CPE 323
MODULE 08
MSP430 Parallel Ports

Aleksandar Milenković

Email: milenka@uah.edu

Web: http://www.ece.uah.edu/~milenka

Overview

This module discusses standard digital input/output through parallel ports. MSP430 family of microcontrollers interfaces the outside world through parallel ports that can serve as digital inputs or outputs or special-function pins (e.g., analog signal input for ADC converter, analog output from DAC, a data receive line for serial communication). You will learn organization of parallel ports and their software aspects - how to initialize and utilize them in embedded software.

Objectives

Upon completion of this module learners will be able to:

- *Recognize hardware and software aspects of parallel ports for digital input/output*
- *Configure parallel ports (direction, special functions through I/O multiplexing)*
- *Utilize parallel ports to interface common electronic components (LEDs, switches, buttons, and others)*

Contents

1 Introduction

MSP430 includes a number of 8-bit parallel ports. Depending on the device type, it can have as little as 2 parallel ports P1 and P2 and as many as 12 parallel ports (P1 to P11 and PJ). Most ports contain 8 bits that are connected to 8 I/O lines through chip pins, but some ports can have less than 8 bits/pins. Each digital I/O line is individually configurable for input or output direction, and each can be individually read or written.

A programmer's view of a generic parallel port with index x, Px, is shown in Figure 1. All parallel ports contain the following registers:

- PxDIR – direction register. Each bit in PxDIR selects the direction of the corresponding I/O pin, regardless of the selected function for the pin. PxDIR bits for I/O pins that are selected for other functions must be set as required by the other function. When a PxDIR bit is cleared to a logic 0, the corresponding I/O pin is switched to the input direction. When the bit is set to a logic 1, the corresponding I/O pin is switched to the output direction. Upon power up all bits in the direction register are cleared, and thus all I/O pins switched to the input direction.
- PxIN – input register. Each bit of PxIN reflects the value of the input signal at the corresponding I/O pin when the pin is configured for I/O function. The register is read-only. A bit with a logic value '0' indicates that input is low, a logic '1' indicates that input is high.
- PxOUT – output register. Each bit of PxOUT register contains the value to be output on the corresponding I/O pin, if it is configured as digital I/O, output direction. A logic '0' will output a low voltage (GND) and a logic '1' will output a high voltage (~V_{DD}).
- PxSEL – function select register. Port pins are often multiplexed with other peripheral module functions. Each bit in PxSEL is used to select the pin function – digital I/O function (the PxSEL bit is at a logic '0') or a peripheral module function (the PxSEL bit is at a logic '1'). Please note that setting a PxSEL bit to a logic '1' does not automatically set the pin direction - sometimes additional actions may be required (e.g., setting PxDIR bits) to fully enable peripheral functions. Recommendation is that you always consult technical documents when working with a specific device.

In addition to these registers, ports P1 and P2 support interrupts and have an additional set of registers to support this capability as discussed below.

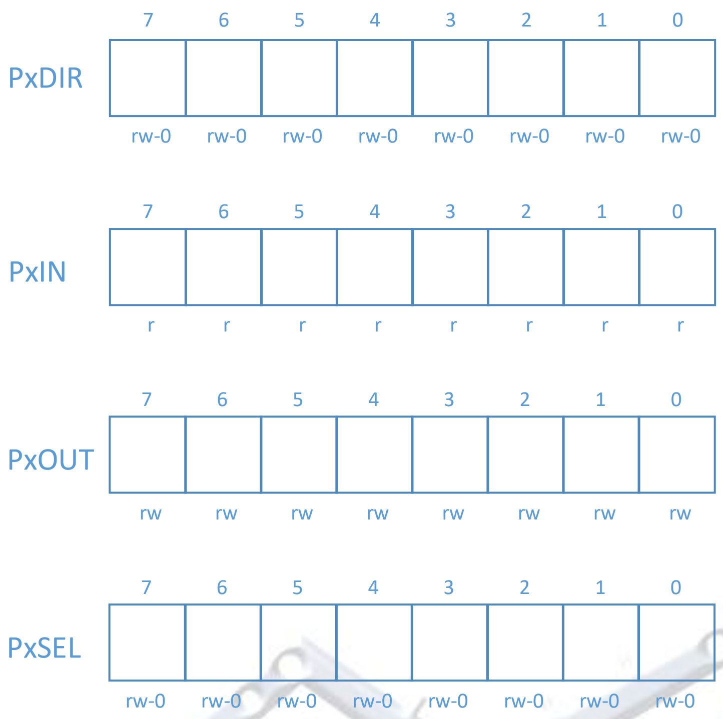


Figure 1. Programmer's view of generic MSP430 parallel port.

2 Port Interrupts

Parallel ports P1 and P2 are special and each pin in these ports has interrupt capability, configured with registers PxIFG, PxIE, and PxIES, where x=1 or 2. Figure 2 shows the port-registers related to interrupts. Their function is described below.

- PxIFG – interrupt flag register. It keeps track of pending interrupts associated with port Px. Each PxIFG bit is the interrupt flag for the corresponding I/O pin and it is set when the selected input signal edge occurs on the pin. PxIFG bits can also be set from software (e.g., using BIS instruction). If the value of a PxIFG bit is a logic '0', no interrupt is pending. Otherwise, an interrupt is pending. Only transitions, not static levels cause interrupts.
- PxIE – interrupt enable register. It allows for selective masking of port interrupts. Each PxIE bit enables the associated PxIFG interrupt flag. A PxIE bit at a logic '0' disables the interrupt for the corresponding PxIFG bit. To enable it, the PxIE bit should be set to a logic '1'.
- PxIES – interrupt edge select register. It allows for specifying the interrupt edge for the corresponding I/O pin. Clearing a bit in PxIES means that a low-to-high transition on the corresponding I/O pin will set the corresponding PxIFG bit. Setting a bit in PxIES to a logic '1' means that a high-to-low transition on the corresponding I/O pin will set the corresponding PxIFG bit.
  Note: please note that changing PxIES in software may trigger setting PxIFG bits

depending on the current state of PxIN lines. For example, changing a bit in PxIES from 0 to 1 when corresponding input bit PxIN is 0 will set the corresponding PxIFG bit. For this boundary conditions check the user manual section discussing interrupt capability of parallel ports P1 and P2.

All interrupt requests from P1 share a single interrupt service routine (entry 47 in MSP430F5529). All interrupt requests from P2 share a single interrupt service routine as well (entry 41 in MSP430F5529). Thus, we have two ISRs, one for P1 and one for P2. These two ISRs are examples of a multi-source interrupt service routine. That means that individual PxIFG bits are not cleared automatically during exception processing, but rather software developers should clear them inside respective ISRs upon servicing the corresponding requests.

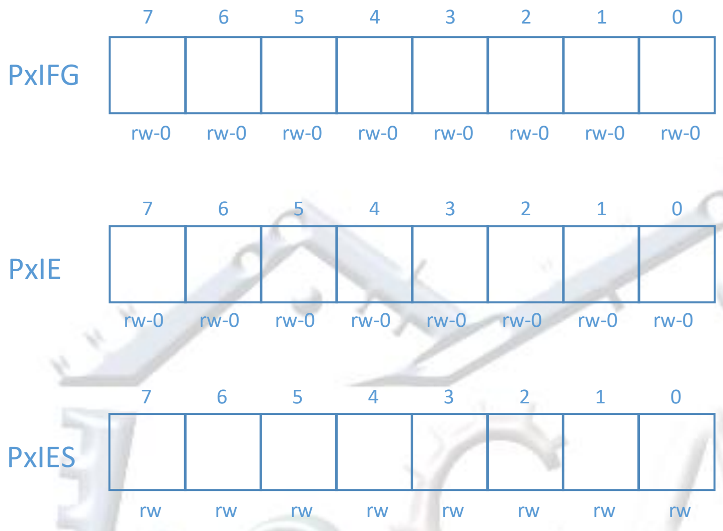


Figure 2. Programmer's view of P1 and P2 port registers related to interrupts.

In case that multiple interrupt request lines connects to a single port, the corresponding ISR needs to examine individual bits in the PxIFG register to determine which requests are pending. It is developer's choice whether to process all pending requests once inside the ISR or to serve the just one of them (highest priority one, where the priority is determined by the developer and then exit the ISR. It is also up to the developer to choose in what order these requests are examined (thus determining priority). The way the software is written will impact the order in which requests are served. Again, clearing the corresponding IFG bit explicitly inside the ISR upon servicing the request is very important. Failing to do so, will trigger re-entering the ISR and this will repeat forever and your software will be stuck.

This process or determining which source is responsible for being inside the ISR can add to the overhead and delay the response to interrupts. To speed up interrupt processing inside ISRs, many peripherals that have multiple interrupt triggers sharing a single interrupt service routine have an additional register, so-called Interrupt Vector or IV register. MSP430F5529 includes P1IV and and P2IV registers as shown in Figure 3. Note: MSP430FG4618 does not have P1IV and P2IV registers.

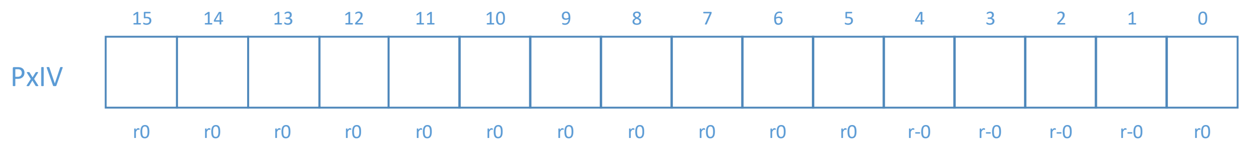


Figure 3. Programmer's view of P1IV and P2IV registers.

We will consider P1IV. It is a 16-bit register, though only bits 4-1 contain an identifier that corresponds to the highest pending interrupt request and all other bits are always 0. This register is read-only and its content is set by the control logic inside P1 peripheral, as follows:

- 0x00 – No interrupt is pending
- 0x02 – P1.0 interrupt is pending (P1IFG.BIT0 is set), highest priority
- 0x04 – P1.1 interrupt is pending (P1IFG.BIT1 is set)
- 0x06 – P1.2 interrupt is pending (P1IFG.BIT2 is set)
- 0x08 – P1.3 interrupt is pending (P1IFG.BIT3 is set)
- 0x0A – P1.4 interrupt is pending (P1IFG.BIT4 is set)
- 0x0C – P1.5 interrupt is pending (P1IFG.BIT5 is set)
- 0x0E – P1.6 interrupt is pending (P1IFG.BIT6 is set)
- 0x10 – P1.7 interrupt is pending (P1IFG.BIT7 is set), lowest priority

Any access (read or write) of the lower byte of the P1IV register, either word or byte access, automatically resets the highest pending interrupt flag. If another interrupt flag is set, another interrupt is immediately generated after servicing the initial interrupt. P1IFG.BIT0 has the highest priority and P1IFG.BIT7 lowest.

For example, assume the P1IFG.BIT0 and P1IFG.BIT2 flags are set. The P1 control logic will set P1IV to 0x02 (value that corresponds to the highest-priority pending interrupt). When the interrupt service routine is entered and the P1IV register is accessed, P1IFG.BIT0 is reset automatically and the register P1IV gets the value that corresponds to the currently highest priority pending interrupt. In this case it is P1IFG.BIT2 or value 0x06. After the RETI instruction of the interrupt service routine is executed, the P1IFG.BIT2 generates another interrupt. The port P2 interrupts behave similarly – share a single interrupt vector and utilize the P2IV register.

The code in Code 1 shows the recommended use of P1IV and a template for the P1_ISR. The first instruction in the handler adds the content of P1IV to the PC. Below it there is a sequence of JMP instructions that will take us immediately to the portion of the ISR where the currently highest-priority interrupt will be handled. This sequence of JMP instructions is called a jump table. This table explains why the P1IV register contains specific values as discussed above. Please note that RETI and JMP instructions are 2 bytes long.

For example, assume that currently P1IFG.BIT2 is set – the content of P1IV will be 0x06. Assuming that P1IE.BIT2 is at a logic '1' and GIE bit is at a logic '1' and no other interrupts are pending, the CPU accepts the interrupt and we enter P1_ISR. When executing the first instruction in line 3, `ADD &P1IV, PC`, the current PC is pointing to the the next instruction `RETI` in line 4. Adding the content of PI1IV (currently 6) to PC will move PC to point to the instruction in line 7, `JMP P1_2_HND`, and this instruction will take us to the portion of the handler where this request is serviced. Please note that ADD instruction reading P1IV will trigger a hardware mechanism that clears P1IFG.BIT2, so we do not have to do it explicitly in software.

The numbers at the right margin show the necessary CPU cycles for each instruction. The software overhead for different interrupt sources includes interrupt latency and return-from-interrupt cycles, but not the task handling itself. This way we have a fixed latency for reaching the portion of the code that processes the request, regardless of which interrupt is requested: 6+3+2=11 clock cycles. This is another example how combining hardware (P1IV and logic managing its content) with corresponding software can improve performance and predictability of interrupt responses.

```
1  ;Interrupt handler for P1                 Cycles
2  P1_HND ...     ; Interrupt latency          6
3  ADD &P1IV,PC ; Add offset to Jump table     3
4  RETI ; Vector 0: No interrupt               5
5  JMP P1_0_HND ; Vector 2: Port 1 bit 0       2
6  JMP P1_1_HND ; Vector 4: Port 1 bit 1       2
7  JMP P1_2_HND ; Vector 6: Port 1 bit 2       2
8  JMP P1_3_HND ; Vector 8: Port 1 bit 3       2
9  JMP P1_4_HND ; Vector 10: Port 1 bit 4      2
10 JMP P1_5_HND ; Vector 12: Port 1 bit 5      2
11 JMP P1_6_HND ; Vector 14: Port 1 bit 6      2
12 JMP P1_7_HND ; Vector 16: Port 1 bit 7      2
13 P1_7_HND ; Vector 16: Port 1 bit 7
14 ... ; Task starts here
15 RETI ; Back to main program                 5
16 P1_6_HND ; Vector 14: Port 1 bit 6
17 ... ; Task starts here
18 RETI ; Back to main program                 5
19 P1_5_HND ; Vector 12: Port 1 bit 5
20 ... ; Task starts here
21 RETI ; Back to main program                 5
22 P1_4_HND ; Vector 10: Port 1 bit 4
23 ... ; Task starts here
24 RETI ; Back to main program               5
25 P1_3_HND ; Vector 8: Port 1 bit 3
26 ... ; Task starts here
27 RETI ; Back to main program               5
28 P1_2_HND ; Vector 6: Port 1 bit 2
29 ... ; Task starts here
30 RETI ; Back to main program               5
31 P1_1_HND ; Vector 4: Port 1 bit 1
32 ... ; Task starts here
33 RETI ; Back to main program               5
34 P1_0_HND ; Vector 2: Port 1 bit 0
35 ... ; Task starts here
36 RETI ; Back to main program               5
37
```

Code 1. Template for P1_ISR using P1IV register.

This approach to writing interrupt service routines is common in many other peripherals (e.g., TimerA, TimerB, ADC12, DMA) that have an interrupt vector register. Please note that with this approach multiple pending requests on P1 will result in executing P1_ISR multiple times, once per each request.

Code 2 shows a modified variant of the program interfacing switch S1 discussed in the Module 06. The program utilizes P2_ISR to turn on LED1 when S1 is pressed; the LED1 is kept on as long as S1 is pressed. This implementation utilizes the template from Code 1. Here we have only one interrupt enables (P2_1). At the beginning of the handler the content of P2IV is added to PC. The jump table has only one active entry in line 64 that leads us to the portion of the program that turns on LED1 and keep checking its status. Please note that here we do not need to explicitly clear P2IFG, bit 1 because it is automatically clear by a read from from the register P2IV (line 61).

```
 1  ;-------------------------------------------------------------------------------
 2  ;   File:        Lab6_D3_IVT.asm
 3  ;   Description: The program demonstrates Press/Release using S1 and LED1.
 4  ;                LED1 is initialized off. The main program enables interrupts
 5  ;                from P2.BIT1 (S1) and remains in an infinite loop doing nothing.
 6  ;                P2_ISR implements keeps LED1 on until S1 is released.
 7  ;
 8  ;   Clocks:      ACLK = 32.768kHz, MCLK = SMCLK = default DCO = 2^20=1,048,576 Hz
 9  ;   Platform:    TI EXP430F5529LP Launchpad
10  ;
11  ;                 MSP430F5529
12  ;             -----------------
13  ;         /|\|                 |
14  ;          | |                 |
15  ;          --|RST              |
16  ;            |             P1.0|-->LED1(RED)
17  ;            |             P2.1|<--S1
18  ;
19  ;   Author:     Aleksandar Milenkovic, milenkovic@computer.org
20  ;   Date:       September 25, 2022
21  ;-------------------------------------------------------------------------------
22              .cdecls C,LIST,"msp430.h"       ; Include device header file
23
24  ;-------------------------------------------------------------------------------
25              .def    RESET                   ; Export program entry-point to
26                                              ; make it known to linker.
27              .def    P2_ISR
28  ;-------------------------------------------------------------------------------
29              .text                           ; Assemble into program memory.
30              .retain                         ; Override ELF conditional linking
31                                              ; and retain current section.
32              .retainrefs                     ; And retain any sections that have
33                                              ; references to current section.
34
35  ;-------------------------------------------------------------------------------
36  RESET:      mov.w   #__STACK_END, SP        ; Initialize stack pointer
37  StopWDT:    mov.w   #WDTPW|WDTHOLD, &WDTCTL ; Stop watchdog timer
38  ;-------------------------------------------------------------------------------
```

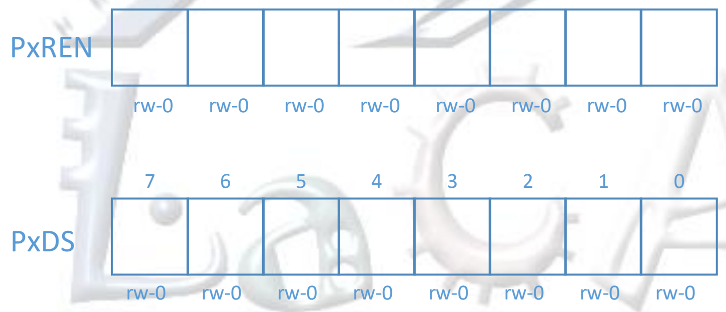
```
Setup:
             bis.b   #001h, &P1DIR          ; Set P1.0 to output
                                            ; direction (0000_0001)
             bic.b   #001h, &P1OUT          ; Set P1OUT to 0x0000_0001

             bic.b   #002h, &P2DIR          ; SET P2.1 as input for S1
             bis.b   #002h, &P2REN          ; Enable Pull-Up resister at P2.1
             bis.b   #002h, &P2OUT          ; required for proper IO set up

             nop
             bis.w   #GIE, SR               ; Enable Global Interrupts
             nop
             bis.b   #002h, &P2IE           ; Enable Port 2 interrupt from bit 1
             bis.b   #002h, &P2IES          ; Set interrupt to call from hi to low
             bic.b   #002h, &P2IFG          ; Clear interrupt flag
InfLoop:
             jmp     $                      ; Loop here until interrupt

;----------------------------------------------------------------------------
; P2_1 (S1) interrupt service routine (ISR)
;----------------------------------------------------------------------------
P2_ISR:
             add.w   &P2IV, PC              ; Clear interrupt flag
             reti                           ; no interrupt is pending
             reti                           ; P2_0 (port 2 bit 0)
             jmp     P2_1_HND               ; P2_1 (port 2 bit 1, switch 1)
             reti                           ; P2_2
             reti                           ; P2_3
             reti                           ; P2_4
             reti                           ; P2_5
             reti                           ; P2_6
             reti                           ; P2_7
P2_1_HND:    bis.b   #001h,&P1OUT           ; Turn on LED1
SW2wait:     bit.b   #002h,&P2IN            ; Test S1
             jz      SW2wait                ; Wait until S1 is released
             bic.b   #001,&P1OUT            ; Turn off LED1
LExit:       reti                           ; Return from interrupt
;-------------------------------------------------------------------------------
; Stack Pointer definition
;-------------------------------------------------------------------------------
             .global __STACK_END
             .sect   .stack

;-------------------------------------------------------------------------------
; Interrupt Vectors
;-------------------------------------------------------------------------------
             .sect   ".reset"        ; MSP430 RESET Vector
             .short  RESET
             .sect   ".int42"        ; PORT1_VECTOR,
             .short  P2_ISR         ; please check the MSP430F5529.h header file
             .end
```

Code 2. Assembly program interfacing switch S1 using P2_ISR that utilizes P2IVT register.

3 Additional Parallel Port Registers

In MSP430F5529 each I/O line can also be individually configurable for pullup or pulldown resistors. In addition, each line can be configurable for drive strength, full or reduced. MSP430F5529 includes additional parallel port registers PxREN and PxDS for dealing with pullup/pulldown resistors and drive strength. Figure 4 shows programmer's view of these registers. Their functionality is described below.

- PxREN – pullup or pulldown register enable. When a respective I/O pin is configured as input, setting its corresponding PxREN bit will enable the pullup or pulldown. That is, a logic '0' means that pullup/pulldown resistor is disabled (default), a logic '1' means that the pullup/pulldown register is enabled. The corresponding bit in the PxOUT register selects if the pin contains a pulldown ('0') or pullup ('1'). Please note, these settings take effect only if the corresponding PxDIR bit is cleared (input mode).
- PxDS – output drive strength register. It allows for controlling Px drive strength. A logic '0' corresponds to reduced output drive strength and a logic '1' to full output drive strength. The default state on power-up is 0x00.



PxREN
rw-0 rw-0 rw-0 rw-0 rw-0 rw-0 rw-0 rw-0
7 6 5 4 3 2 1 0
PxDS
rw-0 rw-0 rw-0 rw-0 rw-0 rw-0 rw-0 rw-0

Figure 4. Programmer's view of PxREN and PxDS registers.

4 Address Mapping of Port Registers

We have discussed parallel ports as 8-bit peripherals. When dealing with port registers we thus use .B instructions. However, neighboring parallel ports can be paired up to create 16-bit logical ports that can be accessed using .W machine instructions. This is achieved by smart placement of registers in the address space. For example, the P1IN register is at the address 0x0000 and the register P2IN is at the address 0x0001. Together we can refer to both as PAIN using a word instruction. Here PA stands for port A and PAIN stands for port A input register, where P1IN represents lower 8 bits and P2IN upper 8 bits of port A. Similarly, we can refer to P1IN as PAIN_L (lower byte or register PA) and to P2IN as PAIN_H (higher byte of register PA). This applies to other registers in paired parallel ports (PxDIR, PxOUT, PxSEL, …). Table 1 shows pairings of parallel ports.

Table 1. Parallel Ports Pairings

8-bit Parallel Ports	16-bit Parallel Ports
P2 | P1	PA
P4 | P3	PB
P6 | P5	PC
P8 | P7	PD
P10 | P9	PE
P12 | P11	PF

5 Internal Port Organization

In this section we will discuss hardware implementation of one I/O lane inside the port P1. Other port I/O lanes have similar implementation and will not be discussed here. For details refer to the device-specific reference manuals.

Figure 5 shows a diagram of one I/O lane inside P1. First, let us discuss digital I/O function (assume P1DIR.BITx is either 1 or 0, P1REN.BITx=0, P1SEL.BITx=0, no interrupts are used). If the P1DIR.BITx=1 (output), P1OUT.BITx goes through the buffer controlled by P1DS.BITx providing low (P1DS.BITx=0) or high (P1DS.BITx=1) drive, to the port pin. This corresponds to digital I/O output mode. Next, let us assume that P1DIR.BITx=0 (input). The signal from the I/O pin goes through the Schmitt trigger to P1IN.BITx. If P1REN.BITx=1, depending on P1DIR.BITx, we will have either pullup or pulldown configuration on the input. If P1SEL.BITx is selected, the signal from/to module is routed to the port pin.

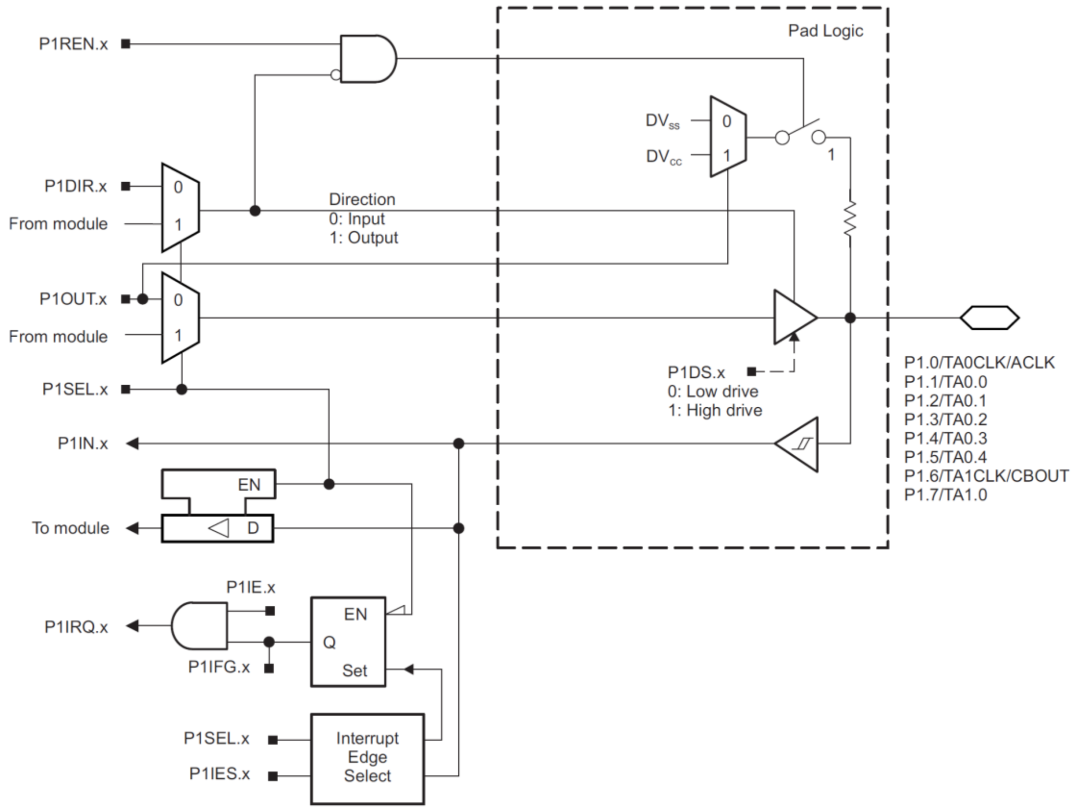


Figure 5. Hardware implementation of an I/O line in P1.