

Quantum Computer Outreach Project

Generated by Doxygen 1.8.13

Contents

1	Todo List	1
2	Data Structure Index	3
2.1	Data Structures	3
3	File Index	5
3.1	File List	5
4	Data Structure Documentation	7
4.1	LED Struct Reference	7
4.1.1	Detailed Description	7
4.2	LED_GLOBAL Struct Reference	8
4.2.1	Detailed Description	8
5	File Documentation	9
5.1	dspic33e/qcomp-sim-c.X/algo.c File Reference	9
5.1.1	Detailed Description	10
5.1.2	Function Documentation	10
5.1.2.1	gate()	10
5.1.2.2	two_gate()	10
5.2	dspic33e/qcomp-sim-c.X/algo.h File Reference	10
5.2.1	Detailed Description	11
5.2.2	Function Documentation	11
5.2.2.1	gate()	11
5.2.2.2	two_gate()	11

5.3	dspic33e/qcomp-sim-c.X/config.h File Reference	12
5.3.1	Detailed Description	12
5.4	dspic33e/qcomp-sim-c.X/io.c File Reference	12
5.4.1	Detailed Description	14
5.4.2	Macro Definition Documentation	14
5.4.2.1	BTN_CHIP_NUM	14
5.4.3	Function Documentation	15
5.4.3.1	__attribute__()	15
5.4.3.2	flash_all()	15
5.4.3.3	flash_led()	15
5.4.3.4	led_color_int()	16
5.4.3.5	led_cycle_test()	16
5.4.3.6	read_btn()	16
5.4.3.7	read_external_buttons()	17
5.4.3.8	set_external_led()	17
5.4.3.9	set_led()	18
5.4.3.10	set_strobe()	18
5.4.3.11	setup_external_leds()	18
5.4.3.12	setup_io()	19
5.4.3.13	TLC591x_mode_switch()	19
5.4.3.14	toggle_strobe()	19
5.4.3.15	update_display_buffer()	20
5.4.3.16	write_display_driver()	20
5.4.4	Variable Documentation	21
5.4.4.1	buttons	21
5.4.4.2	isr_counter	21
5.4.4.3	led_global	21
5.5	dspic33e/qcomp-sim-c.X/io.h File Reference	21
5.5.1	Detailed Description	23
5.5.2	Function Documentation	23

5.5.2.1	flash_all()	23
5.5.2.2	flash_led()	24
5.5.2.3	led_color_int()	24
5.5.2.4	led_cycle_test()	25
5.5.2.5	read_btn()	25
5.5.2.6	read_external_buttons()	25
5.5.2.7	set_external_led()	25
5.5.2.8	set_led()	26
5.5.2.9	set_strobe()	26
5.5.2.10	setup_external_leds()	27
5.5.2.11	setup_io()	27
5.5.2.12	toggle_strobe()	27
5.5.2.13	update_display_buffer()	27
5.5.2.14	write_display_driver()	28
5.6	dspic33e/qcomp-sim-c.X/main.c File Reference	29
5.6.1	Detailed Description	29
5.6.2	Function Documentation	30
5.6.2.1	main()	30
5.7	dspic33e/qcomp-sim-c.X/quantum.c File Reference	30
5.7.1	Detailed Description	31
5.7.2	Macro Definition Documentation	31
5.7.2.1	NUM_MAX_AMPS	31
5.7.3	Function Documentation	32
5.7.3.1	absolute()	32
5.7.3.2	controlled_qubit_op()	32
5.7.3.3	make_ops()	33
5.7.3.4	make_ops_4()	33
5.7.3.5	mat_mul()	34
5.7.3.6	mat_mul_4()	34
5.7.3.7	qubit_display()	34

5.7.3.8	<code>remove_zero_amp_states()</code>	36
5.7.3.9	<code>single_qubit_op()</code>	36
5.7.3.10	<code>sort_states()</code>	38
5.7.3.11	<code>zero_state()</code>	39
5.8	<code>dspic33e/qcomp-sim-c.X/quantum.h</code> File Reference	39
5.8.1	Detailed Description	40
5.8.2	Function Documentation	40
5.8.2.1	<code>controlled_qubit_op()</code>	41
5.8.2.2	<code>make_ops()</code>	41
5.8.2.3	<code>mat_mul()</code>	42
5.8.2.4	<code>mat_mul_4()</code>	42
5.8.2.5	<code>qubit_display()</code>	43
5.8.2.6	<code>remove_zero_amp_states()</code>	45
5.8.2.7	<code>single_qubit_op()</code>	46
5.8.2.8	<code>sort_states()</code>	47
5.8.2.9	<code>zero_state()</code>	48
5.9	<code>dspic33e/qcomp-sim-c.X/spi.c</code> File Reference	48
5.9.1	Detailed Description	49
5.9.2	Function Documentation	49
5.9.2.1	<code>send_byte_spi_1()</code>	49
5.9.2.2	<code>setup_spi()</code>	50
5.10	<code>dspic33e/qcomp-sim-c.X/spi.h</code> File Reference	51
5.10.1	Detailed Description	51
5.10.2	Function Documentation	52
5.10.2.1	<code>send_byte_spi_1()</code>	52
5.10.2.2	<code>setup_spi()</code>	52
5.11	<code>dspic33e/qcomp-sim-c.X/tests.c</code> File Reference	53
5.11.1	Detailed Description	54
5.11.2	Function Documentation	54
5.11.2.1	<code>dim_leds()</code>	54

5.12	dspic33e/qcomp-sim-c.X/tests.h File Reference	55
5.12.1	Detailed Description	56
5.12.2	Function Documentation	56
5.12.2.1	delay()	56
5.12.2.2	dim_leds()	56
5.13	dspic33e/qcomp-sim-c.X/time.c File Reference	57
5.13.1	Detailed Description	57
5.13.2	Function Documentation	57
5.13.2.1	delay()	57
5.13.2.2	setup_timer()	58
5.14	dspic33e/qcomp-sim-c.X/time.h File Reference	58
5.14.1	Detailed Description	59
5.14.2	Function Documentation	59
5.14.2.1	delay()	59
5.14.2.2	setup_timer()	59
	Index	61

Chapter 1

Todo List

Global `absolute` (Complex x)

Check that the complex part is small

Global `BTN_CHIP_NUM`

read buttons

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

This expression can probably be simplified or broken over lines.

Global `led_cycle_test` (void)

This won't work now: `write_display_driver(counter);`

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

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_4` (Complex M[4][4], Complex V[], int i, int j, int k, int l)

remove or make this general? It might not even be needed :(

Global `read_external_buttons` (void)

How long should this be?

button remappings...

Global `setup_timer` ()

distinguish between the two different timers here...

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

wait for John to fix the timer so the microprocessor isn't locked out while displaying state `counter1 = 0;`
`while(counter1 <= 1000000){ counter1++; } }`

Global `TLC591x_mode_switch` (int mode)

mode switcher for `LED` Driver

Global `write_display_driver` (void)

How long should this be?

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

LED	Each LED has the following type	7
LED_GLOBAL	Pin mappings	8

Chapter 3

File Index

3.1 File List

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

dspic33e/qcomp-sim-c.X/ algo.c	
Contains quantum algorithms to be run	9
dspic33e/qcomp-sim-c.X/ algo.h	
Header file for algorithms	10
dspic33e/qcomp-sim-c.X/ config.h	
General config settings #pragma for microcontroller	12
dspic33e/qcomp-sim-c.X/ io.c	
Contains all the functions for reading buttons and writing to LEDs	12
dspic33e/qcomp-sim-c.X/ io.h	
Description: Header file for input output functions	21
dspic33e/qcomp-sim-c.X/ main.c	
The main function	29
dspic33e/qcomp-sim-c.X/ quantum.c	
Description: Contains matrix and vector arithmetic for simulating one qubit	30
dspic33e/qcomp-sim-c.X/ quantum.h	
Description: Header file containing all the matrix arithmetic for simulating a single qubit	39
dspic33e/qcomp-sim-c.X/ spi.c	
Description: Functions for communicating with serial devices	48
dspic33e/qcomp-sim-c.X/ spi.h	
Description: SPI communication functions	51
dspic33e/qcomp-sim-c.X/ tests.c	
Description: Contains all the tests we have performed on the micro- controller	53
dspic33e/qcomp-sim-c.X/ tests.h	
Description: Header file containing all the tests we performed	55
dspic33e/qcomp-sim-c.X/ time.c	
Description: Functions to control the on chip timers	57
dspic33e/qcomp-sim-c.X/ time.h	
Description: Header file containing all the timing functions	58

Chapter 4

Data Structure Documentation

4.1 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.

4.1.1 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](#)

4.2 LED_GLOBAL Struct Reference

pin mappings

```
#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)

4.2.1 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
//
```

Global [LED](#) strobing state parameter

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

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

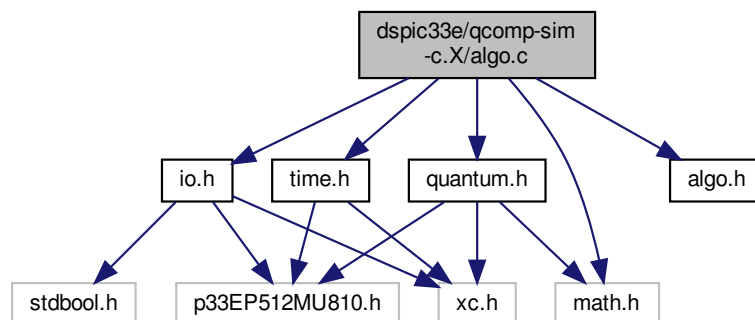
Chapter 5

File Documentation

5.1 dspic33e/qcomp-sim-c.X/algo.c File Reference

Contains quantum algorithms to be run.

```
#include "io.h"  
#include "quantum.h"  
#include "algo.h"  
#include "time.h"  
#include <math.h>  
Include dependency graph for algo.c:
```



Functions

- void `gate` (`Complex` op[2][2], int qubit, `Complex` state[], int num_qubits)
perform single qubit gate
- void `two_gate` (`Complex` op[2][2], int ctrl, int targ, `Complex` state[], int num_qubits)
perform controlled single qubit gate

5.1.1 Detailed Description

Contains quantum algorithms to be run.

5.1.2 Function Documentation

5.1.2.1 gate()

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

perform single qubit gate

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)

5.1.2.2 two_gate()

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

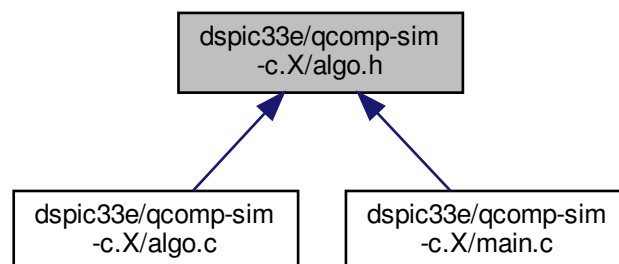
perform controlled single qubit gate

does controlled 2x2 operator

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

header file for algorithms

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



Macros

- #define **NUM_QUBITS** 4
- #define **STATE_LENGTH** 16

Functions

- void [gate](#) ([Complex](#) op[2][2], int qubit, [Complex](#) state[], int num_qubits)
perform single qubit gate
- void [two_gate](#) ([Complex](#) op[2][2], int ctrl, int targ, [Complex](#) state[], int num_qubits)
perform controlled single qubit gate

5.2.1 Detailed Description

header file for algorithms

5.2.2 Function Documentation

5.2.2.1 gate()

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

perform single qubit gate

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)

5.2.2.2 two_gate()

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

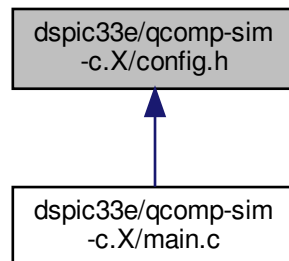
perform controlled single qubit gate

does controlled 2x2 operator

5.3 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:



5.3.1 Detailed Description

General config settings #pragma for microcontroller.

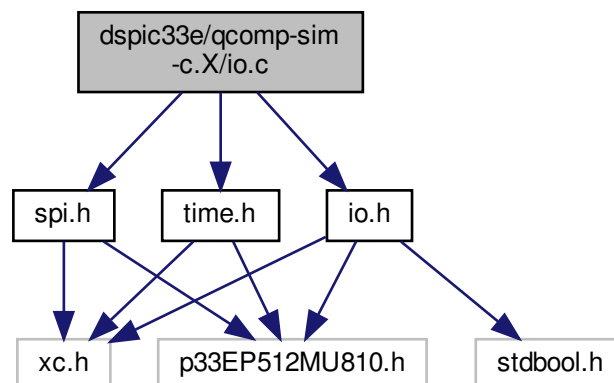
Description: Include this once at the top of main

5.4 dspic33e/qcomp-sim-c.X/io.c File Reference

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

```
#include "io.h"  
#include "time.h"  
#include "spi.h"
```

Include dependency graph for io.c:



Macros

- #define **DISPLAY_CHIP_NUM** 2
- #define **PERIOD** 500000
- #define **BTN_CHIP_NUM** 2

Read external buttons.

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.
- 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)
Update the buttons array (see declaration above)
- 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` [16]
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.

5.4.1 Detailed Description

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

Author

J Scott

Date

8/11/18

5.4.2 Macro Definition Documentation

5.4.2.1 BTN_CHIP_NUM

```
#define BTN_CHIP_NUM 2
```

Read external buttons.

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

5.4.3 Function Documentation

5.4.3.1 `__attribute__()`

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

The max value for `isr_counter`.

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

5.4.3.2 `flash_all()`

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

Flash all the LEDs a number of times.

Parameters

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

5.4.3.3 `flash_led()`

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

Flash **LED** a number of times.

Flash one **LED** a number of times.

5.4.3.4 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 $2^n - 1$

5.4.3.5 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);

5.4.3.6 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

5.4.3.7 read_external_buttons()

```
int read_external_buttons (
    void )
```

Update the buttons array (see declaration above)

SH pin

Todo How long should this be?

Todo button remappings...

5.4.3.8 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>	

5.4.3.9 set_led()

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

Turn a particular LED on or off.

Parameters

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

5.4.3.10 set_strobe()

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

Set an LED strobing.

Parameters

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

5.4.3.11 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

5.4.3.12 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

5.4.3.13 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

5.4.3.14 toggle_strobe()

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

Toggle [LED](#) strobe.

Parameters

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

5.4.3.15 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

5.4.3.16 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?

5.4.4 Variable Documentation

5.4.4.1 buttons

```
int buttons[16]
```

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

5.4.4.2 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).

5.4.4.3 led_global

```
LED_GLOBAL led_global = {0}
```

Parameters

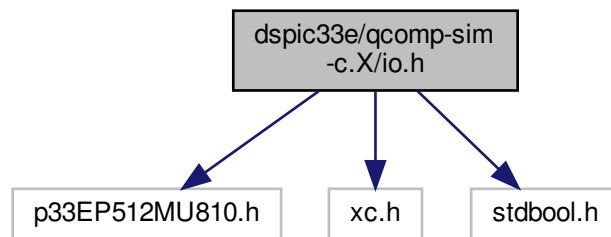
<i>led_global</i>	Global LED strobing state parameter
-------------------	---

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

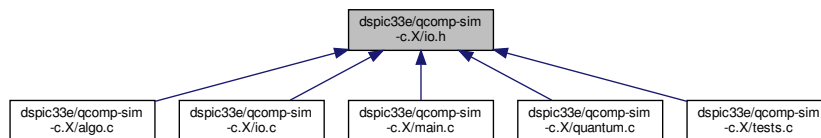
Description: Header file for input output functions.

```
#include "p33EP512MU810.h"
#include "xc.h"
#include <stdbool.h>
```

Include dependency graph for `io.h`:



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



Data Structures

- struct `LED_GLOBAL`
pin mappings
- struct `LED`
Each `LED` has the following type.

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`
- `#define LED_NUM 4`
The number of external LEDs.

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)

5.5.1 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.

5.5.2 Function Documentation

5.5.2.1 `flash_all()`

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

Flash all the LEDs a number of times.

Parameters

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

5.5.2.2 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.

5.5.2.3 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 $2^n - 1$

5.5.2.4 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);

5.5.2.5 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

5.5.2.6 read_external_buttons()

```
int read_external_buttons (
    void )
```

Update the buttons array (see declaration above)

SH pin

Todo How long should this be?

Todo button remappings...

5.5.2.7 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>	

5.5.2.8 set_led()

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

Turn a particular [LED](#) on or off.

Parameters

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

5.5.2.9 set_strobe()

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

Set an [LED](#) strobing.

Parameters

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

5.5.2.10 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

5.5.2.11 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

5.5.2.12 toggle_strobe()

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

Toggle LED strobe.

Parameters

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

5.5.2.13 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

5.5.2.14 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

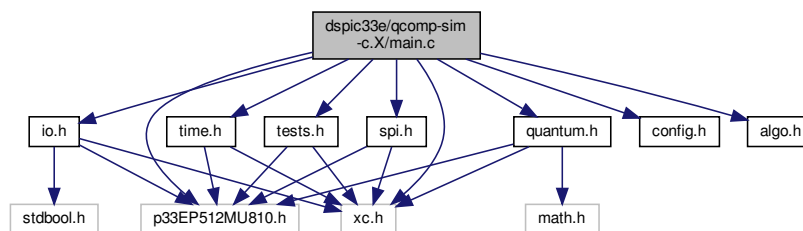
Todo How long should this be?

5.6 dspic33e/qcomp-sim-c.X/main.c File Reference

The main function.

```
#include "p33EP512MU810.h"
#include "xc.h"
#include "config.h"
#include "time.h"
#include "io.h"
#include "quantum.h"
#include "tests.h"
#include "spi.h"
#include "algo.h"
```

Include dependency graph for main.c:



Functions

- int [main](#) (void)

5.6.1 Detailed Description

The main function.

Author

J R Scott

Date

8/11/18

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>

5.6.2 Function Documentation

5.6.2.1 main()

```
int main (
    void )
```

Reading button state

The button states are written into an array of type BUTTON_ARRAY whose

Global variable for button state

Update the buttons variable

Do something if button 0 has been pressed...

Start of the PROGRAM!

do CNOT_01 -> should stay vac

X qubit 2

do CNOT_23 -> should go 11

<

Note

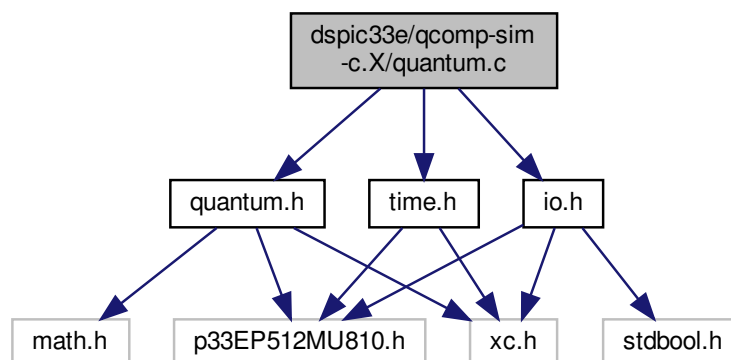
Really important!

5.7 dspic33e/qcomp-sim-c.X/quantum.c File Reference

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

```
#include "io.h"
#include "quantum.h"
#include "time.h"
```

Include dependency graph for quantum.c:



Macros

- `#define NUM_MAX_AMPS 4`

Functions

- void **cadd** (Complex a, Complex b, Complex result)
- void **cmul** (Complex a, Complex b, Complex result)
- **Q15 absolute** (Complex x)
- void **make_ops** (Complex X[2][2], Complex Y[2][2], Complex Z[2][2], Complex H[2][2])
Create complex X, Y, Z and H.
- void **make_ops_4** (Complex CNOT[4][4], Complex CPHASE[4][4], Complex SWAP[4][4])
- void **zero_state** (Complex state[], int Qnum)
Initialise state to the vacuum (zero apart from the first position) Specify the dimension – of the matrix, i.e.
- void **mat_mul** (Complex M[2][2], Complex V[], int i, int j)
2x2 complex matrix multiplication
- void **mat_mul_4** (Complex M[4][4], Complex V[], int i, int j, int k, int l)
4x4 matrix
- void **qubit_display** (Complex state[], int N)
Display the state amplitudes on LEDs.
- void **single_qubit_op** (Complex op[2][2], int k, Complex state[], int N)
apply operator
- void **controlled_qubit_op** (Complex op[2][2], int ctrl, int targ, Complex state[], int N)
selective 2 qubit op function
- int **sort_states** (Complex state[], int num_qubits)
- int **remove_zero_amp_states** (Complex state[], int num_qubits, Complex disp_state[])
takes state vector, number of qubits and vector to write the nonzero elements of the statevector to.

5.7.1 Detailed Description

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

5.7.2 Macro Definition Documentation

5.7.2.1 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.

5.7.3 Function Documentation

5.7.3.1 absolute()

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

Parameters

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

Returns

The absolute value

Todo Check that the complex part is small

5.7.3.2 controlled_qubit_op()

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

selective 2 qubit op function

apply controlled 2x2 op

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

checks that the control qubit is $|1\rangle$ then does 2x2 unitary on remaining state vector 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 $+2^{\wedge}$ (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.

5.7.3.3 make_ops()

```
void make_ops (
    Complex X[2][2],
    Complex Y[2][2],
    Complex Z[2][2],
    Complex H[2][2] )
```

Create complex X, Y, Z and H.

Parameters

<i>X</i>	Pauli X c-Matrix
<i>Z</i>	Pauli Z c-matrix
<i>H</i>	Hadamard c-matrix
<i>Y</i>	Pauli Y c-matrix

Note

IMPLICIT NONE!!!

5.7.3.4 make_ops_4()

```
void make_ops_4 (
    Complex CNOT[4][4],
    Complex CPHASE[4][4],
    Complex SWAP[4][4] )
```

Note

```
/// [][] = row, col
/// [2][3] is (0 0 0 0)
///          (0 0 0 0)
///          (0 0 0 1)
///          (0 0 0 0)
///
```

5.7.3.5 mat_mul()

```
void mat_mul (
    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.

5.7.3.6 mat_mul_4()

```
void mat_mul_4 (
    Complex M[4][4],
    Complex V[],
    int i,
    int j,
    int k,
    int l )
```

4x4 matrix

4x4 complex matrix multiplication store results of each row multiplication

```
/// ( a b c d ) * ( p )
/// ( e f g h )   ( q )
/// ( h i j k )   ( r )
/// ( l m n o )   ( s )
///
```

4 cols of V (i,h,k,l)

add the 4 terms together and put into temp for that row

5.7.3.7 qubit_display()

```
void qubit_display (
    Complex state[],
    int N )
```

Display the state amplitudes on LEDs.

Parameters

<i>state</i>	Pass in the state vector
<i>N</i>	The total number of qubits

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 apparently 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; } Loop over all qubits k = 0, 1, 2, ... N-1
```

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

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

Zeros are at the index root + step

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

update leds for each qubits average zero and one amps

5.7.3.8 remove_zero_amp_states()

```
int remove_zero_amp_states (
    Complex state[],
    int num_qubits,
    Complex 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

5.7.3.9 single_qubit_op()

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

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

```
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 $2^k(k+1)$

First index is ZERO, second index is ONE

5.7.3.10 sort_states()

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

Array for largest amplitudes

To store the position of the amplitudes

/ Sort the state. Look through every element storing it if it is larger / and than the previous largest element for(int j=0; j<N; j++){

Sort the state.

max is always in ascending order. Elemen

int k = NUM_MAX_AMPS - 1; for(int j=0; j<N; j++){ / Compute the magnitude of the element if(absolute(state[j]) >= max[k]) { max[k] = absolute(state[j]); k++; /// Increment k to point to next position in max } }

```
if(max_amp[count] <= state[j][0]){
    count++;
```

update new maximum val max_amp[count] = state[j][0]; save pos of maximal val amp_index[count] = j; }

} / now have arrays of out_state and amplitudes of size count / need to decode int of the out_state to binary to find each quits state

```
while(1){
```

loop over all max vals of the state for(int l=1; l<=count; l++){ / display the states of three qubits for(int k=0; k<4; k++){ / I know this is wrong it returns one_amp as 0x0002 / which means zero_amp becomes 0xFFFF because of overflow?

```
/
```

Note

Changed to unsigned Fract to match set_external_led args in [io.c](#) unsigned _Fract one_amp=(amp_index[l] & (1 << k)); unsigned _Fract zero_amp=1-one_amp; set_external_led(k, 0,zero_amp, one_amp);

```
}
```

let the user see /

Todo wait for John to fix the timer so the microprocessor isn't locked out while displaying state counter1 = 0; while(counter1 <= 1000000){ counter1++; } }

5.7.3.11 zero_state()

```
void zero_state (
    Complex state[],
    int Qnum )
```

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!

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

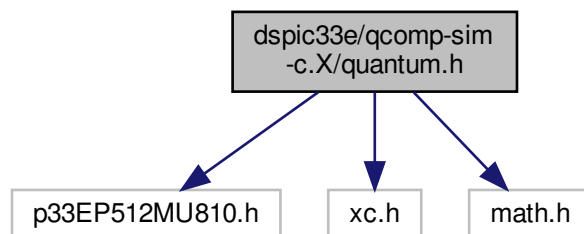
Description: Header file containing all the matrix arithmetic for simulating a single qubit.

```
#include "p33EP512MU810.h"
```

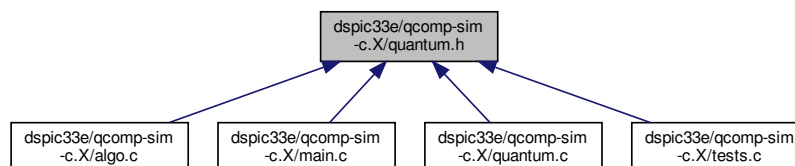
```
#include "xc.h"
```

```
#include <math.h>
```

Include dependency graph for quantum.h:



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



Macros

- `#define ONE_Q15 0.9999694824`

Typedefs

- `typedef signed _Fract Q15`
Basic fractional time.
- `typedef Q15 Complex[2]`
Complex type.

Enumerations

- `enum State {`
 ZERO, ONE, PLUS, MINUS,
 iPLUS, iMINUS }
Basis states.

Functions

- `void make_ops (Complex X[2][2], Complex Y[2][2], Complex Z[2][2], Complex H[2][2])`
Create complex X, Y, Z and H.
- `void zero_state (Complex state[], int Qnum)`
Initialise state to the vacuum (zero apart from the first position) Specify the dimension – of the matrix, i.e.
- `void mat_mul (Complex M[2][2], Complex V[], int i, int j)`
2x2 complex matrix multiplication
- `void mat_mul_4 (Complex M[4][4], Complex V[], int i, int j, int k, int l)`
4x4 complex matrix multiplication
- `void single_qubit_op (Complex op[2][2], int qubit, Complex state[], int Qnum)`
apply operator
- `void controlled_qubit_op (Complex op[2][2], int ctrl, int targ, Complex state[], int N)`
apply controlled 2x2 op
- `void qubit_display (Complex state[], int Qnum)`
Display the state amplitudes on LEDs.
- `int sort_states (Complex state[], int num_qubits)`
- `int val_of_pos_bit (int input, int pos)`
- `int remove_zero_amp_states (Complex state[], int num_qubits, Complex disp_state[])`
updates disp_state where the first 'return value of the function' elements are the nonzero elements of the state vector 'state'

5.8.1 Detailed Description

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

5.8.2 Function Documentation

5.8.2.1 controlled_qubit_op()

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

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
<i>N</i>	total number of qubits

apply controlled 2x2 op

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

checks that the control qubit is $|1\rangle$ then does 2x2 unitary on remaining state vector ROOT loop: starts at 0, increases in steps of 1

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

5.8.2.2 make_ops()

```
void make_ops (
    Complex X[2][2],
    Complex Y[2][2],
    Complex Z[2][2],
    Complex H[2][2] )
```

Create complex X, Y, Z and H.

Parameters

X	Pauli X c-Matrix
Z	Pauli Z c-matrix
H	Hadamard c-matrix
Y	Pauli Y c-matrix

Note

IMPLICIT NONE!!!

5.8.2.3 mat_mul()

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

2x2 complex matrix multiplication

Parameters

M	complex matrix
V	complex vector
i	integer first element of state vector
j	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.

5.8.2.4 mat_mul_4()

```
void mat_mul_4 (
    Complex M[4][4],
    Complex V[],
    int i,
    int j,
    int k,
    int l )
```

4x4 complex matrix multiplication

Parameters

M	4x4 complex matrix
V	complex state vector
i	first element of V
j	second element of V
k	third element of V
l	Fourth element of V

Note

this function is never used as we realised that a 2-qubit gate is a reduced form of a single qubit gate...

Todo remove or make this general? It might not even be needed :(

4x4 complex matrix multiplication store results of each row multiplication

```
/// ( a b c d ) * ( p )
/// ( e f g h ) ( q )
/// ( h i j k ) ( r )
/// ( l m n o ) ( s )
///
```

4 cols of V (i,h,k,l)

add the 4 terms together and put into temp for that row

5.8.2.5 qubit_display()

```
void qubit_display (
    Complex state[],
    int N )
```

Display the state amplitudes on LEDs.

Parameters

$state$	Pass in the state vector
$Qnum$	The total number of qubits

Note

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

Parameters

$state$	Pass in the state vector
N	The total number of qubits

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 apparently 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; } Loop over all qubits k = 0, 1, 2, ... N-1
```

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

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

Zeros are at the index root + step

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

update leds for each qubits average zero and one amps

5.8.2.6 remove_zero_amp_states()

```
int remove_zero_amp_states (
    Complex state[],
    int num_qubits,
    Complex 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>num_qubits</i>	int number of qubits 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

5.8.2.7 single_qubit_op()

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

apply operator

Parameters

<i>state</i>	state vector containing amplitudes
<i>qubit</i>	qubit number to apply 2x2 matrix to
<i>Qnum</i>	total number of qubits in the state
<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

```
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 $2^k(k+1)$

First index is ZERO, second index is ONE

5.8.2.8 sort_states()

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

Array for largest amplitudes

To store the position of the amplitudes

/ Sort the state. Look through every element storing it if it is larger / and than the previous largest element for(int j=0; j<N; j++){

```
Sort the state.
```

```
max is always in ascending order. Elemen
```

```
int k = NUM_MAX_AMPS - 1; for(int j=0; j<N; j++){ / Compute the magnitude of the element if(absolutely(state[j]) >=
max[k]) { max[k] = absolutely(state[j]); k++; /// Increment k to point to next position in max } }
```

```
if(max_amp[count] <= state[j][0]){
    count++;
```

```
update new maximum val max_amp[count] = state[j][0]; save pos of maximal val amp_index[count] = j; }
```

```
} / now have arrays of out_state and amplitudes of size count / need to decode int of the out_state to binary to find
each quits state
```

```
while(1){
```

```
loop over all max vals of the state for(int l=1; l<=count; l++){ / display the states of three qubits for(int k=0; k<4;
k++){ / I know this is wrong it returns one_amp as 0x0002 / which means zero_amp becomes 0xFFFF because of
overflow?
```

```
/
```

Note

Changed to unsigned Fract to match set_external_led args in [io.c](#) unsigned _Fract one_amp=(amp_index[l] & (1 << k)); unsigned _Fract zero_amp=1-one_amp; set_external_led(k, 0,zero_amp, one_amp);

}

let the user see /

Todo wait for John to fix the timer so the microprocessor isn't locked out while displaying state counter1 = 0; while(counter1 <= 1000000){ counter1++; } }

5.8.2.9 zero_state()

```
void zero_state (
    Complex state[],
    int Qnum )
```

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

<i>state</i>	complex state vector
<i>Qnum</i>	int total number of qubits

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

Note

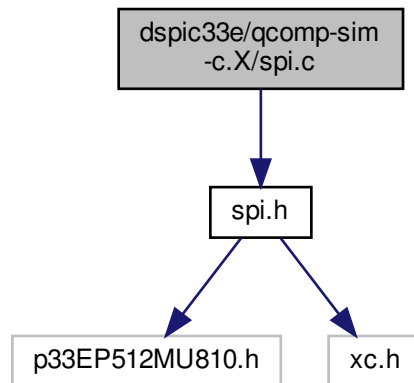
oh the clarity!

5.9 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.

5.9.1 Detailed Description

Description: Functions for communicating with serial devices.

5.9.2 Function Documentation

5.9.2.1 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
-------------	-------------------------

5.9.2.2 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

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

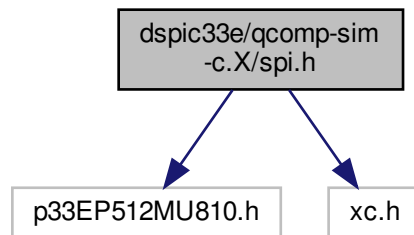
Assuming that $F_CY = 50\text{MHz}$, and the prescalers are 4 and 1, the SPI clock frequency will be 12.5MHz.

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

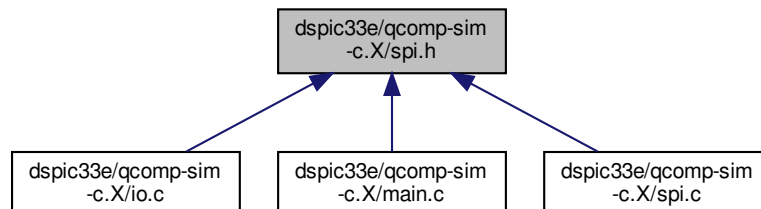
Description: SPI communication functions.

```
#include "p33EP512MU810.h"
#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.

5.10.1 Detailed Description

Description: SPI communication functions.

5.10.2 Function Documentation

5.10.2.1 `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
-------------	-------------------------

5.10.2.2 `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

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

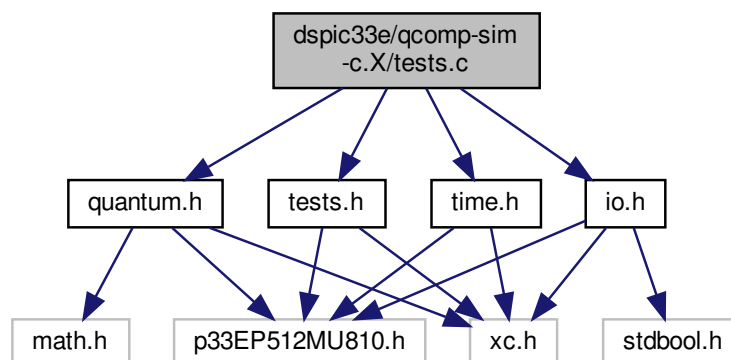
Assuming that $F_{CY} = 50\text{MHz}$, and the prescalers are 4 and 1, the SPI clock frequency will be 12.5MHz.

5.11 dspic33e/qcomp-sim-c.X/tests.c File Reference

Description: Contains all the tests we have performed on the micro- controller.

```
#include "tests.h"
#include "io.h"
#include "quantum.h"
#include "time.h"
```

Include dependency graph for tests.c:



Functions

- void `dim_leds` ()

Testing the speed of 2^{15} 2x2 real matrix multiplications void `mat_mul_test()` {.

5.11.1 Detailed Description

Description: Contains all the tests we have performed on the micro- controller.

5.11.2 Function Documentation

5.11.2.1 `dim_leds()`

```
void dim_leds ( )
```

Testing the speed of 2^{15} 2x2 real matrix multiplications void `mat_mul_test()` {.

Define state vector $|0\rangle = (1,0)$ $|1\rangle = (0,1)$ Vector V; `init_state(V, ZERO);`

Matrix2 X = {{0}}, Z = {{0}}, H = {{0}}; `make_ops(X, Z, H);`

Start the timer `start_timer();`

Do a matrix multiplication test `unsigned int n = 0; while (n < 32768) { mat_mul(X, V); n++; }`

Read the timer `unsigned long int time = read_timer();`

Show that the test is finished `set_led(red, on);`

wait (add a breakpoint here) `while(1 == 1);`

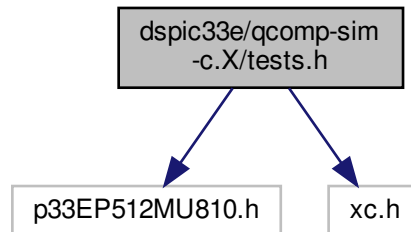
```
}
```

5.12 dspic33e/qcomp-sim-c.X/tests.h File Reference

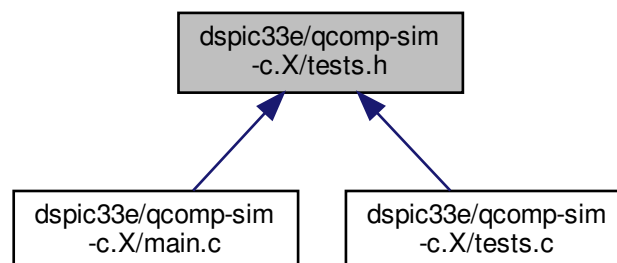
Description: Header file containing all the tests we performed.

```
#include "p33EP512MU810.h"
#include "xc.h"
```

Include dependency graph for tests.h:



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



Functions

- void **mat_mul_test** ()
- void **mat_mul_test_cmplx** ()
- void **one_qubit** ()
- void **one_qubit_cmplx** ()
- void **dim_leds** ()

Testing the speed of 2^{15} 2x2 real matrix multiplications void mat_mul_test() {.

- void **multi_led_strobe** ()
- void **delay** ()

delays for 100,000

5.12.1 Detailed Description

Description: Header file containing all the tests we performed.

5.12.2 Function Documentation

5.12.2.1 delay()

```
void delay ( )
```

delays for 100,000

delays for 100,000

5.12.2.2 dim_leds()

```
void dim_leds ( )
```

Testing the speed of 2^{15} 2x2 real matrix multiplications void mat_mul_test() {.

Define state vector $|0\rangle = (1,0)$ $|1\rangle = (0,1)$ Vector V; init_state(V, ZERO);

Matrix2 X = {{0}}, Z = {{0}}, H = {{0}}; make_ops(X, Z, H);

Start the timer start_timer();

Do a matrix multiplication test unsigned int n = 0; while (n < 32768) { mat_mul(X, V); n++; }

Read the timer unsigned long int time = read_timer();

Show that the test is finished set_led(red, on);

wait (add a breakpoint here) while(1 == 1);

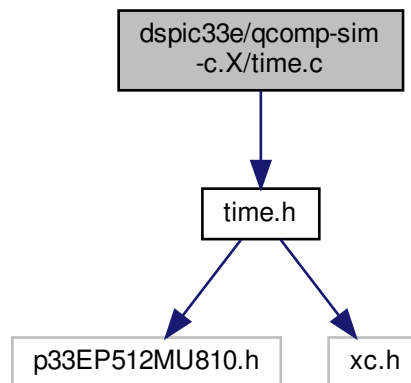
```
}
```


5.13 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!

5.13.1 Detailed Description

Description: Functions to control the on chip timers.

5.13.2 Function Documentation

5.13.2.1 delay()

```
void delay ( )
```

Delay function!

delays for 100,000

5.13.2.2 setup_timer()

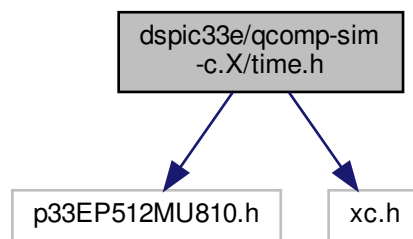
```
void setup_timer ( )
```

Todo distinguish between the two different timers here...

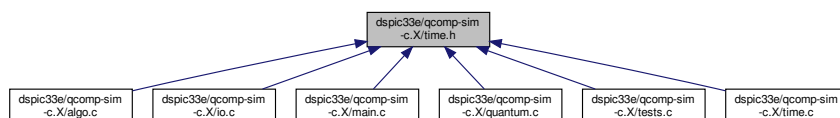
5.14 dspic33e/qcomp-sim-c.X/time.h File Reference

Description: Header file containing all the timing functions.

```
#include "p33EP512MU810.h"
#include "xc.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!*

5.14.1 Detailed Description

Description: Header file containing all the timing functions.

5.14.2 Function Documentation

5.14.2.1 delay()

```
void delay ( )
```

Delay function!

delays for 100,000

5.14.2.2 setup_timer()

```
void setup_timer ( )
```

Todo distinguish between the two different timers here...

Index

`__attribute__`
io.c, [15](#)

absolute
quantum.c, [32](#)

algo.c
gate, [10](#)
two_gate, [10](#)

algo.h
gate, [11](#)
two_gate, [11](#)

BTN_CHIP_NUM
io.c, [14](#)

buttons
io.c, [21](#)

controlled_qubit_op
quantum.c, [32](#)
quantum.h, [40](#)

delay
tests.h, [56](#)
time.c, [57](#)
time.h, [59](#)

dim_leds
tests.c, [54](#)
tests.h, [56](#)

dspic33e/qcomp-sim-c.X/algo.c, [9](#)
dspic33e/qcomp-sim-c.X/algo.h, [10](#)
dspic33e/qcomp-sim-c.X/config.h, [12](#)
dspic33e/qcomp-sim-c.X/io.c, [12](#)
dspic33e/qcomp-sim-c.X/io.h, [21](#)
dspic33e/qcomp-sim-c.X/main.c, [29](#)
dspic33e/qcomp-sim-c.X/quantum.c, [30](#)
dspic33e/qcomp-sim-c.X/quantum.h, [39](#)
dspic33e/qcomp-sim-c.X/spi.c, [48](#)
dspic33e/qcomp-sim-c.X/spi.h, [51](#)
dspic33e/qcomp-sim-c.X/tests.c, [53](#)
dspic33e/qcomp-sim-c.X/tests.h, [55](#)
dspic33e/qcomp-sim-c.X/time.c, [57](#)
dspic33e/qcomp-sim-c.X/time.h, [58](#)

flash_all
io.c, [15](#)
io.h, [23](#)

flash_led
io.c, [15](#)
io.h, [24](#)

gate

algo.c, [10](#)
algo.h, [11](#)

io.c
`__attribute__`, [15](#)
BTN_CHIP_NUM, [14](#)
buttons, [21](#)
flash_all, [15](#)
flash_led, [15](#)
isr_counter, [21](#)
led_color_int, [15](#)
led_cycle_test, [16](#)
led_global, [21](#)
read_btn, [16](#)
read_external_buttons, [17](#)
set_external_led, [17](#)
set_led, [18](#)
set_strobe, [18](#)
setup_external_leds, [18](#)
setup_io, [18](#)
TLC591x_mode_switch, [19](#)
toggle_strobe, [19](#)
update_display_buffer, [20](#)
write_display_driver, [20](#)

io.h
flash_all, [23](#)
flash_led, [24](#)
led_color_int, [24](#)
led_cycle_test, [24](#)
read_btn, [25](#)
read_external_buttons, [25](#)
set_external_led, [25](#)
set_led, [26](#)
set_strobe, [26](#)
setup_external_leds, [27](#)
setup_io, [27](#)
toggle_strobe, [27](#)
update_display_buffer, [27](#)
write_display_driver, [28](#)

isr_counter
io.c, [21](#)

LED_GLOBAL, [8](#)
LED, [7](#)

led_color_int
io.c, [15](#)
io.h, [24](#)

led_cycle_test
io.c, [16](#)
io.h, [24](#)

- led_global
 - io.c, 21
- main
 - main.c, 30
- main.c
 - main, 30
- make_ops
 - quantum.c, 33
 - quantum.h, 41
- make_ops_4
 - quantum.c, 33
- mat_mul
 - quantum.c, 33
 - quantum.h, 42
- mat_mul_4
 - quantum.c, 34
 - quantum.h, 42
- NUM_MAX_AMPS
 - quantum.c, 31
- quantum.c
 - absolute, 32
 - controlled_qubit_op, 32
 - make_ops, 33
 - make_ops_4, 33
 - mat_mul, 33
 - mat_mul_4, 34
 - NUM_MAX_AMPS, 31
 - qubit_display, 34
 - remove_zero_amp_states, 36
 - single_qubit_op, 36
 - sort_states, 37
 - zero_state, 38
- quantum.h
 - controlled_qubit_op, 40
 - make_ops, 41
 - mat_mul, 42
 - mat_mul_4, 42
 - qubit_display, 43
 - remove_zero_amp_states, 45
 - single_qubit_op, 46
 - sort_states, 47
 - zero_state, 48
- qubit_display
 - quantum.c, 34
 - quantum.h, 43
- read_btn
 - io.c, 16
 - io.h, 25
- read_external_buttons
 - io.c, 17
 - io.h, 25
- remove_zero_amp_states
 - quantum.c, 36
 - quantum.h, 45
- send_byte_spi_1
 - spi.c, 49
 - spi.h, 52
- set_external_led
 - io.c, 17
 - io.h, 25
- set_led
 - io.c, 18
 - io.h, 26
- set_strobe
 - io.c, 18
 - io.h, 26
- setup_external_leds
 - io.c, 18
 - io.h, 27
- setup_io
 - io.c, 18
 - io.h, 27
- setup_spi
 - spi.c, 50
 - spi.h, 52
- setup_timer
 - time.c, 57
 - time.h, 59
- single_qubit_op
 - quantum.c, 36
 - quantum.h, 46
- sort_states
 - quantum.c, 37
 - quantum.h, 47
- spi.c
 - send_byte_spi_1, 49
 - setup_spi, 50
- spi.h
 - send_byte_spi_1, 52
 - setup_spi, 52
- TLC591x_mode_switch
 - io.c, 19
- tests.c
 - dim_leds, 54
- tests.h
 - delay, 56
 - dim_leds, 56
- time.c
 - delay, 57
 - setup_timer, 57
- time.h
 - delay, 59
 - setup_timer, 59
- toggle_strobe
 - io.c, 19
 - io.h, 27
- two_gate
 - algo.c, 10
 - algo.h, 11
- update_display_buffer
 - io.c, 20
 - io.h, 27

write_display_driver

io.c, [20](#)

io.h, [28](#)

zero_state

quantum.c, [38](#)

quantum.h, [48](#)