PINS library

Generic interface to hardware specific IO.

26.3.2020/Pekka Lehtikoski

Table of Contents

1. Introduction....2
1.1 About “pins” library....2
1.2 MIT license....2
2. Using “pins” library....3
2.1 JSON configuration....3
2.1.1 Configuring inputs....5
2.1.2 Configuring outputs....5
2.1.3 Configuring analog inputs....5
2.1.4 Configuring analog outputs....6
2.1.5 Configuring PWM....6
2.1.6 Configuring SPI bus....6
2.1.7 Configuring timers....6
2.1.8 Configuring UARTS....7
2.1.9 Configuring pins reserved for debugger and development board....7
2.2 Generating C source and header files from JSON....7
2.3 Pins library types in C....8
2.3.1 Pins library main header file....9
2.3.2 Enumeration of pin types....9
2.3.3 Enumeration of pin attributes....9
2.3.4 Pin definition structure....9
2.4 Pins library functions....10
2.4.1 Initialize pins....10
2.4.2 Set and get pin state....10
2.4.3 Access pin parameters....10
3. Examples....12
3.1 Pins example “jane”....12
3.1.1 Carol hardware....12
3.1.2 JSON configuration and generated C code....13
3.1.3 The jane application....13
4. Implementing pins library stub for a new platform....15

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.
- Connect IO to communication without minimal application code.
- Hardware configuration is written in JSON only once and converted to C code, connected to IOCOM and documented by scripts.

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

IO configuration for a specific hardware is written as JSON file. This JSON file is converted to C code by script. Generated C code has static structure describing each IO pin. The application reads and writes IO pins trough pins library function, giving pointer to static pin structure as argument. Same application code can be used with different HW by including and linking generated C filed for the hardware.

Steps:

- Create JSON file /coderoot/pins/examples/jane/config/pins/carol/pins-io.json. Replace “jane” with name of your application and “carol” with name of your hardware (nick name of your board).
- Generate python script /coderoot/pins/examples/jane/scripts/config-to-c-code.py which converts JSON configuration to C code.
- Include generated .c and .h files in your project. Set /coderoot/pins/examples/jane/config/include/carol in you compiler’s include path.
- Add line #include pins-io.c in applications config.c file and #include pins-io.h in application’s main header file.

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.

A JSON pin setup would look something like ESP32 example below:

```
{
  "io": [{
    "name": "jane",
    "title": "IO pin setup for 'jane' application on 'carol' hardware",
    "groups": [
      {
```

```
 "name": "inputs",
 "pins": [
  {"name": "gazerbeam", "addr": 39, "interrupt": 1},
  {"name": "dip_switch_3", "addr": 34, "pull-up": 1},
  {"name": "dip_switch_4", "addr": 35},
  {"name": "touch_sensor", "addr": 36, "touch": 1}
 ]
},
{
 "name": "outputs",
 "pins": [
  {"name": "led_builtin", "addr": 33}
 ]
},
{
 "name": "analog_inputs",
 "pins": [
  {"name": "potentiometer", "addr": 26, "max": 4095}
 ]
},
{
 "name": "pwm",
 "pins": [
  {"name": "servo", "bank": 0, "addr": 22, "frequency": 50, "resolution": 12, "init": 2048, "max": 4095},
  {"name": "dimmer_led", "bank": 1, "addr": 27, "frequency": 5000, "resolution": 12, "init": 0, "max": 4095}
 ]
},
{
 "name": "uart",
 "pins": [
  {"name": "U2_TXD", "addr": 17},
  {"name": "U2_RXD", "addr": 16}
 ]
},
{
 "name": "debugger",
 "pins": [
    {"name": "AD1_TDI", "addr": 12},
    {"name": "AD0_TCK", "addr": 13},
    {"name": "AD3_TMS", "addr": 14},
    {"name": "AD2_TDO", "addr": 15},
    {"name": "U0_TXD", "addr": 1},
    {"name": "U0_RXD", "addr": 3}
  ]
},
{
 "name": "devboard",
 "pins": [
    {"name": "DEV_SCK", "addr": 6},
    {"name": "DEV_SDO", "addr": 7},
```

```
        {"name": "DEV_SDI", "addr": 8},
        {"name": "DEV_SHD", "addr": 9},
        {"name": "DEV_SWP", "addr": 10},
        {"name": "DEV_CSC", "addr": 11}
       ]
    }
   ]
 }]
}
```

2.1.1 Configuring inputs

Inputs are configured within “inputs” group. Pin name is can be up to 15 characters + terminating ‘\0’. Use only ‘a’ - ‘z’, ‘A’ - ‘Z’, ‘0’ - ‘1’ and underscore ‘_’ characters. The pin name will be used in C code as written. Address “addr” is GPIO pin address.

The attributes for inputs

- pull-up: Set 1 to enable pull-up resistor on input.
- pull-down: Set 1 to enable pull-down resistor on input.
- touch: Set 1 to define this input as touch sensor (this is set up here, even signal may be analog):
- interrupt: Set 1 to trigger HW interrupt when pin state changes, either falling edge, rising edge or both. Edge on which to trigger is in C code.

Examples

```
{"name": "gazerbeam", "addr": 39, "interrupt": 1},
{"name": "dip_switch_3", "addr": 34, "pull-up": 1},
```

2.1.2 Configuring outputs

Configuring outputs under “outputs” group is similar to configuring inputs.

The attributes for inputs

- pull-up: Set 1 to enable pull-up resistor on output.
- pull-down: Set 1 to enable pull-down resistor on output.

Example

```
{"name": "led_builtin", "addr": 33}
```

2.1.3 Configuring analog inputs

Analog inputs are configured within “analog_inputs” group.

The attributes for analog inputs

- addr: GPIO pin address.
- max: Set maximum value for analog input. Typically set by number AD conversion resolution, 1023 for 10-bit ADC and 4095 for 12 bit ADC, etc. Application and IOCOM link can use this setting to scale value to known units, which is useful if for example newer version of the hardware has higher ADC resolution.

Example

```
{"name": "potentiometer", "addr": 26, "max": 4095}
```

2.1.4 Configuring analog outputs

Analog outputs are configured within “analog_outputs” group.

The attributes for analog outputs

- max: Set maximum value for analog output. Typically set by number D/A conversion resolution, 1023 for 10-bit DAC and 4095 for 12 bit DAC, etc.

Example

```
{"name": "myaout", "addr": 25, "max": 4095}
```

2.1.5 Configuring PWM

PWM pins are configured within “pwm” group.
The attributes for PWM pin

- bank: PWM channel?
- addr: GPIO pin address.
- frequency: PWM frequency in Hz, max 65535.
- frequency-kHz: PWM frequency kHz. This is used instead of “frequency” to set higher frequencies than 65kHz.
- resolution: Number of bits in PWM signal. If set to 12, values range from 0 to 4095. For example 2048 would mean that pin is on 50% of time (50% duty cycle). If generating clock pulse, set 1 bit.
- max: Set maximum value for analog output. Typically set by number D/A conversion resolution, 1023 for 10-bit DAC and 4095 for 12 bit DAC, etc.
- init: Set initial value. for example 1024 with 12 bit resolution would mean 25% duty cycle.

Example

```
{"name": "servo", "bank": 0, "addr": 22, "frequency": 50, "resolution": 12, "init": 2048, "max": 4095},
{"name": "ccd_clock",  "bank": 0, "addr": 22, "timer": 0, "frequency-kHz": 10000, "resolution": 1}
```

2.1.6 Configuring SPI bus

SPI buses are configured within “spi” group.

The attributes for SPI bus

- addr: SPI bus number to configure.
- max: Set maximum value for analog output. Typically set by number D/A conversion resolution, 1023 for 10-bit DAC and 4095 for 12 bit DAC, etc.
- miso: Sets GPIO pin address for MISO (master in, slave out) signal.
- mosi: Sets GPIO pin address for MOSI (master out, slave in)signal.
- sclk: Sets GPIO pin address for SCLK (clock) signal.
- cs: Sets GPIO pin address for CS (chip select) signal. This enables a specific SPI device.
- dc: Needed for some devices, like some TFT displays. Sets data control GPIO pin address.

Example

```
{"name": "tft_spi", "addr": 0, "miso": 19, "mosi": 23, "sclk": 18, "dc": 2}
```

2.1.7 Configuring timers

Timers are used to trigger timed interrupts.

Timer attributes

- bank: .
- addr:
- timer: Timer number ?
- frequency: PWM frequency in Hz, max 65535.
- frequency-kHz: PWM frequency kHz. This is used instead of “frequency” to set higher frequencies than 65kHz.

Example

```
{"name": "igc_timer",  "bank": 0, "addr": 0, "timer": 0, "frequency-kHz": 250, "resolution": 1},
```

2.1.8 Configuring UARTS

Serial port UARTS are configured same way regardless of signal levels, difference between TTL level, RS232, RS422.

UART attributes

- addr: UART number.
- rx: Receive data GPIO pin address.
- tx: Transmit data GPIO pin address.
- tc: Transmit control GPIO pin address (reserved for future).
- speed: Baud rate, divided by 100. Set for example 96 for 9600 bps.

Example

```
{"name": "uart2", "addr": 2, "rx": 16, "tx": 17, "speed": 96}
```

2.1.9 Configuring pins reserved for debugger and development board

These are skipped by pins-to-c.py script and cannot be accessed from C code. Reason for these to be defined here is to avoid accidentally using a reserved pin, and in future to include these in IO documentation once we have automatic JSON configuration to document converter written.

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 in C

190918, updated 26.3.2020/pekka

In C code, an IO pin, a SPI bus, timer or UART is referred by static “Pin” structure. Static structure for each “pin” is initialized in C code generated by script from JSON configuration. The “Pin” structure is the same, regardless what this “pin” actually is, “pinType type” member of the structure has the pin type:

2.3.1 Pins library main header file

Use #include “pinsx.h” to include pins library headers for use with IOCOM.

2.3.2 Enumeration of pin types

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

2.3.3 Enumeration of pin attributes

Each “pin” has specific set of attributes, see JSON configuration notes to which attributes can be applied for a specific pin type. Notice that numeric enumeration values can change.

```
typedef enum
{
   PIN_PULL_UP,
   PIN_TOUCH,
   PIN_FREQENCY,
   PIN_FREQENCY_KHZ,
   PIN_RESOLUTION,
   PIN_INIT,
   PIN_INTERRUPT,
   PIN_TIMER_SELECT,
   PIN_MISO,
   PIN_MOSI,
   PIN_SCLK,
   PIN_CS,
   PIN_DC,
   PIN_RX,
   PIN_TX,
   PIN_MIN,
   PIN_MAX
}
pinPrm;
```

2.3.4 Pin definition structure

Structure that holds static information about one IO pin. Pin structure for each IO “pin” is initialized in C code generated from JSON configuration.

```
typedef struct Pin
{
   /** Pint type, like PIN_INPUT, PIN_OUTPUT... See pinType enumeration.
    */
   os_char type;

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

   /** Hardware address for the pin.
```

```
     */
    os_short addr;

    /** Pointer to parameter array, two first os_shorts are reserved for storing value
        as os_int.
     */
    os_short *prm;

    /** Number of items in parameter array. Number of set parameters is this divided by
        two, since each 16 bit number is parameter number amd parameter value.
     */
    os_char prm_n;

    /** Next pin in linked list of pins belonging to same group as this one. OS_NULL
        to indicate that this pin is end of list of not in group.
     */
    const struct Pin *next;

    /** Pointer to IO signal, if this pin is mapped to one.
     */
    const struct iocSignal *signal;
}
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.

All pins of a device can be referred using “extern const IoPinsHdr pins_hdr” in script generated pins-io.h. We use pins_setup(&pins_hdr, 0) this to set up all the IO pins, or “pins_read_all(&pins_hdr, PINS_DEFAULT)” in main loop to read all pins.

2.4 Pins library functions

190917, updated 2.11.2019/pekka

2.4.1 Initialize pins

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

2.4.2 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);
```

2.4.3 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

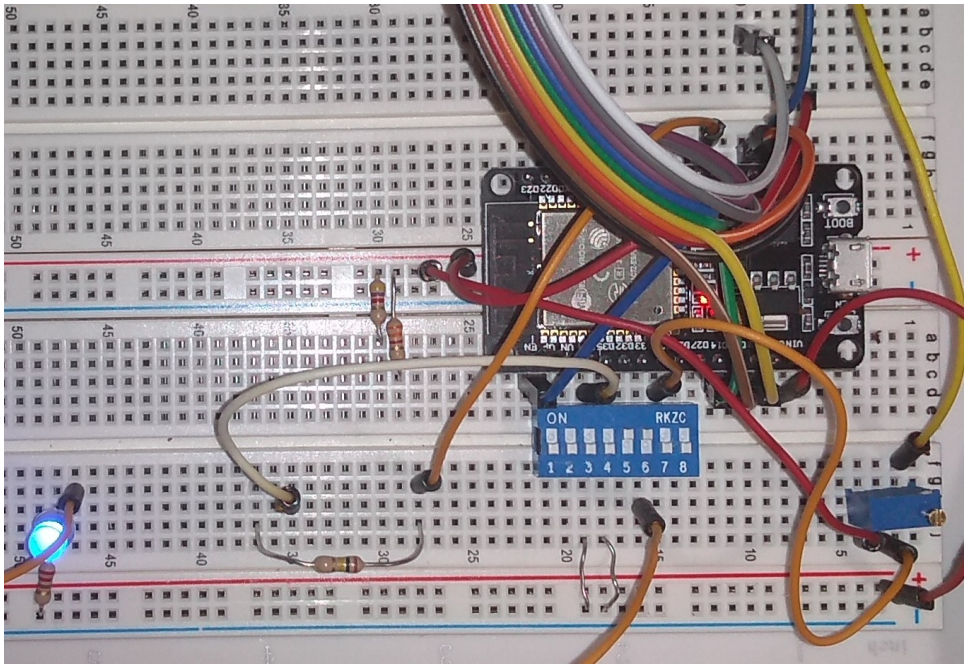
3.1 Pins example “jane”

190918, updated 26.3.2020/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:

THE GPIO PIN ADDRESSESS HERE ARE NOT WELL CHOSEN AND HAVE BEEN CHANGED. THIS TEXT NEEDS TO BE UPDATED.

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.
GAZERBEAM	digital input	36	Photo transistor signal as interrupt.
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)
{
}
```

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.