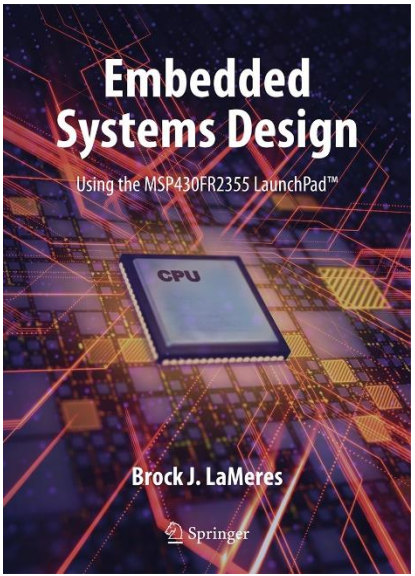
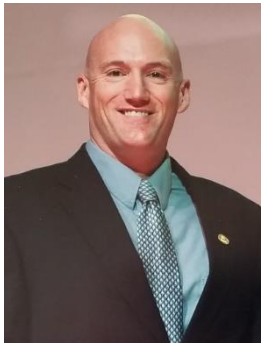
Embedded Systems Design

Chapter 9: Digital I/O

9.1 The MSP430 Digital I/O System



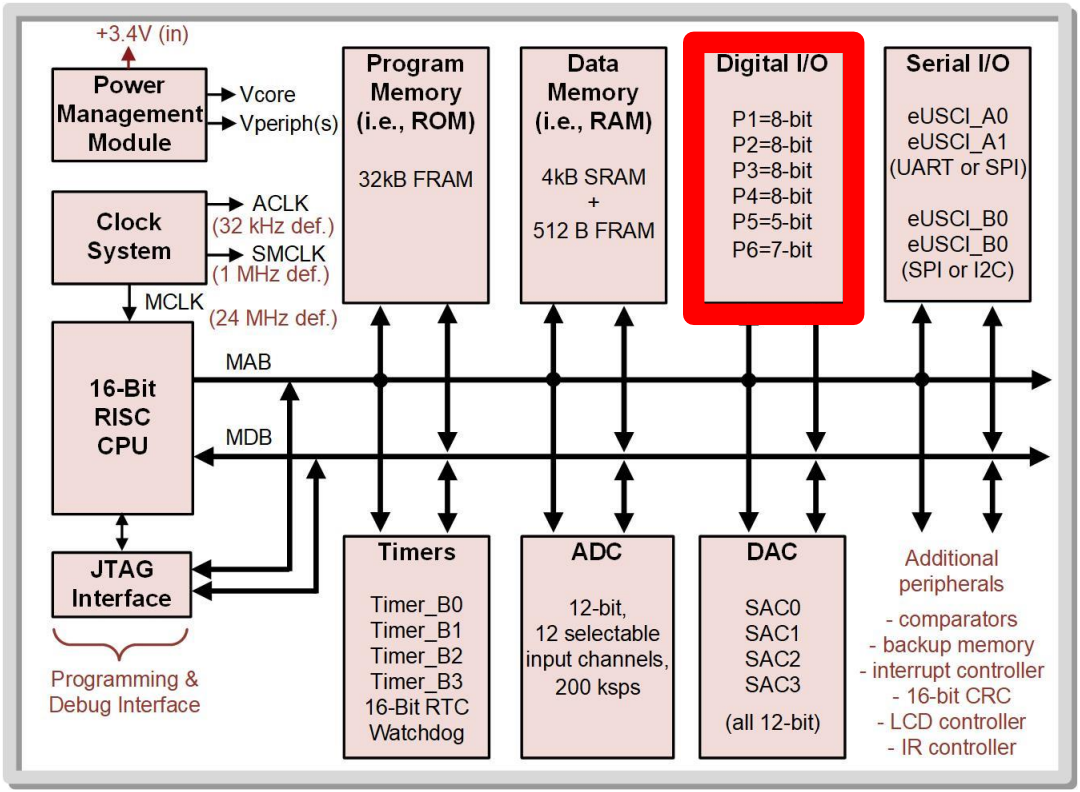


9.1 THE MSP430 DIGITAL I/O SYSTEM

Digital I/O
=
Parallel Ports

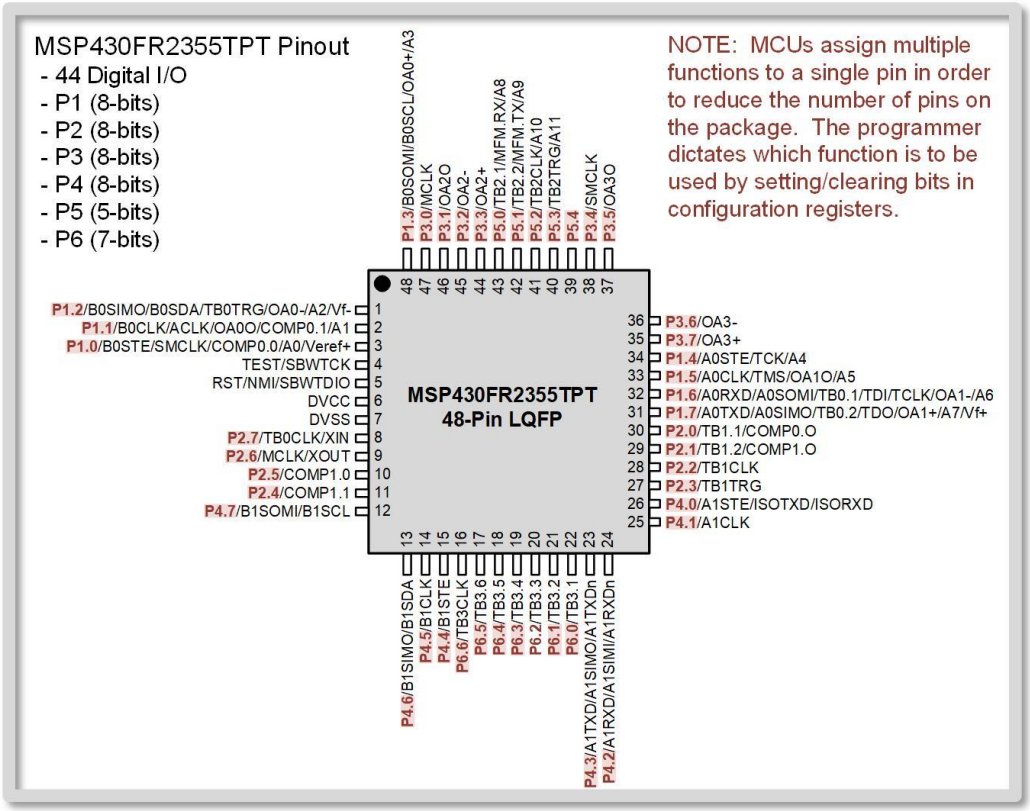
P1, P2, P3,
P4, P5, P6

PA=P1:P2
PB=P3:P4
PC=P5:P6



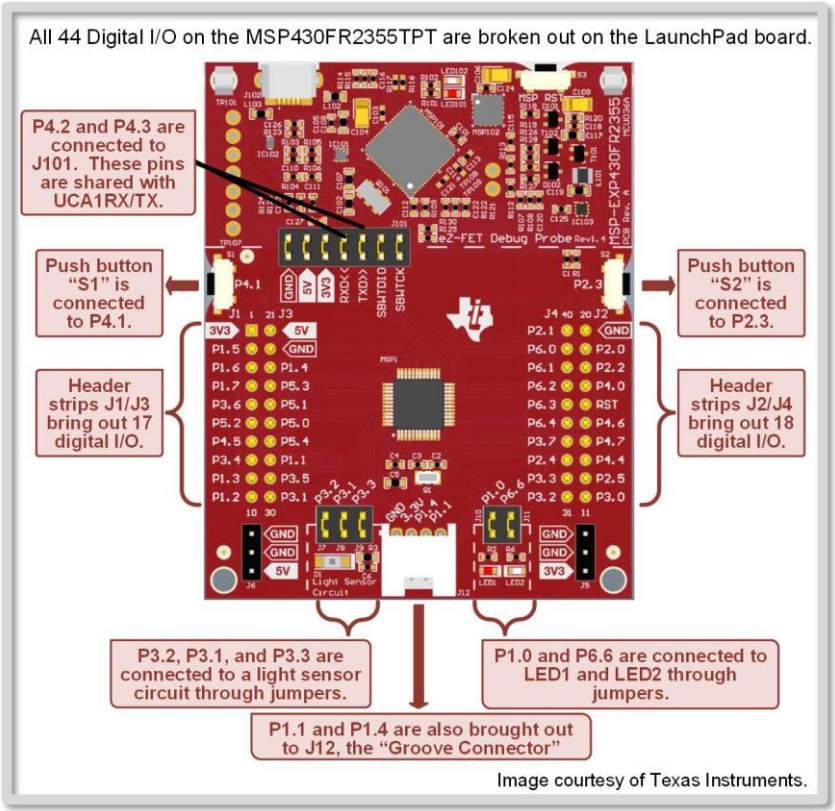
9.1 THE MSP430 DIGITAL I/O SYSTEM

Recall that the MCUs share functionality on pins in order to reduce the size and cost of the device.



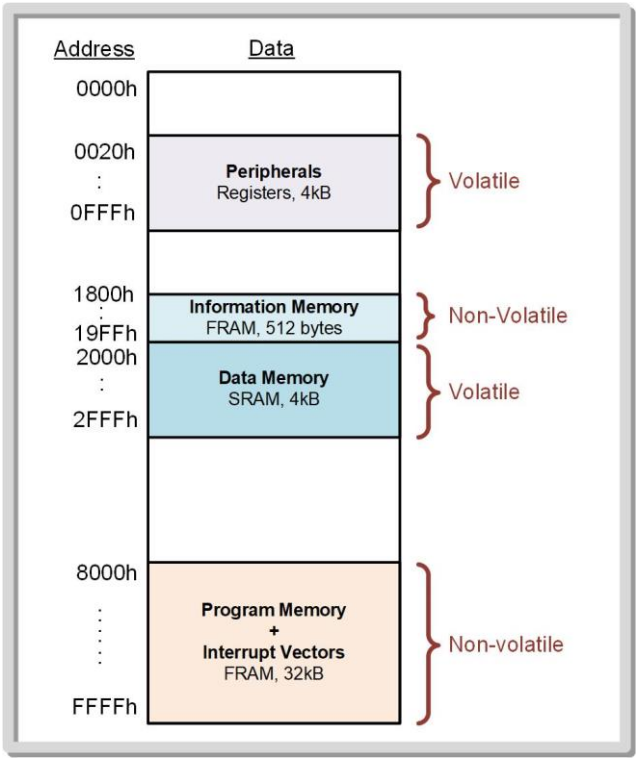
9.1 THE MSP430 DIGITAL I/O SYSTEM

This figure shows the details of the MCU digital I/O breakout on the LaunchpadTM



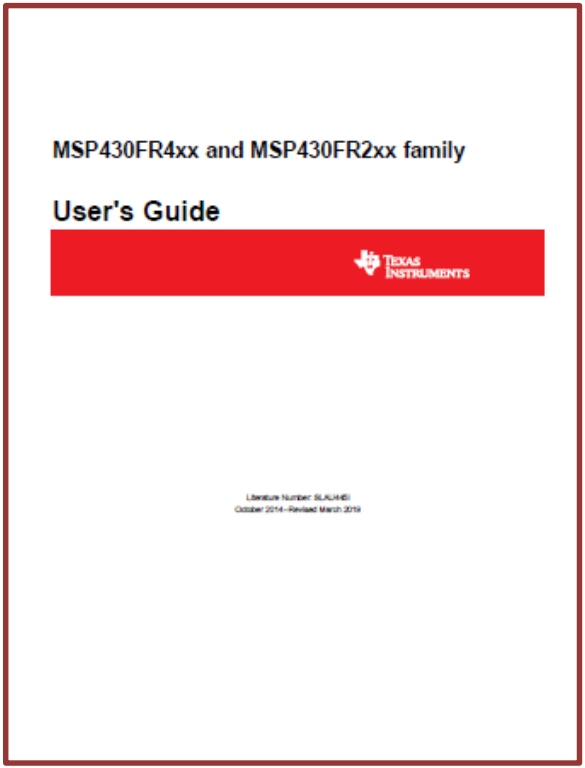
9.1.1 CONFIGURATION REGISTERS

- Peripherals are setup and interfaced with using registers within the memory map.
- Bit Set and Bit Clear instructions are used to configure the registers.

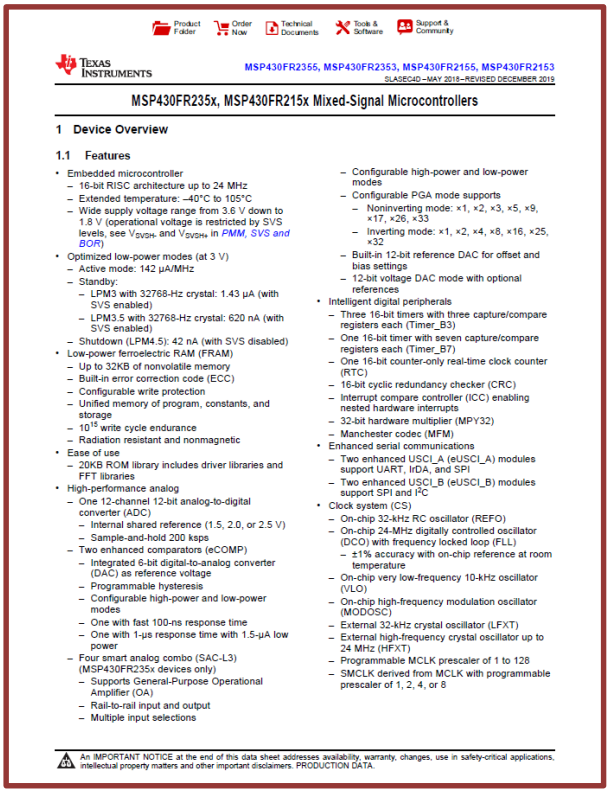


9.1.1 CONFIGURATION REGISTERS – DOCUMENTATION

- The **MSP430FRxx Family User's Guide** gives the name and purpose of each register.



9.1.1 CONFIGURATION REGISTERS – DOCUMENTATION



Product Folder | Order Now | Technical Documents | Tools & Software | Support & Community

TEXAS INSTRUMENTS

MSP430FR2355, MSP430FR2353, MSP430FR2155, MSP430FR2153

SLASEC4D – MAY 2018 – REVISED DECEMBER 2019

MSP430FR235x, MSP430FR215x Mixed-Signal Microcontrollers

1 Device Overview

1.1 Features

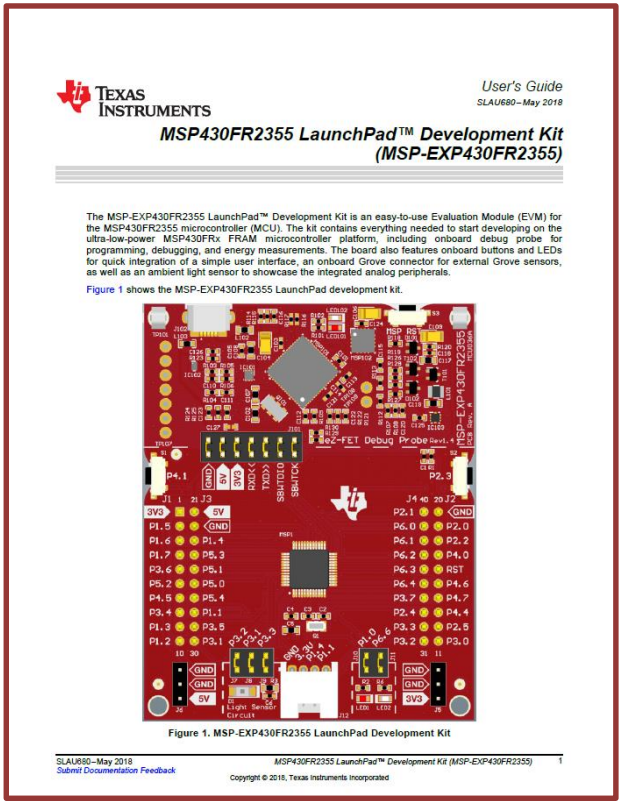
- Embedded microcontroller
  - 16-bit RISC architecture up to 24 MHz
  - Extended temperature: –40°C to 105°C
  - Wide supply voltage range from 3.6 V down to 1.8 V (operational voltage is restricted by SVS levels, see V_{SVSH-} and V_{SVSH+} in *PMM, SVS and BOR*)
- Optimized low-power modes (at 3 V)
  - Active mode: 142 µA/MHz
  - Standby:
    - LPM3 with 32768-Hz crystal: 1.43 µA (with SVS enabled)
    - LPM3.5 with 32768-Hz crystal: 620 nA (with SVS enabled)
  - Shutdown (LPM4.5): 42 nA (with SVS disabled)
- Low-power ferroelectric RAM (FRAM)
  - Up to 32KB of nonvolatile memory
  - Built-in error correction code (ECC)
  - Configurable write protection
  - Unified memory of program, constants, and storage
  - 10^{15} write cycle endurance
  - Radiation resistant and nonmagnetic
- Ease of use
  - 20KB ROM library includes driver libraries and FFT libraries
- High-performance analog
  - One 12-channel 12-bit analog-to-digital converter (ADC)
    - Internal shared reference (1.5, 2.0, or 2.5 V)
    - Sample-and-hold 200 ksps
  - Two enhanced comparators (eCOMP)
    - Integrated 6-bit digital-to-analog converter (DAC) as reference voltage
    - Programmable hysteresis
    - Configurable high-power and low-power modes
    - One with fast 100-ns response time
    - One with 1-µs response time with 1.5-µA low power
  - Four smart analog combo (SAC-L3) (MSP430FR235x devices only)
    - Supports General-Purpose Operational Amplifier (OA)
    - Rail-to-rail input and output
    - Multiple input selections
    - Configurable high-power and low-power modes
    - Configurable PGA mode supports
      - Noninverting mode: ×1, ×2, ×3, ×5, ×9, ×17, ×26, ×33
      - Inverting mode: ×1, ×2, ×4, ×8, ×16, ×25, ×32
    - Built-in 12-bit reference DAC for offset and bias settings
    - 12-bit voltage DAC mode with optional references
- Intelligent digital peripherals
  - Three 16-bit timers with three capture/compare registers each (Timer_B3)
  - One 16-bit timer with seven capture/compare registers each (Timer_B7)
  - One 16-bit counter-only real-time clock counter (RTC)
  - 16-bit cyclic redundancy checker (CRC)
  - Interrupt compare controller (ICC) enabling nested hardware interrupts
  - 32-bit hardware multiplier (MPY32)
  - Manchester codec (MFM)
- Enhanced serial communications
  - Two enhanced USCI_A (eUSCI_A) modules support UART, IrDA, and SPI
  - Two enhanced USCI_B (eUSCI_B) modules support SPI and I^2C
- Clock system (CS)
  - On-chip 32-kHz RC oscillator (REFO)
  - On-chip 24-MHz digitally controlled oscillator (DCO) with frequency locked loop (FLL)
    - ±1% accuracy with on-chip reference at room temperature
  - On-chip very low-frequency 10-kHz oscillator (VLO)
  - On-chip high-frequency modulation oscillator (MODOSC)
  - External 32-kHz crystal oscillator (LFXT)
  - External high-frequency crystal oscillator up to 24 MHz (HFXT)
  - Programmable MCLK prescaler of 1 to 128
  - SMCLK derived from MCLK with programmable prescaler of 1, 2, 4, or 8

An IMPORTANT NOTICE at the end of this data sheet addresses availability, warranty, changes, use in safety-critical applications, intellectual property matters and other important disclaimers. PRODUCTION DATA.

- The **MSP430 Device Specific Datasheet** gives the details of which peripherals are available, which pin they are on, and how to select the function.

9.1.1 CONFIGURATION REGISTERS – DOCUMENTATION

- The **LaunchPad Development Kit User's Guide** tells where the I/O are connected on the red board.



TEXAS INSTRUMENTS

User's Guide
SLAU680–May 2018

MSP430FR2355 LaunchPad™ Development Kit (MSP-EXP430FR2355)

The MSP-EXP430FR2355 LaunchPad™ Development Kit is an easy-to-use Evaluation Module (EVM) for the MSP430FR2355 microcontroller (MCU). The kit contains everything needed to start developing on the ultra-low-power MSP430FRx FRAM microcontroller platform, including onboard debug probe for programming, debugging, and energy measurements. The board also features onboard buttons and LEDs for quick integration of a simple user interface, an onboard Grove connector for external Grove sensors, as well as an ambient light sensor to showcase the integrated analog peripherals.

Figure 1 shows the MSP-EXP430FR2355 LaunchPad development kit.

Figure 1. MSP-EXP430FR2355 LaunchPad Development Kit

9.1.1 PORT DIRECTION REGISTERS (PXDIR)

- The **port direction registers** dictate whether the port bits are configured as inputs or outputs.
- The logic for PxDIR is as follows:
- **Bit = 0: Pin is an input (default)**
- **Bit = 1: Pin is an output**

9.1.2 PORT INPUT REGISTERS (PXIN)

- Each bit within PxIN registers represent the logic levels at the input signals pins.

Bit = 0: Logic low **Bit = 1: Logic high**

- **Read only** registers

- In order to *read* from an input port bit, a program can either move the information into a CPU register, or do bit compares on the PxIN memory location.

9.1.3 PORT OUTPUT REGISTERS (PXOUT)

- Each bit within the PxOUT registers is the value to be output to the port's signal pin when the bit is configured to be an output.
- In order to *write* to an output port bit, a program can move information into the PxOUT register, or perform bit set/clear operations on the PxOUT address location.
- When a port bit is configured as an input, the PxOUT register has a secondary role, which is to dictate the polarity of an optional pull-up/down resistor.

9.1.4 PORT PULLUP/PULLDOWN REGISTER ENABLE REGISTERS (PXREN)

- If a port is an input, an optional pull-up or pull-down resistor can be attached.
- The bits within PxREN control the corresponding bit location within PxIN.
- The resistor resides in the MCU.

Bit = 0: Disabled Bit = 1: Enabled

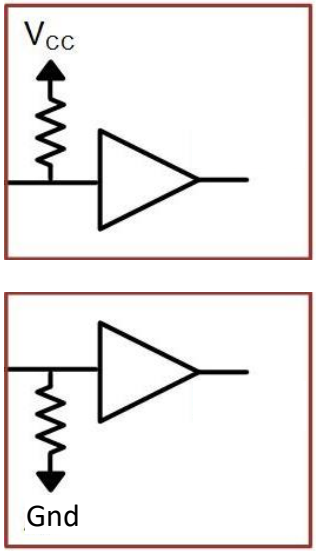
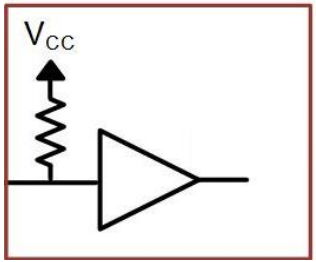


Image Courtesy of https://neodem.wp.horizon.ac.uk/

9.1.4 PORT PULLUP/PULLDOWN REGISTER ENABLE REGISTERS (PXREN)

- Pull-up and pull-down resistors are typically very large (10kΩ → 1MΩ) so that external circuitry can easily override them.

- But the pull-up resistors are “just” strong enough that when nothing is driving the pin, it will pull them to a known logic level (HIGH or LOW).



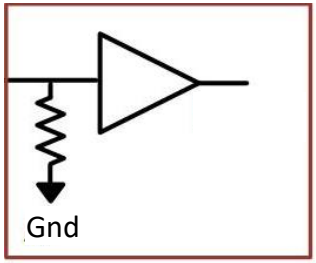


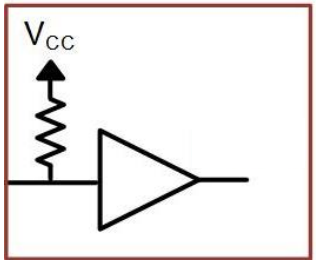
Image Courtesy of https://neodem.wp.horizon.ac.uk/

9.1.4 PORT PULLUP/PULLDOWN REGISTER ENABLE REGISTERS (PXREN)

- To configure whether the resistor is a pull **UP** or pull **DOWN**, the **PxOUT** register is used.

PxOUT = 0: Resistor is a Pull Down

PxOUT = 1: Resistor is a Pull Up



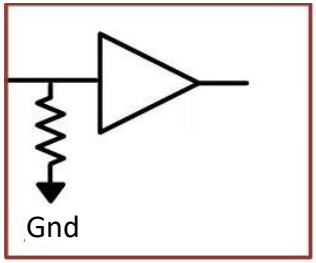
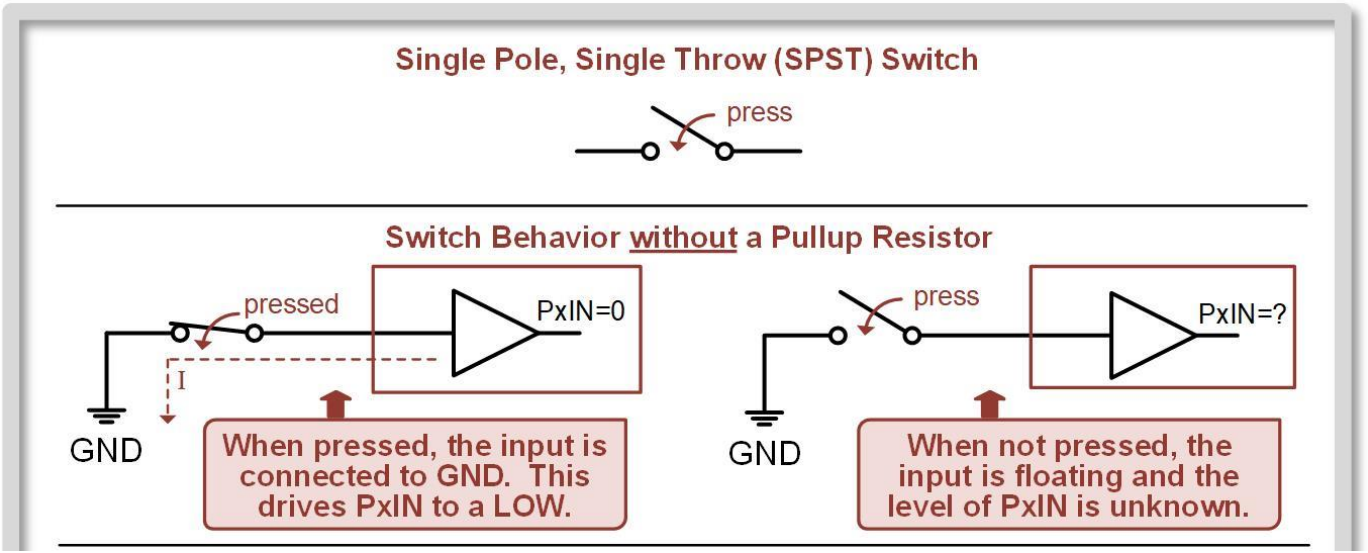


Image Courtesy of https://neodem.wp.horizon.ac.uk/

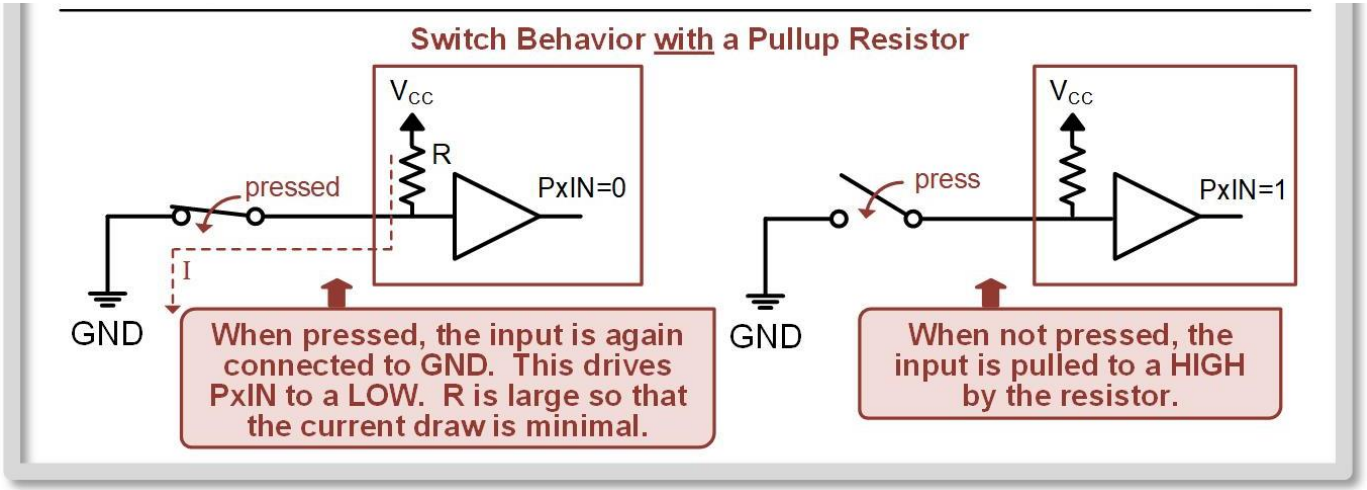
9.1.4 AN EXAMPLE OF USING THE PULL UP/DOWN RESISTORS

- **Single-pole, single-throw (SPST) switch/button** – simplest form of a switch, one input and one output
- When the switch is **open**, the input and output are not connected.
- When the switch is **closed**, the input and output are connected.



9.1.4 AN EXAMPLE OF USING THE PULL UP/DOWN RESISTORS

- Adding a pull-up resistor connected to the power supply can provide a logic high when the switch is open.
- This takes two steps:
  - PxREN = 1 (enable the resistor)
  - PxOUT = 1 (make it a pull **UP**)



9.1.5 PORT FUNCTION SELECT REGISTERS (PXSEL1 AND PXSEL0)

- The Port Function Select (PxSEL) registers tell the MCU which function to use, including whether to make the signal pin a digital input/output.
- PxSEL1 and PxSEL0 hold the two selection bit.
- The function is found in the **Device Specific Datasheet**

PxSEL1	PxSEL0	Function
0	0	Digital I/O (default)
0	1	Secondary Function
1	0	Tertiary Function
1	1	-

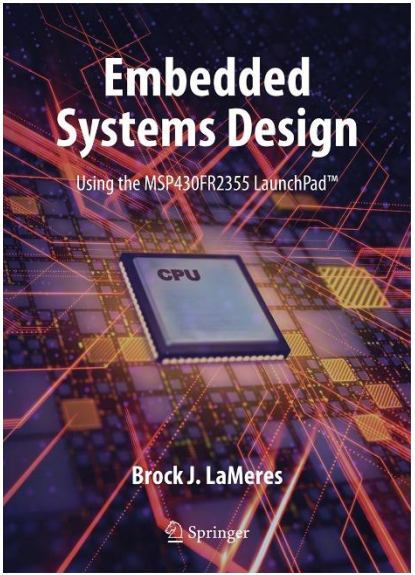
9.1.6 DIGITAL I/O ENABLING AFTER RESET

The steps to fully set up a digital I/O for use can be summarized in the following steps:

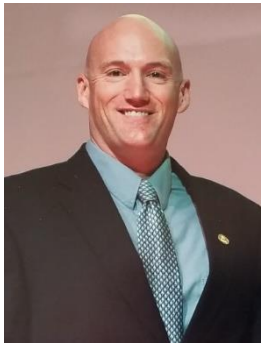
- Initialize Configuration Registers: PxDIR, PxREN, PxOUT if applicable, and PxSEL1: PxSEL0.
- Clear the **LOCKLPM5** bit in the **PM5CTL0** register (low power, high-Z input mode by default)
- Your program may now start using the PxIN or PxOUT register.

CHAPTER 9: DIGITAL I/O

9.1 THE MSP430 DIGITAL I/O SYSTEM



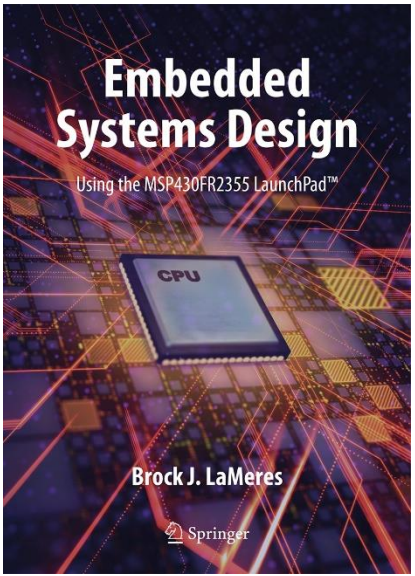
www.youtube.com/c/DigitalLogicProgramming_LaMeres

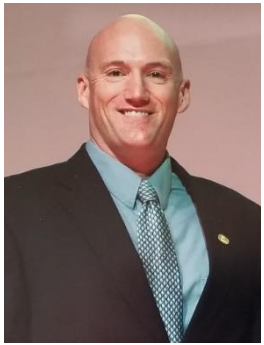


Embedded Systems Design

Chapter 9: Digital I/O

9.1 The MSP430 Digital I/O System – The MSP430.h Header





Brock J. LaMeres, Ph.D.

9.1.7 USING LITERAL DEFINITIONS FROM THE MSP430.H HEADER FILE

- To eliminate the need for the programmer to look up the absolute address of each register within the memory map, Code Composer Studio automatically includes a header file with literal names defined for the memory addresses.

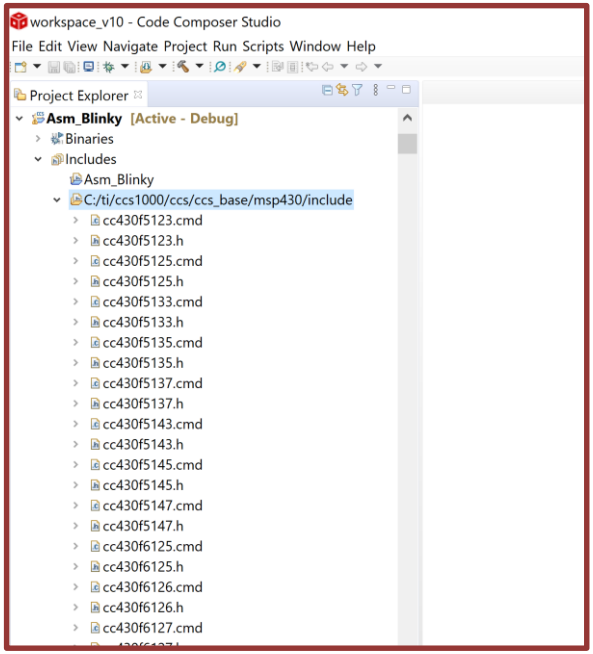
- Main header files we will use: **msp430.h** and **msp430fr2355.h**

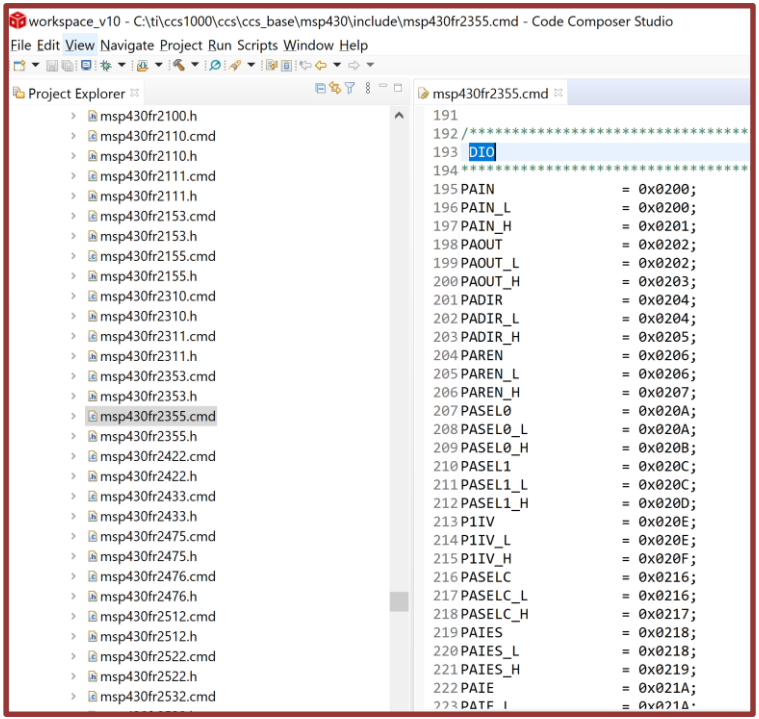
- These files contain the exact spelling of the register names and abbreviations of the memory map.

- These register names are NOT address labels. They are simply name substitutions. So we need to use absolute addressing:

i.e., bis.b #BIT0, &P1DIR

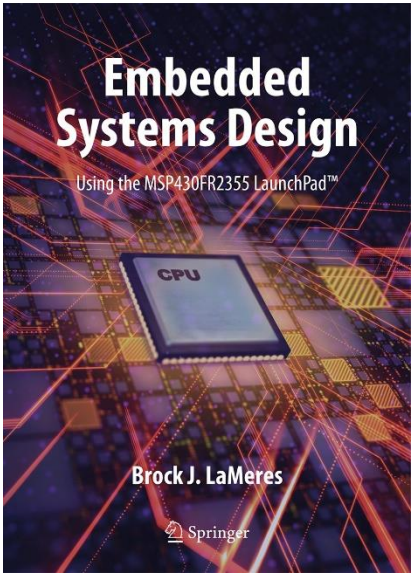
9.1.7 USING LITERAL DEFINITIONS FROM THE MSP430.H HEADER FILE



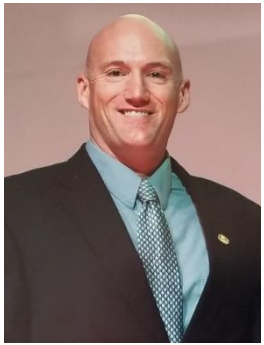


CHAPTER 9: DIGITAL I/O

9.1 THE MSP430 DIGITAL I/O SYSTEM – THE MSP430.H HEADER

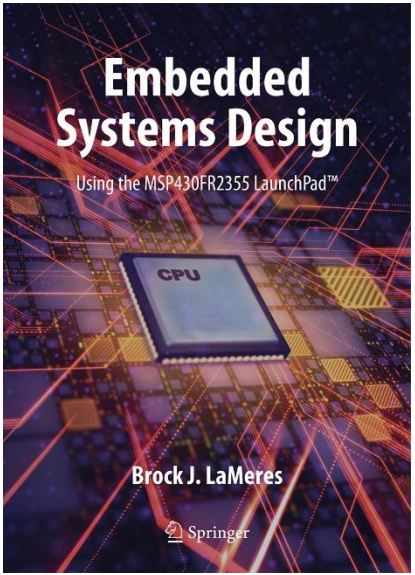


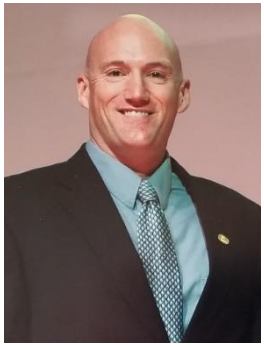
www.youtube.com/c/DigitalLogicProgramming_LaMeres



CHAPTER 9: DIGITAL I/O

9.2 DIGITAL OUTPUT PROGRAMMING





DIGITAL OUTPUT PROGRAM

- Initialize Configuration Registers:

  P1DIR bit 0 = 1 ;Configure P1.0 as an OUTPUT

- Clear LOCKLPM5 in PM5CTL0 register

- Using the names defined in the header file, the assembly code to accomplish this configuration is:

```
bis.b       #BIT0, P1DIR              ; Set P1.0 as an output.  P1.0 = LED1
bic.b       #LOCKLPM5, PM5CTL0        ; Disable Digital I/O low-power default
```

EXAMPLE: USING A DIGITAL OUTPUT TO DRIVE LED1

Step 1: Create a new Empty Assembly-only CCS project titled:

Asm_Dig_IO_Outputs_n_LEDs

Step 2: Type in the following code into the main.asm file where the comments say “Main loop here.”

```
init:
        bis.b   #BIT0, &P1DIR       ; Set P1.0 as an output.  P1.0 = LED1
        bic.b   #LOCKLPM5, &PM5CTL0 ; Disable Digital I/O low-power default

main:
        bis.b   #BIT0, &P1OUT       ; Turn LED1 ON
        bic.b   #BIT0, &P1OUT       ; Turn LED1 OFF

        jmp     main
```

The & is needed when using the literal names from the msp430.h header file as the numeric values are directly substituted into main.asm.

EXAMPLE: USING A DIGITAL OUTPUT TO DRIVE LED1

Step 3: Debug your program.

Step 4: Set a breakpoint before the first instruction

bis.b #BIT0, P1DIR

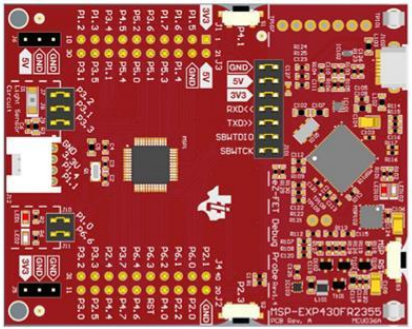


Image Courtesy of https://neodem.wp.horizon.ac.uk/

EXAMPLE: USING A DIGITAL OUTPUT TO DRIVE LED1

Step 5: Run your program to the breakpoint.

Step 6: Step your program to observe its operation.

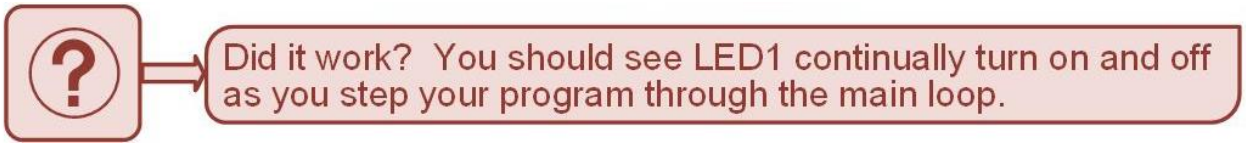


Image Courtesy of https://neodem.wp.horizon.ac.uk/

TIPS AND TRICKS

- The port registers may be viewed during the debug process.
- When the digital I/O is not configured correctly, the result is that the I/O pin simply does nor work.
- When this occurs, the only way to figure out what is going on is to go into the Register Viewer and make sure the configuration registers are set up as desired.

EXAMPLE: VIEWING PORT REGISTERS IN THE DEBUGGER

Step 1: Open the project from the last example titled **Asm_Dig_IO_Outputs_n_LEDs**. If you didn't do the last example, create a new Empty Assembly-only CCS project of this name.

Step 2: If necessary, type in the following code into the main.asm file where the comments say "Main loop here."

```
init:
        bis.b   #BIT0, &P1DIR       ; Set P1.0 as an output.  P1.0 = LED1
        bic.b   #LOCKLPM5, &PM5CTL0 ; Disable Digital I/O low-power default

main:
        bis.b   #BIT0, &P1OUT       ; Turn LED1 ON
        bic.b   #BIT0, &P1OUT       ; Turn LED1 OFF

        jmp     main
```

The & is needed when using the literal names from the msp430.h header file as the numeric values are directly substituted into main.asm.

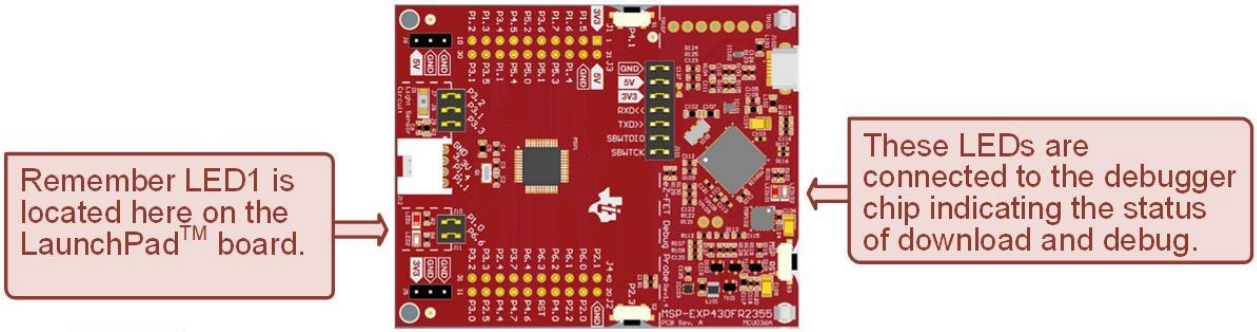
EXAMPLE: VIEWING PORT REGISTERS IN THE DEBUGGER

Step 3: Debug your program.

Step 4: Set a breakpoint before the first instruction

bis.b #BIT0, P1DIR

Step 5: Run to the breakpoint.



EXAMPLE: VIEWING PORT REGISTERS IN THE DEBUGGER

Step 6: Open the Register Viewer and scroll down.

Step 7: Expand the P1 register. You will see all of the configuration registers for this port and their current settings.

Step 8: Change the format to binary. Note that there are additional configuration registers for P1 that have not been covered yet.

Image Courtesy of https://neodem.wp.horizon.ac.uk/

EXAMPLE: VIEWING PORT REGISTERS IN THE DEBUGGER

Step 9: Step your program through the main loop and observe how P1.0 is set and cleared and how this corresponds to LED1 turning on and off.

Each time P1.0 changes values, it is observed in P1OUT.

Note that bit 0 of P1IN mirrors bit 0 of P1OUT.

Variables | Registers

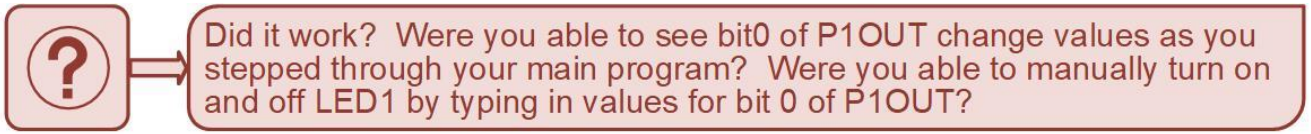
Name	Value	Description
P1		
P1IV	0000000000000000b (Binary)	Port 1 Interrupt Vector Register [Memory Mapped]
P1IN	00000001b (Binary)	Port 1 Input [Memory Mapped]
P1OUT	10001001b (Binary)	Port 1 Output [Memory Mapped]
P1DIR	00000001b (Binary)	Port 1 Direction [Memory Mapped]
P1REN	00000000b (Binary)	Port 1 Resistor Enable [Memory Mapped]
P1SEL0	00000000b (Binary)	Port 1 Select 0 [Memory Mapped]
P1SEL1	00000000b (Binary)	Port 1 Select 1 [Memory Mapped]

Image Courtesy of https://neodem.wp.horizon.ac.uk/

EXAMPLE: VIEWING PORT REGISTERS IN THE DEBUGGER

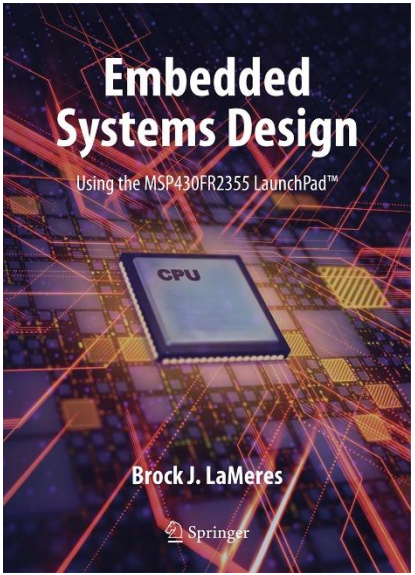
Step 10: Now manually enter 1's and 0's for P1.0 in the P1OUT register.

Step 11: After each entry, hit return. Notice that LED1 turns on and off based on your entry. This is a helpful technique to ensure that the configuration registers are correct and that the LaunchPad™ is working as expected.

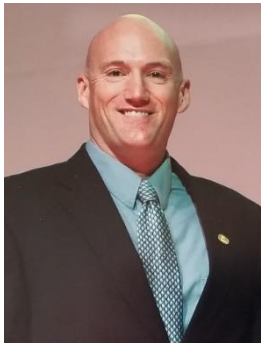


CHAPTER 9: DIGITAL I/O

9.2 DIGITAL OUTPUT PROGRAMMING

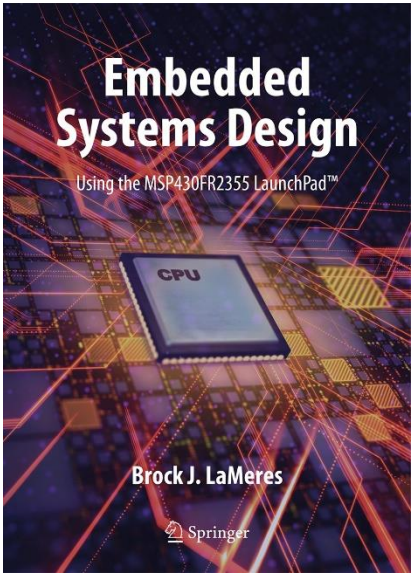


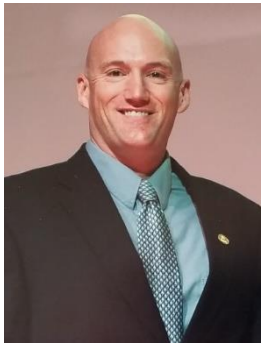
www.youtube.com/c/DigitalLogicProgramming_LaMeres



CHAPTER 9: DIGITAL I/O

9.3 DIGITAL INPUT PROGRAMMING



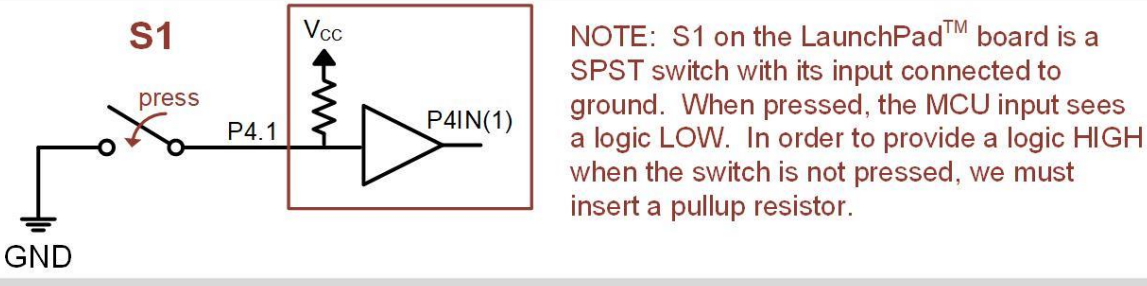


9.3 DIGITAL INPUT PROGRAMMING

- There are two push-button switches provided on the LaunchPad™ board labeled **S1(P4.1)** and **S2(P2.3)**.

- Both push-button switches are SPST with their inputs connected to ground. This provides a logic LOW to the MCU when not pressed.

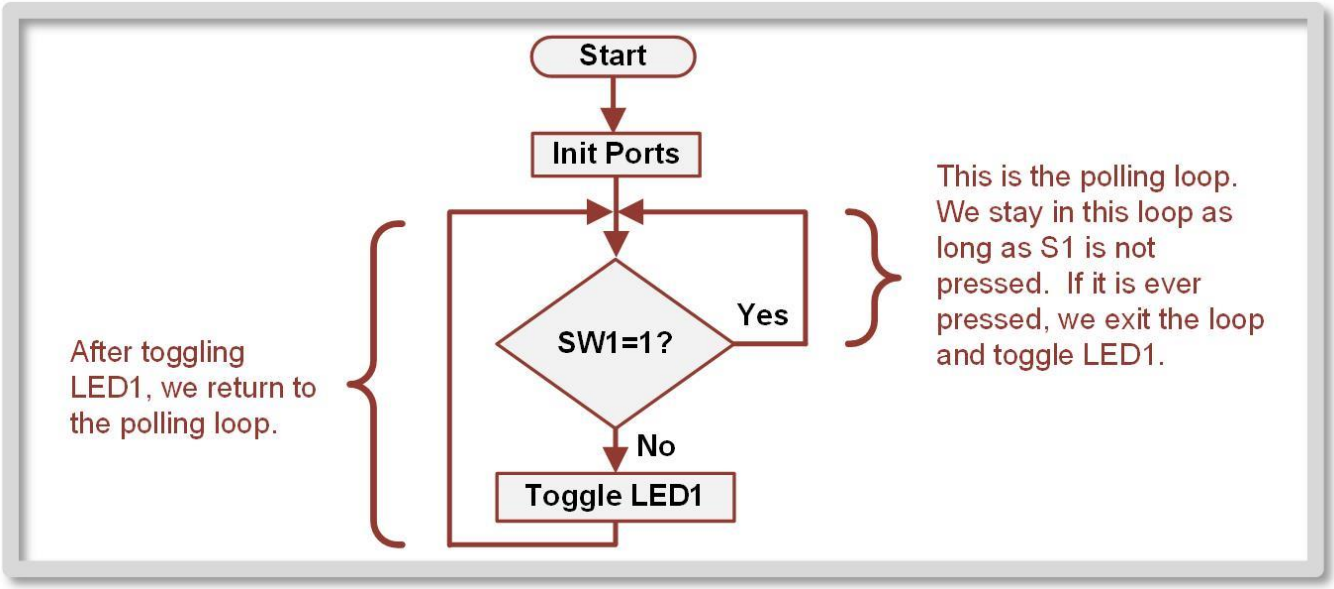
- To use these switches, include pull-up resistors on the MCU to provide a known state when not pressed.



NOTE: S1 on the LaunchPad™ board is a SPST switch with its input connected to ground. When pressed, the MCU input sees a logic LOW. In order to provide a logic HIGH when the switch is not pressed, we must insert a pullup resistor.

9.3 DIGITAL INPUT PROGRAMMING

- **Polling** - creating a program loop that will continually check the value of the input and only exit the loop if the value changes.



EXAMPLE: POLLING THE INPUT S1

Step 1: Create a new Empty Assembly-only CCS project titled:

Asm_Dig_IO_Inputs_n_Polling_S1

Step 2: Type in the following code into the main.asm file where the comments say “Main loop here.”

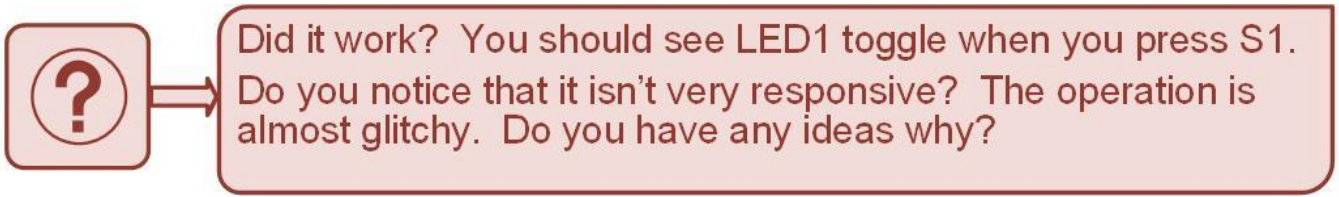
EXAMPLE: POLLING THE INPUT S1

```
init:
        bis.b   #BIT0, &P1DIR       ; Set P1.0 as an output.  P1.0 = LED1
        bic.b   #BIT0, &P1OUT       ; Set initial value of LED1 to 0

        bic.b   #BIT1, &P4DIR       ; Set P4.1 as an input.  P4.1 = S1
        bis.b   #BIT1, &P4REN       ; Enable pullup/pulldown resistor on P4.1
        bis.b   #BIT1, &P4OUT       ; Make the resistor a pullup

        bic.b   #LOCKLPM5, &PM5CTL0 ; Disable Digital I/O low-power default

main:

poll_S1:
        bit.b   #BIT1, &P4IN        ; Test P4.1.  If > 0, no press and Z=0
        jnz     poll_S1             ; Stay in polling loop if Z=0

toggle_LED1:
        xor.b   #BIT0, &P1OUT       ; Toggle P1.1 by XOR'ing it with a 1

        jmp     main
```

EXAMPLE: POLLING THE INPUT S1

Step 3: Debug your program.

Step 4: Run your program and test whether LED1 toggles when you press SW1.



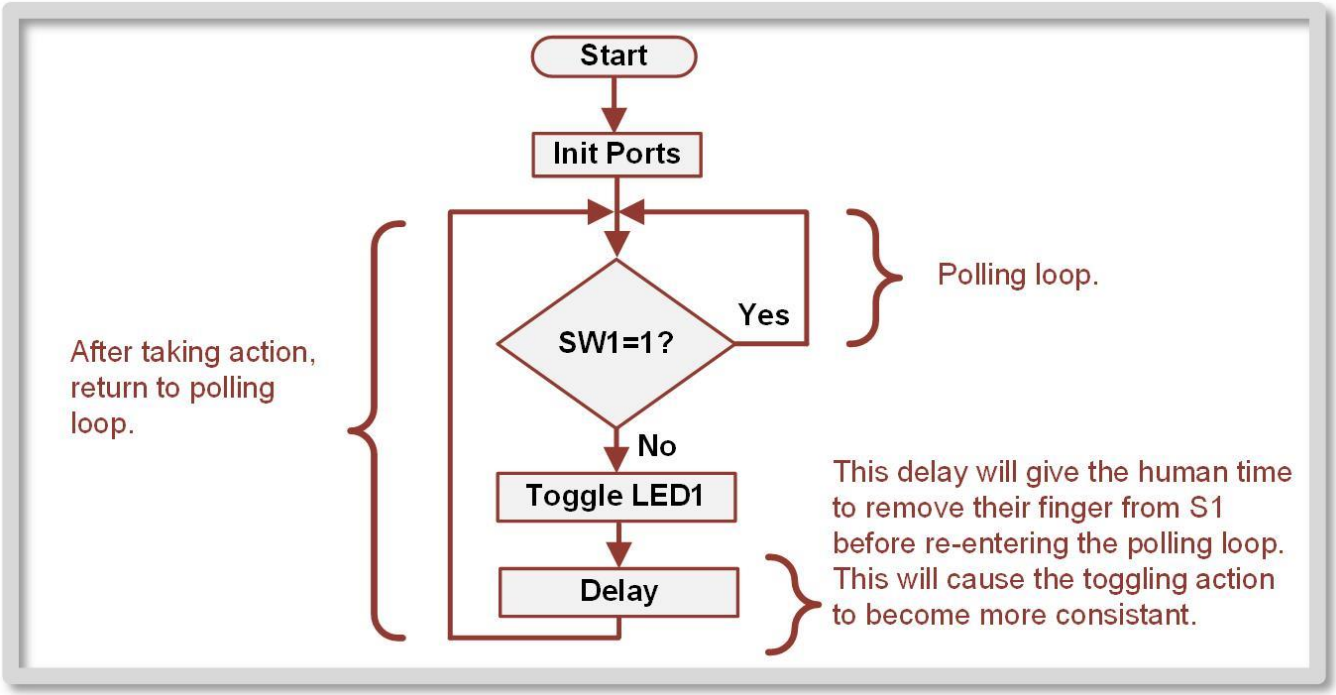
9.3 DIGITAL INPUT PROGRAMMING

- The reason that LED1 is not always toggled when you press and release S1 is because you can never tell what value LED1 is at when you release the button.

- The reason that LED1 appears dimmer when you hold down S1 is because it is being continually turning LED1 on and off millions of times each second while the program checks S1, exists the polling loop, and performs the XOR toggle operation.

9.3 DIGITAL INPUT PROGRAMMING

- Consider this new logic for polling S1 that inserts some delay after the LED1 toggling action.



EXAMPLE: POLLING THE INPUT S1 WITH DELAY

Step 1: Open the project form the last example titled:

Asm_Dig_IO_Inputs_n_Polling_S1

Step 2: Type in the delay portion of the code into the main.asm.

Image Courtesy of https://neodem.wp.horizon.ac.uk/

EXAMPLE: POLLING THE INPUT S1 WITH DELAY

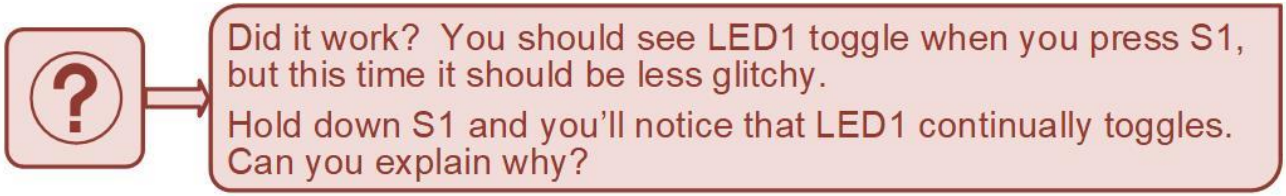
```
init:
        bis.b   #BIT0, &P1DIR       ; Set P1.0 as an output.  P1.0 = LED1
        bic.b   #BIT0, &P1OUT       ; Set initial value of LED1 to 0

        bic.b   #BIT1, &P4DIR       ; Set P4.1 as an input.  P4.1 = S1
        bis.b   #BIT1, &P4REN       ; Enable pullup/pulldown resistor on P4.1
        bis.b   #BIT1, &P4OUT       ; Make the resistor a pullup

        bic.b   #LOCKLPM5, &PM5CTL0 ; Disable Digital I/O low-power default

main:

poll_S1:
        bit.b   #BIT1, &P4IN        ; Test P4.1.  If > 0, no press and Z=0
        jnz     poll_S1             ; Stay in polling loop if Z=0

toggle_LED1:
        xor.b   #BIT0, &P1OUT       ; Toggle P1.1 by XOR'ing it with a 1

        mov.w   #0FFFFh, R4         ; Delay loop
delay:
        dec.w   R4
        jnz     delay

        jmp     main
```

Delay gives time for the human to remove their finger from S1.

EXAMPLE: POLLING THE INPUT S1

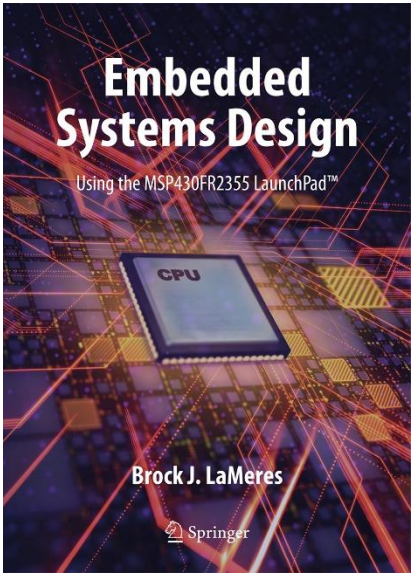
Step 3: Debug your program.

Step 4: Run your program and test whether LED1 toggles when you press SW1 more reliably than before.



CHAPTER 9: DIGITAL I/O

9.3 DIGITAL INPUT PROGRAMMING



www.youtube.com/c/DigitalLogicProgramming_LaMeres