worth noting that the input procedure above contains two instances of the `clc` instructions, and only one is needed. Either one may be deleted, and it is picked at random. Running the same program again might have yielded the instructions `clc` and `lda #$07` in a different order. The `sed` instruction is not needed at all on the 2A03 because this is a chip variant which lacks the decimal mode. On other varieties of the 6502, as emulated by `stoc-6510` for example, the `sed` instruction will be deemed necessary by the equivalence tester.

## Stochastic optimisation

This search strategy walks around the search space by trying a number of mutations at a time, at sees if these mutations together either lower the cost or increase correctness (or both). If so, then the putative program (i.e. the one including the random mutations) replaces the current starting position, and another walk begins. I don't know if this one will prove promising or not. Here are the possible mutations it does:

- Insert a random instruction
- Delete an instruction at random
- Modify a random instruction's operand
- Change a random instruction's opcode for another one, having the same addressing mode
- Pick two random instructions and swap them over
- Pick one instruction, and overwrite it entirely with another one.

This will stop searching when the random walks stop finding improvements. I.e., if it's tried  $n$  times without finding a more optimal program, the search stops and the last found known good program is printed out. Invoke this search with the `.opt` action. So here is an example run:

```
$ ./stoc-2a03 examples/add_two_constants.stoc .dis .opt
; starting at $2000
; 5 instructions
; 7 bytes
; 5 clockticks
    clc
    lda #$07
    sed
    clc
    adc #$05

; starting at $2000
; 1 instructions
; 2 bytes
; 1 clockticks
    lda #$0c
```

There are a few ways to reach the second program from the first; each one is a random walk through the search space:

- Maybe stoc putatively inserted `lda #$0c` at the end of the program, and then deleted the rest of the instructions, having established that they are effectively dead code.
- Maybe stoc putatively altered `lda #$07` to `lda #$0c`, and altered `adc #$05` to something benign, and then deleted the rest of the instructions, having established that they are effectively dead code.

Whatever the case, the search has discovered that adding two constants together is equivalent to loading the sum of those constants, and suggested a replacement program that does so.





## Chapter 2

# Data Structure Index

### 2.1 Data Structures

Here are the data structures with brief descriptions:

<a href="#">decl_t</a>	9
<a href="#">instruction_t</a>	9
<a href="#">iterator_t</a>	10
<a href="#">pick_t</a>	10
<a href="#">rewrite_t</a>	11
<a href="#">stoc_t</a>	12



## Chapter 3

# File Index

### 3.1 File List

Here is a list of all documented files with brief descriptions:

<b>arch.h</b>	.....	??
<a href="#">asm.h</a>		
	Parser for .stoc files	13
<b>decl.h</b>	.....	??
<a href="#">emulator.h</a>		
	Interface to the emulator	14
<a href="#">main.h</a>		
	Miscellaneous functions	16
<a href="#">optimization.h</a>		
	Functions for measuring rewrites	17
<a href="#">pick.h</a>		
	Handling sets of values	18
<a href="#">search.h</a>		
	Search strategies	20
<b>stoc.h</b>	.....	??
<b>tests.h</b>	.....	??



## Chapter 4

# Data Structure Documentation

### 4.1 decl\_t Struct Reference

Collaboration diagram for decl\_t:



#### Data Fields

- int(\* **fn**)(stoc\_t \*c, struct \_decl\_t \*d, uint8\_t \*\*scram)
- int(\* **setup**)(stoc\_t \*c, struct \_decl\_t \*d, uint8\_t \*\*scram)
- char **label** [LABEL\_LEN]
- uint16\_t **start**
- unsigned int **length**
- struct \_decl\_t \* **next**

The documentation for this struct was generated from the following file:

- decl.h

### 4.2 instruction\_t Struct Reference

#### Data Fields

- addr\_t **address**
- uint8\_t **opcode**
- uint16\_t **operand**

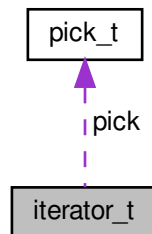
The documentation for this struct was generated from the following file:

- stoc.h

### 4.3 iterator\_t Struct Reference

```
#include <pick.h>
```

Collaboration diagram for iterator\_t:



#### Data Fields

- [pick\\_t \\* pick](#)  
*Pointer to the [pick\\_t](#) object over which we're iterating.*
- int [current](#)  
*Current offset.*

#### 4.3.1 Detailed Description

An iterator

This one is for iterating over a [pick\\_t](#).

The documentation for this struct was generated from the following file:

- [pick.h](#)

### 4.4 pick\_t Struct Reference

```
#include <pick.h>
```

#### Data Fields

- int [count](#)  
*Number of members in the set.*
- uint16\_t [vals](#) [MAXPICKSIZE]  
*the members*

### 4.4.1 Detailed Description

A set of values.

This is intended to be a set, which we can pick a number at random from, or which we can iterate over

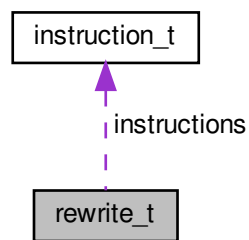
The documentation for this struct was generated from the following file:

- [pick.h](#)

## 4.5 rewrite\_t Struct Reference

```
#include <stoc.h>
```

Collaboration diagram for `rewrite_t`:



### Data Fields

- `uint16_t` [org](#)  
*Where the rewrite starts.*
- `uint16_t` [length](#)  
*Number of instructions in the rewrite.*
- `uint16_t` [end](#)  
*The first address after the last instruction.*
- `instruction_t` [instructions](#) [REWRITE\_LEN]  
*Array of instructions.*
- `long double` [fitness](#)  
*Fitness, or "how correct is the rewrite".*
- `long double` [mcycles](#)  
*machine cycles, or "how long does the program take"*
- `int` [blength](#)  
*The program's length, in bytes.*

### 4.5.1 Detailed Description

#### [rewrite\\_t](#)

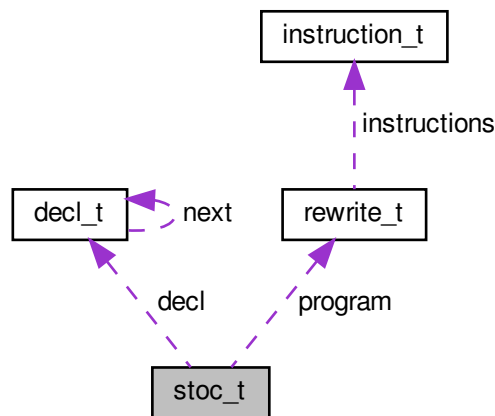
A rewrite is a list of instructions, plus associated data.

The documentation for this struct was generated from the following file:

- `stoc.h`

## 4.6 `stoc_t` Struct Reference

Collaboration diagram for `stoc_t`:



### Data Fields

- `uint8_t a`
- `uint8_t x`
- `uint8_t y`
- `uint8_t flags`
- `uint8_t s`
- `uint16_t pc`
- long long int `clockticks`
- [rewrite\\_t](#) `program`
- `data_t mem [ADDR_SPACE]`
- `uint8_t memf [ADDR_SPACE]`
- `uint16_t ea`
- `uint8_t opcode`
- int `exitcode`
- struct [\\_decl\\_t](#) \* `decl`

The documentation for this struct was generated from the following file:

- `stoc.h`



## Chapter 5

# File Documentation

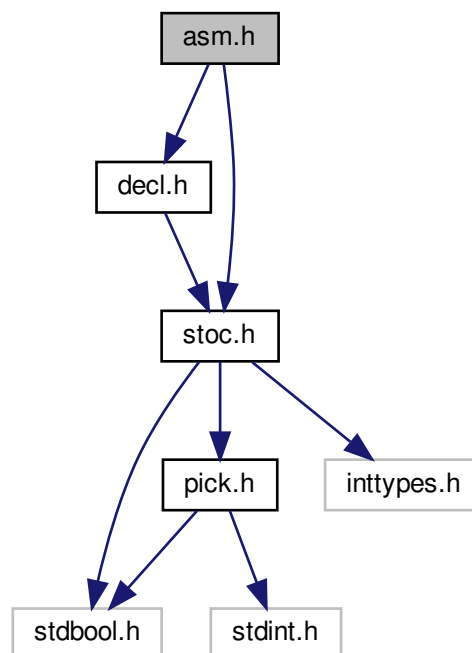
### 5.1 asm.h File Reference

Parser for .stoc files.

```
#include "decl.h"
```

```
#include "stoc.h"
```

Include dependency graph for asm.h:



### Functions

- void [readfile](#) (char \*filename, [stoc\\_t](#) \*reference)

### 5.1.1 Detailed Description

Parser for .stoc files.

### 5.1.2 Function Documentation

#### 5.1.2.1 readfile()

```
void readfile (
    char * filename,
    stoc_t * reference )
```

Load a .stoc file

#### Parameters

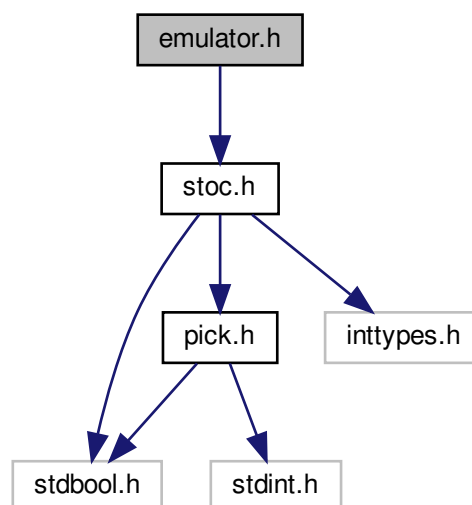
<i>filename</i>	The name of the file to load
<i>reference</i>	A pointer to the <code>stoc_t</code> to load the file into

## 5.2 emulator.h File Reference

Interface to the emulator.

```
#include "stoc.h"
```

Include dependency graph for emulator.h:



## Functions

- void `mem_write` (`stoc_t *c`, `addr_t address`, `data_t val`)
- `uint8_t mem_read` (`stoc_t *c`, `addr_t address`)
- void `step` (`stoc_t *c`)

### 5.2.1 Detailed Description

Interface to the emulator.

### 5.2.2 Function Documentation

#### 5.2.2.1 `mem_read()`

```
uint8_t mem_read (  
    stoc_t * c,  
    addr_t address )
```

Read a value in the emulator's address space

##### Parameters

<i>c</i>	Which emulated machine we're writing to the address space of
<i>address</i>	The address to read from

##### Returns

the value read from the memory location

#### 5.2.2.2 `mem_write()`

```
void mem_write (  
    stoc_t * c,  
    addr_t address,  
    data_t val )
```

Write a value to the emulator's address space

##### Parameters

<i>c</i>	Which emulated machine we're writing to the address space of
<i>address</i>	The address to write to
<i>val</i>	The value to write

### 5.2.2.3 step()

```
void step (  
    stoc_t * c )
```

Single-step the emulator

Execute a single instruction

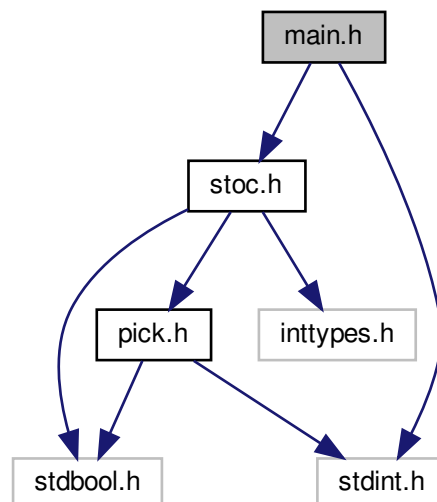
#### Parameters

<code>c</code>	Which emulated machine
----------------	------------------------

## 5.3 main.h File Reference

Miscellaneous functions.

```
#include "stoc.h"  
#include <stdint.h>  
Include dependency graph for main.h:
```



## Functions

- void `hexdump` (`stoc_t *r`)

### 5.3.1 Detailed Description

Miscellaneous functions.

### 5.3.2 Function Documentation

#### 5.3.2.1 hexdump()

```
void hexdump (
    stoc_t * r )
```

Disassemble the file

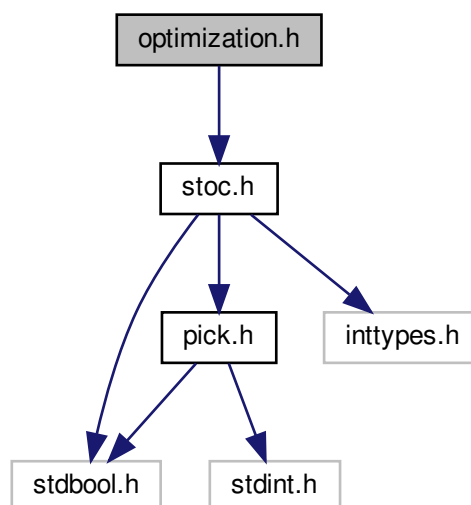
Outputs the rewrite, plus miscellaneous measurements, to stdout.

## 5.4 optimization.h File Reference

Functions for measuring rewrites.

```
#include "stoc.h"
```

Include dependency graph for optimization.h:



## Functions

- int `optimize_size` (`stoc_t *c`)  
*Returns an integer representing the size of the rewrite.*
- int `optimize_speed` (`stoc_t *c`)  
*Returns an integer representing the speed of the rewrite.*
- void `set_optimization` (`int(*fn)(stoc_t *c)`)  
*Sets the optimization function.*
- int `compare` (`stoc_t *a`, `stoc_t *b`)

### 5.4.1 Detailed Description

Functions for measuring rewrites.

### 5.4.2 Function Documentation

#### 5.4.2.1 `compare()`

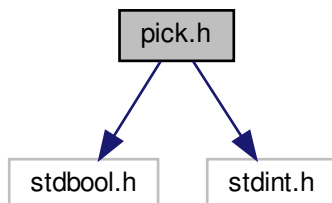
```
int compare (  
    stoc_t * a,  
    stoc_t * b )
```

Returns an integer representing a comparison of two rewrites, according to whichever function was selected by `set_optimization`

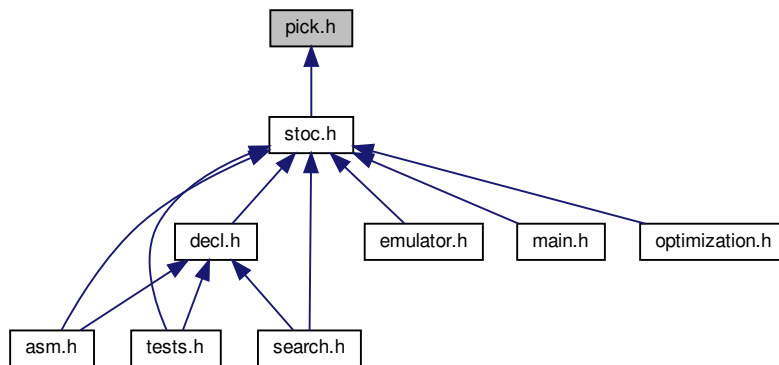
## 5.5 `pick.h` File Reference

Handling sets of values.

```
#include <stdbool.h>  
#include <stdint.h>  
Include dependency graph for pick.h:
```



This graph shows which files directly or indirectly include this file:



## Data Structures

- struct [pick\\_t](#)
- struct [iterator\\_t](#)

## Macros

- `#define MAXPICKSIZE 50`

## Functions

- void **iterator\_init** ([pick\\_t](#) \*pick, [iterator\\_t](#) \*iterator)
- bool **pick\_iterate** ([iterator\\_t](#) \*iter, uint16\_t \*out)
- void **pick\_insert** ([pick\\_t](#) \*pick, uint16\_t val)
- bool **pick\_at\_random** ([pick\\_t](#) \*pick, uint16\_t \*out)
- bool **random\_address** (uint16\_t \*out)
- bool **random\_constant** (uint16\_t \*out)
- void **pickinit** (void)
- void **initialize\_pick** ([pick\\_t](#) \*pick)
- void **pick\_set\_all\_constants** ()
- void **pick\_set\_common\_constants** ()

## Variables

- [pick\\_t](#) constants
- [pick\\_t](#) addresses

### 5.5.1 Detailed Description

Handling sets of values.

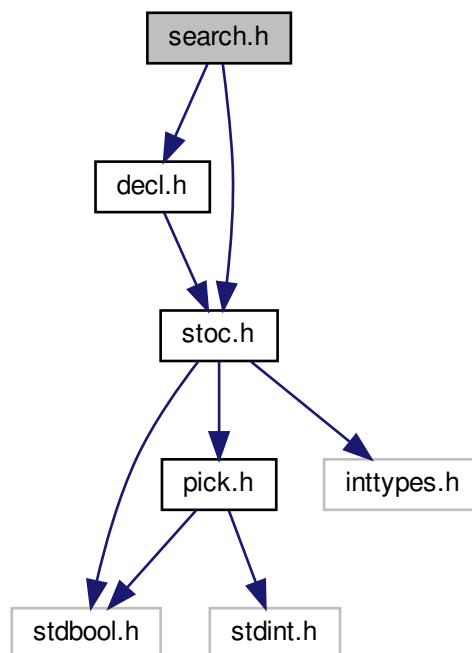
## 5.6 search.h File Reference

search strategies

```
#include "decl.h"
```

```
#include "stoc.h"
```

Include dependency graph for search.h:



### Functions

- void `deadcodeelim` (`stoc_t` \*reference)
- void `stoc_opt` (`stoc_t` \*reference)
- void `stoc_exh` (`stoc_t` \*reference)
- void `search_init` ()

### 5.6.1 Detailed Description

search strategies

### 5.6.2 Function Documentation



### 5.6.2.1 deadcodeelim()

```
void deadcodeelim (
    stoc_t * reference )
```

Eliminate dead code

Removes superfluous instructions from a program

#### Parameters

<i>reference</i>	A <code>stoc_t</code> containing the rewrite you want to eliminate dead code from
------------------	---

### 5.6.2.2 stoc\_exh()

```
void stoc_exh (
    stoc_t * reference )
```

Exhaustive search

Does an exhaustive search of all programs upto a certain length, and prints out the first one

### 5.6.2.3 stoc\_opt()

```
void stoc_opt (
    stoc_t * reference )
```

Optimise

Makes random mutations until an alternative rewrite is discovered that's functionally equivalent

#### Parameters

<i>reference</i>	A <code>stoc_t</code> containing the rewrite you want to optimise
------------------	---



# Index

- asm.h, [13](#)
  - readfile, [14](#)
- compare
  - optimization.h, [18](#)
- deadcodeelim
  - search.h, [20](#)
- decl\_t, [9](#)
- emulator.h, [14](#)
  - mem\_read, [15](#)
  - mem\_write, [15](#)
  - step, [16](#)
- hexdump
  - main.h, [17](#)
- instruction\_t, [9](#)
- iterator\_t, [10](#)
- main.h, [16](#)
  - hexdump, [17](#)
- mem\_read
  - emulator.h, [15](#)
- mem\_write
  - emulator.h, [15](#)
- optimization.h, [17](#)
  - compare, [18](#)
- pick.h, [18](#)
- pick\_t, [10](#)
- readfile
  - asm.h, [14](#)
- rewrite\_t, [11](#)
- search.h, [20](#)
  - deadcodeelim, [20](#)
  - stoc\_exh, [21](#)
  - stoc\_opt, [21](#)
- step
  - emulator.h, [16](#)
- stoc\_exh
  - search.h, [21](#)
- stoc\_opt
  - search.h, [21](#)
- stoc\_t, [12](#)