PicOS
Pin Operations

Version 2.01
October, 2006

Preamble

This note describes the praxis[1] interface to pin operations under PicOS. This interface is described via a collection of functions stored in file PicOS/Libs/Lib/pin_read.c. A program referencing those operations should include the file *pinopts.h* (located in the same directory). The present interface is implemented for the MSP430 architecture. The identical (or a highly similar) interface will be implemented for eCOG1 as soon as the need arises.

1. Pin access types

A GPIO (General Purpose Input Output) pin can be available for at least two types of access: digital 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 declared as accessible by the praxis can be formally used for:

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

Additionally, depending on the microcontroller type, certain pins can be used as ADC input. Such a pin can be also used for:

3. calculating discrete integer samples input 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 voltage on selected pins that is made available outside. In such a case, some pins may also be used for:

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

The high-level interface provides uniform way of accessing GPIO pins for all the purposes listed above, according to the availability of free pins and the capabilities of the microcontroller.

In addition to this standard functionality, the praxis interface optionality implements two special modes for selected pins called *pulse counter* and *notifier*. This functionality, jointly named the *pulse monitor*, is compiled into the system when **`PULSE_MONITOR`** is set to 1 in *options.sys*.

In a nutshell, the pulse counter allows the user to implement a counter whereby signal transitions on a given pin result in incrementing an integer number, which can be thresholded (as to generate events on reaching predetermined values), initialized, saved, and zeroed in a way that avoids losing pulses. The pulse counter is debounced to eliminate noise. The notifier allows the praxis to receive an interrupt if the value on a selected pin changes for more than a certain threshold duration. This, for example, makes it possible to detect simple events like broken wires, again with possible debouncing – to avoid premature "jumps to wrong conclusions."

[1] This is the PicOS term for "application".

2. Pin definitions

The actual configuration of available GPIO pins depends on the particular board. Several configuration parameters of the system are determined by the board type declared in *options.sys*. This declaration has the form:

```
#define TARGET_BOARD       board_name
```

where board_name is one of the known board types (they are listed in *PicOS/options.h*). If the symbol is not defined in *options.sys*, the system assumes **BOARD_UNKNOWN**. A system compiled for such a board cannot reference any peripherals. Also, no pin operations are available on such a system.

Each board type that leaves some GPIO pins available for free access by the praxis must have an entry in file *PicOS/MSP430/pins_sys.h* (*PicOS/eCOG/pins_sys.h* for eCOG1). Here is a sample entry for a hypothetical MSP430 board:

```
#if TARGET_BOARD == BOARD_PROTOTYPE_A
// Departures from default pre-initialization
#define PIN_DEFAULT_P1DIR  0x80
#define PIN_DEFAULT_P3DIR  0xCF
#define PIN_DEFAULT_P5DIR  0xE0
#define PIN_DEFAULT_P6DIR  0x00

#define PINS_BOARD_FOUND   1

#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 (P2, 0),   \
        PIN_RESERVED,      \
        PIN_DEF (P2, 2),   \
        PIN_DEF (P2, 6),   \
        PIN_DEF (P4, 7),   \
};

#define PIN_MAX            13     // Number of pins
#define PIN_MAX_ANALOG     8      // Number of available analog pins
#define PIN_DAC_PINS       0x0706 // Two DAC pins: #6 and #7
#define adc_config_rssi    adc_disable
#endif /* BOARD_PROTOTYPE_A */
```

The first four *defines* specify non-standard initialization for GPIO pins. By default, the system will initialize all ports not explicitly mentioned to “no special function”, direction “out”, value 0. This is the recommended setting of an unused pin to minimize power leaks. In this example, we see that the directions of certain ports are modified. This is not required from the viewpoint of GPIO availability to the praxis, but perhaps there are some modules connected to those pins that expect this kind of setting.

The **PIN_LIST** specifies those pins that are to be available to the praxis via the high-level pin operations. The number of entries on that list is equal **PIN_MAX** and determines how those pins will be numbered logically, i.e., from 0 to **PIN_MAX**-1. Thus, for example, pin number 11 corresponds to P2.6 on MSP430. Note that there may be holes in the list

marked with PIN_RESERVED. They may make sense if for some reason the logical numbering of pins must coincide with some physical property of the board, like the layout of a specific connector, but some of those pins are not available for general access.

If any pins are to be accessible for ADC, they should be included in the front of the list. **PIN_MAX_ANALOG** says how many pins from the front will be usable (in principle) for ADC access. As MSP430 uses P6 for this purpose, it makes sense to start the list from the available pins of P6.

At most two pins can be also used for DAC (e.g., as available on MSP430F1611). If so, they are described by PIN_DAC_PINS. The least significant byte of this value specifies the logical pin number of the pin connected to DAC0, while the more significant byte describes DAC1. This fits the fact that on MSP430F1611 DAC0 is assigned to P6.6 while DAC1 is connected to P6.7. Note that those declarations do not imply that the indicated pins will have to be permanently assigned to the DAC, only that the praxis will be able to use them for that purpose at all.

On some systems, the RF module may want to use the on-chip ADC facility for calculating the numerical value of RSSI (Received Signal Strength Indicator). In such a case, **adc_config_rssi** should be set to the sequence of instructions enabling ADC for this role. If this function is not needed, the symbol should be set to adc_disable. This is because **adc_config_rssi** is executed by the system whenever the high-level pin interface relinquishes the ADC. If it is being used for RSSI calculation, the functions will only borrow it and then return when done. Note, however, that the receiver of such an RF module cannot be active simultaneously with an ADC pin access.

3. Basic operations

The functions listed in this section become available to the praxis with the inclusion of pinopts.h in the program header. As we said before, they require that the board has a description of its pin layout in *PicOS/MSP430/pins_sys.h*.

word pin_write (word pin, word value)

The function sets the value and/or the status of the pin whose logical number is specified 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.

If bit number 2, i.e., 0x04, is set in **value**, the pin is set to ADC or DAC function, depending on whether bit 1 (0x02) is set (ADC) or not (DAC). The 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 demanded 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 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).[2] 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 **NULL**, the function will release the current process and it wake up in the indicated state when it should be re-executed to return the sample value. If state is **NULL**, the function spins internally until the conversion id done. Here is how the function should be used:

```
...
entry (TAKE_SAMPLE)
    val = pin_read_adc (TAKE_SAMPLE, 4, 2, 1);
    ... sample available ...
```

Normally, the function returns a nonnegative integer number between 0 and 4095 inclusively, which is a linear transformation of the input voltage between 0 and 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; however, ADC is being used by the RF module for sampling RSSI. 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)

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 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 such 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 (described in *PicOS/MSP430/pins_sys.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 description in *pins_sys.h* formally allows for this.

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