

# Quantum Computer Outreach Project

Generated by Doxygen 1.8.13



# Contents

<b>1</b>	<b>Todo List</b>	<b>1</b>
<b>2</b>	<b>Bug List</b>	<b>3</b>
<b>3</b>	<b>Data Structure Index</b>	<b>5</b>
3.1	Data Structures . . . . .	5
<b>4</b>	<b>File Index</b>	<b>7</b>
4.1	File List . . . . .	7
<b>5</b>	<b>Data Structure Documentation</b>	<b>9</b>
5.1	BTN Struct Reference . . . . .	9
5.1.1	Detailed Description . . . . .	9
5.2	cycle_node Struct Reference . . . . .	10
5.2.1	Detailed Description . . . . .	10
5.3	LED Struct Reference . . . . .	10
5.3.1	Detailed Description . . . . .	11
5.4	LED_GLOBAL Struct Reference . . . . .	11
5.4.1	Detailed Description . . . . .	11
5.5	RGB Struct Reference . . . . .	12
5.5.1	Detailed Description . . . . .	12

<b>6</b>	<b>File Documentation</b>	<b>13</b>
6.1	dspic33e/qcomp-sim-c.X/algo.c File Reference	13
6.1.1	Detailed Description	15
6.1.2	Function Documentation	15
6.1.2.1	gate()	15
6.1.2.2	gate_display()	15
6.1.2.3	swap_test()	15
6.1.2.4	toffoli_gate()	16
6.1.2.5	two_gate()	16
6.1.2.6	two_gate_display()	17
6.2	dspic33e/qcomp-sim-c.X/algo.h File Reference	17
6.2.1	Detailed Description	18
6.2.2	Function Documentation	18
6.2.2.1	gate()	19
6.2.2.2	gate_display()	19
6.2.2.3	swap_test()	19
6.2.2.4	toffoli_gate()	19
6.2.2.5	two_gate()	20
6.2.2.6	two_gate_display()	20
6.3	dspic33e/qcomp-sim-c.X/config.h File Reference	21
6.3.1	Detailed Description	21
6.4	dspic33e/qcomp-sim-c.X/consts.c File Reference	21
6.4.1	Detailed Description	22
6.4.2	Variable Documentation	22
6.4.2.1	H	22
6.4.2.2	rX	23
6.4.2.3	rXT	23
6.4.2.4	X	23
6.4.2.5	Y	24
6.4.2.6	Z	24

6.5	dspic33e/qcomp-sim-c.X/consts.h File Reference	24
6.5.1	Detailed Description	26
6.5.2	Variable Documentation	26
6.5.2.1	H	26
6.5.2.2	rX	26
6.5.2.3	rXT	27
6.5.2.4	X	27
6.5.2.5	Y	27
6.5.2.6	Z	27
6.6	dspic33e/qcomp-sim-c.X/display.c File Reference	28
6.6.1	Detailed Description	29
6.6.2	Macro Definition Documentation	29
6.6.2.1	NUM_MAX_AMPS	29
6.6.3	Function Documentation	29
6.6.3.1	display_average()	29
6.6.3.2	display_cycle()	31
6.6.3.3	remove_zero_amp_states()	32
6.6.3.4	sort_states()	32
6.7	dspic33e/qcomp-sim-c.X/display.h File Reference	33
6.7.1	Detailed Description	34
6.7.2	Function Documentation	34
6.7.2.1	display_average()	34
6.7.2.2	display_cycle()	36
6.7.2.3	remove_zero_amp_states()	37
6.7.2.4	sort_states()	37
6.8	dspic33e/qcomp-sim-c.X/io.c File Reference	38
6.8.1	Detailed Description	40
6.8.2	Function Documentation	40
6.8.2.1	__attribute__()	40
6.8.2.2	add_to_cycle()	41

6.8.2.3	flash_all()	41
6.8.2.4	flash_led()	41
6.8.2.5	led_color_int()	42
6.8.2.6	led_cycle_test()	42
6.8.2.7	read_btn()	42
6.8.2.8	read_external_buttons()	43
6.8.2.9	reset_cycle()	43
6.8.2.10	set_external_led()	43
6.8.2.11	set_led()	44
6.8.2.12	set_strobe()	44
6.8.2.13	setup_external_leds()	45
6.8.2.14	setup_io()	45
6.8.2.15	TLC591x_mode_switch()	45
6.8.2.16	toggle_strobe()	46
6.8.2.17	update_display_buffer()	46
6.8.2.18	write_display_driver()	47
6.8.3	Variable Documentation	47
6.8.3.1	btn_func	47
6.8.3.2	btn_qubit	47
6.8.3.3	buttons	48
6.8.3.4	chip	48
6.8.3.5	isr_counter	48
6.8.3.6	led_global	48
6.9	dspic33e/qcomp-sim-c.X/io.h File Reference	49
6.9.1	Detailed Description	51
6.9.2	Function Documentation	51
6.9.2.1	add_to_cycle()	51
6.9.2.2	flash_all()	52
6.9.2.3	flash_led()	52
6.9.2.4	led_color_int()	52

6.9.2.5	<a href="#">led_cycle_test()</a>	54
6.9.2.6	<a href="#">read_btn()</a>	54
6.9.2.7	<a href="#">read_external_buttons()</a>	54
6.9.2.8	<a href="#">reset_cycle()</a>	55
6.9.2.9	<a href="#">set_external_led()</a>	55
6.9.2.10	<a href="#">set_led()</a>	56
6.9.2.11	<a href="#">set_strobe()</a>	56
6.9.2.12	<a href="#">setup_external_leds()</a>	56
6.9.2.13	<a href="#">setup_io()</a>	57
6.9.2.14	<a href="#">toggle_strobe()</a>	57
6.9.2.15	<a href="#">update_display_buffer()</a>	57
6.9.2.16	<a href="#">write_display_driver()</a>	58
6.10	<a href="#">dspic33e/qcomp-sim-c.X/main.c File Reference</a>	59
6.10.1	<a href="#">Detailed Description</a>	60
6.10.2	<a href="#">Function Documentation</a>	60
6.10.2.1	<a href="#">main()</a>	60
6.11	<a href="#">dspic33e/qcomp-sim-c.X/quantum.c File Reference</a>	61
6.11.1	<a href="#">Detailed Description</a>	62
6.11.2	<a href="#">Function Documentation</a>	62
6.11.2.1	<a href="#">absolute()</a>	62
6.11.2.2	<a href="#">controlled_qubit_op()</a>	62
6.11.2.3	<a href="#">controlled_qubit_op_new()</a>	63
6.11.2.4	<a href="#">mat_mul()</a>	65
6.11.2.5	<a href="#">mat_mul_old()</a>	65
6.11.2.6	<a href="#">pow2()</a>	66
6.11.2.7	<a href="#">sign()</a>	66
6.11.2.8	<a href="#">single_qubit_op()</a>	66
6.11.2.9	<a href="#">square_magnitude()</a>	67
6.11.2.10	<a href="#">zero_state()</a>	68
6.12	<a href="#">dspic33e/qcomp-sim-c.X/quantum.h File Reference</a>	68

6.12.1 Detailed Description . . . . .	70
6.12.2 Function Documentation . . . . .	70
6.12.2.1 absolute() . . . . .	70
6.12.2.2 controlled_qubit_op() . . . . .	70
6.12.2.3 mat_mul() . . . . .	71
6.12.2.4 pow2() . . . . .	72
6.12.2.5 sign() . . . . .	72
6.12.2.6 single_qubit_op() . . . . .	72
6.12.2.7 square_magnitude() . . . . .	74
6.12.2.8 zero_state() . . . . .	74
6.13 dspic33e/qcomp-sim-c.X/spi.c File Reference . . . . .	75
6.13.1 Detailed Description . . . . .	75
6.13.2 Function Documentation . . . . .	75
6.13.2.1 send_byte_spi_1() . . . . .	76
6.13.2.2 setup_spi() . . . . .	77
6.14 dspic33e/qcomp-sim-c.X/spi.h File Reference . . . . .	78
6.14.1 Detailed Description . . . . .	79
6.14.2 Function Documentation . . . . .	79
6.14.2.1 send_byte_spi_1() . . . . .	79
6.14.2.2 setup_spi() . . . . .	79
6.15 dspic33e/qcomp-sim-c.X/time.c File Reference . . . . .	81
6.15.1 Detailed Description . . . . .	81
6.15.2 Function Documentation . . . . .	82
6.15.2.1 setup_timer() . . . . .	82
6.16 dspic33e/qcomp-sim-c.X/time.h File Reference . . . . .	82
6.16.1 Detailed Description . . . . .	83
6.16.2 Function Documentation . . . . .	83
6.16.2.1 setup_timer() . . . . .	83



# Chapter 1

## Todo List

### Global `absolute` (Complex x)

Check that the complex part is small

### Global `controlled_qubit_op` (const Complex op[2][2], int ctrl, int targ, Complex state[])

This expression can probably be simplified or broken over lines. The condition for the if statement is that root+step and root + step + root\_max contain 1 in the ctrl-th bit.

### Global `controlled_qubit_op_new` (const Complex op[2][2], int ctrl, int targ, Complex state[])

Replace pow2 with left rotations

The problem is the formula for the increment

### Global `display_average` (Complex state[])

Bring all constants out of the loops. Don't use pow. Loop over all qubits k = 0, 1, 2, ... N-1

Rewrite pow for Q15

rename to display\_average

### Global `led_cycle_test` (void)

This won't work now: write\_display\_driver(counter);

### Global `mat_mul` (const Complex M[2][2], Complex V[], int i, int j)

Is static enough? Or should we declare outside the function?

Should we use for loops? Or is it better not to..?

Because of the way the array types work (you can't pass a multidimensional array of unknown size) we will also need a function for 4x4 matrix multiplication.

### Global `mat_mul_old` (const Complex M[2][2], Complex V[], int i, int j)

Should these be outside the function?

### File `quantum.c`

split into a complex math and operator files

### Global `read_external_buttons` (void)

read buttons

How long should this be?

button remappings...

### Global `reset_cycle` (void)

do it

### Global `setup_external_leds` (void)

CURRENTLY CYCLING IS OFF

**Global `setup_timer` ()**

distinguish between the two different timers here...

**Global `single_qubit_op` (const Complex op[2][2], int qubit, Complex state[])**

Should we inline `mat_mul` here?

**Global `single_qubit_op` (const Complex op[2][2], int qubit, Complex state[])**

Should we inline `mat_mul` here?

**Global `sort_states` (Complex state[], int num\_qubits)**

this function...

this

**Global `square_magnitude` (Complex x)**

Maybe we should inline this

Maybe we should inline this

**Global `TLC591x_mode_switch` (int mode)**

mode switcher for `LED` Driver

**Global `toffoli_gate` (int q1, int q2, int q3, Complex state[])**

Fancy non-blocking Interrupt routine { if(no button) return;

**Global `write_display_driver` (void)**

How long should this be?

## Chapter 2

# Bug List

Global `btn_func` [NUM\_BTNS - NUM\_QUBITS]  
this.



## Chapter 3

# Data Structure Index

### 3.1 Data Structures

Here are the data structures with brief descriptions:

<a href="#">BTN</a>	Pin mappings . . . . .	9
<a href="#">cycle_node</a>	The basis for a linked list of states to cycle . . . . .	10
<a href="#">LED</a>	Each <a href="#">LED</a> has the following type . . . . .	10
<a href="#">LED_GLOBAL</a>	Global <a href="#">LED</a> strobing state parameter . . . . .	11
<a href="#">RGB</a>	A type for holding red, green, blue values . . . . .	12



## Chapter 4

# File Index

### 4.1 File List

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

dspic33e/qcomp-sim-c.X/ <a href="#">algo.c</a>	
Contains quantum algorithms to be run . . . . .	13
dspic33e/qcomp-sim-c.X/ <a href="#">algo.h</a>	
Header file for algorithms . . . . .	17
dspic33e/qcomp-sim-c.X/ <a href="#">config.h</a>	
General config settings #pragma for microcontroller . . . . .	21
dspic33e/qcomp-sim-c.X/ <a href="#">consts.c</a>	
All (global) constants) . . . . .	21
dspic33e/qcomp-sim-c.X/ <a href="#">consts.h</a>	
Header file for (global) constants . . . . .	24
dspic33e/qcomp-sim-c.X/ <a href="#">display.c</a>	
For all the state display functions . . . . .	28
dspic33e/qcomp-sim-c.X/ <a href="#">display.h</a>	
Description: Header file containing all the functions for displaying the qubits state vector . . . .	33
dspic33e/qcomp-sim-c.X/ <a href="#">io.c</a>	
Contains all the functions for reading buttons and writing to LEDs . . . . .	38
dspic33e/qcomp-sim-c.X/ <a href="#">io.h</a>	
Description: Header file for input output functions . . . . .	49
dspic33e/qcomp-sim-c.X/ <a href="#">main.c</a>	
The main function . . . . .	59
dspic33e/qcomp-sim-c.X/ <a href="#">quantum.c</a>	
Description: Contains matrix and vector arithmetic for simulating one qubit . . . . .	61
dspic33e/qcomp-sim-c.X/ <a href="#">quantum.h</a>	
Description: Header file containing all the matrix arithmetic for simulating a single qubit . . . .	68
dspic33e/qcomp-sim-c.X/ <a href="#">spi.c</a>	
Description: Functions for communicating with serial devices . . . . .	75
dspic33e/qcomp-sim-c.X/ <a href="#">spi.h</a>	
Description: SPI communication functions . . . . .	78
dspic33e/qcomp-sim-c.X/ <a href="#">time.c</a>	
Description: Functions to control the on chip timers . . . . .	81
dspic33e/qcomp-sim-c.X/ <a href="#">time.h</a>	
Description: Header file containing all the timing functions . . . . .	82





## Chapter 5

# Data Structure Documentation

### 5.1 BTN Struct Reference

pin mappings

```
#include <io.h>
```

#### Data Fields

- int **chip**
- int [line](#)  
*[chip number]*

#### 5.1.1 Detailed Description

pin mappings

```
// Pins for LE and OE on port D
// OE = RD4 = uC:81 = J1:28 = J10:14
// LE = RD3 = uC:78 = J1:40 = J11:18
//
// Pins for SH and CLK_INH on port D
// SH = RD5 = uC:82 = J1:25 = J10:13
// CLK_INH = RD8 = uC:68 = J1:58 = J11:25
//
```

button mapping type

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

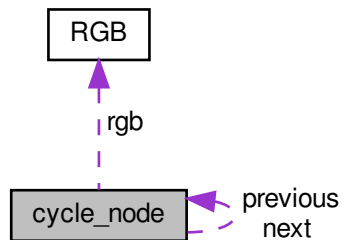
- dspic33e/qcomp-sim-c.X/[io.h](#)

## 5.2 cycle\_node Struct Reference

The basis for a linked list of states to cycle.

```
#include <io.h>
```

Collaboration diagram for cycle\_node:



### Data Fields

- `RGB * rgb`  
*Array of corresponding `RGB` values.*
- `int size`  
*The size of the above arrays.*
- `struct cycle_node * next`  
*Pointer to the next item.*
- `struct cycle_node * previous`  
*Pointer to the previous item.*

### 5.2.1 Detailed Description

The basis for a linked list of states to cycle.

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

- `dspic33e/qcomp-sim-c.X/io.h`

## 5.3 LED Struct Reference

Each `LED` has the following type.

```
#include <io.h>
```

## Data Fields

- int **R** [2]  
*Red mapping array: [chip number, line number].*
- int **G** [2]  
*Green mapping array.*
- unsigned \_Fract **N\_R**  
*Blue mapping array.*
- unsigned \_Fract **N\_G**  
*The R brightness.*
- unsigned \_Fract **N\_B**  
*The G brightness.*

### 5.3.1 Detailed Description

Each **LED** has the following type.

The type holds the information about the position of the **RGB** lines in the display driver array and also the brightness of the **RGB** lines. The counters are used by a timer interrupt service routine pulse the **RGB** LEDs at a specified rate.

The position of the **LED** lines are contained in an array

The type of the counter is *Fract* to facilitate easy comparison with the  $N^*$  variables which used the fractional type.

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

- dspic33e/qcomp-sim-c.X/[io.h](#)

## 5.4 LED\_GLOBAL Struct Reference

Global **LED** strobing state parameter.

```
#include <io.h>
```

## Data Fields

- int **strobe\_leds**  
*Bit set the LEDs which are strobing.*
- int **strobe\_state**  
*Bit zero is the current state (on/off)*

### 5.4.1 Detailed Description

Global **LED** strobing state parameter.

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

- dspic33e/qcomp-sim-c.X/[io.h](#)

## 5.5 RGB Struct Reference

A type for holding red, green, blue values.

```
#include <io.h>
```

### Data Fields

- unsigned \_Fract **R**
- unsigned \_Fract **G**
- unsigned \_Fract **B**

### 5.5.1 Detailed Description

A type for holding red, green, blue values.

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

- dspic33e/qcomp-sim-c.X/[io.h](#)

## Chapter 6

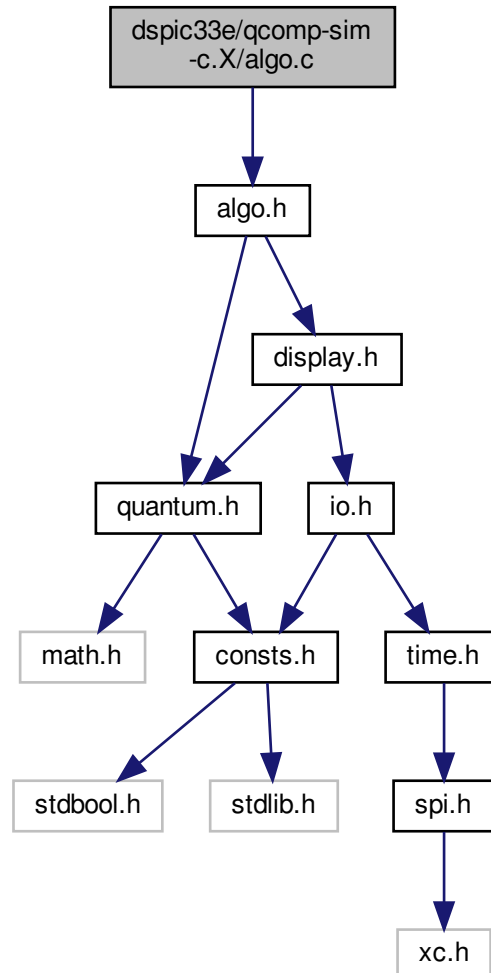
# File Documentation

### 6.1 `dspic33e/qcomp-sim-c.X/algo.c` File Reference

Contains quantum algorithms to be run.

```
#include "algo.h"
```

Include dependency graph for algo.c:



## Functions

- void `gate` (const `Complex` op[2][2], int qubit, `Complex` state[ ])
  - single qubit gate*
- void `gate_display` (const `Complex` op[2][2], int qubit, `Complex` state[ ])
  - single qubit gate with display*
- void `two_gate` (const `Complex` op[2][2], int ctrl, int targ, `Complex` state[ ])
  - two-qubit gate*
- void `two_gate_display` (const `Complex` op[2][2], int ctrl, int targ, `Complex` state[ ])
  - two-qubit gate with display*
- void `swap` (int q1, int q2, `Complex` state[ ])
  - swap using 3 cNots*
- void `swap_test` (`Complex` state[ ])

- from tests.c*
- void `toffoli_gate` (int q1, int q2, int q3, `Complex` state[])  
*Toffoli gate.*
  - void `toffoli_test` (`Complex` state[])

### 6.1.1 Detailed Description

Contains quantum algorithms to be run.

#### Authors

J Scott, O Thomas

#### Date

Nov 2018

### 6.1.2 Function Documentation

#### 6.1.2.1 `gate()`

```
void gate (  
    const Complex op[2][2],  
    int qubit,  
    Complex state[] )
```

single qubit gate

perform single qubit gate does 2x2 operator on state vector

#### 6.1.2.2 `gate_display()`

```
void gate_display (  
    const Complex op[2][2],  
    int qubit,  
    Complex state[] )
```

single qubit gate with display

Display gates!!! does 2x2 operator on state vector displays the average state of the qubit by tracing over all waits to let the user see the state (LEDs)

#### 6.1.2.3 `swap_test()`

```
void swap_test (  
    Complex state[] )
```

from tests.c

swap for ever!

#### 6.1.2.4 toffoli\_gate()

```
void toffoli_gate (
    int q1,
    int q2,
    int q3,
    Complex state[] )
```

Toffoli gate.

**Todo** Fancy non-blocking Interrupt routine { if(no button) return;

pause and do display cycling();

Make this a low priority interrupt so that everything else can interrupt it.

Do stuff for a while

return when you're done.

}

```
/// -o--  -----o-----o-o-----
/// -|--  -----|-----|---|-----
/// -o-- = ---o--X--o--X--|-----
/// -|--  ---|-----|-----|-----
/// -X--  --rX-----rX*---rX-----
///          a   b   c   d   e
///
```

q1 ctrl 1 q2 ctrl 2 q3 target < a

< b

< c

< d

< e

#### 6.1.2.5 two\_gate()

```
void two_gate (
    const Complex op[2][2],
    int ctrl,
    int targ,
    Complex state[] )
```

two-qubit gate

perform controlled single qubit gate does controlled 2x2 operator



## 6.1.2.6 two\_gate\_display()

```
void two_gate_display (
    const Complex op[2][2],
    int ctrl,
    int targ,
    Complex state[] )
```

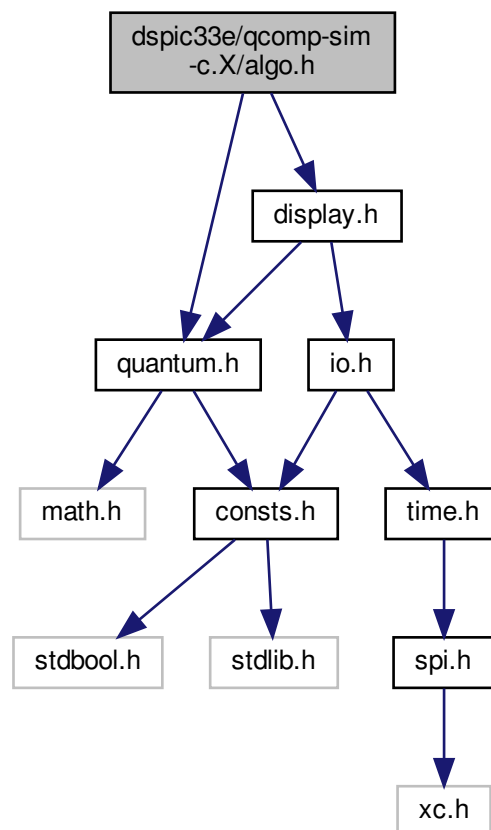
two-qubit gate with display

does controlled 2x2 operator displays the state waits to let the user see the state

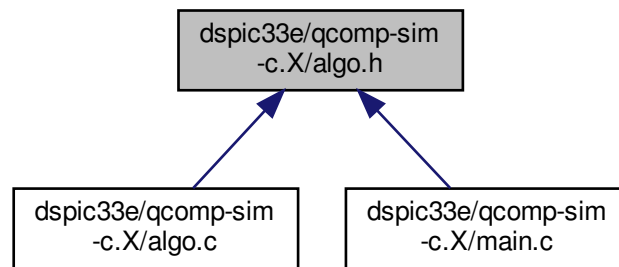
## 6.2 dspic33e/qcomp-sim-c.X/algo.h File Reference

header file for algorithms

```
#include "quantum.h"
#include "display.h"
Include dependency graph for algo.h:
```



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



## Functions

- void `gate` (const `Complex` op[2][2], int qubit, `Complex` state[ ])
  - perform single qubit gate*
- void `two_gate` (const `Complex` op[2][2], int ctrl, int targ, `Complex` state[ ])
  - perform controlled single qubit gate*
- void `gate_display` (const `Complex` op[2][2], int qubit, `Complex` state[ ])
  - Display gates!!!*
- void `two_gate_display` (const `Complex` op[2][2], int ctrl, int targ, `Complex` state[ ])
  - two-qubit gate with display*
- void `swap` (int q1, int q2, `Complex` state[ ])
  - swap using 3 cNots*
- void `swap_test` (`Complex` state[ ])
  - from tests.c*
- void `toffoli_gate` (int q1, int q2, int q3, `Complex` state[ ])
  - Toffoli gate.*
- void `toffoli_test` (`Complex` state[ ])

### 6.2.1 Detailed Description

header file for algorithms

#### Authors

J Scott, O Thomas

#### Date

Nov 2018

### 6.2.2 Function Documentation

## 6.2.2.1 gate()

```
void gate (
    const Complex op[2][2],
    int qubit,
    Complex state[] )
```

perform single qubit gate

perform single qubit gate does 2x2 operator on state vector

## 6.2.2.2 gate\_display()

```
void gate_display (
    const Complex op[2][2],
    int qubit,
    Complex state[] )
```

Display gates!!!

Display gates!!! does 2x2 operator on state vector displays the average state of the qubit by tracing over all waits to let the user see the state (LEDs)

## 6.2.2.3 swap\_test()

```
void swap_test (
    Complex state[] )
```

from tests.c

swap for ever!

## 6.2.2.4 toffoli\_gate()

```
void toffoli_gate (
    int q1,
    int q2,
    int q3,
    Complex state[] )
```

Toffoli gate.

**Todo** Fancy non-blocking Interrupt routine { if(no button) return;

pause and do display cycling();

Make this a low priority interrupt so that everything else can interrupt it.

Do stuff for a while

return when you're done.

```
}
```

```

/// -o--  -----o-----o--o-----
/// -|--  -----|-----|---|-----
/// -o-- = ---o--X--o--X--|-----
/// -|--  ---|-----|-----|-----
/// -X--  --rX-----rX*---rX-----
///      a   b   c   d   e
///

```

q1 ctrl 1 q2 ctrl 2 q3 target < a

< b

< c

< d

< e

#### 6.2.2.5 two\_gate()

```

void two_gate (
    const Complex op[2][2],
    int ctrl,
    int targ,
    Complex state[] )

```

perform controlled single qubit gate

perform controlled single qubit gate does controlled 2x2 operator

#### 6.2.2.6 two\_gate\_display()

```

void two_gate_display (
    const Complex op[2][2],
    int ctrl,
    int targ,
    Complex state[] )

```

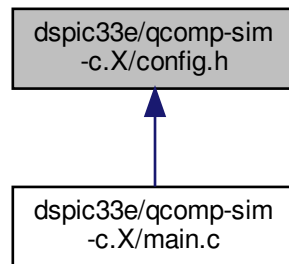
two-qubit gate with display

does controlled 2x2 operator displays the state waits to let the user see the state

## 6.3 dspic33e/qcomp-sim-c.X/config.h File Reference

General config settings #pragma for microcontroller.

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



### 6.3.1 Detailed Description

General config settings #pragma for microcontroller.

#### Authors

J Scott, O Thomas

#### Date

Nov 2018

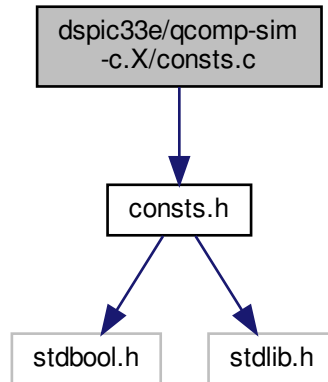
Description: Include this once at the top of main

## 6.4 dspic33e/qcomp-sim-c.X/consts.c File Reference

contains all (global) constants)

```
#include "consts.h"
```

Include dependency graph for consts.c:



## Variables

- const [Complex rX](#) [2][2]
- const [Complex rXT](#) [2][2]
- const [Complex X](#) [2][2]
- const [Complex Y](#) [2][2]
- const [Complex Z](#) [2][2]
- const [Complex H](#) [2][2]

### 6.4.1 Detailed Description

contains all (global) constants)

#### Authors

J Scott, O Thomas

#### Date

Nov 2018

### 6.4.2 Variable Documentation

#### 6.4.2.1 H

```
const Complex H[2][2]
```

#### Initial value:

```
= {{ {0.7071067812, 0.0}, {0.7071067812, 0.0}},
    {{0.7071067812, 0.0}, {-0.7071067812, 0.0}} }
```

## Parameters

$H$	Hadamard gate
-----	---------------

6.4.2.2  $rX$ 

```
const Complex rX[2][2]
```

## Initial value:

```
= {{0.5, 0.5},{0.5, -0.5}},
   {{0.5, -0.5},{0.5, 0.5}}
```

## Parameters

$rX$	sqrt X gate ( 0.5+0.5i 0.5-0.5i ) ( 0.5-0.5i 0.5+0.5i )
------	---

6.4.2.3  $rXT$ 

```
const Complex rXT[2][2]
```

## Initial value:

```
= {{0.5, -0.5},{0.5, 0.5}},
   {{0.5, 0.5},{0.5, -0.5}}
```

## Parameters

$rXT$	Adjoint of $rX$
-------	-----------------

6.4.2.4  $X$ 

```
const Complex X[2][2]
```

## Initial value:

```
= {{0.0, 0.0},{ONE_Q15, 0.0}},
   {{ONE_Q15, 0.0},{0.0, 0.0}}
```

**Parameters**

X	Pauli X gate
---	--------------

**6.4.2.5 Y**

```
const Complex Y[2][2]
```

**Initial value:**

```
= {{{0.0, 0.0}, {0.0, -1.0}},
    {{0.0, ONE_Q15}, {0.0, 0.0}}}
```

**Parameters**

Y	Pauli y gate
---	--------------

**6.4.2.6 Z**

```
const Complex Z[2][2]
```

**Initial value:**

```
= {{{ONE_Q15, 0.0}, {0.0, 0.0}},
    {{0.0, 0.0}, {-1.0, 0.0}}}
```

**Parameters**

Z	Pauli z gate
---	--------------

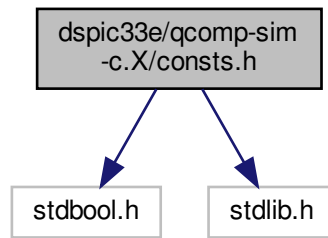
**6.5 dspic33e/qcomp-sim-c.X/consts.h File Reference**

header file for (global) constants

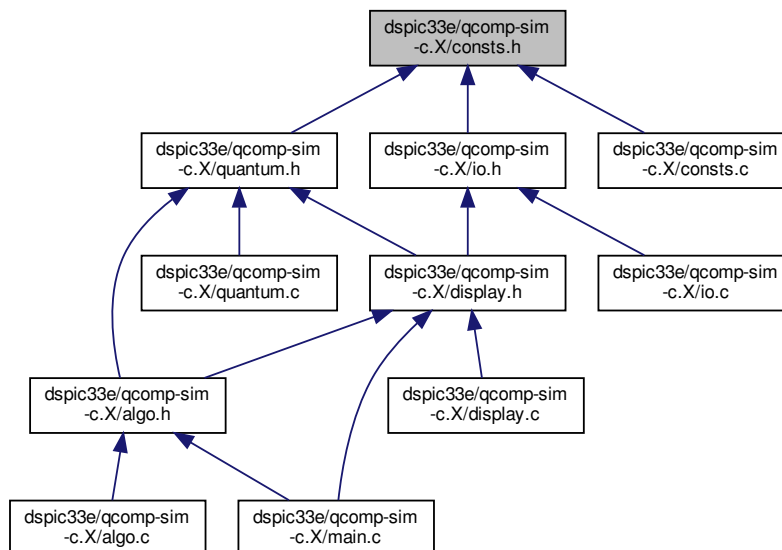
```
#include <stdbool.h>
#include <stdlib.h>
```



Include dependency graph for consts.h:



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



## Macros

- #define **NUM\_QUBITS** 4
- #define **STATE\_LENGTH** 16
- #define **FULL\_PHASE** 0.124996185
- #define **HALF\_PHASE** 0.062498093
- #define **LED\_NUM** 4  
*The number of external LEDs.*
- #define **NUM\_BTNS** 9  
*number of total buttons*
- #define **ONE\_Q15** 0.9999694824
- #define **BTN\_CHIP\_NUM** 2

## Typedefs

- typedef signed \_Fract [Q15](#)  
*Basic fractional time.*
- typedef [Q15 Complex](#)[2]  
*Complex type.*

## Variables

- const [Complex](#) rX [2][2]
- const [Complex](#) rXT [2][2]
- const [Complex](#) X [2][2]
- const [Complex](#) Y [2][2]
- const [Complex](#) Z [2][2]
- const [Complex](#) H [2][2]

### 6.5.1 Detailed Description

header file for (global) constants

#### Authors

J Scott, O Thomas

#### Date

Nov 2018

### 6.5.2 Variable Documentation

#### 6.5.2.1 H

```
const Complex H[2][2]
```

##### Parameters

<i>H</i>	Hadamard gate
----------	---------------

#### 6.5.2.2 rX

```
const Complex rX[2][2]
```

## Parameters

$rX$	is square root of $X$
$rX$	sqrt $X$ gate ( $0.5+0.5i$ $0.5-0.5i$ ) ( $0.5-0.5i$ $0.5+0.5i$ )

6.5.2.3  $rXT$ 

```
const Complex rXT[2][2]
```

## Parameters

$rXT$	Adjoint of $rX$
-------	-----------------

6.5.2.4  $X$ 

```
const Complex X[2][2]
```

## Parameters

$X$	pauli $X$ gate
-----	----------------

6.5.2.5  $Y$ 

```
const Complex Y[2][2]
```

## Parameters

$Y$	Pauli $y$ gate
-----	----------------

6.5.2.6  $Z$ 

```
const Complex Z[2][2]
```

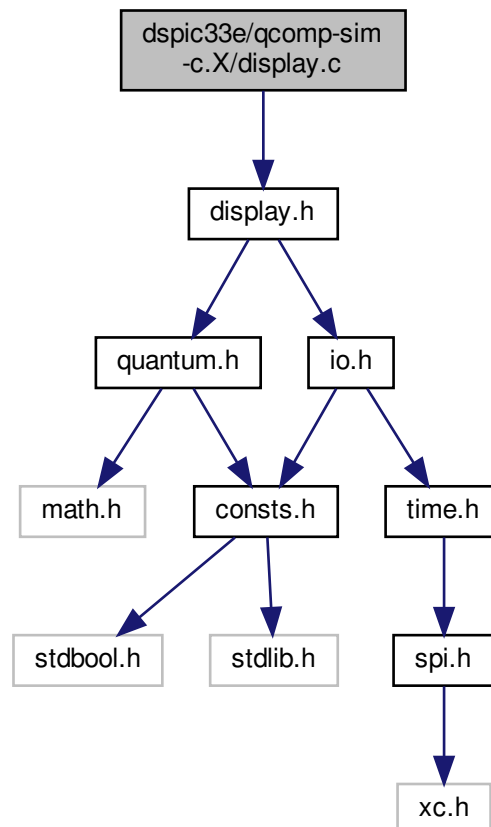
## Parameters

$Z$	Pauli $z$ gate
-----	----------------

## 6.6 dspic33e/qcomp-sim-c.X/display.c File Reference

for all the state display functions

```
#include "display.h"
Include dependency graph for display.c:
```



### Macros

- `#define NUM_MAX_AMPS 4`

### Functions

- void `display_average` (Complex state[])  
*Display the state amplitudes on LEDs.*
- void `display_cycle` (Complex state[])  
*cycles through the non-zero amplitude states*
- int `sort_states` (Complex state[], int num\_qubits)
- int `remove_zero_amp_states` (Complex state[], int disp\_state[])  
*takes state vector, number of qubits and vector to write the nonzero elements of the statevector to.*

## 6.6.1 Detailed Description

for all the state display functions

## 6.6.2 Macro Definition Documentation

### 6.6.2.1 NUM\_MAX\_AMPS

```
#define NUM_MAX_AMPS 4
```

#### Parameters

<i>state</i>	The state vector
<i>num_qubits</i>	The number of qubits in the state vector

#### Returns

This function finds the amplitude of the state vector with the largest magnitude.

## 6.6.3 Function Documentation

### 6.6.3.1 display\_average()

```
void display_average (
    Complex state[] )
```

Display the state amplitudes on LEDs.

#### Parameters

<i>state</i>	Pass in the state vector
--------------	--------------------------

#### Note

Currently the function only displays superpositions using the red and blue colors.

The routine works by adding up the squares of the amplitudes corresponding to each state of a given qubit. Suppose there are three qubits. Then the state vector is given by

*	index	binary	amplitude
*	0	0 0 0	a0
*	1	0 0 1	a1
*	2	0 1 0	a2
*	3	0 1 1	a3
*	4	1 0 0	a4
*	5	1 0 1	a5
*	6	1 1 0	a6
*	7	1 1 1	a7
*	-----		
*	Qubit:	2 1 0	
*			

Consider qubit 2. The value of the ZERO state is formed by adding up all the amplitudes corresponding to its ZERO state. That is, indices 0, 1, 2 and 3. The ONE state is obtained by adding up the other indices: 4, 5, 6 and

1.

So the amplitudes for qubit 2 are

ZERO:  $(a_0)^2 + (a_1)^2 + (a_2)^2 + (a_3)^2$  ONE:  $(a_4)^2 + (a_5)^2 + (a_6)^2 + (a_7)^2$

Corresponding to the following indices:

ZERO: 0+0, 1+0, 2+0, 3+0 ONE: 4+0, 5+0, 6+0, 7+0

For qubit 1 the indices are:

ZERO: 0+0, 0+4, 1+0, 1+4 ONE: 2+0, 2+4, 3+0, 3+4

And for qubit 0 the indices are:

ZERO: 0+0, 0+2, 0+4, 0+6 ONE: 1+0, 1+2, 1+4, 1+6

The examples above are supposed to show the general pattern. For N qubits, qubit number k, the ZERO and ONE states are given by summing all the square amplitudes corresponding to the following indices:

ZERO:  $n + (2^{(k+1)} * j)$ , where  $n = 0, 1, \dots, 2^k - 1$  and  $j = 0, 1, \dots, 2^{(N-k-2)}$

ONE:  $n + (2^{(k+1)} * j)$ , where  $n = 2^k, 2^k + 1, \dots, 2^{(k+1)} - 1$  and  $j = 0, 1, \dots, 2^{(N-k-2)}$

The amplitudes are obtained by summing over both n and j. Notice that there is an edge condition when  $k = N-1$ . There, j acycle\_lengthpparently ranges from 0 to -1. In this case, the only value of j is 0. The condition arises because of the way that  $2^{(N-k-2)}$  is obtained (i.e. such that multiplying it by  $2^{(k+1)}$  gives  $2^{(N-1)}$ .) However, if  $k = N-1$ , then  $2^{(k+1)} = 2^N$  already, so it must be multiplied by  $2^{(-1)}$ . The key point is that the second term should not ever equal  $2^N$ , so j should stop at 0.

The above indices can be expressed as the sum of a ROOT and a STEP as follows:

index = ROOT + STEP

where ROOT ranges from 0 to  $2^k - 1$ . This corresponds to the n values that give rise to ZERO. The indices for ONE can be obtained by adding  $2^k$  to root. The STEP = j is a multiple of  $2^{(k+1)}$  starting from zero that does not equal or exceed  $2^N$ . ROOT can be realised using the following for loop:

```
for(int root = 0; root < 2^k; root++) { ... // ZERO index root; // ONE index root + 2^k; }
```

Then the STEP component can be realised as

```
for(int step = 0; step < 2^N; step += 2^(k+1)) { // Add the following to root... step; }
```

**Todo** Bring all constants out of the loops. Don't use pow. Loop over all qubits  $k = 0, 1, 2, \dots N-1$

Compute powers of 2

ROOT loop: starts at 0, increases in steps of 1

STEP loop: starts at 0, increases in steps of  $2^{(k+1)}$

sign returns an int between 0 & 3 depending which quadrant the amp is in get the difference between quadrants if 0&3 do modulo 2 to get 1.

absolute value of the difference phase zero state - phase 1 state

```
/// c now equals 0      - no phase diff
///                   1 or 3 - re or im phase diff
///                   2      - complete phase diff
///
```

Zeros are at the index root + step

**Todo** Rewrite pow for Q15

Ones are at the index root +  $2^k$  + step

write phase update leds for each qubits average zero and one amps

### 6.6.3.2 display\_cycle()

```
void display_cycle (
    Complex state[] )
```

cycles through the non-zero amplitude states

#### Parameters

<i>state</i>	The state to display
<i>N</i>	The length of the state vector

Filter the state

Allocate RGB array

Decode

Look at the jth bit

Reset the cycle

Each iteration of this loop writes

Loop here to add stuff

### 6.6.3.3 remove\_zero\_amp\_states()

```
int remove_zero_amp_states (
    Complex state[],
    int disp_state[] )
```

takes state vector, number of qubits and vector to write the nonzero elements of the statevector to.

updates disp\_state where the first 'return value of the function' elements are the nonzero elements of the state vector 'state'

the disp\_state elements are the nonzero elements of the state

```
/// e.g. state = (00) = (1/r2) (Bell state)
///              (01)  ( 0 )
///              (10)  ( 0 )
///              (11)  (1/r2)
/// Then displ_state would have 2 elements
/// disp_state = (0) standing for (00)
///              (3)              (11)
///
```

#### Note

we have to allocate disp\_state to be the size of state, the function returns count which tells us the first 'count' elements of disp\_state to use. In the Bell state example there are 2 values in disp\_state, 0 & 3, count is returned as 3 which means take the first count-1 elements (in this case 2) of disp\_state which is 0,1 which is the correct elements

### 6.6.3.4 sort\_states()

```
int sort_states (
    Complex state[],
    int num_qubits )
```

**Todo** this

**Todo** this function...



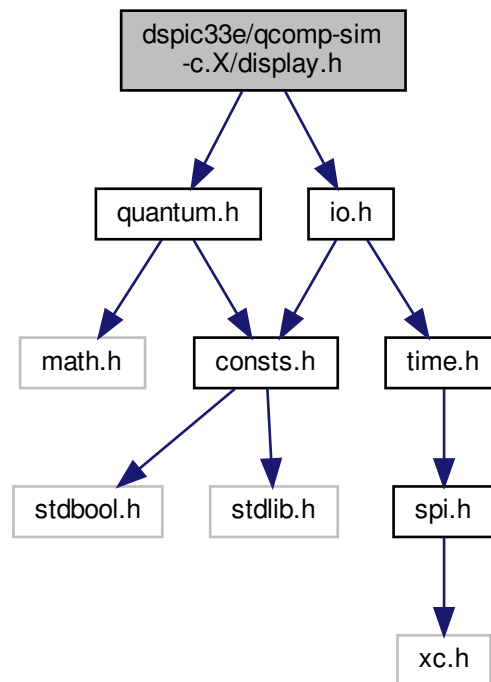
## 6.7 dspic33e/qcomp-sim-c.X/display.h File Reference

Description: Header file containing all the functions for displaying the qubits state vector.

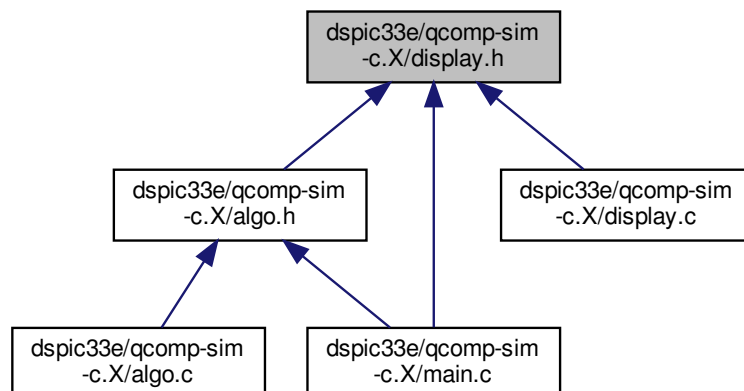
```
#include "quantum.h"
```

```
#include "io.h"
```

Include dependency graph for display.h:



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



## Functions

- void `display_average` (`Complex` state[])  
*Display the state amplitudes on LEDs.*
- void `display_cycle` (`Complex` state[])  
*cycles through the non-zero amplitude states*
- int `remove_zero_amp_states` (`Complex` state[], int disp\_state[])  
*updates disp\_state where the first 'return value of the function' elements are the nonzero elements of the state vector 'state'*
- int `sort_states` (`Complex` state[], int num\_qubits)

### 6.7.1 Detailed Description

Description: Header file containing all the functions for displaying the qubits state vector.

#### Authors

J Scott, O Thomas

#### Date

Nov 2018

### 6.7.2 Function Documentation

#### 6.7.2.1 `display_average()`

```
void display_average (
    Complex state[] )
```

Display the state amplitudes on LEDs.

#### Parameters

<i>state</i>	Pass in the state vector
--------------	--------------------------

#### Note

Currently the function only displays superpositions using the red and blue colors.

**Todo** rename to `display_average`

#### Parameters

<i>state</i>	Pass in the state vector
--------------	--------------------------

**Note**

Currently the function only displays superpositions using the red and blue colors.

The routine works by adding up the squares of the amplitudes corresponding to each state of a given qubit. Suppose there are three qubits. Then the state vector is given by

*	index	binary	amplitude
*	0	0 0 0	a0
*	1	0 0 1	a1
*	2	0 1 0	a2
*	3	0 1 1	a3
*	4	1 0 0	a4
*	5	1 0 1	a5
*	6	1 1 0	a6
*	7	1 1 1	a7
*	-----		
*	Qubit:	2 1 0	
*	-----		

Consider qubit 2. The value of the ZERO state is formed by adding up all the amplitudes corresponding to its ZERO state. That is, indices 0, 1, 2 and 3. The ONE state is obtained by adding up the other indices: 4, 5, 6 and

1.

So the amplitudes for qubit 2 are

ZERO:  $(a_0)^2 + (a_1)^2 + (a_2)^2 + (a_3)^2$  ONE:  $(a_4)^2 + (a_5)^2 + (a_6)^2 + (a_7)^2$

Corresponding to the following indices:

ZERO: 0+0, 1+0, 2+0, 3+0 ONE: 4+0, 5+0, 6+0, 7+0

For qubit 1 the indices are:

ZERO: 0+0, 0+4, 1+0, 1+4 ONE: 2+0, 2+4, 3+0, 3+4

And for qubit 0 the indices are:

ZERO: 0+0, 0+2, 0+4, 0+6 ONE: 1+0, 1+2, 1+4, 1+6

The examples above are supposed to show the general pattern. For N qubits, qubit number k, the ZERO and ONE states are given by summing all the square amplitudes corresponding to the following indices:

ZERO:  $n + (2^{(k+1)} * j)$ , where  $n = 0, 1, \dots, 2^k - 1$  and  $j = 0, 1, \dots, 2^{(N-k-2)}$

ONE:  $n + (2^{(k+1)} * j)$ , where  $n = 2^k, 2^k + 1, \dots, 2^{(k+1)} - 1$  and  $j = 0, 1, \dots, 2^{(N-k-2)}$

The amplitudes are obtained by summing over both n and j. Notice that there is an edge condition when  $k = N-1$ . There, j acycle\_lengthpparently ranges from 0 to -1. In this case, the only value of j is 0. The condition arises because of the way that  $2^{(N-k-2)}$  is obtained (i.e. such that multiplying it by  $2^{(k+1)}$  gives  $2^{(N-1)}$ .) However, if  $k = N-1$ , then  $2^{(k+1)} = 2^N$  already, so it must be multiplied by  $2^{(-1)}$ . The key point is that the second term should not ever equal  $2^N$ , so j should stop at 0.

The above indices can be expressed as the sum of a ROOT and a STEP as follows:

index = ROOT + STEP

where ROOT ranges from 0 to  $2^k-1$ . This corresponds to the n values that give rise to ZERO. The indices for ONE can be obtained by adding  $2^k$  to root. The STEP = j is a multiple of  $2^{(k+1)}$  starting from zero that does not equal or exceed  $2^N$ . ROOT can be realised using the following for loop:

```
for(int root = 0; root < 2^k; root++) { ... // ZERO index root; // ONE index root + 2^k; }
```

Then the STEP component can be realised as

```
for(int step = 0; step < 2^N; step += 2^(k+1)) { // Add the following to root... step; }
```

**Todo** Bring all constants out of the loops. Don't use pow. Loop over all qubits  $k = 0, 1, 2, \dots N-1$

Compute powers of 2

ROOT loop: starts at 0, increases in steps of 1

STEP loop: starts at 0, increases in steps of  $2^k$

sign returns an int between 0 & 3 depending which quadrant the amp is in get the difference between quadrants if 0&3 do modulo 2 to get 1.

absolute value of the difference phase zero state - phase 1 state

```
/// c now equals 0      - no phase diff
///                   1 or 3 - re or im phase diff
///                   2      - complete phase diff
///
```

Zeros are at the index root + step

**Todo** Rewrite pow for Q15

Ones are at the index root +  $2^k$  + step

write phase update leds for each qubits average zero and one amps

#### 6.7.2.2 display\_cycle()

```
void display_cycle (
    Complex state[] )
```

cycles through the non-zero amplitude states

##### Parameters

<i>state</i>	The state to display
<i>N</i>	The length of the state vector

Filter the state

Allocate RGB array

Decode

Look at the jth bit

Reset the cycle

Each iteration of this loop writes

Loop here to add stuff

## 6.7.2.3 remove\_zero\_amp\_states()

```
int remove_zero_amp_states (
    Complex state[],
    int disp_state[] )
```

updates disp\_state where the first 'return value of the function'elements are the nonzero elements of the state vector 'state'

## Parameters

<i>state</i>	complex state vector in
<i>disp_state</i>	complex inout vector where the first n entries are the nonzero elements of 'state'

## Returns

returns the number of elements to look at in disp\_state.

updates disp\_state where the first 'return value of the function'elements are the nonzero elements of the state vector 'state'

the disp\_state elements are the nonzero elements of the state

```
/// e.g. state = (00) = (1/r2) (Bell state)
///              (01)  ( 0 )
///              (10)  ( 0 )
///              (11)  (1/r2)
/// Then displ_state would have 2 elements
/// disp_state = (0) standing for (00)
///              (3)              (11)
///
```

## Note

we have to allocate disp\_state to be the size of state, the function returns count which tells us the first 'count' elements of disp\_state to use. In the Bell state example there are 2 values in disp\_state, 0 & 3, count is returned as 3 which means take the first count-1 elements (in this case 2) of disp\_state which is 0,1 which is the correct elements

## 6.7.2.4 sort\_states()

```
int sort_states (
    Complex state[],
    int num_qubits )
```

**Todo** this

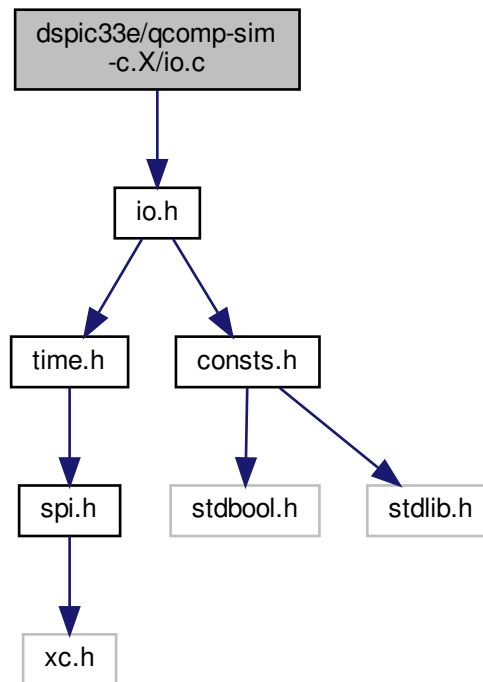
**Todo** this function...

## 6.8 dspic33e/qcomp-sim-c.X/io.c File Reference

Contains all the functions for reading buttons and writing to LEDs.

```
#include "io.h"
```

Include dependency graph for io.c:



### Macros

- `#define DISPLAY_CHIP_NUM 2`
- `#define MAX_CYCLE_LENGTH 16`
- `#define PERIOD 500000`

### Functions

- `int led_color_int (int device, int R, int G, int B)`  
Takes led number & *RGB* -> returns integer for sending via SPI to set the *LED*.
- `int setup_io (void)`  
Set up LEDs and buttons on port D.
- `void __attribute__ ((__interrupt__, no_auto_psv))`  
The max value for *isr\_counter*.
- `void setup_external_leds (void)`  
Set external variable *RGB* LEDs.

- int `add_to_cycle` (RGB colors[], int size)  
*Add an item to the list of states to cycle.*
- int `reset_cycle` (void)  
*Reset the LED display cycle \_\*.*
- void `stop_external_leds` (void)  
*Stop LEDs flashing.*
- void `set_strobe` (int color, int state)  
*Set an LED strobing.*
- void `toggle_strobe` (int color)  
*Toggle LED strobe.*
- int `set_led` (int color, int state)  
*Turn a particular LED on or off.*
- int `read_btn` (int btn)  
*Read the state of a push button.*
- void `leds_off` (void)  
*Turn all the LEDs off.*
- void `flash_led` (int color, int number)  
*Flash LED a number of times.*
- void `flash_all` (int number)  
*Flash all the LEDs a number of times.*
- int `update_display_buffer` (int n, bool R, bool G, bool B)
- int `write_display_driver` (void)  
*Turn on an LED via the external display driver.*
- int `TLC591x_mode_switch` (int mode)  
*Switch between normal and special mode.*
- int `set_external_led` (int index, unsigned \_Fract R, unsigned \_Fract G, unsigned \_Fract B)  
*Updates color properties of global led array.*
- int `read_external_buttons` (void)  
*Read external buttons.*
- int `led_cycle_test` (void)  
*Loop to cycle through LEDs 0 - 15.*
- void `varying_leds` (void)  
*Routine to test the set\_external\_led function.*

## Variables

- int `buttons` [BTN\_CHIP\_NUM]  
*Contains the button states.*
- `LED_GLOBAL led_global` = {0}
- `LED led` [LED\_NUM]  
*The LED array – global in this file.*
- int `display_buf` [DISPLAY\_CHIP\_NUM] = {0}  
*Display buffer to be written to display driver.*
- unsigned \_Fract `isr_counter` = 0  
*Counter for the interrupt service routine \_T5Interrupt.*
- unsigned \_Fract `isr_res` = 0.01  
*Counter value.*
- const unsigned \_Fract `isr_limit` = 0.95  
*Counter resolution.*
- `RGB cycle_colors` [MAX\_CYCLE\_LENGTH][NUM\_QUBITS]

- int **last\_row** = 0
- int **cycle\_counter** = 0
- **BTN btn\_qubit** [NUM\_QUBITS] = 1  
*button mapping 1st byte 00000100 btn A26-28 -> logical 0 00000010 btn A7-9 -> logical 6 00000001 btn A4-6 -> logical 7 00001000 btn A1-3 -> logical 8*
- **btn\_qubit** [0] **chip** = 0  
*logical 0*
- **btn\_qubit** [0] **line** = 2
- **BTN btn\_func** [NUM\_BTNS - NUM\_QUBITS] = 1  
*logical 4*

### 6.8.1 Detailed Description

Contains all the functions for reading buttons and writing to LEDs.

#### Authors

J Scott, O Thomas

#### Date

Nov 2018

### 6.8.2 Function Documentation

#### 6.8.2.1 \_\_attribute\_\_()

```
void __attribute__ (
    (__interrupt__, no_auto_psv) )
```

The max value for isr\_counter.

Timer 6 and 7 for cycling superposition states.

Interrupt service routine for timer 4

Interrupt service routines are automatically called by the microcontroller when an event occurs. In this case, `T5Interrupt` is called when the 32 bit timer formed from T4 and T5 reaches its preset period. The silly name and sill attributes are so that the compiler can correctly map the function in the microcontroller memory. More details of interrupts and interrupt vectors can be found in the compiler manual and the dsPIC33E datasheet.

The job of this routine is to control the modulated brightnesses of the RBG LEDs. This routine is set to be called periodically with a very long period on the time scale of microcontroller operations, but very fast in comparison to what the eye can see. For example, once every 100us. Loop over all the LEDs (the index i).

Decide whether R, G or B should be turned off

Write the display buffer data to the display drivers It's important this line goes here rather than after the the final `update_display_buffer` below. Otherwise you get a flicker due to the LEDs all coming on at the start of this loop

Reset the counter

Turn on all the LEDs back on

Write a row to the leds



### 6.8.2.2 add\_to\_cycle()

```
int add_to_cycle (
    RGB colors[],
    int size )
```

Add an item to the list of states to cycle.

Add an element to the states to be cycled.

#### Parameters

<i>leds</i>	An array of LED indices
<i>colors</i>	Corresponding RGB values for each LED
<i>size</i>	The size of both the above arrays

This function is used to add a set of LED states (RGB values) into the list of states being cycled.

Repeatedly calling this function adds a new state to the end of the list of displayed states. LED states are shown in the order this function is called.

The implementation uses the linked list type `cycle_node`. Each call of this function adds a new element to the end of cycle node Add the new colors to top of array

### 6.8.2.3 flash\_all()

```
void flash_all (
    int number )
```

Flash all the LEDs a number of times.

#### Parameters

<i>number</i>	
---------------	--

### 6.8.2.4 flash\_led()

```
void flash_led (
    int color,
    int number )
```

Flash LED a number of times.

Flash one LED a number of times.

### 6.8.2.5 led\_color\_int()

```
int led_color_int (
    int device,
    int R,
    int G,
    int B )
```

Takes led number & **RGB** -> returns integer for sending via SPI to set the **LED**.

#### Parameters

<i>device</i>	input <b>LED</b> number to change
<i>R</i>	red value between 0 & 1
<i>G</i>	green value between 0 & 1
<i>B</i>	blue value between 0 & 1

#### Returns

Returns int to be sent to **LED** Driver

convention **RGB** -> 000

Each **LED** takes 3 lines, assumes there are no gaps between **LED** channels "device" goes between 0 to  $2^n - 1$

### 6.8.2.6 led\_cycle\_test()

```
int led_cycle_test (
    void )
```

Loop to cycle through LEDs 0 - 15.

**Todo** This won't work now: write\_display\_driver(counter);

### 6.8.2.7 read\_btn()

```
int read_btn (
    int btn )
```

Read the state of a push button.

#### Parameters

<i>btn</i>	
------------	--

**Note**

How well do you know C

**6.8.2.8 read\_external\_buttons()**

```
int read_external_buttons (
    void )
```

Read external buttons.

Update the buttons array (see declaration above)

The external buttons are interfaced to the microcontroller via a shift register. Data is shifted in a byte at a time using the SPI 3 module. The sequence to read the buttons is as follows:

1) Momentarily bring SH low to latch button data into the shift registers 2) Bring CLK\_INH low to enable the clock input on the shift register 3) Start the SPI 3 clock and read data in via the SDI 3 line

The control lines SH and CLK\_INH are on port D

**Todo** read buttons

SH pin

**Todo** How long should this be?

**Todo** button remappings...

**6.8.2.9 reset\_cycle()**

```
int reset_cycle (
    void )
```

Reset the **LED** display cycle \_\*.

Reset the display cycle. Called before adding anything.

**Todo** do it

**6.8.2.10 set\_external\_led()**

```
int set_external_led (
    int index,
    unsigned _Fract R,
    unsigned _Fract G,
    unsigned _Fract B )
```

Updates color properties of global led array.

**Parameters**

<i>led_index</i>	
<i>R</i>	red value between 0 & 1
<i>G</i>	green value between 0 & 1
<i>B</i>	blue value between 0 & 1

**Returns**

0 if successful, -1 otherwise

Use the function to set the **RGB** level of an **LED**. The **LED** is chosen using the

**Parameters**

<i>led_index.</i>	The
<i>R</i>	

**6.8.2.11 set\_led()**

```
int set_led (
    int color,
    int state )
```

Turn a particular **LED** on or off.

**Parameters**

<i>color</i>	
<i>state</i>	

**6.8.2.12 set\_strobe()**

```
void set_strobe (
    int color,
    int state )
```

Set an **LED** strobing.

**Parameters**

<i>color</i>	
<i>state</i>	

**6.8.2.13 setup\_external\_leds()**

```
void setup_external_leds (
    void )
```

Set external variable **RGB** LEDs.

Initialise **LED** lines

Initialise parameters to zero

Initialise display buffer to zero

Set flashing period

Turn timer 6 on

**Todo** CURRENTLY CYCLING IS OFF

**6.8.2.14 setup\_io()**

```
int setup_io (
    void )
```

Set up LEDs and buttons on port D.

< Set port c digital for spi3

Set the OE pin high

Set OE(ED2) pin

Set the SH pin high

Set SH pin

set CLK\_INH high while buttons are pressed

**6.8.2.15 TLC591x\_mode\_switch()**

```
int TLC591x_mode_switch (
    int mode )
```

Switch between normal and special mode.

The mode switch for the TLC591x chip is a bit tricky because it involves synchronising the control lines **LE(ED1)** and OE(ED2) on Port D with the SPI 1 clock. To initiate a mode switch, OE(ED2) must be brought low for one clock cycle, and then the value of **LE(ED1)** two clock cycles later determines the new mode. See the diagrams on page 19 of the datasheet

So long as the timing is not strict, we can probably implement the mode switch by starting a non-blocking transfer of 1 byte to the device (which starts the SPI 1 clock), followed by clearing OE(ED2) momentarily and then setting the value of **LE(ED1)** as required. So long as those two things happen before the SPI 1 clock finishes the procedure will probably work. (The reason is the lack of max timing parameters on page 9 for the setup and hold time for ED1 and ED2, which can therefore presumably be longer than one clock cycle.)

## Parameters

<i>mode</i>	
-------------	--

**Todo** mode switcher for LED Driver

## 6.8.2.16 toggle\_strobe()

```
void toggle_strobe (
    int color )
```

Toggle LED strobe.

## Parameters

<i>color</i>	
--------------	--

## 6.8.2.17 update\_display\_buffer()

```
int update_display_buffer (
    int n,
    bool R,
    bool G,
    bool B )
```

## Parameters

<i>index</i>	LED number to modify
<i>R</i>	Intended value of the R led
<i>G</i>	Intended value of the G led
<i>B</i>	Intended value of the B led

## Returns

0 if successful

Could this get any worse!

This function is supposed to make the display writing process more efficient. It updates a global display buffer which is written periodically to the led display drivers. Instead of the display driver function re-reading the desired state of all the LED lines every time it is called, this function can be used to update only the lines that have changed.

There are quite a few potential bugs in here, mainly array out of bounds if the DISPLAY\_CHIP\_NUM is not set correctly or the LED RGB lines are wrong. (Or if there are just bugs.) Set or clear the red LED of the nth LED

Set or clear the red LED of the nth LED

Set or clear the red LED of the nth LED

### 6.8.2.18 write\_display\_driver()

```
int write_display_driver (
    void )
```

Turn on an [LED](#) via the external display driver.

Send a byte to the display driver.

On power on, the chip (TLC591x) is in normal mode which means that the clocked bytes sent to the chip set which LEDs are on and which are off (as opposed to setting the current of the LEDs)

To write to the device, use the SPI module to write a byte to the SDI 1 pin on the chip. Then momentarily set the [LE\(ED1\)](#) pin to latch the data onto the output register. Finally, bring the OE(ED2) pin low to enable the current sinking to turn on the LEDs. See the timing diagram on page 17 of the datasheet for details.

[LE\(ED1\)](#) and OE(ED2) will be on Port D Set [LE\(ED1\)](#) pin

**Todo** How long should this be?

## 6.8.3 Variable Documentation

### 6.8.3.1 btn\_func

```
btn_func = 1
```

logical 4

logical 5

**Bug** this.

### 6.8.3.2 btn\_qubit

```
btn_qubit = 1
```

button mapping 1st byte 00000100 btn A26-28 -> logical 0 00000010 btn A7-9 -> logical 6 00000001 btn A4-6 -> logical 7 00001000 btn A1-3 -> logical 8

logical 3

logical 2

logical 1

2nd byte 10000000 btn A23-25 -> logical 1 00000010 btn A20-22 -> logical 2 00000100 btn A17-19 -> logical 3 00000001 btn A13-15 -> logical 4 00001000 btn A10-12 -> logical 5

### 6.8.3.3 buttons

```
int buttons[BTN_CHIP_NUM]
```

Contains the button states.

Each entry in the array is either 1 if the button is pressed or 0 if not. The array is accessed globally using 'extern buttons;' in a \*.c file. Read buttons array us updated by calling read\_external\_buttons

### 6.8.3.4 chip

```
btn func [4] chip = 0
```

logical 0

logical 8

logical 7

logical 6

logical 5

logical 4

logical 3

logical 2

logical 1

### 6.8.3.5 isr\_counter

```
unsigned _Fract isr_counter = 0
```

Counter for the interrupt service routine \_T5Interrupt.

These variables are for keeping track of the interrupt based [LED](#) pulsing. The type is \_Fract because it is easier to directly compare two \_Fracts than attempt multiplication of integers and \_Fracts (which isn't supported) The limit is not 1 because \_Fract types do not go up to 1.

It's probably a good idea to make sure the isr\_res counter doesn't overflow (by ensuring that isr\_res + isr\_limit does not exceed 0.999..., the max value of unsigned \_Fract).

### 6.8.3.6 led\_global

```
LED\_GLOBAL led_global = {0}
```

#### Parameters

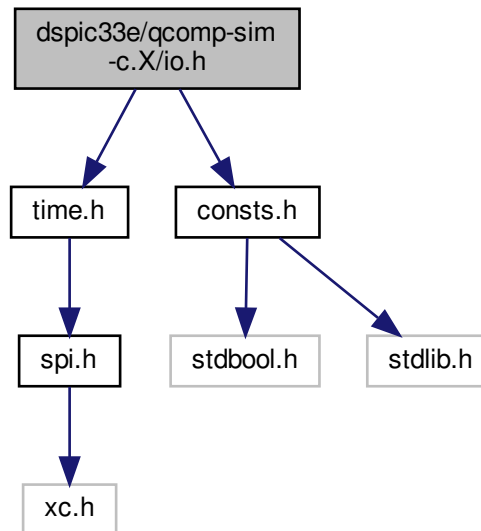
<i>led_global</i>	Global <a href="#">LED</a> strobing state parameter
-------------------	---



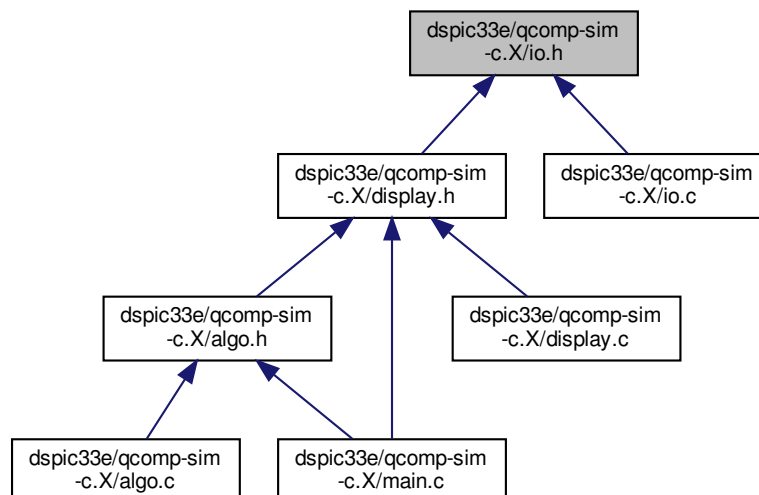
## 6.9 dspic33e/qcomp-sim-c.X/io.h File Reference

Description: Header file for input output functions.

```
#include "time.h"  
#include "consts.h"  
Include dependency graph for io.h:
```



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



## Data Structures

- struct [BTN](#)  
*pin mappings*
- struct [LED\\_GLOBAL](#)  
*Global [LED](#) strobing state parameter.*
- struct [LED](#)  
*Each [LED](#) has the following type.*
- struct [RGB](#)  
*A type for holding red, green, blue values.*
- struct [cycle\\_node](#)  
*The basis for a linked list of states to cycle.*

## Macros

- `#define red 0`  
*Locations of LEDs and buttons on Port D.*
- `#define amber 1`
- `#define green 2`
- `#define sw1 6`
- `#define sw2 7`
- `#define sw3 13`
- `#define off 0`
- `#define on 1`
- `#define LE 3`  
*Control for TLC591x chip on Port D.*
- `#define OE 4`
- `#define SH 5`  
*Control lines for SNx4HC165 chip.*
- `#define CLK_INH 8`

## Typedefs

- `typedef struct cycle\_node cycle\_node\_t`  
*The basis for a linked list of states to cycle.*

## Functions

- `int setup\_io (void)`  
*Set up LEDs and buttons on port D.*
- `void setup\_external\_leds (void)`  
*Set external variable [RGB](#) LEDs.*
- `int set\_led (int color, int state)`  
*Turn a particular [LED](#) on or off.*
- `int read\_btn (int btn)`  
*Read the state of a push button.*
- `void leds\_off (void)`  
*Turn all the LEDs off.*
- `void flash\_led (int color, int number)`

- Flash one [LED](#) a number of times.
- void [flash\\_all](#) (int number)
  - Flash all the LEDs a number of times.
- void [set\\_strobe](#) (int color, int state)
  - Set an [LED](#) strobing.
- void [toggle\\_strobe](#) (int color)
  - Toggle [LED](#) strobe.
- int [update\\_display\\_buffer](#) (int led\_index, bool R, bool G, bool B)
- int [write\\_display\\_driver](#) (void)
  - Send a byte to the display driver.
- int [set\\_external\\_led](#) (int led\_index, unsigned \_Fract R, unsigned \_Fract G, unsigned \_Fract B)
  - Updates color properties of global led array.
- int [led\\_color\\_int](#) (int device, int R, int G, int B)
  - Takes led number & [RGB](#) -> returns integer for sending via SPI to set the [LED](#).
- int [led\\_cycle\\_test](#) (void)
  - Loop to cycle through LEDs 0 - 15.
- int [read\\_external\\_buttons](#) (void)
  - Update the buttons array (see declaration above)
- int [add\\_to\\_cycle](#) ([RGB](#) colors[], int size)
  - Add an element to the states to be cycled.
- int [reset\\_cycle](#) (void)
  - Reset the display cycle. Called before adding anything.

### 6.9.1 Detailed Description

Description: Header file for input output functions.

Include it at the top of any C source file which uses buttons and LEDs. It also defines various constants representing the positions of the buttons and LEDs on port D.

#### Authors

J Scott, O Thomas

#### Date

Nov 2018

### 6.9.2 Function Documentation

#### 6.9.2.1 [add\\_to\\_cycle\(\)](#)

```
int add_to_cycle (
    RGB colors[],
    int size )
```

Add an element to the states to be cycled.

Add an element to the states to be cycled.

**Parameters**

<i>leds</i>	An array of <a href="#">LED</a> indices
<i>colors</i>	Corresponding <a href="#">RGB</a> values for each <a href="#">LED</a>
<i>size</i>	The size of both the above arrays

This function is used to add a set of [LED](#) states ([RGB](#) values) into the list of states being cycled.

Repeatedly calling this function adds a new state to the end of the list of displayed states. [LED](#) states are shown in the order this function is called.

The implementation uses the linked list type [cycle\\_node](#). Each call of this function adds a new element to the end of cycle node Add the new colors to top of array

**6.9.2.2 flash\_all()**

```
void flash_all (
    int number )
```

Flash all the LEDs a number of times.

**Parameters**

<i>number</i>	
---------------	--

**6.9.2.3 flash\_led()**

```
void flash_led (
    int color,
    int number )
```

Flash one [LED](#) a number of times.

**Parameters**

<i>color</i>	
<i>number</i>	

Flash one [LED](#) a number of times.

**6.9.2.4 led\_color\_int()**

```
int led_color_int (
    int device,
    int R,
```

```
int G,  
int B )
```

Takes led number & [RGB](#) -> returns integer for sending via SPI to set the [LED](#).

**Parameters**

<i>device</i>	input LED number to change
<i>R</i>	red value between 0 & 1
<i>G</i>	green value between 0 & 1
<i>B</i>	blue value between 0 & 1

**Returns**

Returns int to be sent to LED Driver

convention RGB -> 000

Each LED takes 3 lines, assumes there are no gaps between LED channels "device" goes between 0 to  $2^n - 1$

**6.9.2.5 led\_cycle\_test()**

```
int led_cycle_test (
    void )
```

Loop to cycle through LEDs 0 - 15.

**Todo** This won't work now: write\_display\_driver(counter);

**6.9.2.6 read\_btn()**

```
int read_btn (
    int btn )
```

Read the state of a push button.

**Parameters**

<i>btn</i>	
------------	--

**Note**

How well do you know C

**6.9.2.7 read\_external\_buttons()**

```
int read_external_buttons (
    void )
```

Update the buttons array (see declaration above)

Update the buttons array (see declaration above)

The external buttons are interfaced to the microcontroller via a shift register. Data is shifted in a byte at a time using the SPI 3 module. The sequence to read the buttons is as follows:

1) Momentarily bring SH low to latch button data into the shift registers 2) Bring CLK\_INH low to enable the clock input on the shift register 3) Start the SPI 3 clock and read data in via the SDI 3 line

The control lines SH and CLK\_INH are on port D

**Todo** read buttons

SH pin

**Todo** How long should this be?

**Todo** button remappings...

#### 6.9.2.8 reset\_cycle()

```
int reset_cycle (
    void )
```

Reset the display cycle. Called before adding anything.

Reset the display cycle. Called before adding anything.

**Todo** do it

#### 6.9.2.9 set\_external\_led()

```
int set_external_led (
    int index,
    unsigned _Fract R,
    unsigned _Fract G,
    unsigned _Fract B )
```

Updates color properties of global led array.

##### Parameters

<i>led_index</i>	
<i>R</i>	red value between 0 & 1
<i>G</i>	green value between 0 & 1
<i>B</i>	blue value between 0 & 1

**Returns**

0 if successful, -1 otherwise

Use the function to set the **RGB** level of an **LED**. The **LED** is chosen using the

**Parameters**

<i>led_index.</i>	The
<i>R</i>	

**6.9.2.10 set\_led()**

```
int set_led (
    int color,
    int state )
```

Turn a particular **LED** on or off.

**Parameters**

<i>color</i>	
<i>state</i>	

**6.9.2.11 set\_strobe()**

```
void set_strobe (
    int color,
    int state )
```

Set an **LED** strobing.

**Parameters**

<i>color</i>	
<i>state</i>	

**6.9.2.12 setup\_external\_leds()**

```
void setup_external_leds (
    void )
```

Set external variable **RGB** LEDs.



Initialise LED lines

Initialise parameters to zero

Initialise display buffer to zero

Set flashing period

Turn timer 6 on

**Todo** CURRENTLY CYCLING IS OFF

#### 6.9.2.13 setup\_io()

```
int setup_io (
    void )
```

Set up LEDs and buttons on port D.

< Set port c digital for spi3

Set the OE pin high

Set OE(ED2) pin

Set the SH pin high

Set SH pin

set CLK\_INH high while buttons are pressed

#### 6.9.2.14 toggle\_strobe()

```
void toggle_strobe (
    int color )
```

Toggle LED strobe.

##### Parameters

<i>color</i>	
--------------	--

#### 6.9.2.15 update\_display\_buffer()

```
int update_display_buffer (
    int n,
```

```

    bool R,
    bool G,
    bool B )

```

#### Parameters

<i>led_index</i>	LED number to modify
<i>R</i>	Intended value of the R led
<i>G</i>	Intended value of the G led
<i>B</i>	Intended value of the B led

#### Returns

0 if successful

#### Parameters

<i>index</i>	LED number to modify
<i>R</i>	Intended value of the R led
<i>G</i>	Intended value of the G led
<i>B</i>	Intended value of the B led

#### Returns

0 if successful

Could this get any worse!

This function is supposed to make the display writing process more efficient. It updates a global display buffer which is written periodically to the led display drivers. Instead of the display driver function re-reading the desired state of all the LED lines every time it is called, this function can be used to update only the lines that have changed.

There are quite a few potential bugs in here, mainly array out of bounds if the DISPLAY\_CHIP\_NUM is not set correctly or the LED RGB lines are wrong. (Or if there are just bugs.) Set or clear the red LED of the nth LED

Set or clear the red LED of the nth LED

Set or clear the red LED of the nth LED

#### 6.9.2.16 write\_display\_driver()

```

int write_display_driver (
    void )

```

Send a byte to the display driver.

Don't use this function to write to LEDs – use the set\_external\_led function

Send a byte to the display driver.

On power on, the chip (TLC591x) is in normal mode which means that the clocked bytes sent to the chip set which LEDs are on and which are off (as opposed to setting the current of the LEDs)

To write to the device, use the SPI module to write a byte to the SDI 1 pin on the chip. Then momentarily set the LE(ED1) pin to latch the data onto the output register. Finally, bring the OE(ED2) pin low to enable the current sinking to turn on the LEDs. See the timing diagram on page 17 of the datasheet for details.

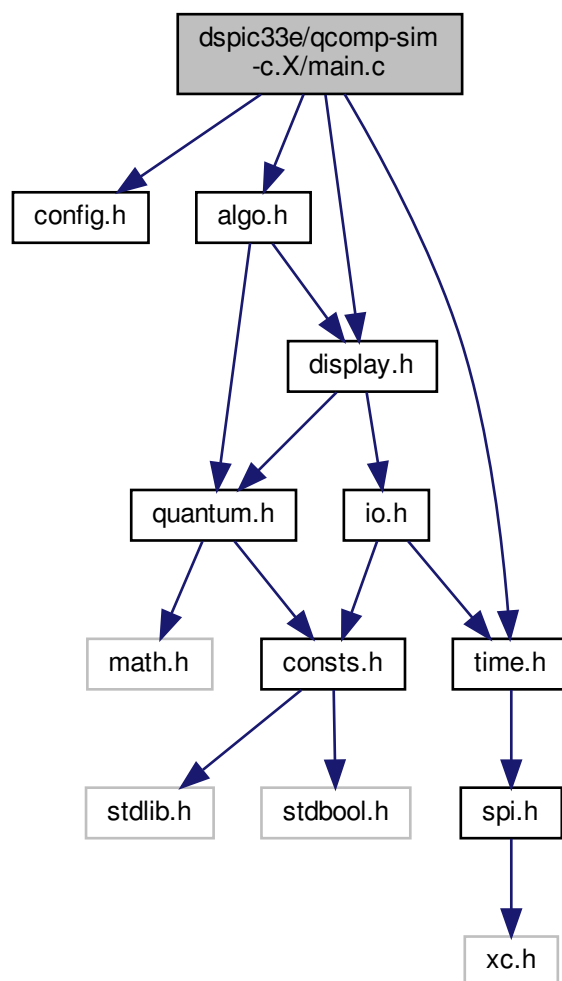
LE(ED1) and OE(ED2) will be on Port D Set LE(ED1) pin

**Todo** How long should this be?

## 6.10 dspic33e/qcomp-sim-c.X/main.c File Reference

The main function.

```
#include "config.h"  
#include "time.h"  
#include "algo.h"  
#include "display.h"  
Include dependency graph for main.c:
```



### Functions

- int `main` (void)

### 6.10.1 Detailed Description

The main function.

#### Authors

J Scott, O Thomas

#### Date

Nov 2018

Contains an example of fixed precision 2x2 matrix multiplication for applying operations to a single qubit. The only operations included are H, X and Z so that everything is real (this can be extended later).

All the functions have now been moved into separate files. [io.h](#) and [io.c](#) contain functions for reading and controlling the buttons and LEDs, and [quantum.h/quantum.c](#) contain the matrix arithmetic for simulating one qubit.

Compile command: make (on linux). But if you want to program the micro- controller too or if you're using windows you're better of downloading and installing MPLAB-X <https://www.microchip.com/mplab/mplab-x-ide>.

#### Note

You also need the microchip xc16 compilers which are available from <https://www.microchip.com/mplab/compilers>

### 6.10.2 Function Documentation

#### 6.10.2.1 main()

```
int main (
    void )
```

button reading test.

<

#### Note

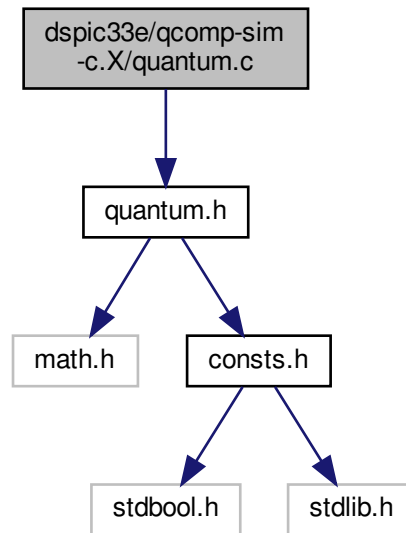
Really important!

## 6.11 dspic33e/qcomp-sim-c.X/quantum.c File Reference

Description: Contains matrix and vector arithmetic for simulating one qubit.

```
#include "quantum.h"
```

Include dependency graph for quantum.c:



### Functions

- `int pow2 (int k)`  
*A simple function to compute integer powers of 2.*
- `int sign (Complex a)`  
*returns phase quadrant*
- `void cadd (const Complex a, const Complex b, Complex result)`
- `void cmul (const Complex a, const Complex b, Complex result)`
- `Q15 absolute (Complex x)`  
*abs function*
- `Q15 square_magnitude (Complex x)`  
*Compute the magnitude squared of a complex number.*
- `void zero_state (Complex state[])`  
*Initialise state to the vacuum (zero apart from the first position) Specify the dimension – of the matrix, i.e.*
- `void mat_mul_old (const Complex M[2][2], Complex V[], int i, int j)`  
*This is an old version of the mat\_mul function.*
- `void mat_mul (const Complex M[2][2], Complex V[], int i, int j)`  
*This version uses inlined cadd and cmul.*
- `void single_qubit_op (const Complex op[2][2], int k, Complex state[])`  
*apply operator*
- `void controlled_qubit_op_new (const Complex op[2][2], int ctrl, int targ, Complex state[])`  
*selective 2 qubit op function*
- `void controlled_qubit_op (const Complex op[2][2], int ctrl, int targ, Complex state[])`  
*Old controlled qubit operations.*

### 6.11.1 Detailed Description

Description: Contains matrix and vector arithmetic for simulating one qubit.

#### Authors

J Scott, O Thomas

#### Date

Nov 2018

**Todo** split into a complex math and operator files

### 6.11.2 Function Documentation

#### 6.11.2.1 absolute()

```
Q15 absolute (
    Complex x )
```

abs function

#### Parameters

x	A complex number to find the absolute value of
---	--

#### Returns

The absolute value

**Todo** Check that the complex part is small

#### 6.11.2.2 controlled\_qubit\_op()

```
void controlled_qubit_op (
    const Complex op[2][2],
    int ctrl,
    int targ,
    Complex state[] )
```

Old controlled qubit operations.

apply controlled 2x2 op ROOT loop: starts at 0, increases in steps of 1

STEP loop: starts at 0, increases in steps of  $2^k$

First index is ZERO, second index is ONE

**Note**

for 2 qubit case check if the index in the ctrl qubit is a 1 then apply the 2x2 unitary else do nothing  
 sorry. this checks for the first element of the state vector i.e. the target qubits  $|0\rangle$  and checks that the state vector element is one which the control qubit has a  $|1\rangle$  state -> (root + step)

The second element of the state vector to take is then the first  $+2^{\wedge}(\text{target qubit number})$ . This also needs to be checked that the control qubit is in the  $|1\rangle$ .

**Todo** This expression can probably be simplified or broken over lines. The condition for the if statement is that root+step and root + step + root\_max contain 1 in the ctrl-th bit.

**6.11.2.3 controlled\_qubit\_op\_new()**

```
void controlled_qubit_op_new (
    const Complex op[2][2],
    int ctrl,
    int targ,
    Complex state[] )
```

selective 2 qubit op function

checks that the control qubit is  $|1\rangle$  then does 2x2 unitary on remaining state vector elements

This routine implements a controlled unitary gate. Controlled unitaries can be expressed as single qubit unitaries that are conditionally applied if the control qubit state (ctrl) is 1. Otherwise no operation is performed.

The following example is for the three qubit case. Suppose the following operation is performed.

```
*      00 01 10 11
* 00 ( 1 0 0 0 )
* 01 ( 0 1 0 0 )
* 10 ( 0 0 u00 u01 )
* 11 ( 0 0 u10 u11 )
*
```

The first qubit is the control (ctrl) and the second qubit is the target (targ). If the control is 0 the identity operation is performed. If the control qubit is 1, then a unitary U (the second block above) is performed.

For three qubits, the state vector is shown below:

```
*      index      binary      amplitude
*      -----
*      0          0 0 0        a0
*      1          0 0 1        a1
*      2          0 1 0        a2
*      3          0 1 1        a3
*      4          1 0 0        a4
*      5          1 0 1        a5
*      6          1 1 0        a6
*      7          1 1 1        a7
*      -----
*      Qubit:      2 1 0
*
```

Suppose the controlled unitary is to be performed between qubits 0 and 1, with the control qubit on 0. Suppose the controlled gate is a CNOT, so that the 2x2 matrices involved are I and X. X and I are performed on the following (vertical) pairs of indices

I                      X

i: (0+0) (0+4) (1+0) (1+4) (ctrl = 0, targ = 1) j: (0+2) (0+6) (1+2) (1+6)

If the control and target are reversed (ctrl on 1), then the pairings of the indices are

I                      X

i: (0+0) (0+4) (2+0) (2+4) (ctrl = 1, targ = 0) j: (0+1) (0+5) (2+1) (2+5)

For control and target qubits on 0 and 2 the indices are

I                      X

i: (0+0) (0+2) (1+0) (1+2) (ctrl = 0, targ = 2) j: (0+4) (0+6) (1+4) (1+6)

If the control and target are reversed (ctrl on 2), then the pairings of the indices are

I                      X

i: (0+0) (0+2) (4+0) (4+4) (ctrl = 2, targ = 0) j: (0+1) (0+3) (4+1) (4+5)

Finally, if the control and target are 1 and 2, then

I                      X

i: (0+0) (0+1) (2+0) (2+1) (ctrl = 1, targ = 2) j: (0+4) (0+5) (2+4) (2+5)

If the control and target are reversed (ctrl on 2), then the pairings of the indices are

I                      X

i: (0+0) (0+1) (4+0) (4+1) (ctrl = 2, targ = 1) j: (0+2) (0+3) (4+2) (4+3)

The pattern in the general case is as follows. Firstly, similarly to the single qubit case, the index required can be expressed as the sum of a root and another contribution. In this case, the root depends only on the ctrl qubit number:

$$\text{root} = x * 2^{\text{ctrl}}$$

where x is the state of the ctrl qubit (either 1 or 0). This will determine whether I or (in the case of CNOT) X is applied. That the root only depends on the ctrl qubit number is due to the interpretation of root – it is the base index of all the ctrl states of a particular value. For example, whatever the qubit number, the starting index of the zero ctrl state is always zero. Then, the first occurrence of a 1 in the ctrl qubit depends on the ctrl qubit number, and is just a power of 2 into the state vector.

The other contributions to the index depend on the target qubit number (targ). The offset between indices of the same operation (either I or X) are separated by

$$\text{sep} = 2^{\text{targ}}$$

The logic for this is similar to the case for ctrl: the way to get from a 0 in the target to a 1 in the target is to add  $2^{\text{targ}}$  to the index in the state vector.

Finally, there is the offset due to moving from the 0 to 1 state within a particular operation (I or X). This depends on both the values of the ctrl and targ qubit numbers as follows:

$$\text{offset} = 2^{(N-\text{ctrl}-\text{targ})} * y$$

where N is the number of \_qubits (3 in the above case). Here, y is either zero or one, and enumerates the operations that must be performed. In other words, the index is given by the following expression

$$\text{i: root} + \text{offset} = x * 2^{\text{ctrl}} + y * 2^{(N-\text{ctrl}-\text{targ})} \quad \text{j: root} + \text{sep} + \text{offset} = x * 2^{\text{ctrl}} + 2^{\text{targ}} + y * 2^{(N-\text{ctrl}-\text{targ})}$$

where x is the value of the ctrl qubit (do X when x is 1, I when x is zero) and y ranges from 0 to  $2^{(N-1)}$  where N is the number of qubits. Since it is only necessary to do the non-trivial unitary, x is always 1.

**Todo** Replace pow2 with left rotations

**Todo** The problem is the formula for the increment



## 6.11.2.4 mat\_mul()

```
void mat_mul (
    const Complex M[2][2],
    Complex V[],
    int i,
    int j )
```

This version uses inlined cadd and cmul.

2x2 complex matrix multiplication

## Parameters

<i>M</i>	A 2x2 complex matrix
<i>V</i>	A Nx1 complex vector
<i>i</i>	The first index to pick from the vector V
<i>j</i>	The second index to pick from the vector V

**Todo** Is static enough? Or should we declare outside the function?

**Todo** Should we use for loops? Or is it better not to..?

This is necessary because the previous computations use V

## 6.11.2.5 mat\_mul\_old()

```
void mat_mul_old (
    const Complex M[2][2],
    Complex V[],
    int i,
    int j )
```

This is an old version of the mat\_mul function.

## Parameters

<i>M</i>	A 2x2 complex matrix
<i>V</i>	A Nx1 complex vector
<i>i</i>	The first index to pick from the vector V
<i>j</i>	The second index to pick from the vector V

The function uses cadd and cmul

**Todo** Should these be outside the function?

### 6.11.2.6 pow2()

```
int pow2 (
    int k )
```

A simple function to compute integer powers of 2.

#### Parameters

<i>k</i>	The exponent of 2 to compute
----------	------------------------------

#### Returns

$2^k$

Multiply by 2

### 6.11.2.7 sign()

```
int sign (
    Complex a )
```

returns phase quadrant

```
///          Im
//          |
//      1   |   0
//          |
//  ----- Re
//          |
//      2   |   3
//          |
///
```

if real negative and im neg return -1 if real negative and im pos return -0.5

else if real pos and im negative return -0.5

else if both pos return 0

### 6.11.2.8 single\_qubit\_op()

```
void single_qubit_op (
    const Complex op[2][2],
    int k,
    Complex state[] )
```

apply operator

#### Parameters

<i>state</i>	state vector containing amplitudes
<i>qubit</i>	qubit number to apply 2x2 matrix to
<i>N</i>	total number of qubits in the state
<i>op</i>	2x2 operator to be applied

This routine applies a single qubit gate to the state vector

#### Parameters

<i>state.</i>	Consider the three qubit case, with amplitudes shown in the table below:
---------------	--

*	index	binary	amplitude
*	-----	-----	-----
*	0	0 0 0	a0
*	1	0 0 1	a1
*	2	0 1 0	a2
*	3	0 1 1	a3
*	4	1 0 0	a4
*	5	1 0 1	a5
*	6	1 1 0	a6
*	7	1 1 1	a7
*	-----	-----	-----
*	Qubit:	2 1 0	
*			

If a single qubit operation is applied to qubit 2, then the 2x2 matrix must be applied to all pairs of (0,1) in the first column, with the numbers in the other columns fixed. In other words, the following indices are paired:

```
(0+0) (1+0) (2+0) (3+0)
(4+0) (5+0) (6+0) (7+0)
```

where the top line corresponds to the ZERO amplitude and the bottom row corresponds to the ONE amplitude.

Similarly, for qubit 1 the pairings are:

```
(0+0) (0+4) (1+0) (1+4)
(2+0) (2+4) (3+0) (3+4)
```

And for qubit 0 the pairings are:

```
(0+0) (0+2) (0+4) (0+6)
(1+0) (1+2) (1+4) (1+6)
```

These numbers are exactly the same as the previous function, which means the same nested loops can be used to perform operation. Now the index

```
root + step
```

refers to the ZERO amplitude (the first element in the column vector to be multiplied by the 2x2 matrix), and the index `Complex state[], int N root + 2^k + step`

corresponds to the ONE entry. ROOT loop: starts at 0, increases in steps of 1

STEP loop: starts at 0, increases in steps of  $2^{(k+1)}$

First index is ZERO, second index is ONE

**Todo** Should we inline `mat_mul` here?

#### 6.11.2.9 square\_magnitude()

```
Q15 square_magnitude (
    Complex x )
```

Compute the magnitude squared of a complex number.

**Parameters**

<code>x</code>	The input complex number <code>x</code>
----------------	---

**Returns**

The value of  $|x|^2$

**Todo** Maybe we should inline this

**6.11.2.10 zero\_state()**

```
void zero_state (
    Complex state[] )
```

Initialise state to the vacuum (zero apart from the first position) Specify the dimension – of the matrix, i.e.

$2^{\text{(number of qubits)}}$

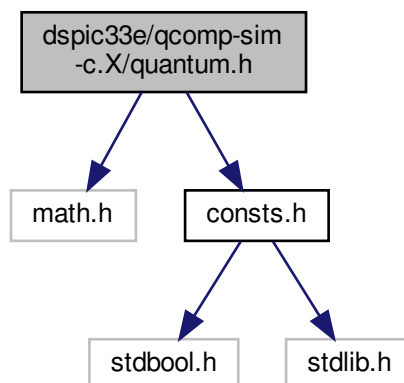
**Note**

oh the clarity!

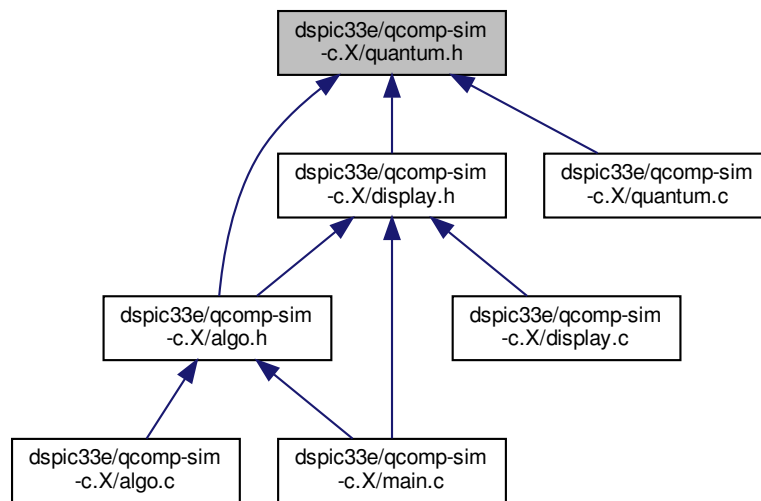
**6.12 dspic33e/qcomp-sim-c.X/quantum.h File Reference**

Description: Header file containing all the matrix arithmetic for simulating a single qubit.

```
#include <math.h>
#include "consts.h"
Include dependency graph for quantum.h:
```



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



## Enumerations

- enum `State` {  
**ZERO**, **ONE**, **PLUS**, **MINUS**,  
**iPLUS**, **iMINUS** }

*Basis states.*

## Functions

- void `zero_state` (`Complex` state[])  
*Initialise state to the vacuum (zero apart from the first position) Specify the dimension – of the matrix, i.e.*
- int `sign` (`Complex` a)  
*returns phase quadrant*
- void `mat_mul` (const `Complex` M[2][2], `Complex` V[], int i, int j)  
*2x2 complex matrix multiplication*
- void `single_qubit_op` (const `Complex` op[2][2], int qubit, `Complex` state[])  
*apply operator*
- void `controlled_qubit_op` (const `Complex` op[2][2], int ctrl, int targ, `Complex` state[])  
*apply controlled 2x2 op*
- `Q15 absolute` (`Complex` x)  
*abs function*
- int `pow2` (int k)  
*A simple function to compute integer powers of 2.*
- `Q15 square_magnitude` (`Complex` x)  
*Compute the magnitude squared of a complex number.*

### 6.12.1 Detailed Description

Description: Header file containing all the matrix arithmetic for simulating a single qubit.

#### Authors

J Scott, O Thomas

#### Date

Nov 2018

### 6.12.2 Function Documentation

#### 6.12.2.1 absolute()

```
Q15 absolute (
    Complex x )
```

abs function

#### Parameters

<i>x</i>	A complex number to find the absolute value of
----------	--

#### Returns

The absolute value

**Todo** Check that the complex part is small

#### 6.12.2.2 controlled\_qubit\_op()

```
void controlled_qubit_op (
    const Complex op[2][2],
    int ctrl,
    int targ,
    Complex state[] )
```

apply controlled 2x2 op

#### Parameters

<i>op</i>	single qubit unitary 2x2
<i>ctrl</i>	control qubit number (0,1,...,n-1)
<i>targ</i>	target qubit number (0,1,...,n-1)
<i>state</i>	complex state vector

apply controlled 2x2 op ROOT loop: starts at 0, increases in steps of 1

STEP loop: starts at 0, increases in steps of  $2^{(k+1)}$

First index is ZERO, second index is ONE

#### Note

for 2 qubit case check if the index in the ctrl qubit is a 1 then apply the 2x2 unitary else do nothing  
sorry. this checks for the first element of the state vector i.e. the target qubits  $|0\rangle$  and checks that the state vector element is one which the control qubit has a  $|1\rangle$  state -> (root + step)

The second element of the state vector to take is then the first  $+2^{(target\ qubit\ number)}$ . This also needs to be checked that the control qubit is in the  $|1\rangle$ .

**Todo** This expression can probably be simplified or broken over lines. The condition for the if statement is that root+step and root + step + root\_max contain 1 in the ctrl-th bit.

#### 6.12.2.3 mat\_mul()

```
void mat_mul (
    const Complex M[2][2],
    Complex V[],
    int i,
    int j )
```

2x2 complex matrix multiplication

#### Parameters

<i>M</i>	complex matrix
<i>V</i>	complex vector
<i>i</i>	integer first element of state vector
<i>j</i>	integer second element of state vector

**Todo** Because of the way the array types work (you can't pass a multidimensional array of unknown size) we will also need a function for 4x4 matrix multiplication.

2x2 complex matrix multiplication

#### Parameters

<i>M</i>	A 2x2 complex matrix
<i>V</i>	A Nx1 complex vector
<i>i</i>	The first index to pick from the vector V
<i>j</i>	The second index to pick from the vector V

**Todo** Is static enough? Or should we declare outside the function?

**Todo** Should we use for loops? Or is it better not to..?

This is necessary because the previous computations use V

#### 6.12.2.4 pow2()

```
int pow2 (
    int k )
```

A simple function to compute integer powers of 2.

##### Parameters

<i>k</i>	The exponent of 2 to compute
----------	------------------------------

##### Returns

$2^k$

Multiply by 2

#### 6.12.2.5 sign()

```
int sign (
    Complex a )
```

returns phase quadrant

```
///          Im
//          |
//      1   |   0
//          |
//  ----- Re
//          |
//      2   |   3
//          |
///
```

if real negative and im neg return -1 if real negative and im pos return -0.5

else if real pos and im negative return -0.5

else if both pos return 0

#### 6.12.2.6 single\_qubit\_op()

```
void single_qubit_op (
    const Complex op[2][2],
    int k,
    Complex state[ ] )
```

apply operator



## Parameters

<i>state</i>	state vector containing amplitudes
<i>qubit</i>	qubit number to apply 2x2 matrix to
<i>op</i>	2x2 operator to be applied
<i>state</i>	state vector containing amplitudes
<i>qubit</i>	qubit number to apply 2x2 matrix to
<i>N</i>	total number of qubits in the state
<i>op</i>	2x2 operator to be applied

This routine applies a single qubit gate to the state vector

## Parameters

<i>state.</i>	Consider the three qubit case, with amplitudes shown in the table below:
---------------	--

```

*      index      binary      amplitude
*      -----
*      0          0 0 0        a0
*      1          0 0 1        a1
*      2          0 1 0        a2
*      3          0 1 1        a3
*      4          1 0 0        a4
*      5          1 0 1        a5
*      6          1 1 0        a6
*      7          1 1 1        a7
*      -----
*      Qubit:      2 1 0
*

```

If a single qubit operation is applied to qubit 2, then the 2x2 matrix must be applied to all pairs of (0,1) in the first column, with the numbers in the other columns fixed. In other words, the following indices are paired:

```

(0+0) (1+0) (2+0) (3+0)
(4+0) (5+0) (6+0) (7+0)

```

where the top line corresponds to the ZERO amplitude and the bottom row corresponds to the ONE amplitude.

Similarly, for qubit 1 the pairings are:

```

(0+0) (0+4) (1+0) (1+4)
(2+0) (2+4) (3+0) (3+4)

```

And for qubit 0 the pairings are:

```

(0+0) (0+2) (0+4) (0+6)
(1+0) (1+2) (1+4) (1+6)

```

These numbers are exactly the same as the previous function, which means the same nested loops can be used to perform operation. Now the index

```
root + step
```

refers to the ZERO amplitude (the first element in the column vector to be multiplied by the 2x2 matrix), and the index `Complex state[], int N root + 2^k + step`

corresponds to the ONE entry. ROOT loop: starts at 0, increases in steps of 1

STEP loop: starts at 0, increases in steps of  $2^{(k+1)}$

First index is ZERO, second index is ONE

**Todo** Should we inline `mat_mul` here?

#### 6.12.2.7 square\_magnitude()

```
Q15 square_magnitude (
    Complex x )
```

Compute the magnitude squared of a complex number.

##### Parameters

x	The input complex number x
---	----------------------------

##### Returns

The value of  $|x|^2$

**Todo** Maybe we should inline this

##### Parameters

x	The input complex number x
---	----------------------------

##### Returns

The value of  $|x|^2$

**Todo** Maybe we should inline this

#### 6.12.2.8 zero\_state()

```
void zero_state (
    Complex state[] )
```

Initialise state to the vacuum (zero apart from the first position) Specify the dimension – of the matrix, i.e.

$2^{\text{(number of qubits)}}$

##### Parameters

state	complex state vector
-------	----------------------

$2^{\text{(number of qubits)}}$

##### Note

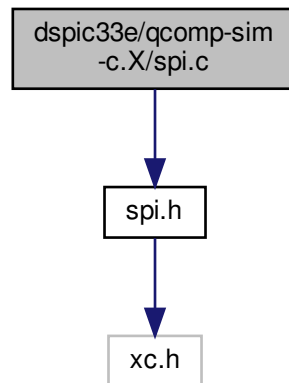
oh the clarity!

## 6.13 dspic33e/qcomp-sim-c.X/spi.c File Reference

Description: Functions for communicating with serial devices.

```
#include "spi.h"
```

Include dependency graph for spi.c:



### Functions

- int [setup\\_spi](#) (void)  
*Set up serial peripheral interface.*
- int [send\\_byte\\_spi\\_1](#) (int data)  
*Send a byte to the SPI1 peripheral.*
- int [read\\_byte\\_spi\\_3](#) ()  
*Recieve a byte from the SPI3 peripheral.*

#### 6.13.1 Detailed Description

Description: Functions for communicating with serial devices.

##### Authors

J Scott, O Thomas

##### Date

Nov 2018

#### 6.13.2 Function Documentation

#### 6.13.2.1 `send_byte_spi_1()`

```
int send_byte_spi_1 (  
    int data )
```

Send a byte to the SPI1 peripheral.

## Parameters

<i>data</i>	byte to be sent to SPI1
-------------	-------------------------

## 6.13.2.2 setup\_spi()

```
int setup_spi (
    void )
```

Set up serial peripheral interface.

Pin mappings — Pin mappings and codes —

J10:41 = J1:91 = uC:70 = RPI74 (PPS code: 0100 1010)  
 J10:44 = J1:93 = uC:9 = RPI52 (PPS code: 0011 0100)  
 J10:47 = J1:101 = uC:34 = RPI42 (PPS code: 0010 1010)  
 J10:43 = J1:95 = uC:72 = RP64 (PPS reg: RPOR0\_L; code: 0100 0000)  
 J10:46 = J1:97 = uC:69 = RPI73 (PPS code: 0100 1001)  
 J10:7 = J1:13 = uC:3 = RP85 (PPS reg: RPOR6\_L; code: 0101 0101)  
 J10:5 = J1:7 = uC:5 = RP87 (PPS reg: RPOR6\_H)  
 J10:55 = J1:117 = uC:10 = RP118 (PPS reg: RPOR13\_H)

— Pin mappings for SPI 1 module —

SPI 1 Clock Out (SCK1) PPS code: 000110 (0x06)  
 SPI 1 Data Out (SDO1) PPS code: 000101 (0x05)  
 SPI 1 Slave Select PPS code: 000111

— Pin mappings for SPI 3 module —

SPI 3 Clock Out (SCK3) PPS code: 100000 (0x20)  
 SPI 3 Data Out (SDO3) PPS code: 011111 (0x1F)  
 SPI 3 Slave Select PPS code: 100001

Configure the SPI 1 pins

< Put SCK1 on J10:43

< Put SDO1 on J10:55

The clock pin also needs to be configured as an input

< Set SCK1 on J10:43 as input

Configure the SPI 3 output pins

< Put SCK3 on J10:7

< Put SDO3 on J10:5

< Put SDI3 on J10:44

< Set SCK3 on J10:7 as input

@note

SPI 1 clock configuration

$SCK1 = F_{CY} / (\text{Primary Prescaler} * \text{Secondary Prescaler})$

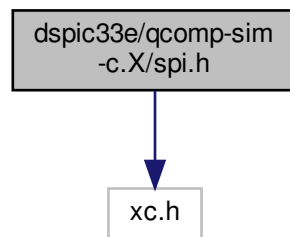
Assuming that  $F_{CY} = 50\text{MHz}$ , and the prescalers are 4 and 1, the SPI clock frequency will be 12.5MHz.

## 6.14 dspic33e/qcomp-sim-c.X/spi.h File Reference

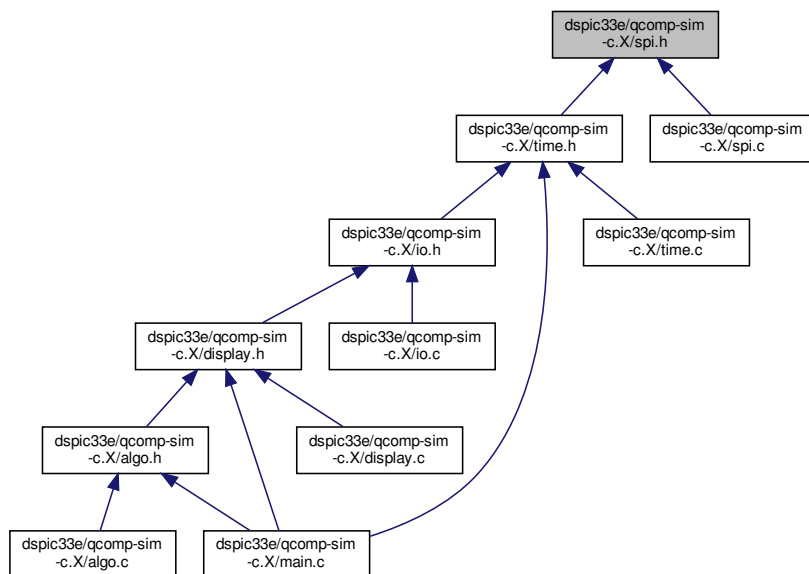
Description: SPI communication functions.

```
#include "xc.h"
```

Include dependency graph for spi.h:



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



### Functions

- int [setup\\_spi](#) (void)  
Set up serial peripheral interface.
- int [send\\_byte\\_spi\\_1](#) (int data)  
Send a byte to the SPI1 peripheral.
- int [read\\_byte\\_spi\\_3](#) ()  
Recieve a byte from the SPI3 peripheral.

### 6.14.1 Detailed Description

Description: SPI communication functions.

#### Authors

J Scott, O Thomas

#### Date

Nov 2018

### 6.14.2 Function Documentation

#### 6.14.2.1 send\_byte\_spi\_1()

```
int send_byte_spi_1 (
    int data )
```

Send a byte to the SPI1 peripheral.

#### Parameters

<i>data</i>	byte to be sent to SPI1
-------------	-------------------------

#### 6.14.2.2 setup\_spi()

```
int setup_spi (
    void )
```

Set up serial peripheral interface.

Pin mappings — Pin mappings and codes —

J10:41 = J1:91 = uC:70 = RPI74 (PPS code: 0100 1010)  
 J10:44 = J1:93 = uC:9 = RPI52 (PPS code: 0011 0100)  
 J10:47 = J1:101 = uC:34 = RPI42 (PPS code: 0010 1010)  
 J10:43 = J1:95 = uC:72 = RP64 (PPS reg: RPOR0\_L; code: 0100 0000)  
 J10:46 = J1:97 = uC:69 = RPI73 (PPS code: 0100 1001)  
 J10:7 = J1:13 = uC:3 = RP85 (PPS reg: RPOR6\_L; code: 0101 0101)  
 J10:5 = J1:7 = uC:5 = RP87 (PPS reg: RPOR6\_H)  
 J10:55 = J1:117 = uC:10 = RP118 (PPS reg: RPOR13\_H)

— Pin mappings for SPI 1 module —

SPI 1 Clock Out (SCK1) PPS code: 000110 (0x06)

SPI 1 Data Out (SDO1) PPS code: 000101 (0x05)  
SPI 1 Slave Select PPS code: 000111

— Pin mappings for SPI 3 module —

SPI 3 Clock Out (SCK3) PPS code: 100000 (0x20)  
SPI 3 Data Out (SDO3) PPS code: 011111 (0x1F)  
SPI 3 Slave Select PPS code: 100001

Configure the SPI 1 pins

< Put SCK1 on J10:43

< Put SDO1 on J10:55

The clock pin also needs to be configured as an input

< Set SCK1 on J10:43 as input

Configure the SPI 3 output pins

< Put SCK3 on J10:7

< Put SDO3 on J10:5

< Put SDI3 on J10:44

< Set SCK3 on J10:7 as input

@note

SPI 1 clock configuration

$$SCK1 = F_{CY} / (\text{Primary Prescaler} * \text{Secondary Prescaler})$$

Assuming that  $F_{CY} = 50\text{MHz}$ , and the prescalers are 4 and 1, the SPI clock frequency will be 12.5MHz.

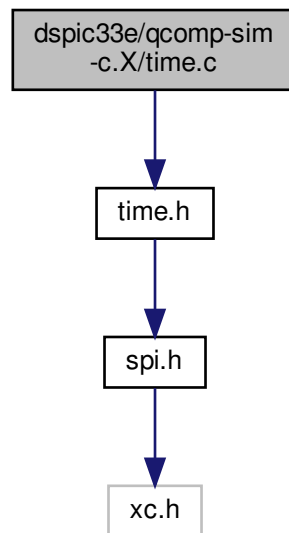


## 6.15 dspic33e/qcomp-sim-c.X/time.c File Reference

Description: Functions to control the on chip timers.

```
#include "time.h"
```

Include dependency graph for time.c:



### Functions

- void **setup\_clock** ()
- void **setup\_timer** ()
- void **reset\_timer** ()
- void **start\_timer** ()
- void **stop\_timer** ()
- unsigned long int **read\_timer** ()
- void **delay** ()

*Delay function!*

### 6.15.1 Detailed Description

Description: Functions to control the on chip timers.

#### Authors

J Scott, O Thomas

#### Date

Nov 2018

## 6.15.2 Function Documentation

### 6.15.2.1 setup\_timer()

```
void setup_timer ( )
```

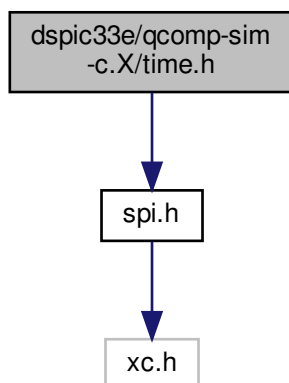
**Todo** distinguish between the two different timers here...

## 6.16 dspic33e/qcomp-sim-c.X/time.h File Reference

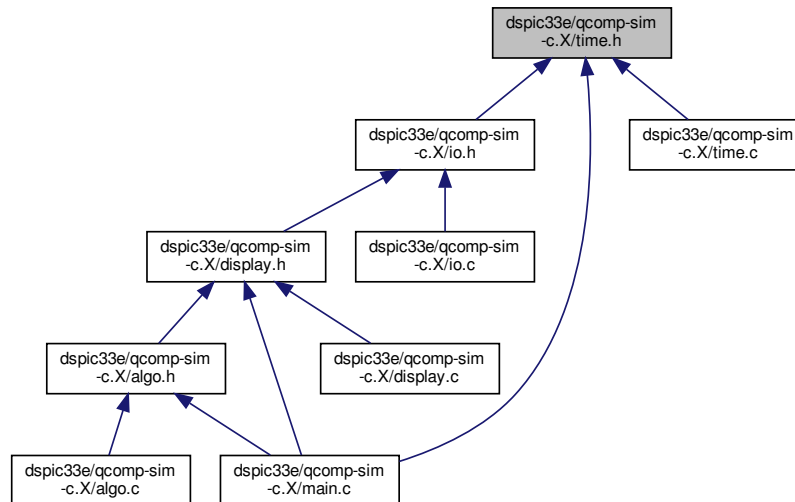
Description: Header file containing all the timing functions.

```
#include "spi.h"
```

Include dependency graph for time.h:



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



## Functions

- void **setup\_clock** ()
- void **setup\_timer** ()
- void **reset\_timer** ()
- void **start\_timer** ()
- void **stop\_timer** ()
- unsigned long int **read\_timer** ()
- void **delay** ()

*Delay function!*

### 6.16.1 Detailed Description

Description: Header file containing all the timing functions.

#### Authors

J Scott, O Thomas

#### Date

Nov 2018

### 6.16.2 Function Documentation

#### 6.16.2.1 setup\_timer()

```
void setup_timer ( )
```

**Todo** distinguish between the two different timers here...



# Index

- `__attribute__`
    - `io.c`, [40](#)
- `absolute`
  - `quantum.c`, [62](#)
  - `quantum.h`, [70](#)
- `add_to_cycle`
  - `io.c`, [40](#)
  - `io.h`, [51](#)
- `algo.c`
  - `gate`, [15](#)
  - `gate_display`, [15](#)
  - `swap_test`, [15](#)
  - `toffoli_gate`, [15](#)
  - `two_gate`, [16](#)
  - `two_gate_display`, [16](#)
- `algo.h`
  - `gate`, [18](#)
  - `gate_display`, [19](#)
  - `swap_test`, [19](#)
  - `toffoli_gate`, [19](#)
  - `two_gate`, [20](#)
  - `two_gate_display`, [20](#)
- `BTN`, [9](#)
- `btn_func`
  - `io.c`, [47](#)
- `btn_qubit`
  - `io.c`, [47](#)
- `buttons`
  - `io.c`, [47](#)
- `chip`
  - `io.c`, [48](#)
- `consts.c`
  - `H`, [22](#)
  - `rXT`, [23](#)
  - `rX`, [23](#)
  - `X`, [23](#)
  - `Y`, [24](#)
  - `Z`, [24](#)
- `consts.h`
  - `H`, [26](#)
  - `rXT`, [27](#)
  - `rX`, [26](#)
  - `X`, [27](#)
  - `Y`, [27](#)
  - `Z`, [27](#)
- `controlled_qubit_op`
  - `quantum.c`, [62](#)
  - `quantum.h`, [70](#)
- `controlled_qubit_op_new`
  - `quantum.c`, [63](#)
- `cycle_node`, [10](#)
- `display.c`
  - `display_average`, [29](#)
  - `display_cycle`, [31](#)
  - `NUM_MAX_AMPS`, [29](#)
  - `remove_zero_amp_states`, [31](#)
  - `sort_states`, [32](#)
- `display.h`
  - `display_average`, [34](#)
  - `display_cycle`, [36](#)
  - `remove_zero_amp_states`, [36](#)
  - `sort_states`, [37](#)
- `display_average`
  - `display.c`, [29](#)
  - `display.h`, [34](#)
- `display_cycle`
  - `display.c`, [31](#)
  - `display.h`, [36](#)
- `dspic33e/qcomp-sim-c.X/algo.c`, [13](#)
- `dspic33e/qcomp-sim-c.X/algo.h`, [17](#)
- `dspic33e/qcomp-sim-c.X/config.h`, [21](#)
- `dspic33e/qcomp-sim-c.X/consts.c`, [21](#)
- `dspic33e/qcomp-sim-c.X/consts.h`, [24](#)
- `dspic33e/qcomp-sim-c.X/display.c`, [28](#)
- `dspic33e/qcomp-sim-c.X/display.h`, [33](#)
- `dspic33e/qcomp-sim-c.X/io.c`, [38](#)
- `dspic33e/qcomp-sim-c.X/io.h`, [49](#)
- `dspic33e/qcomp-sim-c.X/main.c`, [59](#)
- `dspic33e/qcomp-sim-c.X/quantum.c`, [61](#)
- `dspic33e/qcomp-sim-c.X/quantum.h`, [68](#)
- `dspic33e/qcomp-sim-c.X/spi.c`, [75](#)
- `dspic33e/qcomp-sim-c.X/spi.h`, [78](#)
- `dspic33e/qcomp-sim-c.X/time.c`, [81](#)
- `dspic33e/qcomp-sim-c.X/time.h`, [82](#)
- `flash_all`
  - `io.c`, [41](#)
  - `io.h`, [52](#)
- `flash_led`
  - `io.c`, [41](#)
  - `io.h`, [52](#)
- `gate`
  - `algo.c`, [15](#)
  - `algo.h`, [18](#)
- `gate_display`

- algo.c, 15
- algo.h, 19
- H
  - consts.c, 22
  - consts.h, 26
- io.c
  - \_\_attribute\_\_, 40
  - add\_to\_cycle, 40
  - btn\_func, 47
  - btn\_qubit, 47
  - buttons, 47
  - chip, 48
  - flash\_all, 41
  - flash\_led, 41
  - isr\_counter, 48
  - led\_color\_int, 41
  - led\_cycle\_test, 42
  - led\_global, 48
  - read\_btn, 42
  - read\_external\_buttons, 43
  - reset\_cycle, 43
  - set\_external\_led, 43
  - set\_led, 44
  - set\_strobe, 44
  - setup\_external\_leds, 45
  - setup\_io, 45
  - TLC591x\_mode\_switch, 45
  - toggle\_strobe, 46
  - update\_display\_buffer, 46
  - write\_display\_driver, 46
- io.h
  - add\_to\_cycle, 51
  - flash\_all, 52
  - flash\_led, 52
  - led\_color\_int, 52
  - led\_cycle\_test, 54
  - read\_btn, 54
  - read\_external\_buttons, 54
  - reset\_cycle, 55
  - set\_external\_led, 55
  - set\_led, 56
  - set\_strobe, 56
  - setup\_external\_leds, 56
  - setup\_io, 57
  - toggle\_strobe, 57
  - update\_display\_buffer, 57
  - write\_display\_driver, 58
- isr\_counter
  - io.c, 48
- LED\_GLOBAL, 11
- LED, 10
- led\_color\_int
  - io.c, 41
  - io.h, 52
- led\_cycle\_test
  - io.c, 42
- io.h, 54
- led\_global
  - io.c, 48
- main
  - main.c, 60
- main.c
  - main, 60
- mat\_mul
  - quantum.c, 64
  - quantum.h, 71
- mat\_mul\_old
  - quantum.c, 65
- NUM\_MAX\_AMPS
  - display.c, 29
- pow2
  - quantum.c, 65
  - quantum.h, 72
- quantum.c
  - absolute, 62
  - controlled\_qubit\_op, 62
  - controlled\_qubit\_op\_new, 63
  - mat\_mul, 64
  - mat\_mul\_old, 65
  - pow2, 65
  - sign, 66
  - single\_qubit\_op, 66
  - square\_magnitude, 67
  - zero\_state, 68
- quantum.h
  - absolute, 70
  - controlled\_qubit\_op, 70
  - mat\_mul, 71
  - pow2, 72
  - sign, 72
  - single\_qubit\_op, 72
  - square\_magnitude, 73
  - zero\_state, 74
- RGB, 12
- rXT
  - consts.c, 23
  - consts.h, 27
- read\_btn
  - io.c, 42
  - io.h, 54
- read\_external\_buttons
  - io.c, 43
  - io.h, 54
- remove\_zero\_amp\_states
  - display.c, 31
  - display.h, 36
- reset\_cycle
  - io.c, 43
  - io.h, 55
- rX

- consts.c, [23](#)
  - consts.h, [26](#)
- send\_byte\_spi\_1
  - spi.c, [75](#)
  - spi.h, [79](#)
- set\_external\_led
  - io.c, [43](#)
  - io.h, [55](#)
- set\_led
  - io.c, [44](#)
  - io.h, [56](#)
- set\_strobe
  - io.c, [44](#)
  - io.h, [56](#)
- setup\_external\_leds
  - io.c, [45](#)
  - io.h, [56](#)
- setup\_io
  - io.c, [45](#)
  - io.h, [57](#)
- setup\_spi
  - spi.c, [77](#)
  - spi.h, [79](#)
- setup\_timer
  - time.c, [82](#)
  - time.h, [83](#)
- sign
  - quantum.c, [66](#)
  - quantum.h, [72](#)
- single\_qubit\_op
  - quantum.c, [66](#)
  - quantum.h, [72](#)
- sort\_states
  - display.c, [32](#)
  - display.h, [37](#)
- spi.c
  - send\_byte\_spi\_1, [75](#)
  - setup\_spi, [77](#)
- spi.h
  - send\_byte\_spi\_1, [79](#)
  - setup\_spi, [79](#)
- square\_magnitude
  - quantum.c, [67](#)
  - quantum.h, [73](#)
- swap\_test
  - algo.c, [15](#)
  - algo.h, [19](#)
- TLC591x\_mode\_switch
  - io.c, [45](#)
- time.c
  - setup\_timer, [82](#)
- time.h
  - setup\_timer, [83](#)
- toffoli\_gate
  - algo.c, [15](#)
  - algo.h, [19](#)
- toggle\_strobe
  - io.c, [46](#)
  - io.h, [57](#)
- two\_gate
  - algo.c, [16](#)
  - algo.h, [20](#)
- two\_gate\_display
  - algo.c, [16](#)
  - algo.h, [20](#)
- update\_display\_buffer
  - io.c, [46](#)
  - io.h, [57](#)
- write\_display\_driver
  - io.c, [46](#)
  - io.h, [58](#)
- X
  - consts.c, [23](#)
  - consts.h, [27](#)
- Y
  - consts.c, [24](#)
  - consts.h, [27](#)
- Z
  - consts.c, [24](#)
  - consts.h, [27](#)
- zero\_state
  - quantum.c, [68](#)
  - quantum.h, [74](#)