GPIO Module

Since	Origin / Contributor	Maintainer	Source
2014-12-22	Zeroday	Zeroday	gpio.c

This module provides access to the GPIO (General Purpose Input/Output) subsystem.

All access is based on the I/O index number on the NodeMCU dev kits, not the internal GPIO pin. For example, the D0 pin on the dev kit is mapped to the internal GPIO pin 16.

If not using a NodeMCU dev kit, please refer to the below GPIO pin maps for the index↔gpio mapping.

IO index	ESP8266 pin	IO index	ESP8266 pin
0 [*]	GPIO16	7	GPIO13
1	GPIO5	8	GPIO15
2	GPIO4	9	GPIO3
3	GPIO0	10	GPIO1
4	GPIO2	11	GPIO9
5	GPIO14	12	GPIO10
6	GPIO12		

[*] D0(GPIO16) can only be used as gpio read/write. No support for open-drain/interrupt/pwm/i2c/ow.

gpio.mode()	Initialize pin to GPIO mode, set the pin in/out direction, and optional internal wea
gpio.read()	Read digital GPIO pin value.
gpio.serout()	Serialize output based on a sequence of delay-times in µs.
gpio.trig()	Establish or clear a callback function to run on interrupt for a pin.
gpio.write()	Set digital GPIO pin value.

v: release

gpio.pulse	This covers a set of APIs that allow generation of pulse trains with accurate timin
gpio.pulse.build	This builds the gpio.
gpio.pulse:start	This starts the output operations.
gpio.pulse:getstate	This returns the current state.
gpio.pulse:stop	This stops the output operation at some future time.
gpio.pulse:cancel	This stops the output operation immediately.
gpio.pulse:adjust	This adds (or subtracts) time that will get used in the min / max delay case.
gpio.pulse:update	This can change the contents of a particular step in the output program.

gpio.mode()

Initialize pin to GPIO mode, set the pin in/out direction, and optional internal weak pull-up.

Syntax

`gpio.mode(pin, mode [, pullup])`

Parameters

- `pin` pin to configure, IO index
- `mode` one of gpio.OUTPUT, gpio.OPENDRAIN, gpio.INPUT, or gpio.INT (interrupt mode)
- `pullup` gpio.PULLUP enables the weak pull-up resistor; default is gpio.FLOAT

Returns

`nil`

Example

```
gpio.mode(0, gpio.OUTPUT)
```

See also

- `gpio.read()`
- `gpio.write()`

v: release

gpio.read()

Read digital GPIO pin value.

Syntax

`gpio.read(pin)`

Parameters

`pin` pin to read, IO index

Returns

a number, 0 = low, 1 = high

Example

```
-- read value of gpio 0.
gpio.read(0)
```

See also

`gpio.mode()`

gpio.serout()

Serialize output based on a sequence of delay-times in µs. After each delay, the pin is toggled. After the last cycle and last delay the pin is not toggled.

The function works in two modes: * synchronous - for sub-50 µs resolution, restricted to max. overall duration, * asynchrounous - synchronous operation with less granularity but virtually unrestricted duration.

Whether the asynchronous mode is chosen is defined by presence of the `callback` parameter. If present and is of function type the function goes asynchronous and the callback function is invoked when sequence finishes. If the parameter is numeric the function still goes asynchronous but no callback is invoked when done.

For the asynchronous version, the minimum delay time should not be shorter than 50 µs and maximum delay time is 0x7fffff µs (~8.3 seconds). In this mode the function does not block the stack and returns immediately before the output sequence is finalized. HW timer `FRC1_SOURCE` mode is used to change the states. As there is only a single hardware timer, there are restrictions on which modules can be used at the same time. An error will be raised if the timer is already in use.

Note that the synchronous variant (no or nil `callback` parameter) function blocks the stack and as such any use of it must adhere to the SDK guidelines (also explained here). Failure to do so may lead to WiFi issues or outright to crashes/reboots. In short it means that the sum of all delay times multiplied by the number of cycles should not exceed 15 ms.

Syntax

`gpio.serout(pin, start_level, delay_times [, cycle_num[, callback]])`

Parameters

- `pin` pin to use, IO index
- `start_level` level to start on, either `gpio.HIGH` or `gpio.LOW`
- `delay_times` an array of delay times in µs between each toggle of the gpio pin.
- `cycle_num` an optional number of times to run through the sequence. (default is 1)
- `callback` an optional callback function or number, if present the function returns immediately and goes asynchronous.

Returns

`nil`

Example

```
gpio.mode(1,gpio.OUTPUT,gpio.PULLUP)
gpio.serout(1,gpio.HIGH,{30,30,60,60,30,30})  -- serial one byte, b10110010
gpio.serout(1,gpio.HIGH,{30,70},8)  -- serial 30% pwm 10k, lasts 8 cycles
gpio.serout(1,gpio.HIGH,{3,7},8)  -- serial 30% pwm 100k, lasts 8 cycles
gpio.serout(1,gpio.HIGH,{0,0},8)  -- serial 50% pwm as fast as possible, lasts 8 cycles
gpio.serout(1,gpio.LOW,{20,10,10,20,10,10,10,100}) -- sim uart one byte 0x5A at about 100kbps
gpio.serout(1,gpio.HIGH,{8,18},8) -- serial 30% pwm 38k, lasts 8 cycles

gpio.serout(1,gpio.HIGH,{5000,995000},100, function() print("done") end) -- asynchronous 100 flashes
gpio.serout(1,gpio.HIGH,{5000,995000},100, 1) -- asynchronous 100 flashes 5 ms long, no callback
```

v: release

gpio.trig()

Establish or clear a callback function to run on interrupt for a pin.

This function is not available if GPIO_INTERRUPT_ENABLE was undefined at compile time.

Syntax

`gpio.trig(pin, [type [, callback_function]])`

Parameters

- `pin` **1-12**, pin to trigger on, IO index. Note that pin 0 does not support interrupts.
- `type` "up", "down", "both", "low", "high", which represent *rising edge*, *falling edge*, *both edges*, *low level*, and *high level* trigger modes respectivey. If the type is "none" or omitted then the callback function is removed and the interrupt is disabled.
- `callback_function(level, when, eventcount)` callback function when trigger occurs. The level of the specified pin at the interrupt passed as the first parameter to the callback. The timestamp of the event is passed as the second parameter. This is in microseconds and has the same base as for `tmr.now()`. This timestamp is grabbed at interrupt level and is more consistent than getting the time in the callback function. This timestamp is normally of the first interrupt detected, but, under overload conditions, might be a later one. The eventcount is the number of interrupts that were elided for this callback. This works best for edge triggered interrupts and enables counting of edges. However, beware of switch bounces -- you can get multiple pulses for a single switch closure. Counting works best when the edges are digitally generated. The previous callback function will be used if the function is omitted.

Returns

`nil`

Example

```
do
  -- use pin 1 as the input pulse width counter
  local pin, pulse1, du, now, trig = 1, 0, 0, tmr.now, gpio.trig
  gpio.mode(pin,gpio.INT)
  local function pin1cb(level, pulse2)
    print( level, pulse2 - pulse1 )
    pulse1 = pulse2
    trig(pin, level == gpio.HIGH  and "down" or "up")
  end
  trig(pin, "down", pin1cb)
end
```

See also

v: release

`gpio.mode()`

gpio.write()

Set digital GPIO pin value.

Syntax

`gpio.write(pin, level)`

Parameters

- `pin` pin to write, IO index
- `level` `gpio.HIGH` or `gpio.LOW`

Returns

`nil`

Example

```
-- set pin index 1 to GPIO mode, and set the pin to high.
pin=1
gpio.mode(pin, gpio.OUTPUT)
gpio.write(pin, gpio.HIGH)
```

See also

- `gpio.mode()`
- `gpio.read()`

gpio.pulse

This covers a set of APIs that allow generation of pulse trains with accurate timing on multiple pins. It is similar to the `serout` API, but can handle multiple pins and has better timing control.

The basic idea is to build a `gpio.pulse` object and then control it with methods on that object. Only one `gpio.pulse` object can be active at a time. The object is built from an array of tables where each inner table represents an action to take and the time to delay before moving to the next action.

One of the uses for this is to generate bipolar impulse for driving clock movements where you want (say) a pulse on Pin 1 on the even second, and a pulse on Pin 2 on the odd second. `:getstate` and `:adjust` can be used to keep the pulse synchronized to the RTC clock (that is itself synchronized with NTP).

Attention

This sub module is disabled by default. Uncomment `LUA_USE_MODULES_GPIO_PULSE` in `app/include/user_modules.h` before building the firmware to enable it.

To make use of this feature, decide on the sort of pulse train that you need to generate -- hopefully it repeats a number of times. Decide on the number of GPIO pins that you will be using. Then draw up a chart of what you want to happen, and in what order. Then you can construct the table struct that you pass into `gpio.pulse.build`. For example, for the two out of phase square waves, you might do:

Step	Pin 1	Pin 2	Duration (µS)	Next Step
1	High	Low	100,000	2
2	Low	High	100,000	**1**

This would (when built and started) just runs step 1 (by setting the output pins as specified), and then after 100,000µS, it changes to step 2i. This alters the output pins and then waits for 100,000µS before going back to step 1. This has the effect of outputting to Pin 1 and Pin 2 a 5Hz square wave with the pins being out of phase. The frequency will be slightly lower than 5Hz as this is software generated and interrupt masking can delay the move to the next step. To get much closer to 5Hz, you want to allow the duration of each step to vary slightly. This will then adjust the length of each step so that, overall, the output is at 5Hz.

Step	Pin 1	Pin 2	Duration (µS)	Range	Next Step
1	High	Low	100,000	90,000 - 110,000	2
2	Low	High	100,000	90,000 - 110,000	**1**

When turning this into the table structure as described below, you don't need to specify anything special when the number of the next step is one more than the current step. When specifying an out of order step, you must specify how often you want this to be performed. The number of iterations can be up to around 4,000,000,000 (actually any value that fits into an unisgned 32 bit integer). If this isn't enough repeats, then loops can be nested as below

```
{
  { [1] = gpio.HIGH, [2] = gpio.LOW, delay=500 },
  { [1] = gpio.LOW, [2] = gpio.HIGH, delay=500, loop=1, count=1000000000, min=400, max=600 },
  { loop=1, count=1000000000 }
}
```

The loop/count in step 2 will cause 1,000,000,000 pulses to be output (at 1kHz). This is around 11 days. At this point, it will continue onto step 3 which triggers the 11 days of 1kHz. THis process will repeat for 1,000,000,000 times (which is roughly 30 Million years).

The looping model is that associated with each loop there is a hidden variable which starts at the `count` value and decrements on each iteration until it gets to zero when it then proceeds to the next step. If control reaches that loop again, then the hidden variable is reset to the value of `count` again.

gpio.pulse.build

This builds the `gpio.pulse` object from the supplied argument (a table as described below).

Syntax

`gpio.pulse.build(table)`

Parameter

`table` this is view as an array of instructions. Each instruction is represented by a table as follows:

- All numeric keys are considered to be pin numbers. The values of each are the value to be set onto the respective GPIO line. For example `{ [1] = gpio.HIGH }` would set pin 1 to be high. Note this that is the NodeMCU pin number and *not* the ESP8266 GPIO number. Multiple pins can be set at the same time. Note that any valid GPIO pin can be used, including pin 0.
- `delay` specifies the number of microseconds after setting the pin values to wait until moving to the next state. The actual delay may be longer than this value depending on whether interrupts are enabled at the end time. The maximum value is 64,000,000 -- i.e. a bit more than a minute.
- `min` and `max` can be used to specify (along with `delay`) that this time can be varied. If one time interval overruns, then the extra time will be deducted from a time period which has a `min` or `max` specified. The actual time can also be adjusted with the `:adjust` API below.
- `count` and `loop` allow simple looping. When a state with `count` and `loop` is completed, the next state is at `loop` (provided that `count` has not decremented to zero). The coun v: release
  implemented as an unsigned 32 bit integer -- i.e. it has a range up to around 4,000,000,000.

The first state is state 1. The `loop` is rather like a goto instruction as it specifies the next instruction to be executed.

Returns

`gpio.pulse` object.

Example

```
gpio.mode(1, gpio.OUTPUT)
gpio.mode(2, gpio.OUTPUT)

pulser = gpio.pulse.build( {
  { [1] = gpio.HIGH, [2] = gpio.LOW, delay=250000 },
  { [1] = gpio.LOW, [2] = gpio.HIGH, delay=250000, loop=1, count=20, min=240000, max=260000 }
})

pulser:start(function() print ('done') end)
```

This will generate a square wave on pins 1 and 2, but they will be exactly out of phase. After 10 seconds, the sequence will end, with pin 2 being high.

Note that you *must* set the pins into output mode (either gpio.OUTPUT or gpio.OPENDRAIN) before starting the output sequence, otherwise nothing will appear to happen.

gpio.pulse:start

This starts the output operations.

Syntax

`pulser:start([adjust, ] callback)`

Parameter

- `adjust` This is the number of microseconds to add to the next adjustable period. If this value is so large that it would push the delay past the `min` or `max`, then the remainder is held over until the next adjustable period.
- `callback` This callback is executed when the pulses are complete. The callback is invoked with the same four parameters that are described as the return values of `gpio.pulse:getstate`.

Returns

v: release

`nil`

Example

```
pulser:start(function(pos, steps, offset, now)
             print (pos, steps, offset, now)
             end)
```

gpio.pulse:getstate

This returns the current state. These four values are also passed into the callback functions.

Syntax

`pulser:getstate()`

Returns

- `position` is the index of the currently active state. The first state is state 1. This is `nil` if the output operation is complete.
- `steps` is the number of states that have been executed (including the current one). This allows monitoring of progress when there are loops.
- `offset` is the time (in microseconds) until the next state transition. This will be negative once the output operation is complete.
- `now` is the value of the `tmr.now()` function at the instant when the `offset` was calculated.

Example

```
pos, steps, offset, now = pulser:getstate()
print (pos, steps, offset, now)
```

gpio.pulse:stop

This stops the output operation at some future time.

Syntax

`pulser:stop([position ,] callback)`

Parameters

- `position` is the index to stop at. The stopping happens on entry to this state. If not then stops on the next state transition.

- `callback` is invoked (with the same arguments as are returned by `:getstate`) when the operation has been stopped.

Returns

`true` if the stop will happen.

Example

```
pulser:stop(function(pos, steps, offset, now)
            print (pos, steps, offset, now)
            end)
```

gpio.pulse:cancel

This stops the output operation immediately.

Syntax

`pulser:cancel()`

Returns

- `position` is the index of the currently active state. The first state is state 1. This is `nil` if the output operation is complete.
- `steps` is the number of states that have been executed (including the current one). This allows monitoring of progress when there are loops.
- `offset` is the time (in microseconds) until the next state transition. This will be negative once the output operation is complete.
- `now` is the value of the `tmr.now()` function at the instant when the `offset` was calculated.

Example

```
pulser:cancel(function(pos, steps, offset, now)
              print (pos, steps, offset, now)
              end)
```

gpio.pulse:adjust

This adds (or subtracts) time that will get used in the `min` / `max` delay case. This is usef
are trying to synchronize a particular state to a particular time or external event.

v: release

Syntax

`pulser:adjust(offset)`

Parameters

- `offset` is the number of microseconds to be used in subsequent `min` / `max` delays. This overwrites any pending offset.

Returns

- `position` is the index of the currently active state. The first state is state 1. This is `nil` if the output operation is complete.
- `steps` is the number of states that have been executed (including the current one). This allows monitoring of progress when there are loops.
- `offset` is the time (in microseconds) until the next state transition. This will be negative once the output operation is complete.
- `now` is the value of the `tmr.now()` function at the instant when the `offset` was calculated.

Example

```
pulser:adjust(177)
```

gpio.pulse:update

This can change the contents of a particular step in the output program. This can be used to adjust the delay times, or even the pin changes. This cannot be used to remove entries or add new entries. Changing the `count` will change the initial value, but not change the current decrementing value;

Syntax

`pulser:update(entrynum, entrytable)`

Parameters

- `entrynum` is the number of the entry in the original pulse sequence definition. The first entry is numbered 1.
- `entrytable` is a table containing the same keys as for `gpio.pulse.build`

Returns

Nothing

Example

```
pulser:update(1, { delay=1000 })
```