PINS library

Interface to hardware specific IO.

2.11.2019/Pekka Lehtikoski

Table of Contents

1. Introduction

190916, updated 16.9.2019/pekka

The “pins” library separates IO application from hardware level IO functions, so the same IO application can be run on different micro-controllers or as Linux/Windows simulation. This library is optional: Any other IO library can be used as well. Typically IO hardware accessed tough the “pins” interface are GPIO, ADC/DAC, PWM, timers, etc.

1.1 About “pins” library

Function of the "pins" library is to:

- Provide an application with hardware independent access to GPIO, analogs, PWM, timers, etc.
- Provide subs to call forward hardware dependent IO library, like pigpio on Raspberry PI, HAL on STM32, IO simulation on PC and so forth.
- Define hardware pin configuration in such way that it needs to be written only once, and is then available from C code, memory maps and in documentation.

Key points

- Hardware specific IO headers are not included in application (except when there is need to bypass "pins").
- Somewhere we need to map hardware pin numbers, etc. need to the application. This could be simply written as one C file. Here approach is to write it as JSON and then generate the C files automatically by Python script. This enable using the PIN information to automate mapping IO pins to IOCOM signals, generate documentation by scripts, etc, and not write same information twice.

1.2 MIT license

190625, updated 27.6.2019/pekka

2. Using “pins” library

190918, updated 18.9.2019/pekka

2.1 JSON configuration

190917, updated 2.11.2019/pekka

The idea is to write application’s hardware configurations as JSON, and then generate matching C code and header files by Python script pins-to-c.py. It is possible to write directly C code, writing JSON just makes it easier because information needs to be written only in one place.

For example I have application “jane”, which controls “led_builtin” binary output to on a LED on IO board. I want to test this on hardware I named “carol”, which is ESP32, Arduino libraries and the LED connected to pin 2. I need to bind pin address 2 and other pin attributes (like pull-up), etc to name “led_builtin”. I write this information a JSON file jane/config/pins/carol/jane-io.json. Then I run the pins-to-c.py script to generate C code. I use same JSON file together with IOCOM signal configuration JSON to map the IO pins to IOCOM communication signals.

If I would need to run same “jane” application on different hardware, let’s say “alice”: Separate JSON HW configuration file would written for it.

- The LED_BUILTIN output is mapped to specific IO pin by pin number, and/or bank number and. We may also have some attributes for IO pin. For example SERVO PWM has frequency, resolution, initial value, etc.
- One pin, or more generally IO item since we can include timers etc, is configured by name, address and and various parameters.
- Group attribute can be used to generate groups of pins.
- Attribute values excluding group are integers.
- Some parameters can be changed while the code runs.
- IO pins can be mapped directly to IOCOM signals.

As JSON this pin setup would look something like:

```
{
  "io": [{
    "name": "jane",
    "title": "IO pin setup for 'jane' application on 'carol' hardware",
    "groups": [
      {
        "name": "inputs",
        "pins": [
          {"name": "dip_switch_3", "addr": 34, "pull-up": 1},
          {"name": "dip_switch_4", "addr": 35},
          {"name": "touch_sensor", "addr": 4, "touch": 1}
        ]
      },
      {
        "name": "outputs",
        "pins": [
          {"name": "led_builtin", "addr": 2}
        ]
      },
      {
        "name": "analog_inputs",
        "pins": [
          {"name": "potentiometer", "addr": 25, "speed": 3, "delay": 11, "max": 4095}
        ]
      },
      {
        "name": "pwm",
        "pins": [
          {"name": "servo", "bank": 0, "addr": 32, "frequency": 50, "resolution": 12, "init": 2048, "max": 4095},
          {"name": "dimmer_led", "bank": 1, "addr": 33, "frequency": 5000, "resolution": 12, "init": 0, "max": 4095}
        ]
      }
    ]
  }]
}
```

Note: Min and max can be used to set range of values. These do not effect anything to IO, but allow the hardware configuration JSON to pass these values to application.

2.2 Generating C source and header files from JSON

190917, updated 2.11.2019/pekka

The hardware specific IO configuration, like jane-io.json, is converted to C files by pins-to-c.py script. This will generate jane-io.c and jane-io.h file, which can be compiled into the application.

jane-io.h

```
/* This file is gerated by pins-to-c.py script, do not modify. */
OSAL_C_HEADER_BEGINS

typedef struct
{
  struct
  {
    PinGroupHdr hdr;
    Pin dip_switch_3;
    Pin dip_switch_4;
    Pin touch_sensor;
  }
  inputs;

  struct
  {
    PinGroupHdr hdr;
    Pin led_builtin;
  }
  outputs;

  struct
  {
    PinGroupHdr hdr;
    Pin potentiometer;
  }
  analog_inputs;

  struct
  {
    PinGroupHdr hdr;
    Pin servo;
    Pin dimmer_led;
  }
  pwm;
}
pins_t;

extern const IoPinsHdr pins_hdr;
extern const pins_t pins;

OSAL_C_HEADER_ENDS
```

And in C file

```
/* This file is gerated by pins-to-c.py script, do not modify. */
#include "pins.h"

static os_short pins_inputs_dip_switch_3_prm[]= {PIN_PULL_UP, 1};
static os_short pins_inputs_touch_sensor_prm[]= {PIN_TOUCH, 1};
static os_short pins_analog_inputs_potentiometer_prm[]= {PIN_SPEED, 3, PIN_DELAY, 11};
static os_short pins_pwm_servo_prm[]= {PIN_RESOLUTION, 12, PIN_FREQENCY, 50, PIN_INIT, 2048};
static os_short pins_pwm_dimmer_led_prm[]= {PIN_RESOLUTION, 12, PIN_FREQENCY, 5000, PIN_INIT, 0};

const pins_t pins =
{
  {{3, &pins.inputs.dip_switch_3},
    {"dip_switch_3", PIN_INPUT, 0, 34, pins_inputs_dip_switch_3_prm, sizeof(pins_inputs_dip_switch_3_prm)/sizeof(os_short),
OS_NULL},
    {"dip_switch_4", PIN_INPUT, 0, 35, OS_NULL, 0, OS_NULL},
    {"touch_sensor", PIN_INPUT, 0, 4, pins_inputs_touch_sensor_prm, sizeof(pins_inputs_touch_sensor_prm)/sizeof(os_short),
OS_NULL}
  },
  {{1, &pins.outputs.led_builtin},
    {"led_builtin", PIN_OUTPUT, 0, 2, OS_NULL, 0, OS_NULL}
  },


  {{1, &pins.analog_inputs.potentiometer},
    {"potentiometer", PIN_ANALOG_INPUT, 0, 25, pins_analog_inputs_potentiometer_prm,
sizeof(pins_analog_inputs_potentiometer_prm)/sizeof(os_short), OS_NULL}
  },


  {{2, &pins.pwm.servo},
    {"servo", PIN_PWM, 0, 32, pins_pwm_servo_prm, sizeof(pins_pwm_servo_prm)/sizeof(os_short), OS_NULL},
    {"dimmer_led", PIN_PWM, 1, 33, pins_pwm_dimmer_led_prm, sizeof(pins_pwm_dimmer_led_prm)/sizeof(os_short), OS_NULL}
  }
};

static const PinGroupHdr *pins_group_list[] =
{
  &pins.inputs.hdr,
  &pins.outputs.hdr,
  &pins.analog_inputs.hdr,
  &pins.pwm.hdr
};

const IoPinsHdr pins_hdr = {pins_group_list, sizeof(pins_group_list)/sizeof(PinGroupHdr*)};
```

2.3 Pins library types

190918, updated 18.9.2019/pekka

Include “pins.h”

2.3.1 Enumeration of pin types

```
typedef enum
{
  PIN_INPUT,
  PIN_OUTPUT,
  PIN_ANALOG_INPUT,
  PIN_ANALOG_OUTPUT,
  PIN_PWM,
  PIN_TIMER
}
pinType;
```

2.3.2 Enumeration of pin parameters

```
typedef enum
{
  PIN_PULL_UP,
  PIN_TOUCH,
  PIN_FREQENCY,
  PIN_RESOLUTION,
  PIN_INIT,
  PIN_SPEED, /* not used */
  PIN_DELAY, /* not used */
  PIN_MIN,   /* Minimum value for signal */
  PIN_MAX    /* Maximum value for signal, 0 if not set */
}
pinPrm;
```

2.4 Pin definition structure

```
/* Structure to set up static information about one IO pin or other IO item.
 */
typedef struct Pin
{
  /* Pointer to pin name string.  */
```

```
    os_char *name;

    /* Pint type, like PIN_INPUT, PIN_OUTPUT... See pinType enumeration.   */
    pinType type;

    /* Hardware bank number for the pin, if applies.   */
    os_short bank;

    /* Hardware address for the pin.  */
    os_short addr;

    /* Pointer to parameter array, OS_NULL if none.  */
    os_short *prm;

    /* Number of items in parameter array. */
    os_char prm_n;

    /* Next pin in linked list of pins belonging to same group as this one. */
    const struct Pin *congroup_next;

    /* Next pin in linked list of pins belonging to the same IO configuration. */
    const struct Pin *board_next;
}
Pin;
```

There is group_next and board_next. Often is handy to loop trough all pins, like when making memory map for IO com. Or reading group of inputs with one command. To facilitate this we can group pins together. Pins with same PIN_GROUP number set go generate linked list and all pins of IO board a second linked list.

These appear in automatically generated .h file as:

extern const Pin *jane_pins; Pointer to Jane's first IO pin. We use this to initialize the IO pins, like pins_setup(jane_pins)

2.5 Pins library functions

190917, updated 2.11.2019/pekka

2.5.1 Initialize pins

```
/* Set up IO hardware.
 */
void pins_setup(
    const IoPinsHdr *pins_hdr,
    os_int flags);
```

2.5.2 Set and get pin state

```
/* Set IO pin state.
 */
void pin_set(
    const Pin *pin,
    os_int state);

/* Get pin state.
 */
os_int pin_get(
    const Pin *pin);
```

2.5.3 Access pin parameters

```
/* Modify IO pin parameter.
```

```
 */
void pin_set_prm(
    const Pin *pin,
    pinPrm prm,
    os_int value);

/* Get value of IO pin parmeter.
 */
os_int pin_get_prm(
    const Pin *pin,
    pinPrm prm);
```

3. Examples

190918, updated 18.9.2019/pekka

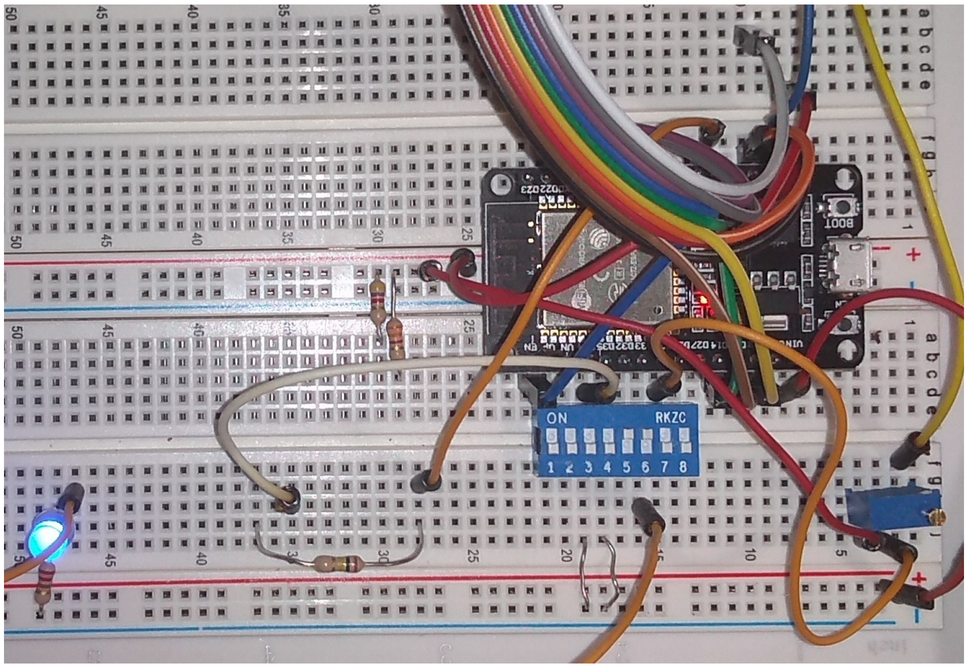
3.1 Pins example "jane"

190918, updated 31.10.2019/pekka

The "jane" is name for "pins" library example IO device application. "Carol" is name for a specific IO board, here MELIFE ESP32 development board which has specific uses for it's pin.

3.1.1 Carol hardware

It is useful to give name to a IO board so it can be called. My breadboard MELIFE ESP32 test is called "carol", so the name "carol" means ESP32 with Arduino libraries. This combination can be developed by either Visual Studio Code + Platform IO or Arduino IDE.



"Carol" is a bit of a mess, it needs to survive for a day of testing

Carol hardware wiring is as follows:

IO pin name	IO pin type	IO address	Description
DIP_SWITCH_3	digital input	34	Blue 8 dip switch block, switch pin 3 grounds the input. Dip switch is also declared as pull-up, but this doesn't seem to do anything on ESP32.
DIP_SWITCH_4	digital input	35	Blue 8 dip switch block, switch pin 4 grounds the input.
TOUCH_SENSOR	digital input	4	Touch the sensor T0 (GPIO4)
POTENTIOMETER	analog input	25	10k potentiometer wired to give voltage from 0 to 3.3V, ADC channel 8
SERVO	PWM	22	Servo PWM control. PWM channel 0 (set as bank) is used with 50 Hz frequency and 12 bits precision, and initialized at middle (2048)
DIMMER_LED	PWM	33	LED dimmer PWM control. PWM channel 1 (set as bank) is used with 5000 Hz frequency and 12 bits precision, and initialized at dark (0)

The IO pin name is name what Jane application uses for specific functionality, like potentiometer, DIP switch 3, or PWM controlled dimmed blue led. IO address is hardware specific, that tells what has been wired to which pin of the ESP32. For Arduino libraries, it is Arduino GPIO pin number.

3.1.2 JSON configuration and generated C code

The Jane example was used for this pins library and JSON configuration and generated C code can be found earlier in this document.

3.1.3 The jane application

The jane-example.c is simple single thread example IO device application. It demonstrates how to use IO trough “pins” interface.

- The pins_setup() initialize the IO pins.
- pin_set() set IO pin state
- pin_get() get IO pin state

```
#include "jane.h"
/* Here we include hardware specific IO code. The file name is always same for jane, but
   pins/<hardware> is added compiler's include paths
 */
#include "jane-io.c"


os_timer t;
os_boolean state;
os_int dip3, dip4, touch, dimmer, dimmer_dir, potentiometer;


osalStatus osal_main(
    os_int argc,
    os_char *argv[])
{
    pins_setup(&pins_hdr, 0);


    os_get_timer(&t);
    state = OS_FALSE;
    dip3 = dip4 = touch = -1;
    dimmer = 0;
    dimmer_dir = 1;
    potentiometer = -4095;


    /* When emulating micro-controller on PC, run loop. Just save context pointer on
       real micro-controller.
     */
    osal_simulated_loop(OS_NULL);
    return 0;
}


osalStatus osal_loop(
    void *app_context)
{
    os_int x, delta;
    os_char buf[32];


    /* Digital output */
    if (os_elapsed(&t, 50))
    {
        os_get_timer(&t);
        state = !state;
        pin_set(&pins.outputs.led_builtin, state);
    }


    /* Digital input */
    x = pin_get(&pins.inputs.dip_switch_3);
    if (x != dip3)
    {
        dip3 = x;
        osal_console_write(dip3 ? "DIP switch 3 turned ON\n" : "DIP switch 3 turned OFF\n");
    }
    x = pin_get(&pins.inputs.dip_switch_4);
    if (x != dip4)
    {
        dip4 = x;
        osal_console_write(dip4 ? "DIP switch 4 turned ON\n" : "DIP switch 4 turned OFF\n");
    }

    /* Touch sensor */
    x = pin_get(&pins.inputs.touch_sensor);
    delta = touch - x;
    if (delta < 0) delta = -delta;
    if (delta > 20)
    {
        touch = x;
        if (touch)
        {
          osal_console_write("TOUCH_SENSOR: ");
          osal_int_to_string(buf, sizeof(buf), touch);
          osal_console_write(buf);
          osal_console_write("\n");
        }
    }

    /* Analog input */
    x = pin_get(&pins.analog_inputs.potentiometer);
    delta = potentiometer - x;
    if (delta < 0) delta = -delta;
    if (delta > 100)
    {
        potentiometer = x;
        osal_console_write("POTENTIOMETER: ");
        osal_int_to_string(buf, sizeof(buf), potentiometer);
        osal_console_write(buf);
        osal_console_write("\n");
    }

    /* PWM */
    dimmer += dimmer_dir;
    if (dimmer > 4095 || dimmer < 0) dimmer_dir = -dimmer_dir;
    pin_set(&pins.pwm.dimmer_led, dimmer);

    return OSAL_SUCCESS;
}

void osal_main_cleanup(
    void *app_context)
{
}
```

3.1.4 What was point of all this complexity?

The goal here is to separate IO device application from specific hardware and write it only once: If we have many somewhat complex IO boards, an IO board application can survive hardware, platform or development tool changes. Secondary goals are to run IO board application as PC simulation, easily to bind IO to communication and to generate documentation from same source data.
This is quite useful for developing more complex applications with long life time. In simple cases, one is probably better off calling hardware specific IO direly from application and something like “pins” library is just unnecessary complexity.

4. Implementing pins library stub for a new platform

190918, updated 2.11.2019/pekka

Example: Implement Raspberry PI IO trough pins library

- Copy pins/code/arduino/pins_basics.c to code/linux/pins_basics_pi.c
- If there is no pins_hw_defs.h in linux folder, copy it also.

Example: Run Jane on Raspberry pi.

Jane was originally tested with hardware named “carol”, which is ESP32 with Arduino libraries wired to control specific set of inputs and outputs. I name the new hardware “david”.

- Copy jane-io.json from jane/pins/carol to jane/pins/david.
- Modify the jane-io.json to reflect Raspberry PI IO wiring.
- Add call to pins-to-c.py script for “david” thto jane/scripts/pin-config-to-c-code.sh.