Pawel Gburzynski

PicOS

Pin Operations

August, 2013

1 Introduction

This note describes the various standardized interfaces to pin operations under PicOS. Simple digital I/O pins are most conveniently accessed directly; however, some usage patterns ask for specialized handling. For example, for some digital input pins, the praxis may want to perceive events when the value of a pin changes. Then, the changes may have to be debounced to compensate for the inherent imperfections of mechanical switches. ADC (analog-to-digital) and DAC (digital-to-analog) conversion is another issue that requires some system support.

A GPIO (General Purpose Input Output) pin can be available for at least two types of access: digital (binary) reading and writing. We look at these issues from the viewpoint of the microcontroller, i.e., when we say “reading” or “writing,” we mean reading or writing by a program running in the microcontroller. Thus any GPIO pin accessible by the praxis can be formally used for:

1. binary digital input, assuming the pin is driven by a digital signal from the outside
2. binary digital output, assuming the pin sends a digital signal out

Additionally, depending on the microcontroller type, certain pins can be configured as ADC input. In addition to 1 and 2, such a pin can be also used for:

3. calculating discrete integer samples whose values are determined by the voltage level fed into the pin from the outside

Some microcontrollers may be equipped with DAC converters that generate digitally-controllable voltage on selected output pins. In such a case, those pins may additionally be used for:

4. sending out voltage determined by discrete integer values stored in some internal registers of the microcontroller

As viewed by the praxis, functions 3 and 4 are best served by (analog) sensors and actuators.

2 Unified access

The tools described in this section provide for flexible access to any available pin where the direction and function of the pin can be defined dynamically by the praxis. They are seldom used in what we would call production praxes, but are handy for testing boards and their interfaces.

There are two reasons why production praxes are not crazy about the unified access tools:

1. in a specific board configured for a specific application, pin configurations (directions and roles) are typically hardwired and fixed
2. the tools are relatively expensive in terms of code size (being an overkill for inflexible, statically preconfigured pins)

One (legacy) exception is perhaps the pulse monitor/notifier providing a nontrivial mechanism for counting pulses and triggering events on debounced signals appearing on selected input pins. This function comes as part of the complete unified access functionality. Should it be needed in a (new) production praxis, it can be easily reimplemented in a more frugal way.

The unified access interface is described by a collection of functions in *PicOS/Libs/Lib/pin_read.c*. A program referencing those functions should include the header file *pinopts.h* (located in the same directory).

2.1 Declaring the pins (system level)

The pins made accessible to the unified access tools must be properly described within the board description. Such a description takes the form of a list which is typically defined in file *board_pins.h*. Here's an example from *BOARDS/WARSAW/board_pins.h*:

```
#define PIN_LIST  {           \
      PIN_DEF     (P6, 0),    \
      PIN_DEF     (P6, 1),    \
      PIN_DEF     (P6, 2),    \
      PIN_DEF     (P6, 3),    \
      PIN_DEF     (P6, 4),    \
      PIN_DEF     (P6, 5),    \
      PIN_DEF     (P6, 6),    \
      PIN_DEF     (P6, 7),    \
      PIN_DEF     (P1, 6),    \
      PIN_DEF     (P1, 7),    \
      PIN_DEF     (P2, 2),    \
      PIN_DEF     (P2, 3),    \
      PIN_DEF     (P2, 5),    \
      PIN_DEF     (P2, 6),    \
      PIN_DEF     (P4, 0),    \
      PIN_DEF     (P4, 4),    \
      PIN_DEF     (P4, 5),    \
      PIN_DEF     (P4, 6),    \
      PIN_DEF     (P4, 7),    \
      PIN_DEF     (P5, 4)     \
}

#define PIN_MAX         20     // Total number of pins
#define PIN_MAX_ANALOG  8      // ADC-capable
#define PIN_DAC_PINS    0x0706 // DAC-capable
```

The first argument of *PIN_DEF* identifies the port, and the second argument tells the pin number within the port (0–7). From the praxis, the pins will be identified by integer numbers 0 through 19 (the total number of pins in *PIN_LIST* is 20) counting from the top of the list.

Three more constants follow the pin list declaration. *PIN_MAX* tells the list length, i.e., the total number of pins made accessible to the praxis.[1] *PIN_MAX_ANALOG* gives the number of ADC-capable pins. All such pins, if any, must appear at the top of the list.

The last constant, *PIN_DAC_PINS*, identifies up to two DAC-capable pins encoded as two bytes of a word (6 and 7 in the above example). Those values represent indexes in the list, i.e., pins P6.6 and P6.7 are declared as DAC-capable. There can be no more than two pins with this property and, of course, the capability must be available in the CPU.

If no pins are to be ADC-capable (or DAC-capable), the respective constants (*PIN_MAX_ANALOG* and *PIN_DAC_PINS*) need not be defined, or they can be set to zero. Note that pin number zero (in the list) cannot be DAC-capable: the zero value of a byte in *PIN_DAC_PINS* means that the slot represented by the byte is not used. For example,

```
#define PIN_DAC_PINS 0x0700
```

means that there is only one DAC-capable pin, namely pin number 7 (counting from the top of the list).

[1] This specification is optional and intended as a parameter available to the praxis. The library functions determine the number of pins based on the list length.

If for whatever reason you want to have holes in the list, i.e., empty slots affecting the numeration of the pins, but representing no pins, put *PIN_RESERVED* in place of *PIN_DEF*, e.g.,[2]

```
#define PIN_LIST  {           \
      PIN_DEF     (P6, 0),    \
      PIN_RESERVED,           \
      PIN_RESERVED,           \
      PIN_DEF     (P6, 3),    \
      PIN_DEF     (P6, 4),    \
      PIN_DEF     (P6, 5)     \
}
```

In the above case, the logical pins numbers 1 and 2 are unused (they will be treated as unavailable by the tool functions). Note that the total formal number of pins (the range of pin numbers for the operations listed in Section 2.4) is still determined by the length of the list, so it is 6 (0–5) in the above example.

2.2 Declaring a pin monitor

One optional functionality of the toolbox is the so-called *pin monitor* using two selected pins as the so-called *counter* and *notifier*. The idea is that the counter counts debounced pulses on its pin optionally matching the count against a comparator, while the notifier is a simple trigger for events caused by pulses on the pin.

There are two implementation options.[3] In the first (recommended) case, the two pins must be both on P2 (they are thus interrupt-capable). This implementation is simpler and more efficient, because it takes advantage of hardware interrupts. Here is a way to declare this variant of the pin monitor:

```
#define PULSE_MONITOR PINS_MONITOR_INT (3, 4)
#define MONITOR_PINS_SEND_INTERRUPTS 1
```

The declaration should be put into *board_pins.h*, e.g., after the declaration of pins available for general access (Section 2.1). The arguments of *PINS_MONITOR_INT* are the two pin numbers assumed to refer to P2, i.e., P2.3 and P2.4 in the above case. The pins need not be mentioned in the pin list (Section 2.1). If they are, the praxis should not touch them while their monitor functions are active (see Section 2.5). There is no way to have only one function (just the counter or just the notifier), i.e., the two pins come as an inseparable pair.

In the second case, if the pins are not interrupt capable,[4] the declaration looks like this:

```
#define PULSE_MONITOR PINS_MONITOR (P6, 1, P6, 2)
#define MONITOR_PINS_SEND_INTERRUPTS 0
```

The arguments of *PINS_MONITOR* are two pairs: port designator and pin number (referring to that port). The second constant is optional.

The timing parameters (debouncing) are described by some constants in *PicOS/pins.h*. These constants are not configurable on the per-board/per-praxis basis, but can be changed globally, if needed. Here are their standard values:

```
#define PMON_DEBOUNCE_UNIT    16
#define PMON_DEBOUNCE_CNT_ON  3
#define PMON_DEBOUNCE_CNT_OFF 3
#define PMON_DEBOUNCE_NOT_ON  4
#define PMON_DEBOUNCE_NOT_OFF 100
```

[2] I have no idea why I implemented this feature. It has never been used and it is a bit of a nuisance. Oh well, the whole thing is legacy junk.

[3] Reflecting the mess with an old RFM project.

[4] The functionality of interrupts in this case is emulated by the millisecond clock.

PMON_DEBOUNCE_UNIT tells the unit of time for the remaining values. Thus, the value of *PMON_DEBOUNCE_CNT_ON* in the above example means that the counter pulse must be on for at least 48 milliseconds (3 x 16) to be considered valid. Then, when it goes off , no other pulse will be considered until the signal remains off for another 48 milliseconds or more (as prescribed by *PMON_DEBOUNCE_CNT_OFF*). The next two constants apply (in the same way) to the notifier.

2.3 Configuring pin interrupts

This part of board configuration is general and applies in any situation when some GPIO pins trigger interrupts that must be perceived by some driver (e.g., signal pins connected to external modules). The system assumes that only P1 and P2 are interrupt capable.

The board-specific view of the pin interrupt configuration is described in file *board_pins_interrupts.h* in the board's directory. That file has the following generic layout:

```
#ifdef P1_INTERRUPT_SERVICE
… interrupt capture code for P1
#endif
#ifdef P2_INTERRUPT_SERVICE
… interrupt capture code for P2
#endif
```

The simple idea is that the same file is included in the generic interrupt service routines for P1 and P2 with the respective symbol defined in each case. The capture code should recognize whether the interrupt is of interest (typically by examining the IF flags) and invoke the necessary routines. Note that the routines are executed as an interrupt service (preempting all FSMs).

The configuration files should also make sure that the needed interrupt service routines are compiled in, which is typically done by putting:

```
//+++ p1irq.c
```

and/or

```
//+++ p2irq.c
```

preferably somewhere into *board_pins.h* (see [2]). Needless to say, the pins must be configured for receiving interrupts on the pertinent signal edges.

The above “special statements” are usually accompanied with:

```
REQUEST_EXTERNAL (p1irq);
REQUEST_EXTERNAL (p2irq);
```

The statement is a macro which makes sure that the specified argument is formally referenced from the present program file as an external symbol. This is not needed for normal compilation of a praxis, but becomes necessary when the board description is turned into a library (see [2]).

For the pin monitor, the present section only applies when the monitor pins are interrupt-capable.The *P2_INTERRUPT_SERVICE* section of *board_pins_interrupts.h* should then include this line:

```
#include "irq_pins.h"
```

which provides the code to recognize and handle the interrupts coming from the monitor pins.

2.4 Basic operations

The functions listed in this section become available to the praxis with the inclusion of *pinopts.h* in the program header.

int pin_write (word pin, word value)

The function sets the value and/or the status of the pin whose logical number (i.e., its index on the pin list – see Section 2.1) is provided as the first argument. If the second argument has no other bits set than possibly the least significant bit, the pin is made an output digital pin, and the least significant bit of **value** determines its status: 1-high, 0-low.

Note that the function reconfigures the pin dynamically as needed, regardless of its role described statically by the system (e.g., in *board_pins.h*).

If bit number 2, i.e., 0x04, in **value** is set, the pin is reconfigured to ADC or DAC function, depending on whether bit 1 (0x02) is set (ADC) or not (DAC). Any other bits of **value** are ignored.

If bit number 2 is zero, but bit number 1 (0x02) is set, the pin is made digital input.

The function returns **ERROR** (−1) if the pin is unavailable (out of range) or incapable of performing the requested function. Otherwise, if everything was OK, the function returns 0.

word pin_read (word pin)

The function determines the status of the pin and, if the pin is digital input, reads its current value. If the returned value has no other bits set than possibly the least significant bit, it means that the pin is in fact digital input, and the least significant bit tells its status: 0-low, 1-high.

If bit number 2 (0x04) is set, it means that the pin is ADC or DAC, depending on whether bit 1 (0x02) is set (DAC) or not (ADC).[5] If bit number 2 is zero, but bit 1 is set, it means that the pin is digital output. In that case, the least significant bit tells its current status: 0-low, 1-high. The function returns **ERROR** (−1) if the pin is unavailable (out of range).

int pin_read_adc (word state, word pin, word ref, word smpt)

The function samples the voltage on the **pin** and returns the ADC value. The third argument specifies the source of reference voltage: 0 – 1.5V, 1 – 2.5V, 2 – Vcc. The third argument gives the sampling time in PicOS milliseconds, with zero being equivalent to 1. If state is not **WNONE**, the function will release the current FSM and wake it up in the indicated state when the operation should be re-tried to return the sample value. If state is **NULL**, the function spins internally until the conversion has been accomplished. Here is how the function should be used:

```
…
state: TAKE_SAMPLE:
    val = pin_read_adc (TAKE_SAMPLE, 4, 2, 1);
    … sample available …
```

Normally, the function returns a non-negative integer number between 0 and 4095 inclusively, which is a linear transformation of the input voltage between 0 and the reference. Needless to say, the operation only makes sense if issued to a pin declared as capable of handling ADC signals. Otherwise, the function returns **ERROR** (−1). This will also happen if the pin is in principle available for ADC input, but the ADC is being used by the RF module for sampling RSSI (this only applies to some legacy radio modules, e.g., DM2200). If this is the case, **pin_read_adc** can only be used while the receiver is switched off.

int pin_write_dac (word pin, word val, word fac)

[5] Note that the meaning of this bit is inverted with respect to **pin_write**.

The function sets DAC voltage on the **pin**, which must be one of the two pins declared as being capable of outputting DAC signals. The specified value must be between 0 and 4095 inclusively and is scaled to voltage between 0 and the reference. The reference is taken from the last ADC operation (**pin_read_adc**) and it cannot be Vcc, i.e., it must be 1.5V or 2.5V. This means that the function should be preceded by a (possibly dummy) call to **pin_read_adc** with the **ref** argument being 0 or 1. A single call will do for an arbitrary number of subsequent calls to **pin_write_dac**: all of them will use the last set ADC reference for setting the output voltage. If the last argument is nonzero, the output voltage will be multiplied by 3, but not above Vcc.

Regardless of the formal capabilities of a given pin (e.g., as predeclared in *board_pins.h*) its actual dynamic status should be set by **pin_write** before using it for a given purpose (calling one of the remaining functions listed above). Notably, **pin_read_adc** and **pin_write_dac** do force this status to agree with their operation, even if it didn't agree when the function was called. What this means is that, for example, P6.0 may have been used as a digital input pin and a call to **pin_read_adc** will make it ADC input, as long as its declaration in *board_pins.h* formally allows that.

2.5 The pin monitor

These operations become available (the inclusion of *pinopts.h* brings them in) if the pin monitor functionality is defined for the board (see Section 2.2).

void pmon_start_cnt (lint count, Boolean edge)

The function starts the counter. Note that before the counter has been activated, any signals arriving at the counter pin are ignored. The first argument is the initial value to be put into the counter, unless it is negative, in which case the counter is not reset (it retains the previous value). Note that the counter is a long (32-bit) signed integer.

The second argument tells the signal edge: NO – down (i.e., high-to-low transition), YES – up.

void pmon_stop_cnt ()

The function stops the counter. From now on any signals arriving on the counter pin will be ignored.

void pmon_set_cmp (lint value)

If the argument is non-negative, the function sets the comparator (associated with the counter) to the specified value and activates the comparator. The comparator will be generating an event whenever the counter reaches its value. The identifier of this event (usable as the first argument of *when*) is *PMON_CNTEVENT* (aka. *PMON_CMPEVENT*).

If the argument is negative, the comparator is deactivated. Note that initially the comparator is inactive.

void pmon_dec_cnt ()

The function safely (i.e., avoiding races with interrupts) decrements the counter by the value currently stored in the comparator.

void pmon_sub_cnt (lint value)

The function safely (i.e., avoiding races with interrupts) subtracts the argument from the counter.

void pmon_add_cmp (lint value)

The function safely (i.e., avoiding races with interrupts) adds the argument to the comparator.

lword pmon_get_cnt ()

The function returns the current value of the counter.

Boolean pmon_pending_cmp ()

The function tells whether a comparator event is pending and, if so, removes the pending event. It should be called by an FSM awakened by a comparator event (*PMON_CNTEVENT*). It can also be used in polling mode.

void pmon_start_not (Boolean edge)

The function starts the notifier. Note that the notifier is dormant (any signals arriving at the pin are ignored) until started explicitly. The argument indicates the signal edge: NO – down (i.e., high-to-low transition), YES – up.

void pmon_stop_not ()

The function stops the notifier. From now on any signals arriving at the notifier pin will be ignored.

Boolean pmon_pending_not ()

The function tells whether a notifier event is pending and, if so, removes the pending event. Notifier events are triggered by the pertinent signal transitions on the pin and can be awaited by FSMs (with the standard *when* operation) as *PMON_NOTEVENT*. The function should be called by an FSM awakened by the event. It can also be used in polling mode.

word pmon_get_state ()

The function retrieves and returns the state of the pin monitor. The returned value is an OR of these bits (the constants can be AND-ed with the value to check if the respective conditions hold):

PMON_STATE_CNT_ON

the counter is on

PMON_STATE_CNT_RISING

the counter signal edge is up (as opposed to down)

PMON_STATE_CMP_ON

the comparator is on

PMON_STATE_CMP_PENDING

the comparator event is pending

PMON_STATE_NOT_ON

the notifier is on

PMON_STATE_NOT_RISING

the notifier signal edge is up

PMON_STATE_NOT_PENDING

the notifier event is pending

3 Raw access

The most straightforward way to reference a pin directly is to access the respective port location and the respective bit within the port. For example:

```
P2IN & 0x02
```

extracts the bit number 1 from the input register of port 2, i.e., port P2.1. The following two operations:

```
_BIS (P3OUT, 0x18);
_BIC (P3OUT, 0xE0);
```

set the (output) ports P3.3 and P3.4 to 1 and clear ports P3.5 through P3.7. Note that for the above operations to work, the directions (and functions) of the ports must be set up properly.

3.1 Static port configuration (system level)

The static pre-configuration of ports is described by a set of constants whose default values are stored in *portinit.h* (in *PicOS/MSP430*). Some of those constants are often redefined in the board configuration files (typically in *board_pins.h*). By default, for every port:

- special function is deselected (i.e., the port is used in the standard way)
- the direction is set to output
- the value is set to zero
- interrupts are disabled (for the interrupt-capable ports)
- internal pull-up/pull-down is disabled (for pins that have this capability)
- the drive strength is set to low (for pins that have this capability)

If you want to change the default configuration of some pins, you can redefine any of the following generic constants (the letter x stands for the port identifier). The redefinitions are typically inserted at the top of *board_pins.h*.

PIN_DEFAULT_PxSEL (special function select)

The default value of this constant is 0x00 (it deselects special functions). For example:

```
#define PIN_DEFAULT_P6SEL 0x0F
```

selects the special function for ports P6.0 through P6.3.[6]

PIN_DEFAULT_PxDIR (direction)

The default value is 0xFF, which means that the direction of all pins is output.

PIN_DEFAULT_PxOUT (output level)

The default value is 0x00, which means that the value of all pins is zero (low).

PIN_DEFAULT_PxIE (interrupt enable, applicable to capable ports only)

The default value is 0x00, which means that interrupts are disabled.

PIN_DEFAULT_PxIES (interrupt edge select, capable ports only)

The default value is 0x00 meaning the rising edge.

PIN_DEFAULT_PxREN (pull-up/pull-down setting, capable ports only)

The default value is 0x00 meaning no pulling.

PIN_DEFAULT_PxDS (drive strength, capable pins only)

The default value is 0x00 meaning low strength.

[6] For MSP430F1..., the special function of a pin on port 6 is ADC input.

To reconfigure a pin from the praxis, you can directly set/clear the respective bits in the respective registers. The register name is implied by the constant suffix. For example:

```
_BIS (P2IE, 0x44);
```

enables interrupts for pins P2.2 and P2.6. By default those interrupts will be triggered on a rising edge (see the default value of *PIN_DEFAULT_P2IES*).

3.2 Single-argument reference

The operation listed in this section allow the praxis (or the system) to reference any pin by its simple number represented by a single symbolic constant. The constant looks like P*x_n* where *x* is the port number (or name) and *n* is the pin number within the port. This way the pin becomes an entity with a single name which can be easily used in macros. Examples: *P3_1*, *PJ_0*, *P10_7*. The same constants are defined for all boards. If a particular pin does not exist on a given board, the operations on it are void.

Note that normally, to access a pin, you have to provide two arguments, i.e., the port and the pin bit within the port, e.g.,

```
_BIS (P6, 0x01);
```

This makes it difficult to parameterize some operations by pins (two constants are needed to describe a single pin).

The raw access operation are available always (the standard inclusion of *sysio.h* is all that is needed). Their availability costs no code, because they are all implemented as macros. They are primarily intended for quick (debugging) hacks. Here is the list:

_PDS (p, v)

Sets the direction of pin *p* to *v* (0 – input, 1 – output). For example:

```
_PDS (P3_0, 1);
_PDS (PJ_1, 0);
```

_PFS (p, v)

Sets the special function of pin *p* to *v* (0 – normal, 1 – special).

_PVS (p, v)

Sets the output value of pin *p* to *v*, which should be 0 or 1. The pin's direction must have been set to output previously, i.e., the operation does not automatically force the output direction for the pin.

_PHS (p, v)

Sets the drive strength of pin *p* to *v*, which should be 0 (low) or 1 (high).

_PPS (p, v)

Sets the pull-up/pull-down of pin *p* to *v*, which should be 0 (off) or 1 (on).

_PD (p)

Returns the direction of pin *p* (0 – input, 1 – output).

_PF (p)

Returns the special function status of pin *p* (0 – normal, 1 – special).

_PV (p)

Returns the input value of pin *p* (0 or 1). The pin's direction must have been set to input previously.

_PH (p)

Returns the drive strength of pin *p* (0 or 1).

_PP (p)

Returns the pull-up/pull-down status of pin *p* (0 or 1).

Any operation referencing a non-existent pin on a given board or a non-existent capability (e.g., drive strength, pull-up/pull-down) is void. If the operation returns a value, that value is zero.

4 Sensors and actuators

For an ADC-capable pin used for its ADC function, the most natural view is as a sensor. The sensor/actuator view can be extended onto any GPIO pin whose role is to pass simple signals between the microcontroller and the external world.

One difference with respect to the unified view (Section 2) is that the functionality of pins accessed in the sensor/actuator view is (usually) static, i.e., it is typically predefined for the board (in *board_pins.h*) via constants. This isn't really a hard restriction. In special cases, pins can be reconfigured, e.g., via prelude/postlude macros for an analog sensor (see Page 13). For a digital sensor, the implementation (driver) functions can perform arbitrary actions on the pins.

Sensors are typically configured in *board_pins.h* (sometimes in a separate header file included from *board_pins.h* which, by convention, is called *board_sensors.h*). Here is a sample way to define a bunch of sensors:

```
#define SENSOR_LIST {                                   \
      INTERNAL_TEMPERATURE_SENSOR,                      \
      INTERNAL_VOLTAGE_SENSOR,                          \
      ANALOG_SENSOR ( 1,                                \
                    512,                                \
                      0,                                \
        SREF_VEREF_AVSS,                                \
                      4,                                \
                      0),                               \
      DIGITAL_SENSOR (0, shtxx_init, shtxx_temp),       \
      DIGITAL_SENSOR (0, NULL, shtxx_humid),            \
      ANALOG_SENSOR ( 0,                                \
                     16,                                \
                      7,                                \
        SREF_VEREF_AVSS,                                \
                      4,                                \
                   REFON)                               \
      }
#define N_HIDDEN_SENSORS      2
#define SENSOR_ANALOG
#define SENSOR_DIGITAL
#define SENSOR_INITIALIZERS
```

The definition takes the form of a list which also implicitly numbers the sensors. Normally, the first sensor in the list receives the number 0, the second 1, and so on. If the constant *N_HIDDEN_SENSORS* is defined and set to something positive, then the numbering starts from –*N_HIDDEN_SENSORS*, i.e., the first two sensors in the above example are numbered –2 and –1. They are considered “hidden”, and the first “visible” (i.e., non-negatively numbered) sensor is the third sensor from the top. The idea is that some special (“internal” in the above example) sensors are referenced by negative numbers (numbered from –1 down) while the “normal” sensors are numbered from zero up.

A sensor can be either analog or digital (the first two system macros in the above example translate to references to *ANALOG_SENSOR*. All analog sensors are assumed to represent some ADC-capable pins receiving some voltage to be sampled and

compared against some range. Digital sensors, on the other hand, generally represent some specific external modules (requiring special protocols) and are handled by dedicated functions (drivers). One special case of a digital sensor provides an interface to a group of digital input I/O pins.

The last three constants defined in the above example (note that their values are irrelevant) indicate to the system that: 1) analog sensors are present (*SENSOR_ANALOG*), 2) digital sensors are present (*SENSOR_DIGITAL*), and initializers for digital sensors are present (*SENSOR_INITIALIZERS*, note *shtxx_init* in the declaration of the SHT sensor – explained in Section 4.2). These constants make sure that the code required to handle the respective components is compiled in.

4.1 Analog sensors

An analog sensor is declared with this operation:

```
ANALOG_SENSOR (isi, nsa, pin, ure, sht, ere);
```

where:

isi is the inter-sample interval in milliseconds and can be between 0 and 255. It is relevant when ***nsa*** is greater than 1.

nsa is the number of samples to average for a single reading (between 1 and 65537). The value returned by the sensor will be an average of that many readings taken at ***isi*** millisecond intervals.

pin is the signal source, typically a pin number applicable to the ADC-capable port for the given microcontroller (e.g., P6 on MSP430F1611). The pin must be properly configured, i.e., its special function must be selected. The range of *pin* is 0–15. The first 8 values refer to pins, the remaining values identify special sources (see Figure 1 and [1]).

ure selects the source of reference voltage. The range is 0–7 (see Figure 2, also refer to [1] for details). Note that the file *include/msp430/adc12.h* within the *mspgcc* package defines symbolic names for the selection values for signal source, reference voltage, and other fields and flags applicable to ADC settings.

sht describes the sample holding time, i.e., the amount of time during which the sample signal is being held charging the capacitors of the converter. Generally, longer time gives better accuracy, especially when the signal comes from a source with high impedance, although short intervals are OK in most cases. The range is 0–15 translated according to Figure 3. The ADC clock cycle is about 625 kHz.

ere is a collection of auxiliary flags (a byte) describing the contents of the lower half of register *ADC12CTL0* (see [1]). If the internal reference generator has been selected in ***ure*** (values 1 and 5), the ***ere*** flags should include at least *REFON* (bit 0x20) which turns on the internal generator. Then, if *REF2_5V* (bit 0x40) is also set, the reference voltage will be 2.5V instead of 1.5V.

Some analog sensors may require special actions before (and after) a measurement. For example, some external reference voltage may have to be switched on (e.g., by setting a GPIO pin high) before the measurement, and then, after the measurement, switched off to reduce current drain. Such actions can be described by defining two macros: *sensor_adc_prelude* and *sensor_adc_postlude*. If any of them is defined, it will be inserted before/after the code for taking the measurement. Note that the macro will be executed once per “logical” readout, even if that readout is an average of multiple samples.

Input channel select	
0000	A0
0001	A1
0010	A2
0011	A3
0100	A4
0101	A5
0110	A6
0111	A7
1000	Ve_{REF+}
1001	V_{REF-}/Ve_{REF-}
1010	Temperature sensor
1011	$(AV_{CC} - AV_{SS}) / 2$
1100	$(AV_{CC} - AV_{SS}) / 2$
1101	$(AV_{CC} - AV_{SS}) / 2$
1110	$(AV_{CC} - AV_{SS}) / 2$
1111	$(AV_{CC} - AV_{SS}) / 2$

Figure 1: Signal source selection for ADC.

Select reference	
000	$V_{R+} = AV_{CC}$ and $V_{R-} = AV_{SS}$
001	$V_{R+} = V_{REF+}$ and $V_{R-} = AV_{SS}$
010	$V_{R+} = Ve_{REF+}$ and $V_{R-} = AV_{SS}$
011	$V_{R+} = Ve_{REF+}$ and $V_{R-} = AV_{SS}$
100	$V_{R+} = AV_{CC}$ and $V_{R-} = V_{REF-}/ Ve_{REF-}$
101	$V_{R+} = V_{REF+}$ and $V_{R-} = V_{REF-}/ Ve_{REF-}$
110	$V_{R+} = Ve_{REF+}$ and $V_{R-} = V_{REF-}/ Ve_{REF-}$
111	$V_{R+} = Ve_{REF+}$ and $V_{R-} = V_{REF-}/ Ve_{REF-}$

Figure 2: Reference voltage selection for ADC.

Each of the two macros takes an argument which is substituted with a packed set of sensor parameters. Note that the same pair of macros is executed for all analog sensors, so they may have to tell which sensor is being handled at the moment. That is best accomplished by examining the sensor's pin, which is its unique feature. Here is an example:

```
#define sensor_adc_prelude(p) \
      do { \
            if (ANALOG_SENSOR_PIN(p) == 0) { \
                  _BIS (P5OUT, 0x10); \
            } else { \
                  _BIS (P4OUT, 0x80); \
            } \
            mdelay (20); \
      } while (0)

#define sensor_adc_postlude(p) \
      do { \
            _BIC (P5OUT, 0x10); \
```


```
            _BIC (P4OUT, 0x80); \
    } while (0)
```

The *if* statement in the first macro shows the way to get hold of the sensor's pin number. If this is pin number 0 (the first “normal” sensor in the example above), then P5.4 is set high, otherwise (any other analog sensor), P4.7 goes high. This happens before the readout. After the readout, regardless of the sensor, both pins go down.

SHTx Bits	ADC12CLK cycles
0000	4
0001	8
0010	16
0011	32
0100	64
0101	96
0110	128
0111	192
1000	256
1001	384
1010	512
1011	768
1100	1024
1101	1024
1110	1024
1111	1024

Figure 3: Sample holding time for ADC.

4.2 Digital sensors

A digital sensor is defined with this statement:

```
DIGITAL_SENSOR (val, init_fun, readout_fun);
```

where:

val is some value that is passed to the readout function (*readout_fun*). Its purpose is to tell apart different instances or roles of the same sensor.

init_fun is a (pointer to a) function which will be called at system's initialization, whose role is to initialize the sensor. The function is optional (the pointer can be *NULL*). The function should be declared as *void fun (void)*.

readout_fun is a function to be called to return the sensor's value or to await an event on the sensor. It should be declared as *word fun (word st, const byte *par, address res)*. If the second argument is not *NULL*, the function is called to return the sensor's value; otherwise, it is called to await an event.

The two functions comprise the sensor's driver and have to be provided for every new digital sensor type. Sometimes they are accompanied by an interrupt-handling function.

One standard type of digital sensor related to pin access is the pin sensor. At most one such sensor can be defined. The sensor groups up to 16 input pins and presents their values as a 16-bit unsigned word.

4.2.1 Declaring a pin sensor (system level)

The pins available through the sensor have to be preconfigured as input (see Section 3.1). The list of the pins whose input values will contribute to the sensor is declared (typically in *board_pins.h*) as a special constant, e.g.,

```
#define INPUT_PIN_LIST {      \
      INPUT_PIN (P1, 1, 0),   \
      INPUT_PIN (P1, 2, 1),   \
      INPUT_PIN (P2, 0, 0),   \
      INPUT_PIN (P5, 3, 0)    \
}
```

Every declaration (*INPUT_PIN*) specifies three parameters: the port, the pin number (0–7), and the "polarization" or "edge" (0 or 1) indicating whether the high (0) or low (1) value of the pin should translate into 1 on the respective bit position of the sensor value.

Then, the sensor should be declared as a digital sensor in this manner:

```
DIGITAL_SENSOR (0, pin_sensor_init, pin_sensor_read),
```

The two functions (the driver of the pin sensor) are provided in *pin_sensor.c* (in directory *PicOS*). Here is a more complete example including all the requisites:

```
#include "analog_sensor.h"
#include "sensors.h"
#include "pin_sensor.h"

#define SENSOR_LIST {                                          \
      INTERNAL_TEMPERATURE_SENSOR,                             \
      INTERNAL_VOLTAGE_SENSOR,                                 \
      DIGITAL_SENSOR (0, pin_sensor_init, pin_sensor_read), \
}

#define SENSOR_ANALOG
#define SENSOR_DIGITAL
#define SENSOR_INITIALIZERS
#define N_HIDDEN_SENSORS  2
```

Note that the first two (hidden) sensors are analog ones, so the "analog" attributes of the declaration must be present.

The above declaration is all that is needed in those situations when we do not care about the events triggered by the pin sensor. For something amounting to a button functionality, you may need events (interrupts sent on pin status changes), unless you want to poll the sensor continuously in a loop.

To augment the above setting with event capability, include the following declarations, e.g., after the declaration of *PIN_LIST*:

```
#define INPUT_PIN_P1_IRQ 0x06
#define INPUT_PIN_P2_IRQ 0x01
#define INPUT_PIN_P1_EDGES 0x04
//+++ p1irq.c p2irq.c
REQUEST_EXTERNAL (p1irq);
REQUEST_EXTERNAL (p2irq);
```

The first two constants declare the interrupt pins for ports P1 and P2 corresponding to the pins from those ports contributing to the sensor. Note that port P5 cannot send interrupt, so the fourth pin from the set cannot trigger events (although its status is still available when the sensor value is read).

The third constants says that the interrupt edge for P1.2 (the second pin from the list above) should be reversed from the default low-high, i.e., this particular pin should be sending interrupts on high-to-low transitions. The special comment + the two *REQUEST_EXTERNAL* statements (see [2]) makes sure that the pin interrupts handlers for P1 and P2 are compiled in.

If all the interrupt-capable pins of the sensor belonged to a single port (P1 or P2) then only one of the IRQ constants would be needed (and only one interrupt service function would have to be requested). The *EDGES* constant is only required if some pins of the respective port have reverse polarity, i.e., they are supposed to show as “up” (the corresponding bit in the sensor value set to 1) in the low state. The constant only concerns those pins that are interrupt capable (are supposed to trigger sensor event). For example, the last pin on the list above (P5.3) is not interrupt capable which means that its status belongs to the sensor value, but the changes of that status won't trigger sensor events. If the polarity of that pin were reversed, as in:

```
INPUT_PIN (P5, 3, 1),
```

there would be no need to reflect that fact in an *EDGE* constant, i.e., no declaration like this one:

```
#define INPUT_PIN_P5_EDGES 0x08
```

would be required.

4.2.2 Pin sensor value and events

The value returned by the pin sensor (Section 4.3) is a 16-bit unsigned number whose individual bits, counting from the least significant position, correspond to the pins contributing to the sensor in the order of appearance of those pins on *INPUT_PIN_LIST* (Section 4.2.1). Thus (in reference to the example in Section 4.2.1), the value 0x000B would mean that P1.1 is high, P1.2 is low (note the reverse polarity of that pin), P2.0 is low, and P5.3 is high. As there are only four pins in the sensor, only the least significant nibble of the value can ever be nonzero.

If sensor events are enabled (Section 4.2.1), then an event is triggered if these conditions hold simultaneously:

1. There is a pending *wait_sensor* request to the sensor (Section 4.3), i.e., the praxis is (already) waiting for the event.
2. The state of one of the pins declared with the *IRQ/EDGES* constants (Section 4.2.1) changes in accordance with the declaration (i.e., low-to-high or high-to-low).

This means, for example, that when a pin changes its status while the praxis is not waiting for the event, the event will be lost. This provides a way to debounce pin events (in contrast to button events, they are not debounced by the system). For example, having responded to an event, the praxis may delay issuing the next *wait_sensor* request for a while.

4.3 Sensor interface to the praxis

Both types of sensors (analog and digital) offer the same interface to the praxis, at least as far as reading values is concerned. This function:

```
void read_sensor (word state, sint id, address value);
```

retrieves the current value of the sensor number *id*, according to the ordering of sensors on *SENSOR_LIST* (Section 4). Note that if hidden sensors are present, *id* can be sensibly negative.

The *state* argument provides a way to suspend the value retrieval, if the operation cannot be carried out immediately. In such a case, the system will block the FSM and

resume it at the indicated state when it makes sense to retry the operation. According to the philosophy of PicOS, the code for reading a sensor value should resemble this:

```
…
state GET_VALUE:
      read_sensor (GET_VALUE, HUMIDITY_SENSOR, &result);
      … result available …
```

For some sensors, notably the pin sensor, the value is available immediately (so the state argument is ignored), but even for some apparently simple sensors (e.g., most analog sensors), the retrieval procedure will often go through several stages (looping through the state), so it is a good habit to always assume that the state is needed and used in a nontrivial way.

Generally, the size of the value returned by *read_sensor* depends on the sensor. For a digital sensor representing a complex module, that value can be compound (an array of values), so you have to make sure that the last argument points something that can accommodate the result. For some standard sensors, like analog sensors and the pin sensor, the value is always one word. For an analog sensor, the value is always between 0 and 4095 (representing a 12-bit linear transformation with respect to the reference voltage); for the pin sensor, the value is a collection of bits representing the status of the individual pins contributing to the sensor (Section 4.2.1).

Some sensors can trigger events (e.g., this is an option for the pin sensor – Section 4.2.2). Many digital sensors (e.g., accelerometers, motion detectors) are inherently event-oriented, so their events are often more important than values. Analog (ADC-based) sensors never trigger events. They are passive:[7] the mechanism of ADC conversion is only activated for a value readout, and the issue is always simple: the readout produces a 12-bit representation of the current voltage on the pin relative to whatever reference voltage source was declared for the sensor. Note that, depending on the sensor's declaration (Section 4.1), the readout may take some time (like in averaging multiple samples) and the FSM may be blocked while waiting for its completion.

Here is the operation to await a sensor event:

```
void wait_sensor (sint sn, word state);
```

The operation will block the FSM to wake it up in the indicated state when the nearest event on the sensor occurs. For some sensors (not for a pin sensor, however) it is possible that the event is already pending when the operation is issued; in such a case, the state transition will occur immediately.

4.4 Actuators

Actuators are similar to sensors, but they work in the output direction. They are in fact simpler, because there are no actuator events. Controlling an actuator boils down to setting it to some value.

4.4.1 Declaring actuators (system level)

Here is a sample sequence (taken from *board_options.h* of some board configuration) declaring three actuators:

```
#include "analog_actuator.h"
#include "actuators.h"

#define ACTUATOR_LIST {                                  \
    ANALOG_ACTUATOR (0, 0, 0, 0, 0),                     \
    ANALOG_ACTUATOR (512, 1, 1, 0, 0),                   \
    DIGITAL_ACTUATOR (0, NULL, pin_actuator_write),      \
```

[7] There exists an *ADC_SAMPLER* option (not covered in this document) for collecting synchronously large volumes of data, possibly from several ADC pins simultaneously.

```
}

#define     ACTUATOR_ANALOG
#define     ACTUATOR_DIGITAL
```

It closely resembles a declaration of sensors (see Section 4). The two constants at the bottom indicate that both analog and digital actuators are present in the system (so the handling code can be compiled in). There are no initializers (the second argument of the single case of *DIGITAL_ACTUATOR* is *NULL*), but if there were, one more definition:

```
#define ACTUATOR_INITIALIZERS
```

would be necessary.

Similar to sensors, you can have hidden (i.e., negatively numbered) actuators. A definition of the form:

```
#define N_HIDDEN_ACTUATORS x
```

indicates that *x* actuators from the top of the list are hidden.

The arguments of *DIGITAL_ACTUATOR* are exactly as for *DIGITIAL_SENSOR*: they point to two functions (the initializer being optional) amounting to the sensor's driver. The second function is called to write a value to the actuator.

4.4.2 The pin actuator

The pin actuator is the reverse of the pin sensor: it allows you to control a set of (output) pins treated as a single 16-bit unsigned value with different bits representing different pins. Here is a sample sequence declaring such pins:

```
#define OUTPUT_PIN_LIST {     \
      OUTPUT_PIN (P2, 3, 0),  \
      OUTPUT_PIN (P2, 2, 0),  \
      OUTPUT_PIN (P6, 7, 1),  \
      OUTPUT_PIN (P5, 4, 0),  \
      OUTPUT_PIN (P3, 3, 1)   \
}

#include "pin_actuator.h"
```

The above declaration can be put into *board_pins.h* preceding the declaration of the actual pin actuator (Section 4.4.1). The pins contributing to the actuator are numbered according to their order on the above list. Thus, for example, P2.3 is numbered 0 and will be represented by the least significant bit of the actuator's value.

Similar to the pin sensor, a pin may have its polarity reversed (P6.7 and P3.3 in the above example). In such a case, 1 in the pin's bit in the actuator's value will translate into the low setting of the pin (and vice versa).

Note that pins contributing to the pin actuator should have their directions properly preconfigured (Section 3.1).

4.4.3 Actuator interface to the praxis

This is taken care of by a single function declared as:

```
void write_actuator (word state, sint id, address val);
```

where the arguments are as for *read_sensor* (Section 4.3). This time, the last argument provides the value to be written to the sensor. As the length of that value generally depends on the sensor type, it is represented by a pointer.

Similar to sensors, writing a value to some actuators may involve blocking, so the state argument is generally important. In the simple special case of the pin actuator, the value is always written immediately and the state argument is irrelevant. The interpretation of

the value is exactly as for the pin sensor (Section 4.2.2), except that the direction is now reversed.

4.4.4 Analog actuators

An analog actuator is associated with a DAC-capable pin. There can be at most two such pins (at most two analog actuators), and this feature is not available on all CPU types. The declaration of an analog actuator (Section 4.4.1) looks like this:

```
ANALOG_ACTUATOR (time, module, bump, ref, href);
```

If ***time*** is nonzero, it specifies the amount of time in PicOS milliseconds for which the output pin will be set to the voltage prescribed by the actuator's value. After that time, the voltage will be automatically reset to zero. This is intended for generating (rough) pulses. If ***time*** is zero, then once set, the voltage stays put until it is explicitly changed.

The actuator's pin is selected with ***module*** which can be 0 or 1. On MSP430F1611, 0 selects P6.6 and 1 selects P6.7.

The output voltage is generated with respect to the reference voltage (as some fraction thereof) which can be internal or external. If ***bump*** is 1 (or nonzero), the output voltage is amplified 3 times compared to the standard value, i.e., it can exceed the reference voltage up to 3 times (but not above the supply voltage).

If ***ref*** is zero, the reference voltage is taken from ADC (it is either 1.5V (if ***href*** is zero), or 2.5V (if ***href*** is nonzero). If ref is nonzero, the reference voltage is taken from a special external pin (and ***href*** is then ignored).

When a value is written to an analog actuator (by *write_actuator*), the output pin is set to the voltage determined as the reference voltage scaled linearly by the value such that 4095 means max (i.e., the reference voltage), and 0 means 0V. With ***bump***, the output voltage produced this way is multiplied by 3 (up to the supply voltage).

If ***time*** is nonzero, *write_actuator* won't unblock until that many milliseconds have elapsed, and then the pin will be automatically reset to 0V. Otherwise, the pin will be set to the proper voltage and the function will return immediately. The converter remains in the "on" state, sustaining the last value, until it is changed with a new call to *write_actuator*. To switch the converter off, write any value larger than 4095, e.g., *WNONE*. That will bring the output voltage down to 0V and disable the converter.

Note that when using the internal reference for actuators, you should avoid mixing sensor and actuator operations, unless you know very well what you are doing. This is because writing a value to an actuator may reset the ADC.

5 Buttons

A button is an input pin acting as a software-debounced signal trigger. This is one more way of handling binary input pins, mildly recommended for actual buttons (because the signal is automatically debounced preventing spurious events).

5.1 Declaring buttons (system level)

Here is a sample sequence declaring a set of buttons (from *board_pins.h* of the CHRONOS board):

```
#define BUTTON_LIST {           \
    BUTTON_DEF (2, 0x04, 0),    \
    BUTTON_DEF (2, 0x02, 0),    \
    BUTTON_DEF (2, 0x10, 0),    \
    BUTTON_DEF (2, 0x01, 0),    \
    BUTTON_DEF (2, 0x08, 0),    \
}

#define BUTTON_PIN_P2_IRQ 0x1F
```

```
//+++ p2irq.c
REQUEST_EXTERNAL (p2irq);
#define BUTTON_M1 0
#define BUTTON_M2 1
#define BUTTON_S1 2
#define BUTTON_S2 3
#define BUTTON_BL 4
```

The arguments of *BUTTON_DEF*, declaring a single button, are interpreted as:

1. The port number. All the buttons in the above example are on P2.
2. The pin bit within the port, e.g., 0x04 stands for pin 2, i.e., P2.2.
3. The repeat flag (0 or 1) with 1 meaning that holding the button will result in a repeated series of events until the button is released.

Note that only interrupt-capable ports can be used for buttons. The constant *BUTTON_PIN_P2_IRQ* combines the button pins associated with P2 and is interpreted as a mask identifying button interrupts on P2. If P1 bits were also present in the declared set of buttons, then another constant would be needed, *BUTTON_PIN_P1_IRQ*, to identify button interrupts on P2.

The last five define statements provide symbolic names for the buttons and are intended for the praxis.

The following three parameters can be redefined (in *board_pins.h* or *options.sys*):

BUTTON_DEBOUNCE_DELAY (default value 64)

This is the number of PicOS milliseconds for which the pin status must remain stable to be deemed valid.

BUTTON_REPEAT_DELAY (default value 750)

This is the number of PicOS milliseconds for which the button must be continuously pressed to begin the repeat action. This only applies to buttons declared with the repeat flag.

BUTTON_REPEAT_INTERVAL (default value 256)

This is the number of PicOS milliseconds separating two consecutive events triggered when repeating. This only applies to buttons declared with the repeat flag.

By default, the polarity of all buttons is low-to-high, i.e., the button is considered pressed when the pig signal is high. If you insert:

```
#define BUTTON_PRESSED_LOW 1
```

into *board_pins.h* or *options.sys*, then the polarity of all buttons will be reversed. Note that all buttons must have the same polarity.

5.2 Accessing buttons from the praxis

A program willing to use buttons should include *buttons.h* in its header. Then it should invoke at some point this function:

```
void buttons_action (void (*)(word));
```

where the single argument points to a “report function” accepting a single word-type argument. The function will be called whenever a button has been pressed (which also includes repeat events). The argument tells the number of the pressed button reflecting its position on *BUTTON_LIST* (Section 5.1). The first button is numbered 0. If multiple buttons have been pressed at the same time, only the lowest-numbered button is reported. The status of other buttons (button pins) won't be checked and reported for as long as the present button remains pressed. If the button was declared with the repeat

flag, and it remains pressed for at least *BUTTON_REPEAT_DELAY* (Section 5.1), the report function will be called at *BUTTON_REPEAT_INTERVALS* for as long as the button remains pressed.

Example:

```
…
word Buttons;
…
void buttons (word but) {
      Buttons |= (1 << but);
      trigger (&Buttons);
}
…
fsm main {
      state MAIN_INIT:
            …
            buttons_action (buttons);
            …
      state WAIT_BUTTON:
            when (&Buttons, SEE_BUTTONS);
            release;
      state SEE_BUTTONS:
            // Buttons tells which buttons have been
            // pressed since we last looked
            …
            Buttons = 0;
            proceed WAIT_BUTTON;
      …
}
```

Calling *buttons_action* with the argument *NULL* revokes the previously declared report function and effectively disabled the buttons.

References:

[1] Texas Instruments, **MSP430x1xx Family: user's Guide** <slau049f>.

[2] Olsonet Communications, **mkmk – the PicOS Makefile Maker**.