Embedded Systems Design Using Microcontroller MSP430

Ts. Lê Mạnh Hải

Khoa CNTT,

ĐH Kỹ thuật Công nghệ TP HCM

Mở đầu

I Mục đích môn học:

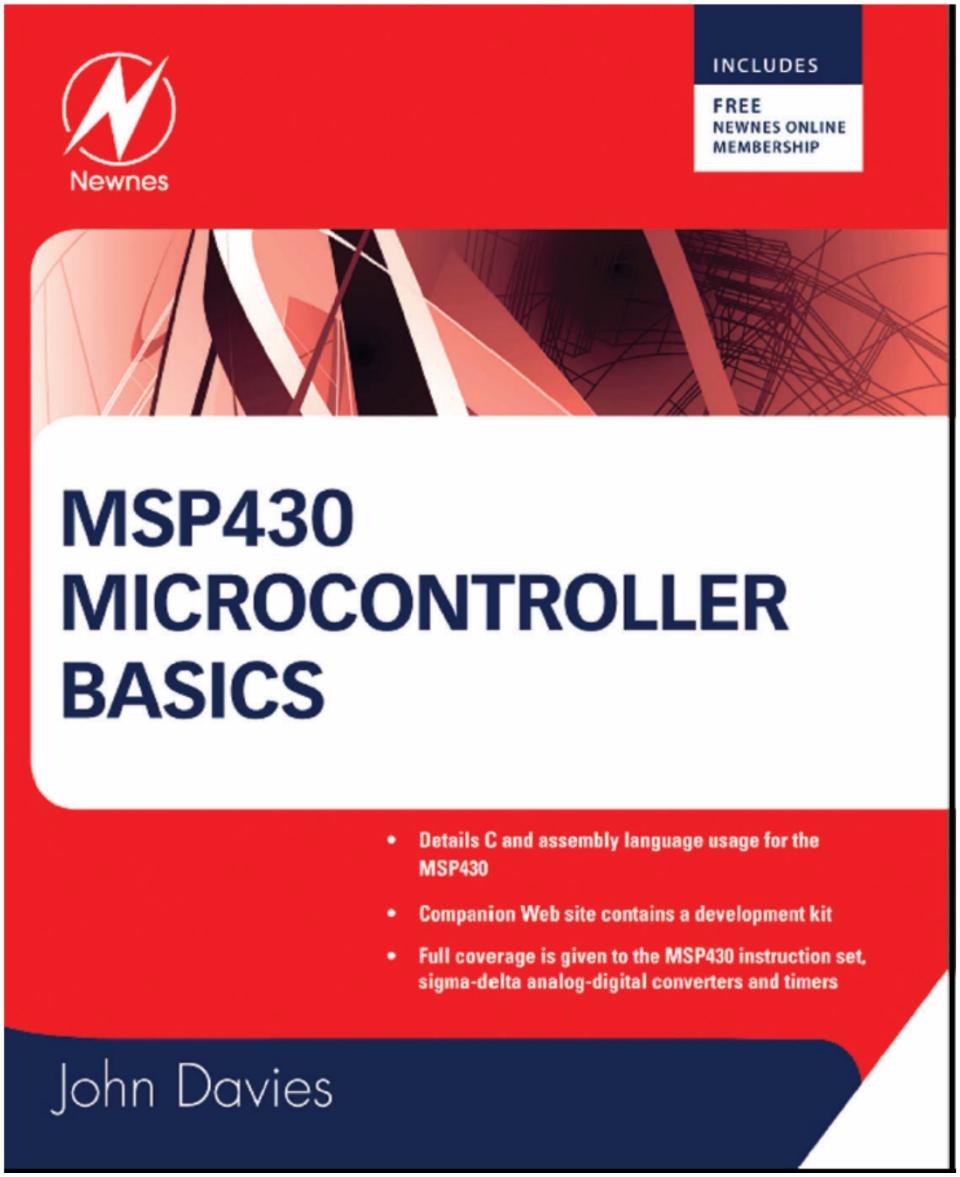
- Cung cấp kiến thức về lập trình vi điều khiển TI MSP430.
- Rèn luyện kỹ năng đọc sách chuyên ngành bằng **tiếng Anh**

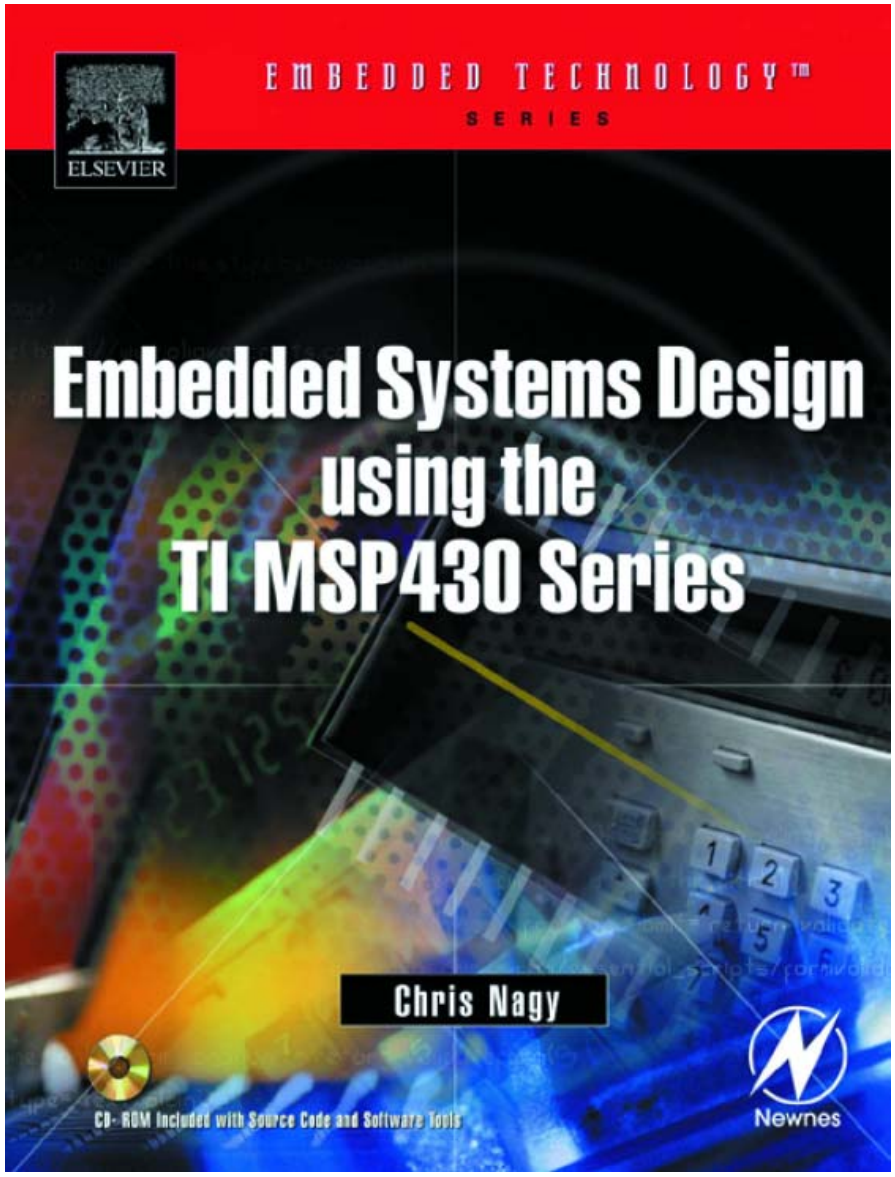
II. Thời gian:

- 30 tiết lý thuyết (2 tín chỉ) + 30 tiết thực hành (1 tín chỉ)

III Giáo trình và tài liệu tham khảo

- ***MSP430 Microcontroller Basics***. John H. Davies. Elsevier. 2008 (685 trang)
- ***Embedded Systems Design using the TI MSP430 Series.*** Chris Nagy. Elsevier. 2003 (296trang)
- Introduction to Embedded Systems - A Cyber-Physical Systems Approach, E. A. Lee and S. A. Seshia. http://LeeSeshia.org. 2011


Newnes
INCLUDES
FREE
NEWNES ONLINE
MEMBERSHIP
MSP430
MICROCONTROLLER
BASICS
Details C and assembly language usage for the MSP430
Companion Web site contains a development kit
Full coverage is given to the MSP430 instruction set, sigma-delta analog-digital converters and timers
John Davies


ELSEVIER
EMBEDDED TECHNOLOGY™
SERIES
Embedded Systems Design
using the
TI MSP430 Series
Chris Nagy
CD- ROM Included with Source Code and Software Tools
Newnes

IV. Đánh giá:

- Thi kết thúc môn: Bài tự luận với 3 câu hỏi.

V. Giáo viên:

- Ts. Lê Mạnh Hải. Tel: 0985399000.
- Không gọi điện thoại để hỏi hay xin điểm, email: hailemanh@yahoo.com, **lm.hai@hutech.edu.vn**
- Website: giangvien.hutech.edu.vn
- GV thực hành: Nguyễn Ngọc Đức. 0978629557

Nội dung chi tiết

Chương 1: Các hệ thống nhúng và vi điều khiển

Chương 2: Chíp Texas Instruments MSP430

Chương 3: Phát triển ứng dụng nhúng

Chương 4: Sơ lược về MSP430

Chương 5: Kiến trúc vi điều khiển MSP430

Chương 6: các hàm và ngắt

Chương 7: Nhập/xuất

Chương 8: Bộ định thời

Chương 9: ADC và DAC

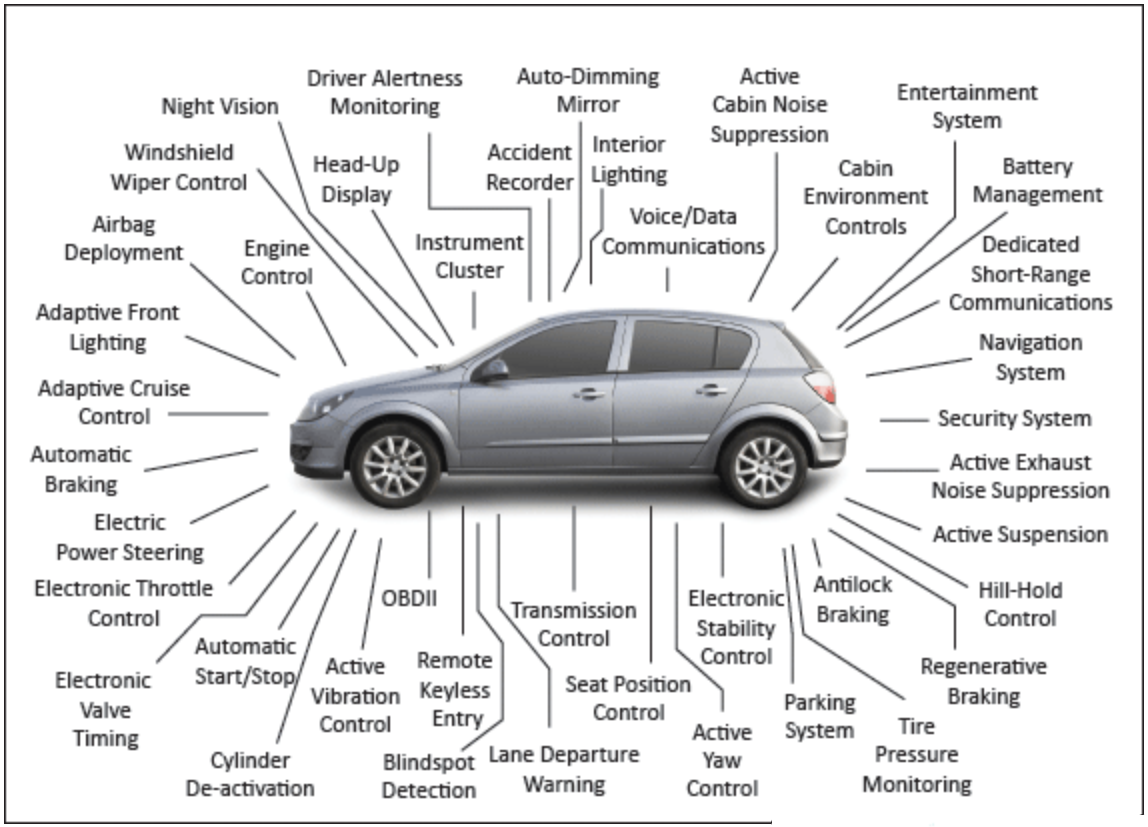
Chương 10: Kết nối

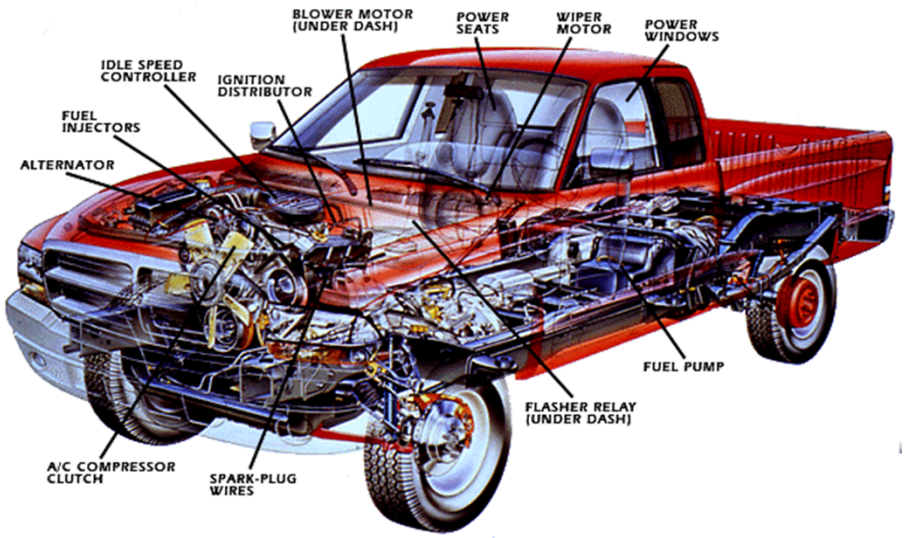
Chapter 1: Embedded Electronic Systems and Microcontrollers

1. **What (and Where) Are Embedded Systems?**
2. **Approaches to Embedded Systems**
3. **Small Microcontrollers**
4. **Anatomy of a Typical Small Microcontroller**
5. **Memory**
6. **Software**
7. **Where Does the MSP430 Fit?**

Embedded Systems

- Washing machines and video recorders.
- Fancy modern cars have approaching 100 processors
- Embedded systems encompass a broad range of computational power.
- *embedded* seems synonymous with *invisible*

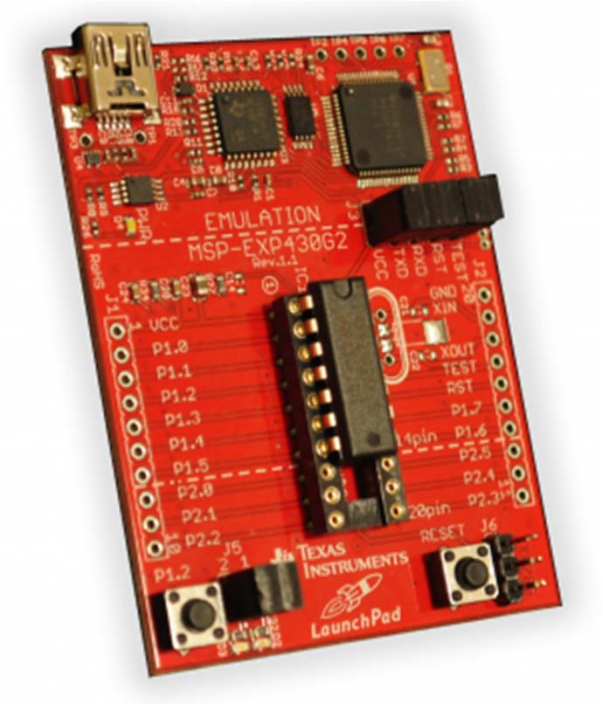

Night Vision
Driver Alertness Monitoring
Auto-Dimming Mirror
Active Cabin Noise Suppression
Entertainment System
Windshield Wiper Control
Head-Up Display
Accident Recorder
Interior Lighting
Cabin Environment Controls
Battery Management
Airbag Deployment
Engine Control
Instrument Cluster
Voice/Data Communications
Dedicated Short-Range Communications
Adaptive Front Lighting
Navigation System
Adaptive Cruise Control
Security System
Automatic Braking
Active Exhaust Noise Suppression
Electric Power Steering
Active Suspension
Electronic Throttle Control
OBDII
Transmission Control
Electronic Stability Control
Antilock Braking
Hill-Hold Control
Electronic Valve Timing
Automatic Start/Stop
Active Vibration Control
Remote Keyless Entry
Seat Position Control
Regenerative Braking
Parking System
Cylinder De-activation
Blindspot Detection
Lane Departure Warning
Active Yaw Control
Tire Pressure Monitoring


BLOWER MOTOR (UNDER DASH)
POWER SEATS
WIPER MOTOR
POWER WINDOWS
IDLE SPEED CONTROLLER
IGNITION DISTRIBUTOR
FUEL INJECTORS
ALTERNATOR
FUEL PUMP
FLASHER RELAY (UNDER DASH)
A/C COMPRESSOR CLUTCH
SPARK-PLUG WIRES

- A cellular phone also has a 32-bit processor and digital signal processing (DSP).
- The subject of this book is the **Texas Instruments MSP430**, which is a straightforward, modern 16-bit processor designed specially for low-power.

LaunchPad: 5USD

MSP430G2211IN14
MSP430G2231IN14



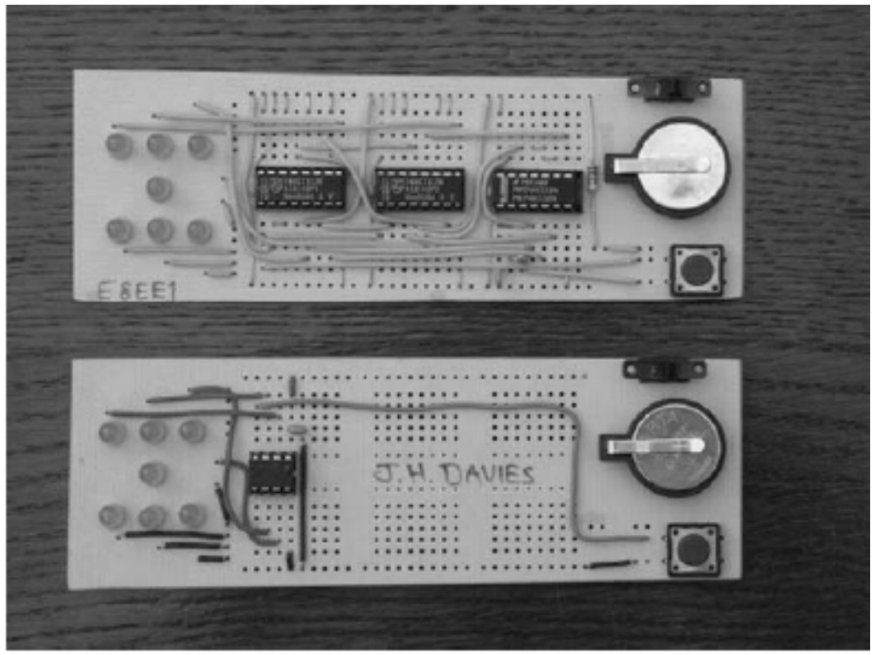
IAR Kickstart or Code Composer Studio Ver 4 (CCS)

MSP-EXP430G2 LaunchPad Experimenter Board

Many different approaches can be taken for the design of embedded systems

- The general trend is toward digital systems and increasing integration: Systems that used analog electronics or small-scale integrated circuits (ICs) in the past
- Now more likely to use larger digital ICs

- ***Small-Scale Integration: The 555***
- ***Medium-Scale Integration: 4000 Series CMOS***
- ***Large-Scale Integration: Small Microcontroller***



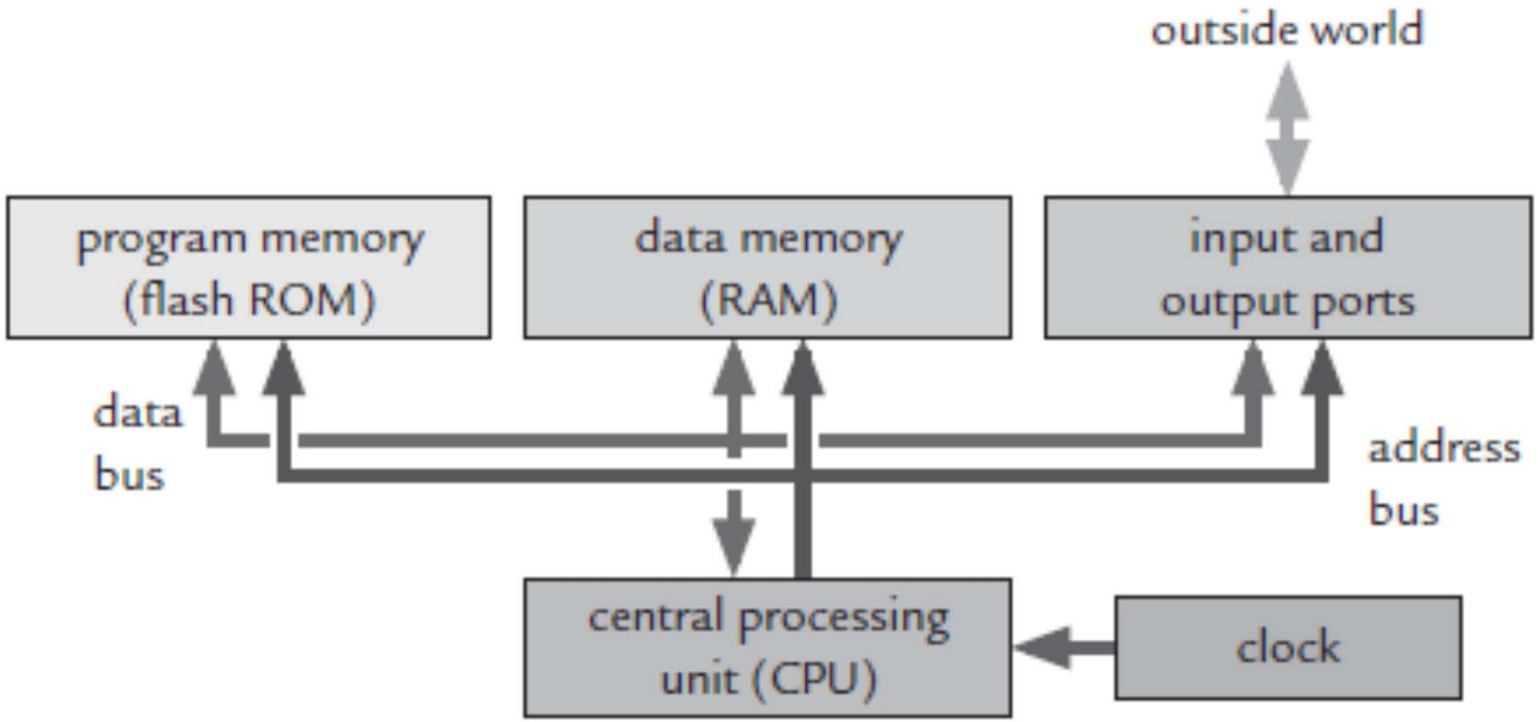
Larger Systems

- **Application-specific integrated circuits (ASICs)**
  - Specially designed for a particular application
- **Field-programmable gate arrays (FPGAs) and programmable logic devices (PLDs)**
  - Essentially an array of gates and flip-flops, which can be connected by programming the device to produce the desired function
- **Microcontrollers**
  - These have nearly fixed hardware built around a central processing unit (CPU)

Small Microcontrollers

- *Microprocessor or microcontroller?*
- Process 8 or 16 bits of data and have a 16-bit address bus
- 64KB of memory
- Main function is likely to be sequential control rather than computation.

Anatomy of a Typical Small Microcontroller



- **Central processing unit**
  - Arithmetic logic unit (ALU), which performs computation.
  - Registers needed for the basic operation of the CPU, such as the program counter (PC), stack pointer (SP), and status register (SR).
  - Further registers to hold temporary results.
  - Instruction decoder and other logic to control the CPU, handle resets, and interrupts, and so on.

- **Memory for the program:** Nonvolatile (read-only memory, ROM), meaning that it retains its contents when power is removed.
- **Memory for data:** Known as random-access memory (RAM) and usually volatile.
- **Input and output ports:** To provide digital communication with the outside world.
- **Address and data buses:** To link these subsystems to transfer data and instructions.
- **Clock:** To keep the whole system synchronized.

Timers: Most microcontrollers have at least one timer because of the wide range of functions that they provide.

Watchdog timer: This is a safety feature, which resets the processor if the program becomes stuck in an infinite loop.

Communication interfaces: A wide choice of interfaces is available to exchange information with another IC or system.

Nonvolatile memory for data: This is used to store data whose value must be retained when power is removed.

Analog-to-digital converter: This is very common because so many quantities in the real world vary continuously.

Digital-to-analog converter: This is much less common, because most analog outputs can be simulated using PWM.

Real-time clock: These are needed in applications that must track the time of day.

Monitor, background debugger, and embedded emulator: These are used to download the program into the MCU

Memory

- Each location can typically store 1 byte (8 bits or 1B) of data and is often called a *register*
- Memory is linked to the CPU by buses for data, address, and control

Memory types

- **Volatile and Nonvolatile Memory**
  - **Volatile:** Loses its contents when power is removed
  - **Nonvolatile:** Retains its contents
    - **Masked ROM**
    - **EPROM (electrically programmable ROM)**
    - **OTP (one-time programmable memory)**
    - **Flash memory**

Harvard and von Neumann Architectures

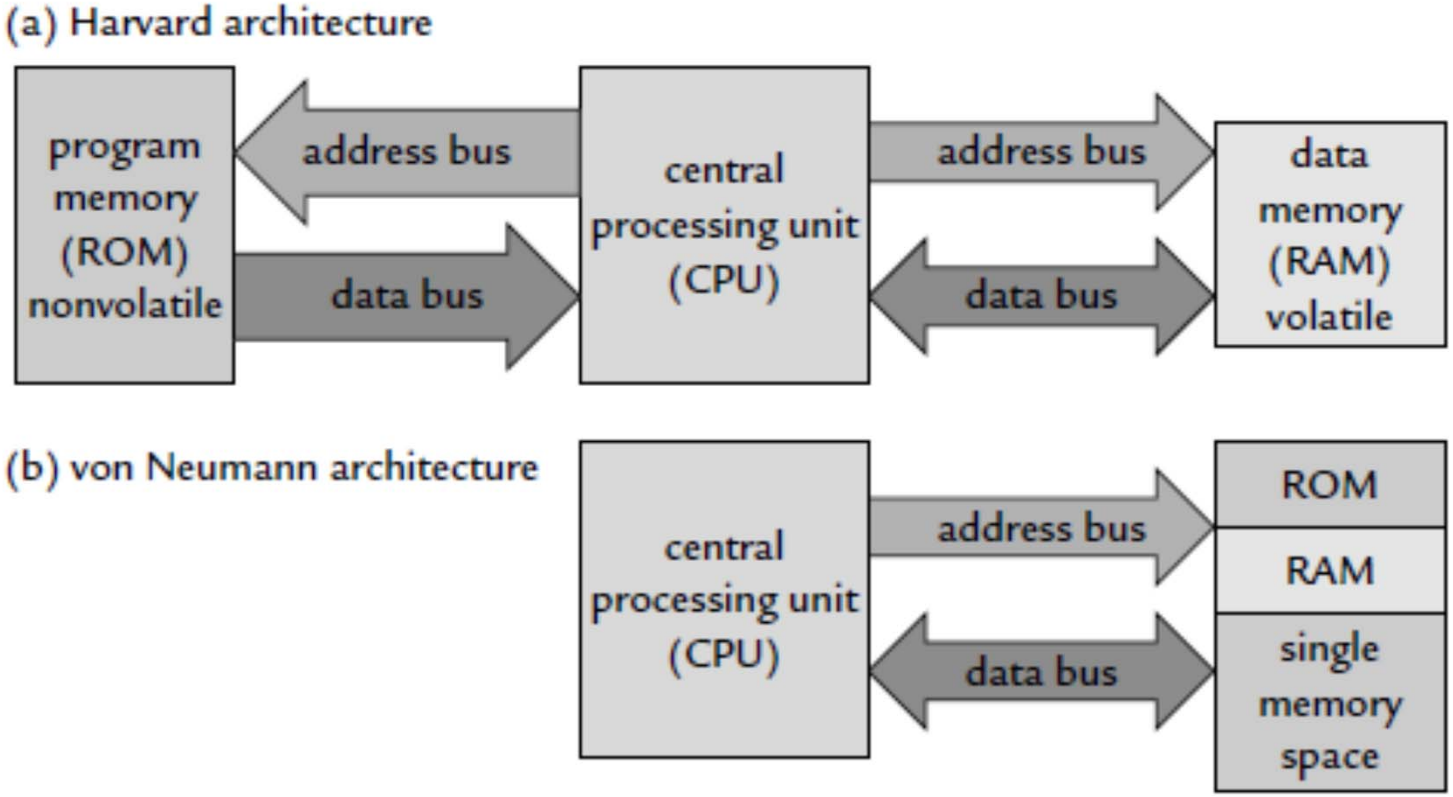


Figure 1.3: Harvard and von Neumann architectures for memory.

Software

- **C:** The most common choice for small microcontrollers nowadays. A compiler translates C into machine code that the CPU can process.
- IAR and CCS

Where Does the MSP430 Fit?

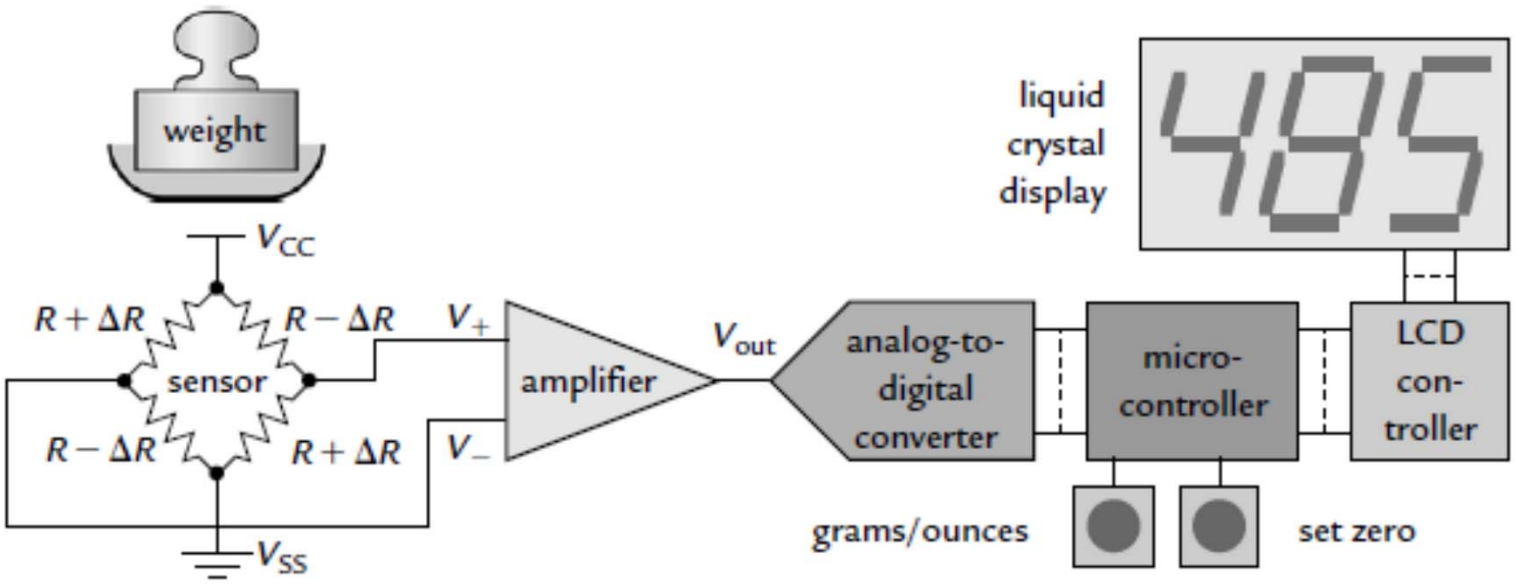


Figure 1.4: Weighing machine with a liquid crystal display, broken down into individual functions.

Quizzes

- Name some **embedded systems**?
- What is **microcontroller**?
- What are differences between **Harvard and von Neumann Architectures**?

What is next?

- ***CHAPTER 2: The Texas Instruments MSP430 (Page 21-42)***

Chíp Texas Instruments MSP430

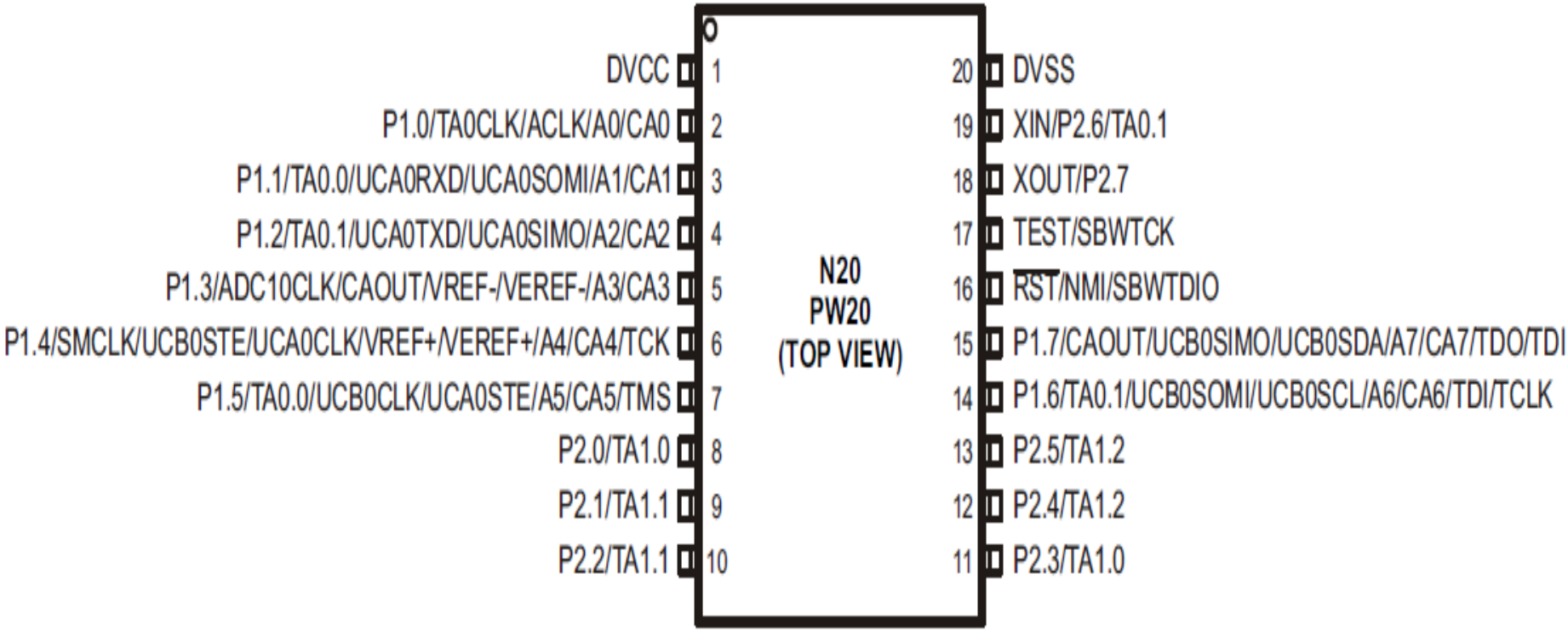
Contents

- **The Outside View—Pin-Out**
- **The Inside View—Functional Block Diagram**
- **Memory**
- **Central Processing Unit**
- **Memory-Mapped Input and Output**
- **Clock Generator**
- **Exceptions: Interrupts and Resets**
- **Where to Find Further Information**

The Outside View—Pin-Out

- *pin-out:* 14-pin plastic dual-in-line package (PDIP)
- Perhaps the most obvious feature is that almost all pins have several functions
- Most applications do not use all the peripherals so, with luck, there is no conflict where a design needs more than one function on a pin simultaneously

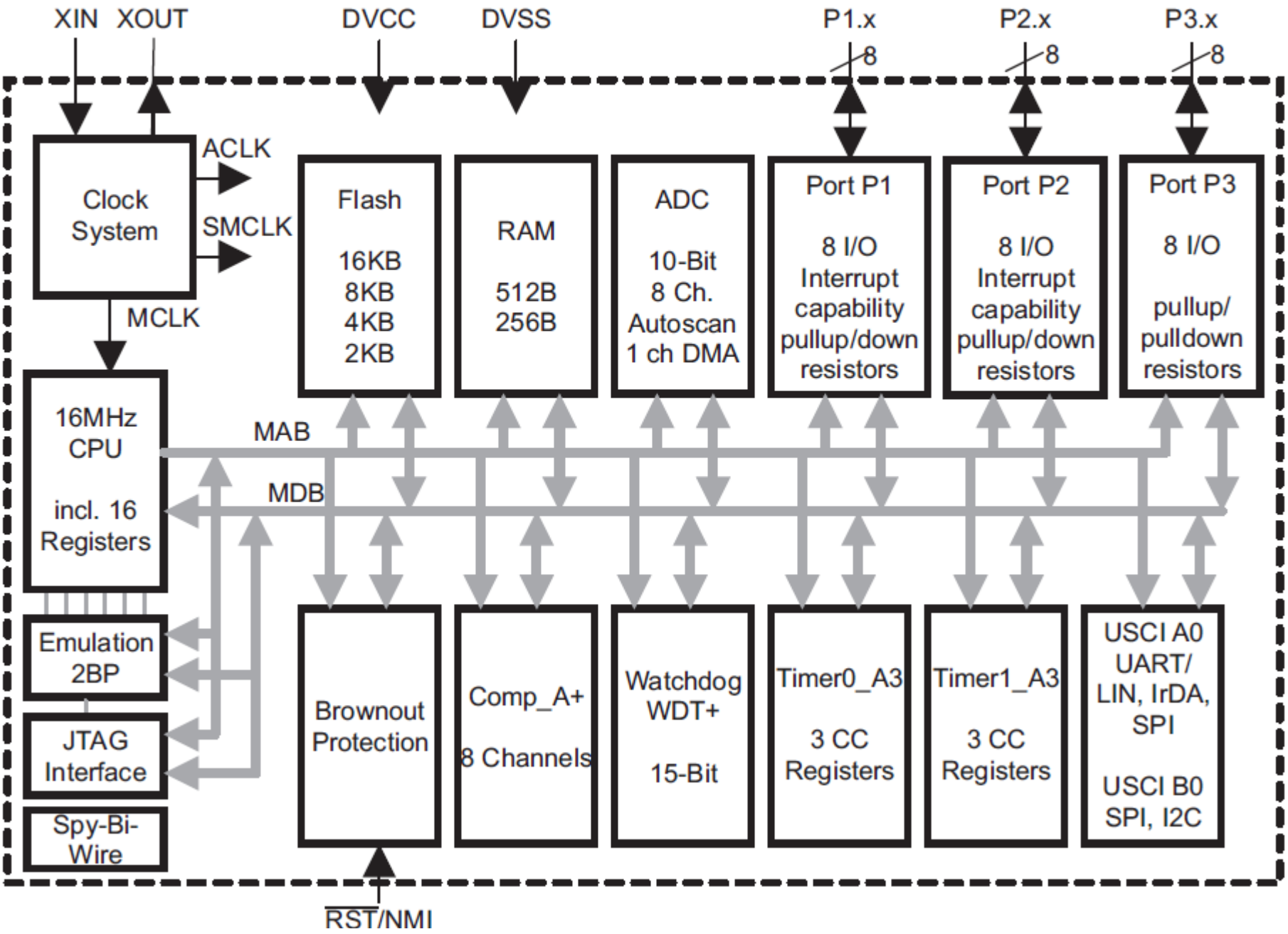
Device Pinout, MSP430G2x13 and MSP430G2x53, 20-Pin Devices, TSSOP and PDIP



NOTE: ADC10 is available on MSP430G2x53 devices only.

NOTE: The pulldown resistors of port P3 should be enabled by setting P3REN.x = 1.

Functional Block Diagram, MSP430G2x53



- *V*CC and *V*SS are the supply voltage and ground for the whole device (the analog and digital supplies are separate in the 16-pin package).
- P1.0–P1.7, P2.6, and P2.7 are for digital input and output, grouped into ports P1and P2.
- TACLK, TA0, and TA1 are associated with Timer_A; TACLK can be used as the clock input to the timer, while TA0 and TA1 can be either inputs or outputs. These can be used on several pins because of the importance of the timer.

The Inside View—Functional Block Diagram

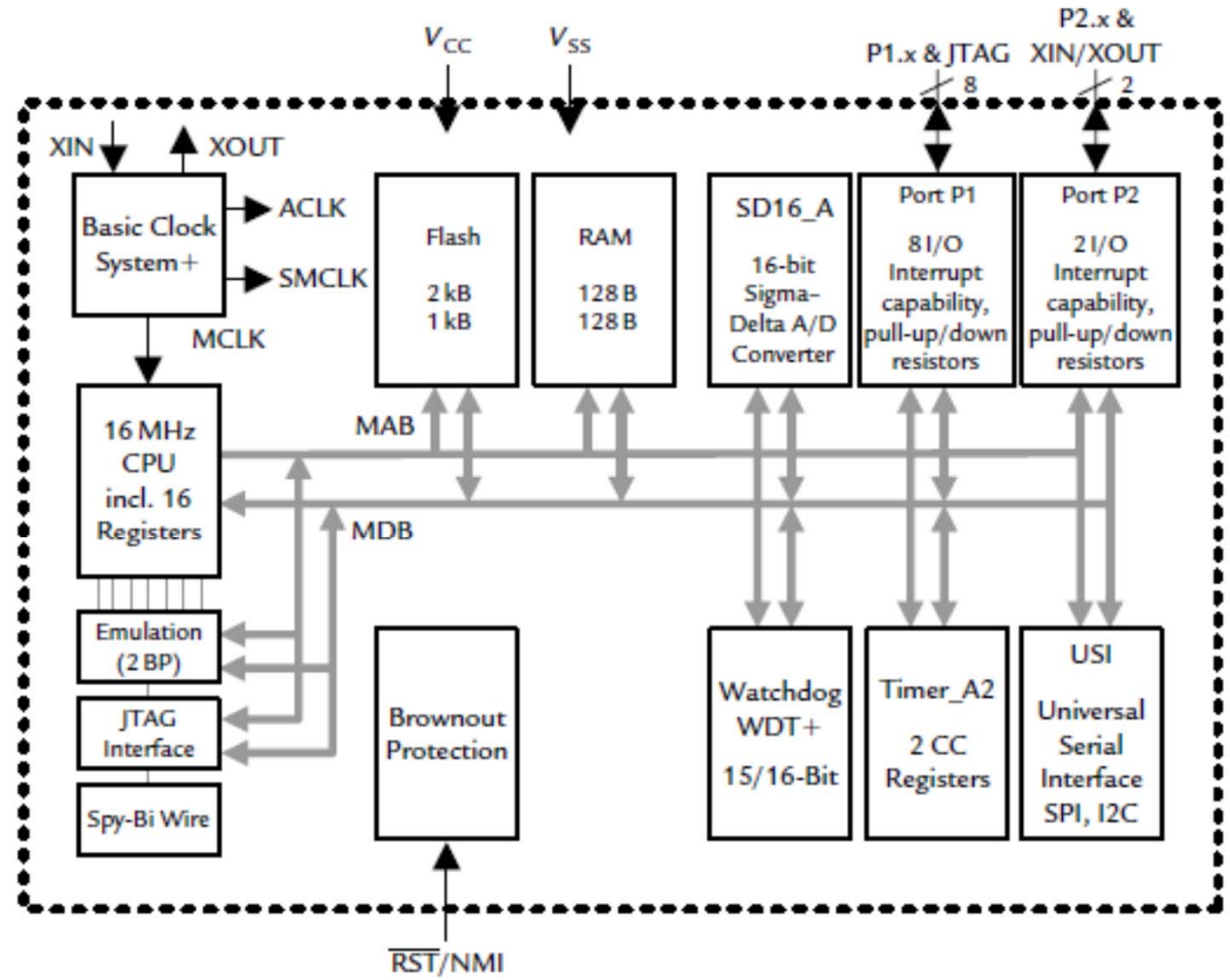
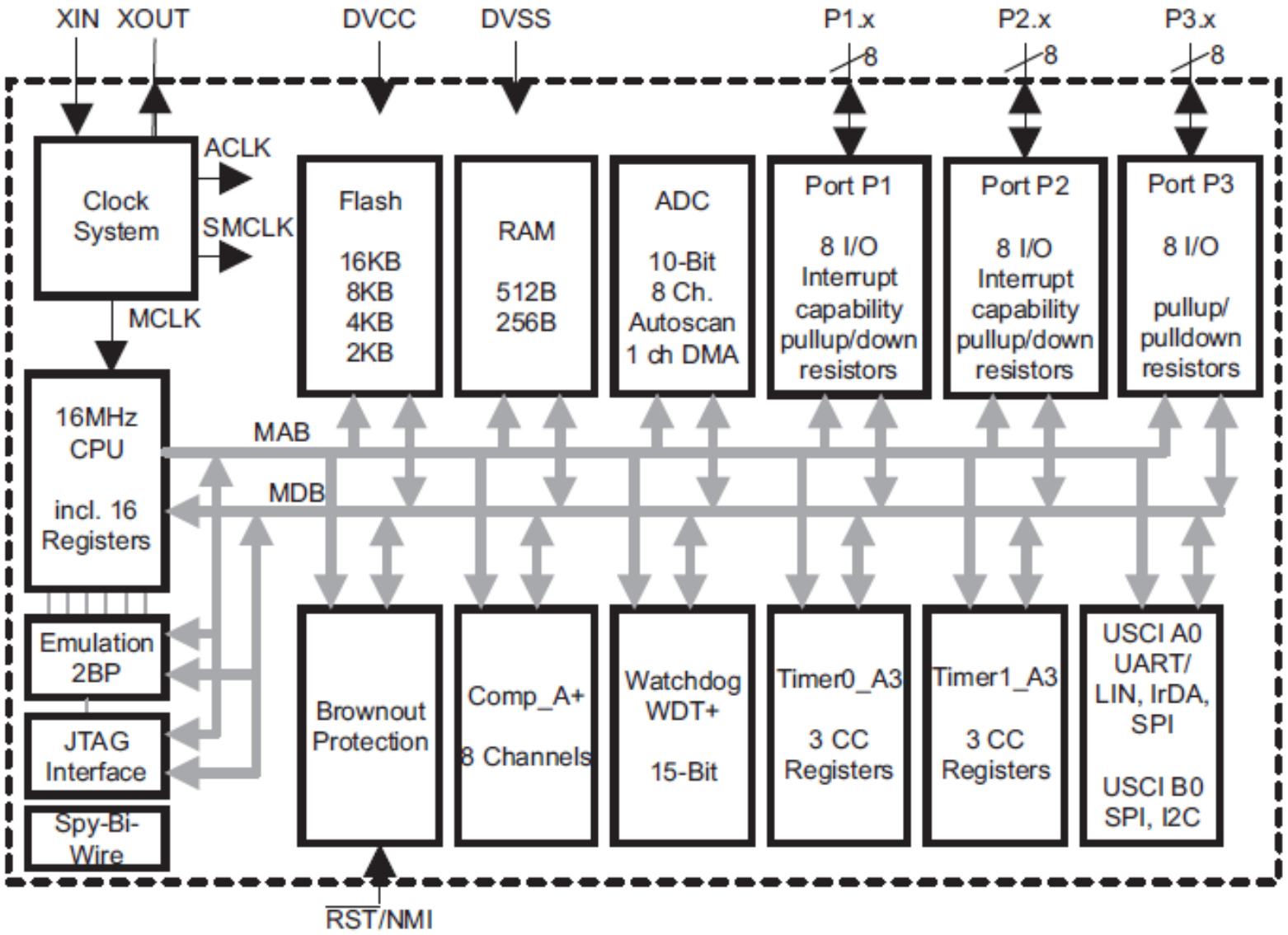


Figure 2.2: Block diagram of the MSP430F2003 and F2013, taken from the data sheet.

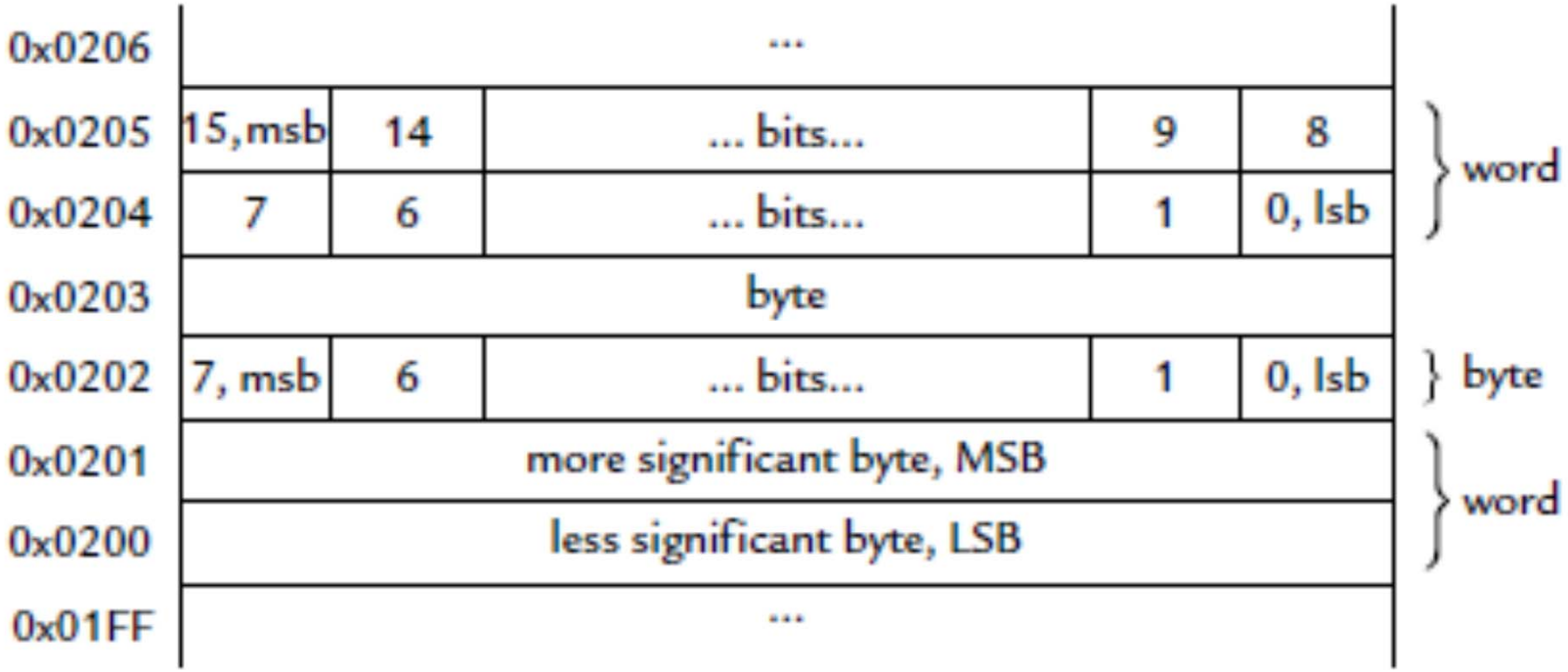
Functional Block Diagram, MSP430G2x53



Memory

- Each register or pigeonhole holds 8 bits or 1 byte and this is the smallest entity that can be transferred to and from memory
- memory address bus is 16 bits: 0x0000 to 0xFFFF
- The memory data bus is 16 bits wide and can transfer either a word of 16 bits or a byte of 8 bits.

Memory address



Ordering

- **Little-endian ordering:** The low-order byte is stored at the lower address and the high-order byte at the higher address. This is used by the MSP430 and is the more common format.
- **Big-endian ordering:** The high-order byte is stored at the lower address. This is used by the Freescale HCS08, for instance.

Memory Map

- **Special function registers:** Mostly concerned with enabling functions of some modules and enabling and signalling interrupts from peripherals.
- **Peripheral registers with byte access and peripheral registers with word access:** Provide the main communication between the CPU and peripherals. Some must be accessed as words and others as bytes.
- **Random access memory:** Used for variables. This always starts at address 0x0200 and the upper limit depends on the size of the RAM. The F2013 has 128 B.
- **Bootstrap loader:** Contains a program to communicate using a standard serial protocol, often with the COM port of a PC.

- **Information memory:** A 256B block of flash memory that is intended for storage of nonvolatile data. This might include serial numbers to identify equipment—an address for a network, for instance—or variables that should be retained even when power is removed. DCO in the MSP430F2xx family and is protected by default.
- **Code memory:** Holds the program, including the executable code itself and any constant data. The F2013 has 2KB but the F2003 only 1KB.
- **Interrupt and reset vectors:** Used to handle "exceptions," when normal operation of the processor is interrupted or when the device is reset. This table was smaller and started at 0xFFE0 in earlier devices.

Address	Type of memory
0xFFFF	interrupt and reset
0xFFC0	vector table
0xFFBF	flash code memory
0xF800	(lower boundary varies)
0xF7FF	
0x1100	
0x10FF	flash
0x1000	information memory
0x0FFF	*bootstrap loader*
0x0C00	*(not in F20xx)*
0x0BFF	
0x0280	
0x027F	RAM
0x0200	(upper boundary varies)
0x01FF	peripheral registers
0x0100	with word access
0x00FF	peripheral registers
0x0100	with byte access
0x000F	special function registers
0x0000	(byte access)

Central Processing Unit

- The central processing unit (CPU) executes the instructions stored in memory. It steps through the instructions in the sequence in which they are stored in memory until it encounters a branch or when an *exception* occurs (interrupt or reset).
- It includes the arithmetic logic unit (ALU), which performs computation, a set of 16 registers designated R0–R15 and the logic needed to decode the instructions and implement them.
- The CPU can run at a maximum clock frequency *f*MCLK of 16MHz

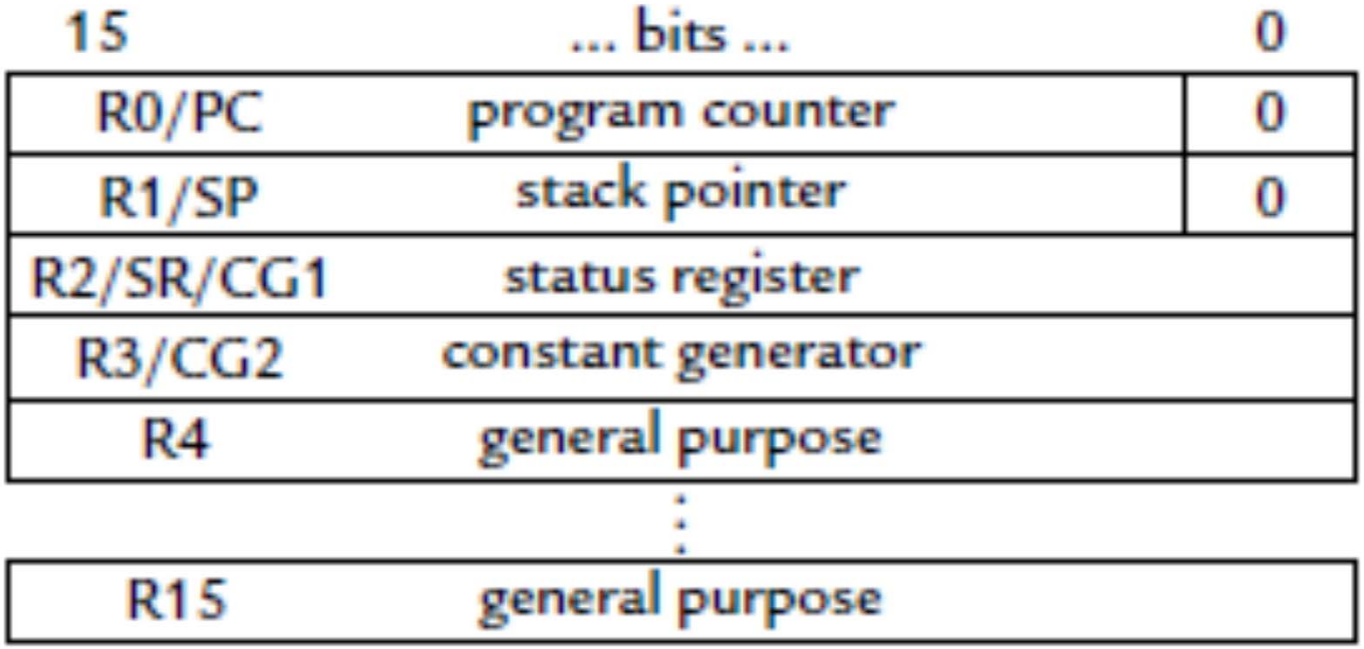


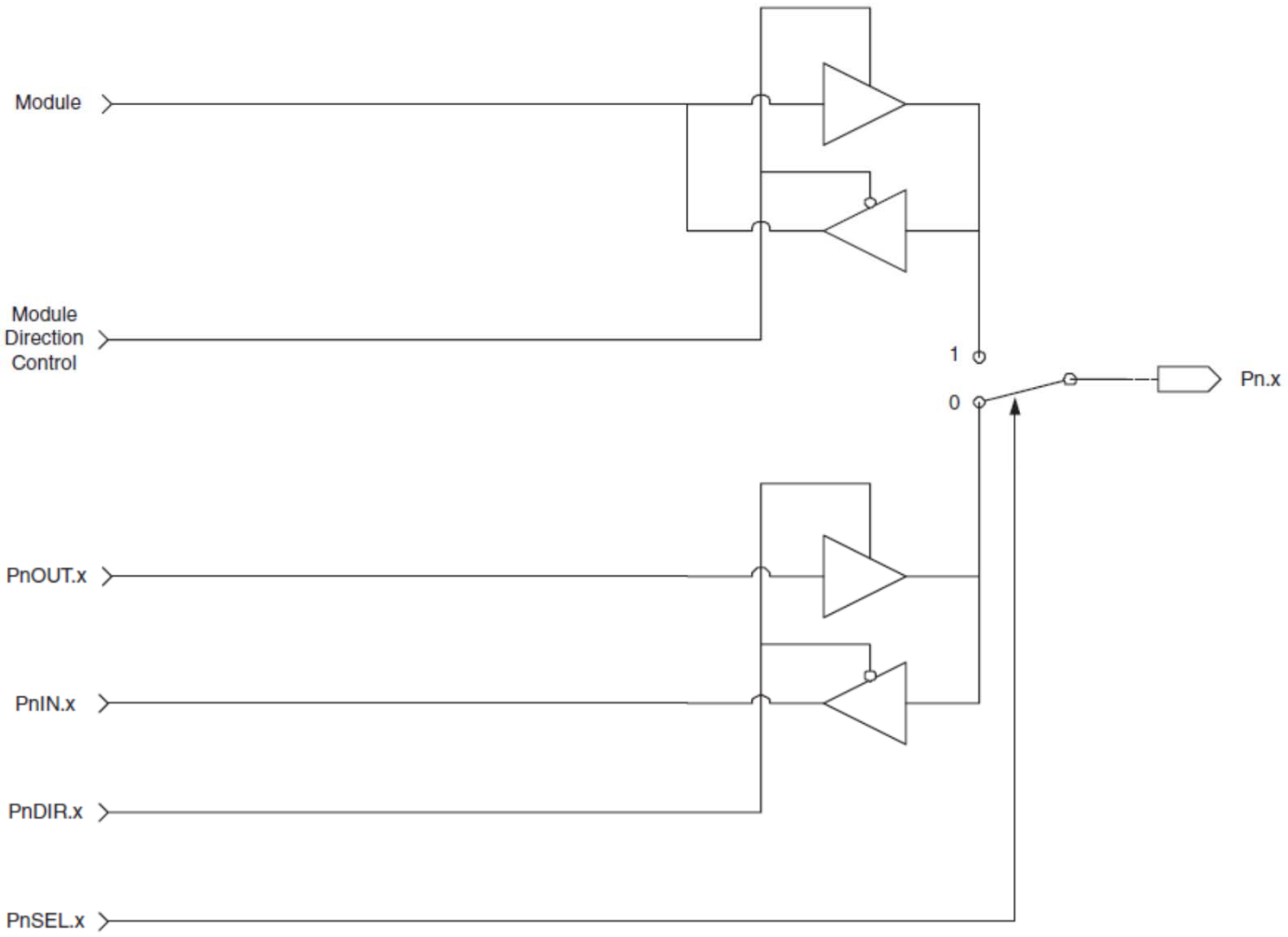
Figure 2.5: Registers in the CPU of the MSP430.

CPU registers

- **Program counter, PC:** This contains the address of the next instruction to be executed, "points to" the instruction in the usual jargon.
- **Stack pointer, SP:** When a subroutine is called, the CPU jumps to the subroutine, executes the code there, then returns to the instruction after the call.
- **Status register, SR:** This contains a set of *flags* (single bits), whose functions fall into three categories. The most commonly used flags are C, Z, N, and V, which give information about the result of the last arithmetic or logical operation.
- **Constant generator:** This provides the six most frequently used values so that they need not be fetched from memory whenever they are needed.
- **General purpose registers:** The remaining 12 registers, R4–R15, are general working registers. They may be used for either data or addresses because both are 16-bit values, which simplifies the operation significantly.

Memory-Mapped Input and Output

- **Port P1 input, P1IN:** Reading returns the logical values on the inputs if they are configured for digital input and output. This register is read-only. It is also *volatile*, which means that it may change at a time that a program cannot predict.
- **Port P1 output, P1OUT:**Writing sends the value to be driven onto the pin if it is configured as a digital output. If the pin is not currently an output, the value is stored in a buffer and appears on the pin if it is later switched to be an output.
- **Port P1 direction, P1DIR:** A bit of 0 configures the pin as an input, which is the default.Writing a 1 switches the pin to become an output.


Module
Module
Direction
Control
1
0
Pn.x
PnOUT.x
PnIN.x
PnDIR.x
PnSEL.x

Clock Generator

- Clock is essential for every synchronous digital system
- Usually a crystal with a frequency of a few MHz would be connected to two pins. It would drive the CPU directly and was typically divided down by a factor of 2 or 4 for the main bus.
- Unfortunately, the conflicting demands for high performance and low power mean that most modern microcontrollers have much more complicated clocks, often with two or more sources.
- In many applications the MCU spends most of its time in a low-power mode until some event occurs, when it must wake up and handle the event rapidly.

- **Master clock, MCLK**, is used by the CPU and a few peripherals.
- **Subsystem master clock, SMCLK**, is distributed to peripherals.
- **Auxiliary clock, ACLK**, is also distributed to peripherals.

- ACLK comes from a low-frequency crystal oscillator, typically at 32 KHz.
- MCLK and SMCLK are supplied from the DCO, which is controlled by a frequency-locked loop (FLL). This locks the frequency at 32 times the ACLK frequency, which is close to 1MHz for the usual watch crystal.

Exceptions: Interrupts and Resets

- **Interrupts:** Usually generated by hardware (although they can be initiated by software) and often indicate that an event has occurred that needs an urgent response. A packet of data might have been received, for instance, and needs to be processed before the next packet arrives. The processor **stops** what it was doing, **stores** enough information (the contents of the program counter and status register) for it to resume later on and **executes** an *interrupt service routine* (ISR). It returns to its previous activity when the ISR has been completed. Thus an ISR is something like a subroutine called by hardware (at an unpredictable time) rather than software.
- **Resets:** Again usually generated by hardware, either when power is applied or when something catastrophic has happened and normal operation cannot continue. This can happen accidentally if the watchdog timer has not been disabled, which is easy to forget. A reset causes the device to (re)start from a well-defined state.

Where to Find Further Information

- **Data Sheet**
- **Family User's Guide**
- **FET User's Guide**
- **Application Notes**
- **Code Examples**

Quizzes

- List CPU registers and their functions
- How many clocks signals in TI MSP430G2553?
- What are Interrupt Service Routines?

Next

- ***Development***
- ***A Simple Tour of the MSP430***

Chapter 3: Development

1. **Development Environment (pg 44-46)**
2. **The C Programming Language (pg46-55)**
3. **Assembly Language (pg55-57)**
4. **Access to the Microcontroller for Programming and Debugging (pg57-59)**
5. **Demonstration Boards (pg59-63)**
6. **Hardware (pg64)**
7. **Equipment (pg65)**

Development Environment

- **Editor:** Used to write and edit programs (usually C or assembly code).
- **Assembler or compiler:** Produces executable code and checks for errors, preferably providing helpful messages.
- **Linker:** Combines compiled files and routines from libraries and arranges them for the correct types of memory in the MCU.
- **Stand-alone simulator:** Simulates the operation of the MCU on a desktop computer without the real hardware.
- **Embedded emulator/debugger:** Allows software to run on the MCU in its target system under the control of a debugger running on a desktop computer,
- **In-circuit emulator:** Specialized and expensive ($1000s) hardware that emulates the operation of the MCU under the control of debugging software running on a desktop computer.
- **Flash programmer:** Downloads ("burns") the program into flash memory on the MCU.

- **IAR EmbeddedWorkbench**

http://www.iar.com

- **Code Composer Essentials**

http://www.ti.com/tool/msp-cce430

Code Composer Essentials v3.1 SR1- Free 16KB

The C Programming Language

```
if ((P1IN & BIT3) == 0) { // Test P1.3
// Actions for P1.3 == 0
} else {
// Actions for P1.3 != 0
}
```

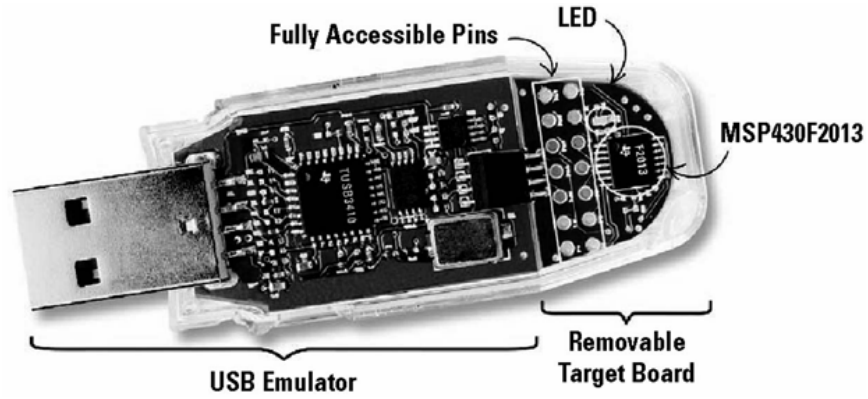
Assembly Language

mov.w #WDTPW|WDTHOLD ,& WDTCTL

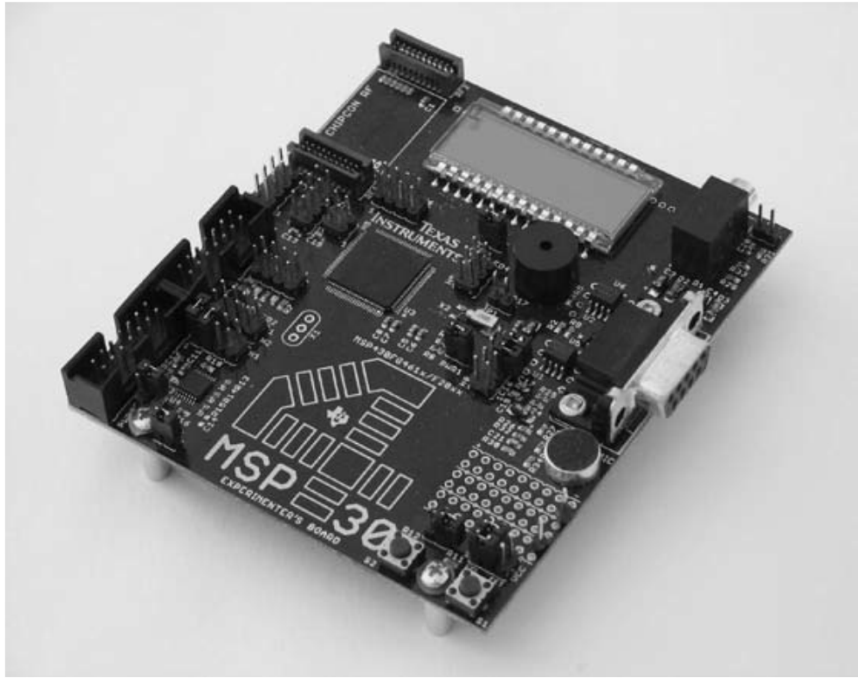
Access to the Microcontroller for Programming and Debugging

- Bootstrap loader (serial interface).
- Conventional JTAG (four wires).
- Spy-Bi-Wire JTAG (two wires).

Demonstration Boards



The TI MSP430FG4618/F2013 Experimenter's Board.

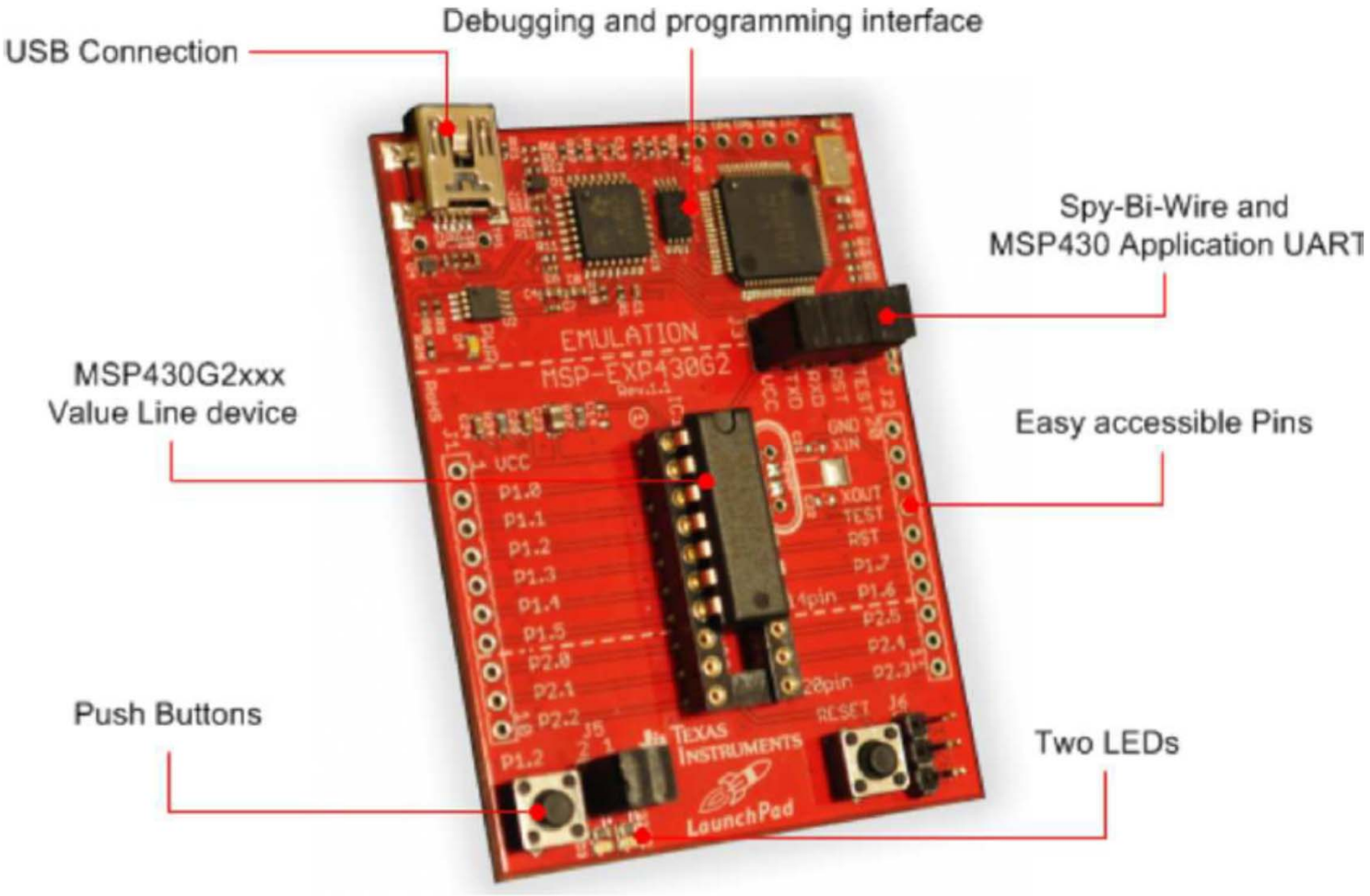


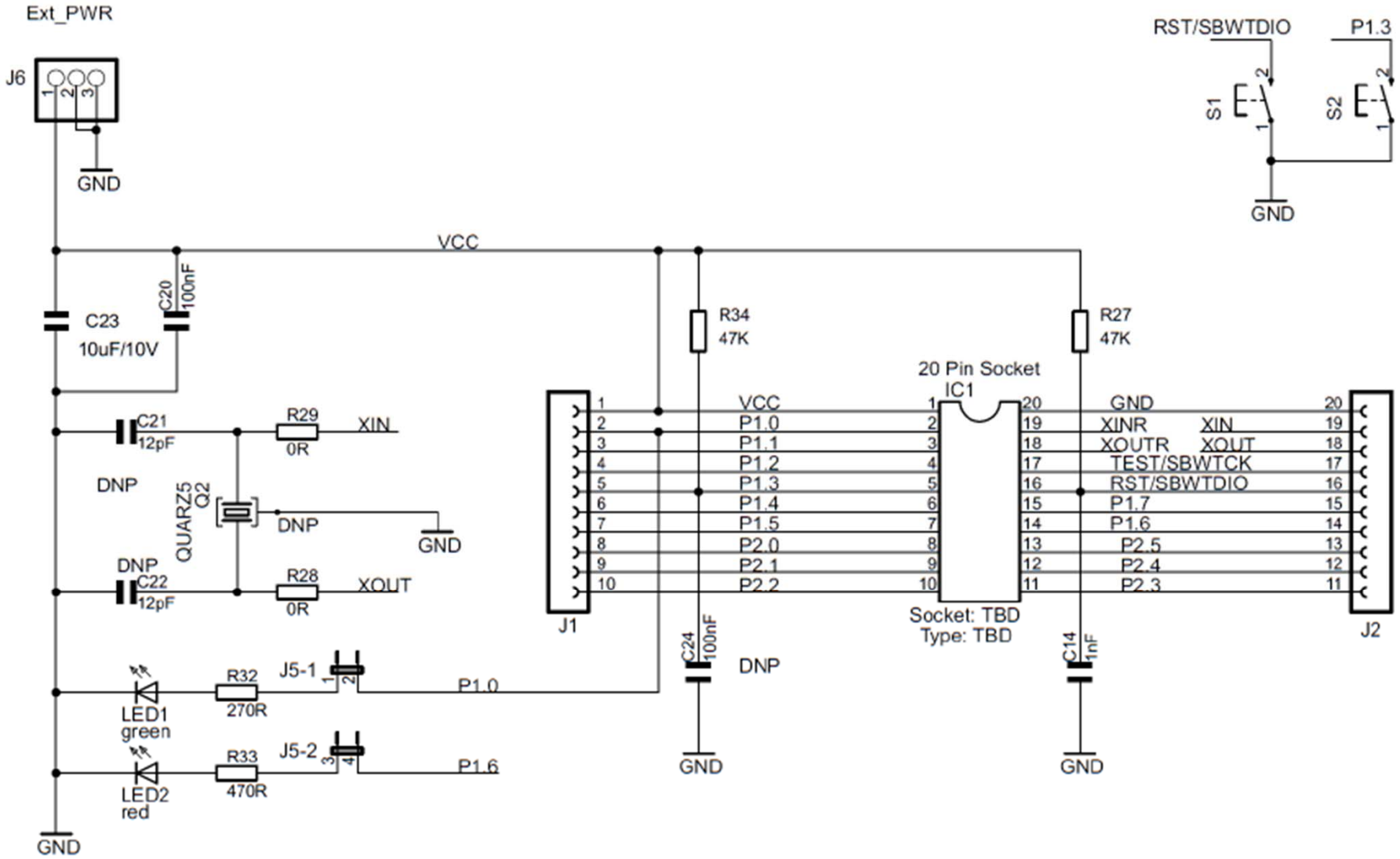
Hardware hint

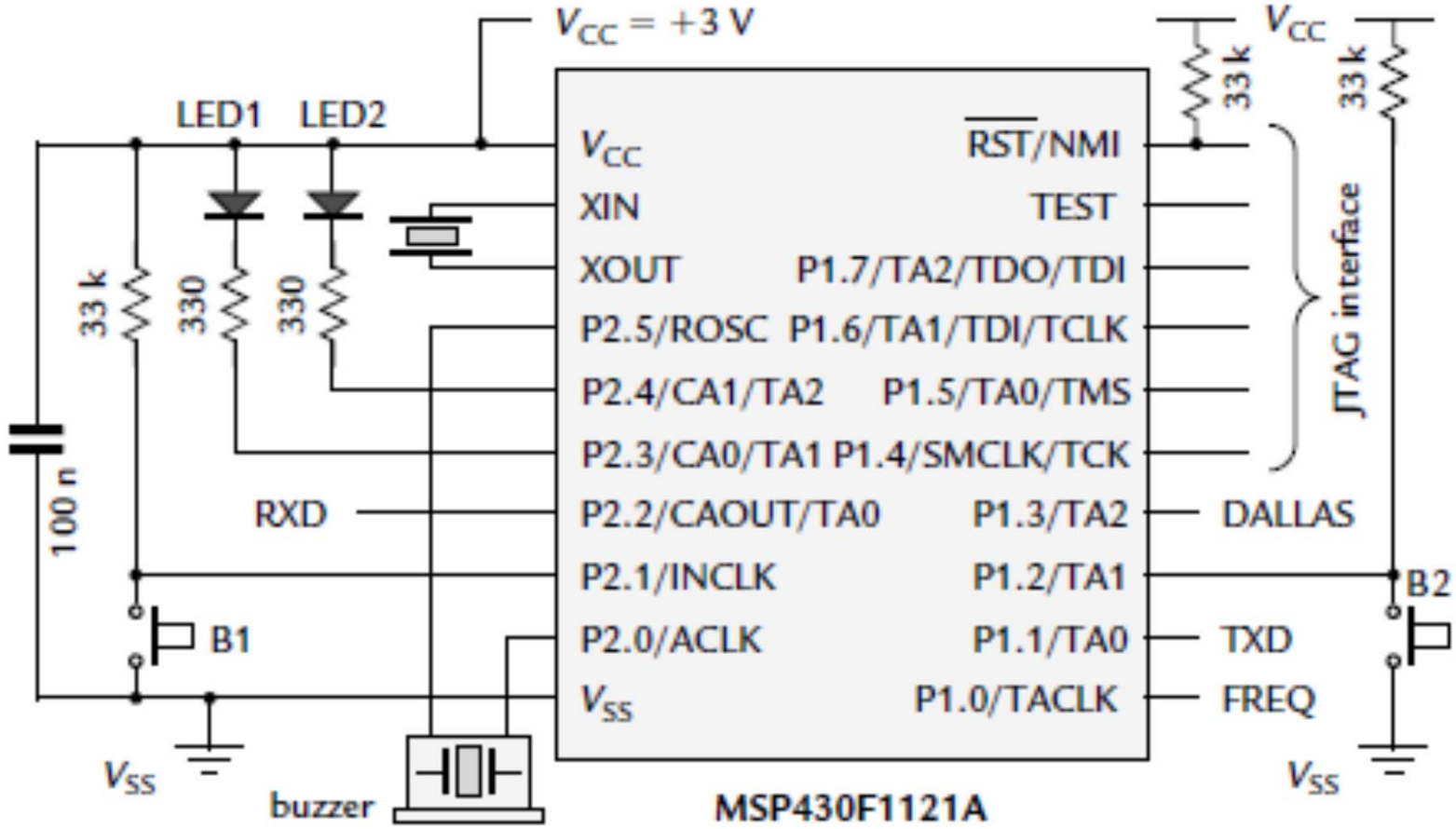
- ?

A Simple Tour of the MSP430

- **First Program**
- **Light LEDs in C**
- **Light LEDs in Assembly Language**
- **Read Input from a Switch**
- **Automatic Control: Flashing Light by Software Delay**
- **Automatic Control: Use of Subroutines**
- **Automatic Control: Flashing a Light by Polling Timer_A**
- **Header Files and Issues Brushed under the Carpet**


USB Connection
Debugging and programming interface
Spy-Bi-Wire and
MSP430 Application UART
MSP430G2xxx
Value Line device
Easy accessible Pins
Push Buttons
Two LEDs
EMULATION
MSP-EXP430G2
TEXAS INSTRUMENTS
LaunchPad


Ext_PWR
J6
GND
VCC
C23
10uF/10V
C20
100nF
C21
12pF
R29
0R
XIN
DNP
QUARZ5
Q2
DNP
GND
DNP
C22
12pF
R28
0R
XOUT
LED1
green
R32
270R
J5-1
P1.0
LED2
red
R33
470R
J5-2
P1.6
GND
R34
47K
R27
47K
20 Pin Socket
IC1
J1
VCC
P1.0
P1.1
P1.2
P1.3
P1.4
P1.5
P2.0
P2.1
P2.2
GND
XINR
XOUTR
TEST/SBWTCK
RST/SBWTDIO
P1.7
P1.6
P2.5
P2.4
P2.3
XIN
XOUT
J2
Socket: TBD
Type: TBD
C24
100nF
DNP
GND
C14
1nF
GND
RST/SBWTDIO
P1.3
S1
S2
GND


V_{CC} = +3 V
LED1
LED2
33 k
330
330
100 n
RXD
B1
V_{SS}
buzzer
V_{CC}
V_{CC}
33 k
33 k
JTAG interface
V_{CC}
XIN
XOUT
P2.5/ROSC
P2.4/CA1/TA2
P2.3/CA0/TA1
P2.2/CAOUT/TA0
P2.1/INCLK
P2.0/ACLK
V_{SS}
RST/NMI
TEST
P1.7/TA2/TDO/TDI
P1.6/TA1/TDI/TCLK
P1.5/TA0/TMS
P1.4/SMCLK/TCK
P1.3/TA2
P1.2/TA1
P1.1/TA0
P1.0/TACLK
DALLAS
TXD
FREQ
B2
V_{SS}
MSP430F1121A

First Program

```
1. #include <stdio.h>
2. void main (void)
3. {
4. printf("hello , world\n");
5. }
```

Light LEDs in C

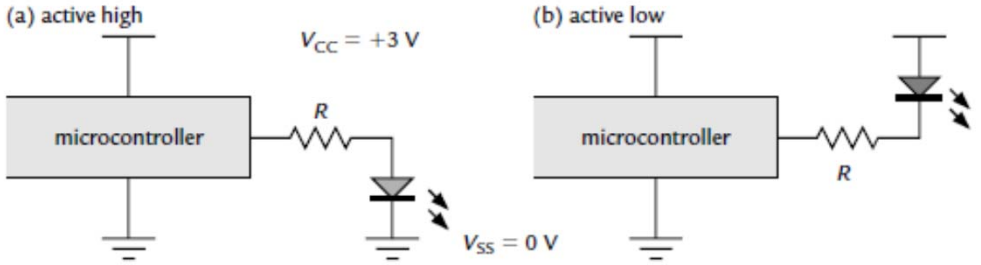


Figure 4.3: Connection of an LED to a pin of a microcontroller: (a) active high and (b) active low.

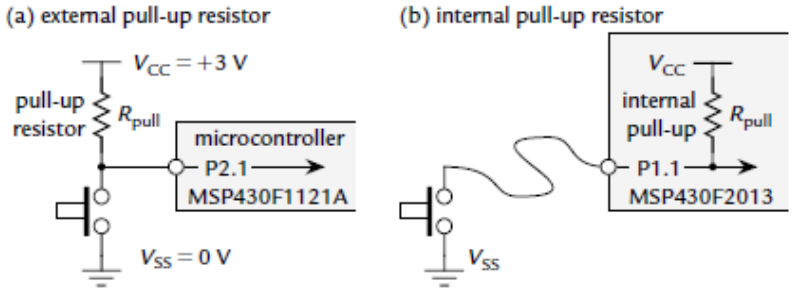
```
#include <msp430x11x1.h> // Specific device
void main (void)
{
WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
P2DIR = 0x18; // Set pins with LEDs to output , 0b00011000
P2OUT = 0x08; // LED2 (P2.4) on , LED1 (P2.3) off (active low!)
for (;;) { // Loop forever ...
} // ... doing nothing
}
```

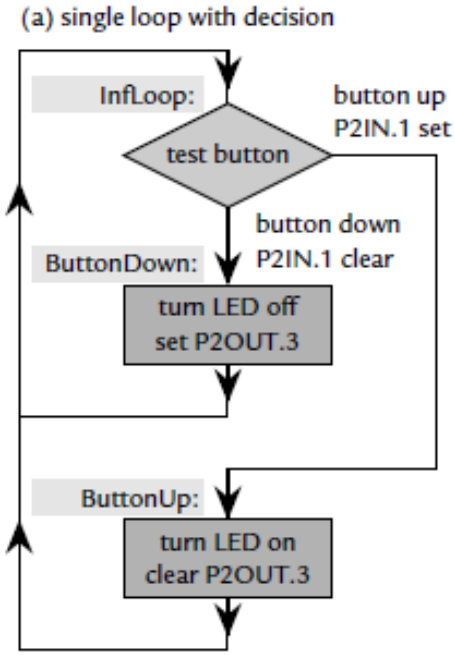
Write correct code for Launchpad !!!

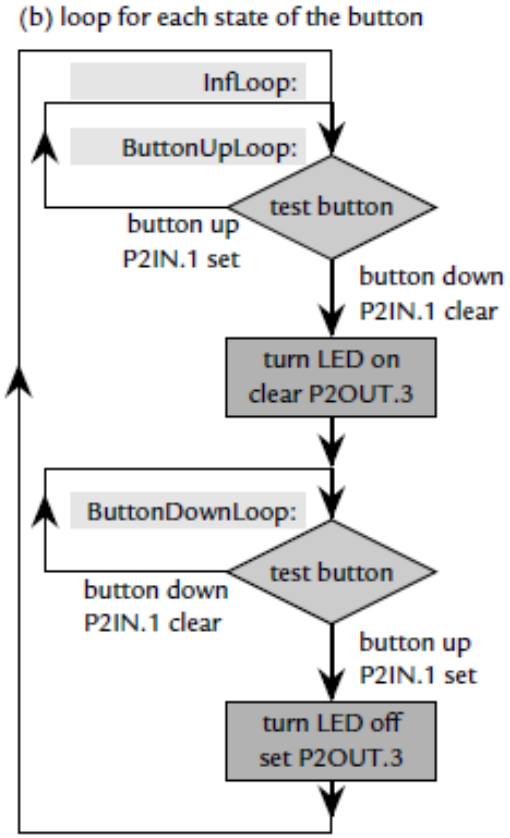
Light LEDs in Assembly Language

```
#include <msp430x11x1.h> ; Header file for this device
ORG 0xF000 ; Start of 4KB flash memory Reset: ; Execution starts here
mov.w #WDTPW|WDTHOLD ,& WDTCTL ; Stop watchdog timer
mov.b #00001000b,& P2OUT; LED2 (P2.4) on , LED1 (P2.3) off (active low!)
mov.b #00011000b,& P2DIR ; Set pins with LEDs to output
InfLoop: ; Loop forever ...
jmp InfLoop ; ... doing nothing
;-------------------------------------------------------------------
ORG 0xFFFE ; Address of MSP430 RESET Vector
DW
```

Read Input from a Switch







```
#include <msp430x11x1.h> // Specific device
// Pins for LED and button on port 2
#define LED1 BIT3
#define B1 BIT1
void main (void)
{
WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
P2OUT |= LED1; // Preload LED1 off (active low!)
P2DIR = LED1; // Set pin with LED1 to output
for (;;) { // Loop forever
if ((P2IN & B1) == 0)
    { // Is button down? (active low)
    P2OUT &= ~LED1; // Yes: Turn LED1 on (active low!)
    }
else       {
    P2OUT |= LED1; // No: Turn LED1 off (active low!)
    }
    }
}
```

Write correct code for Launchpad !!!

```
#include <io430x11x1.h> // Specific device , new format header
// Pins for LED and button
#define LED1 P2OUT_bit.P2OUT_3
#define B1 P2IN_bit.P2IN_1
void main (void)
{
WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
LED1 = 1; // Preload LED1 off (active low!)
P2DIR_bit.P2DIR_3 = 1; // Set pin with LED1 to output
for (;;) { // Loop forever
while (B1 != 0) { // Loop while button up
} // (active low) doing nothing
// actions to be taken when button is pressed
LED1 = 0; // Turn LED1 on (active low!)
while (B1 == 0) { // Loop while button down
} // (active low) doing nothing
// actions to be taken when button is released
LED1 = 1; // Turn LED1 off (active low!)
}
}
```

Write correct code for Launchpad !!!

Automatic Control: Flashing Light by Software Delay

- the simplest task is to flash an LED on and off. Let us do this with a period of about 1 Hz, which needs a delay of 0.5 s while the LED remains on or off

```
#include <msp430x11x1.h> // Specific device
// Pins for LEDs
#define LED1 BIT3
#define LED2 BIT4
// Iterations of delay loop; reduce for simulation
#define DELAYLOOPS 50000
void main (void)
{
volatile unsigned int LoopCtr; // Loop counter: volatile!
WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
P2OUT = ~LED1; // Preload LED1 on , LED2 off
P2DIR = LED1|LED2; // Set pins with LED1 ,2 to output
for (;;) { // Loop forever
for (LoopCtr = 0; LoopCtr < DELAYLOOPS; ++ LoopCtr) {
} // Empty delay loop
P2OUT ^= LED1|LED2; // Toggle LEDs
}
}
```

Automatic Control: Use of Subroutines

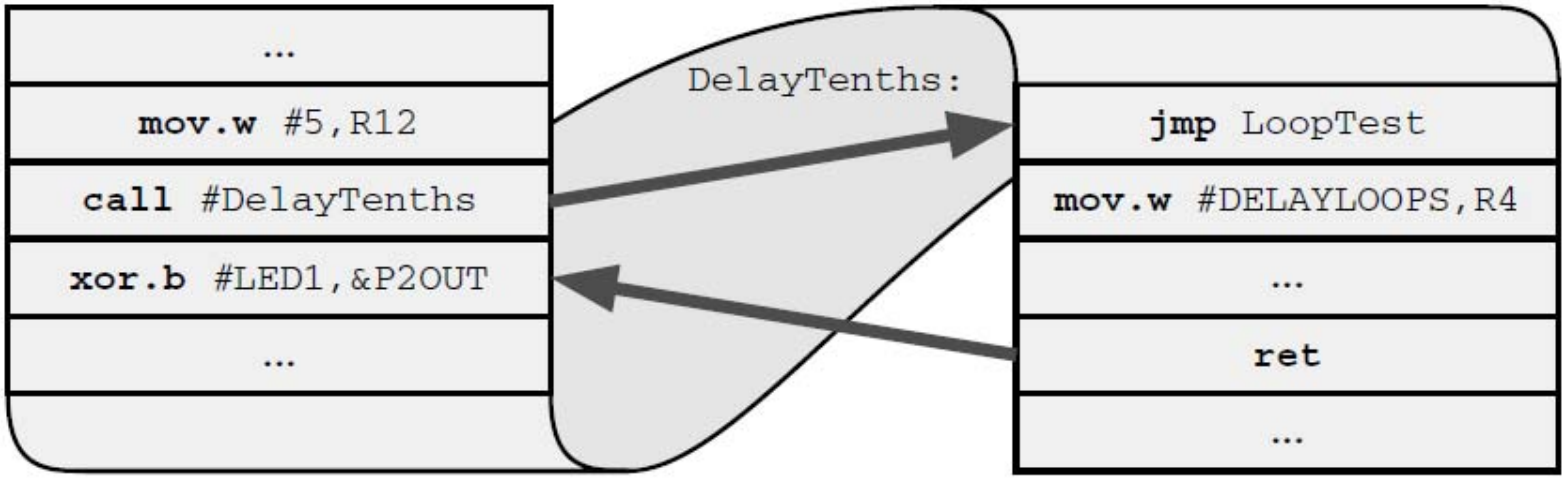
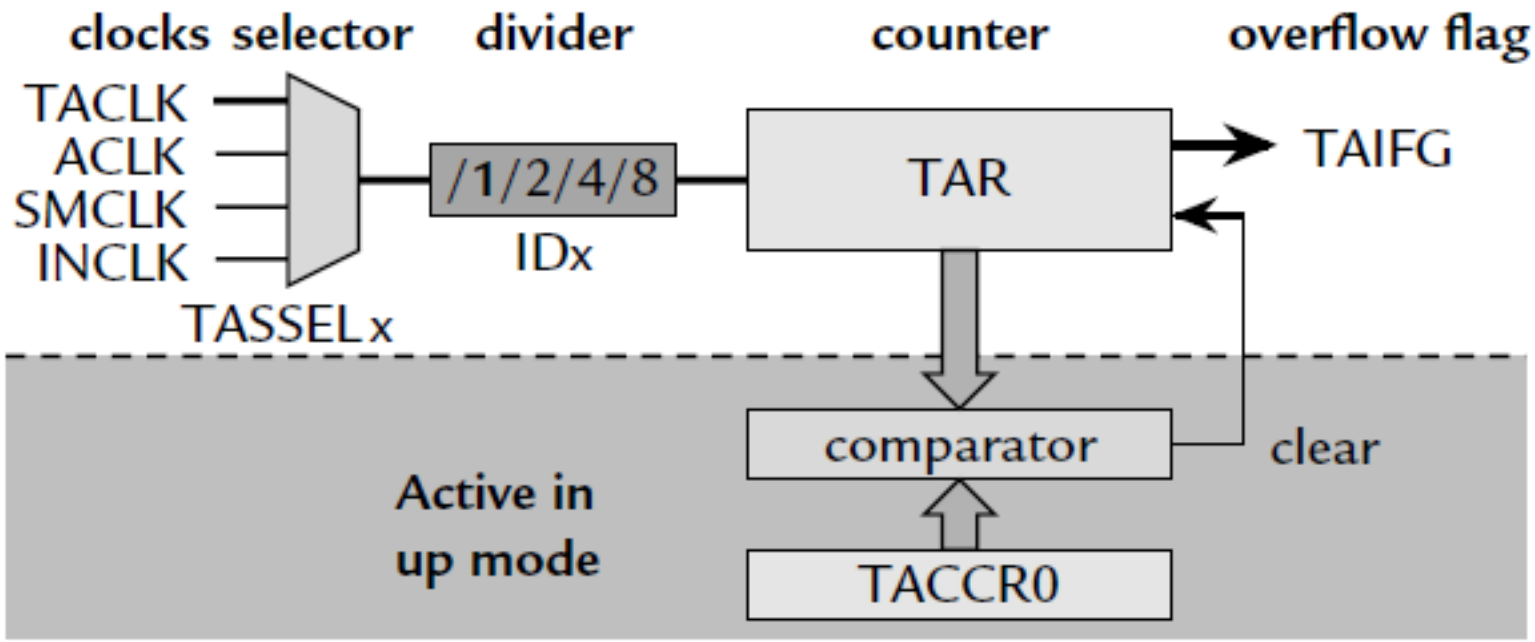


Figure 4.6: Flow of control when a subroutine is called and returns.

Automatic Control: Flashing a Light by Polling Timer_A

- Software delay loops are a waste of the processor because it is not available for more useful actions.
- Unpredictability of delays written in C.
- All microcontrollers therefore have special hardware to act as a timer.


clocks selector
divider
counter
overflow flag
TACLK
ACLK
SMCLK
INCLK
TASSELx
/1/2/4/8
IDx
TAR
TAIFG
comparator
clear
Active in
up mode
TACCR0

```
#include <io430x11x1.h> // Specific device
// Pins for LEDs
#define LED1 BIT3
#define LED2 BIT4
void main (void)
{
WDTCTL = WDTPW|WDTHOLD; // Stop watchdog timer
P2OUT = ~LED1; // Preload LED1 on , LED2 off
P2DIR = LED1|LED2; // Set pins for LED1 ,2 to output
TACTL = MC_2|ID_3|TASSEL_2|TACLR; // Set up and start Timer A
// Continuous up mode , divide clock by 8, clock from SMCLK , clear
    timer
for (;;) { // Loop forever
while (TACTL_bit.TAIFG == 0) { // Wait for overflow
} // doing nothing
TACTL_bit.TAIFG = 0; // Clear overflow flag
P2OUT ^= LED1|LED2; // Toggle LEDs
} // Back around infinite loop
}
```

Timer_A in Up Mode

- The maximum desired value of the count is programmed into another register, TACCR0. In this mode TAR starts from 0 and counts up to the value in TACCR0, after which it returns to 0 and sets TAIFG. Thus the period is TACCR0+1 counts.
- The clock has been divided down to 100 KHz so we need 50,000 counts for a delay of 0.5 s and should therefore store 49,999 in TACCR0.

Quizzes

- List all techniques to flash a led?

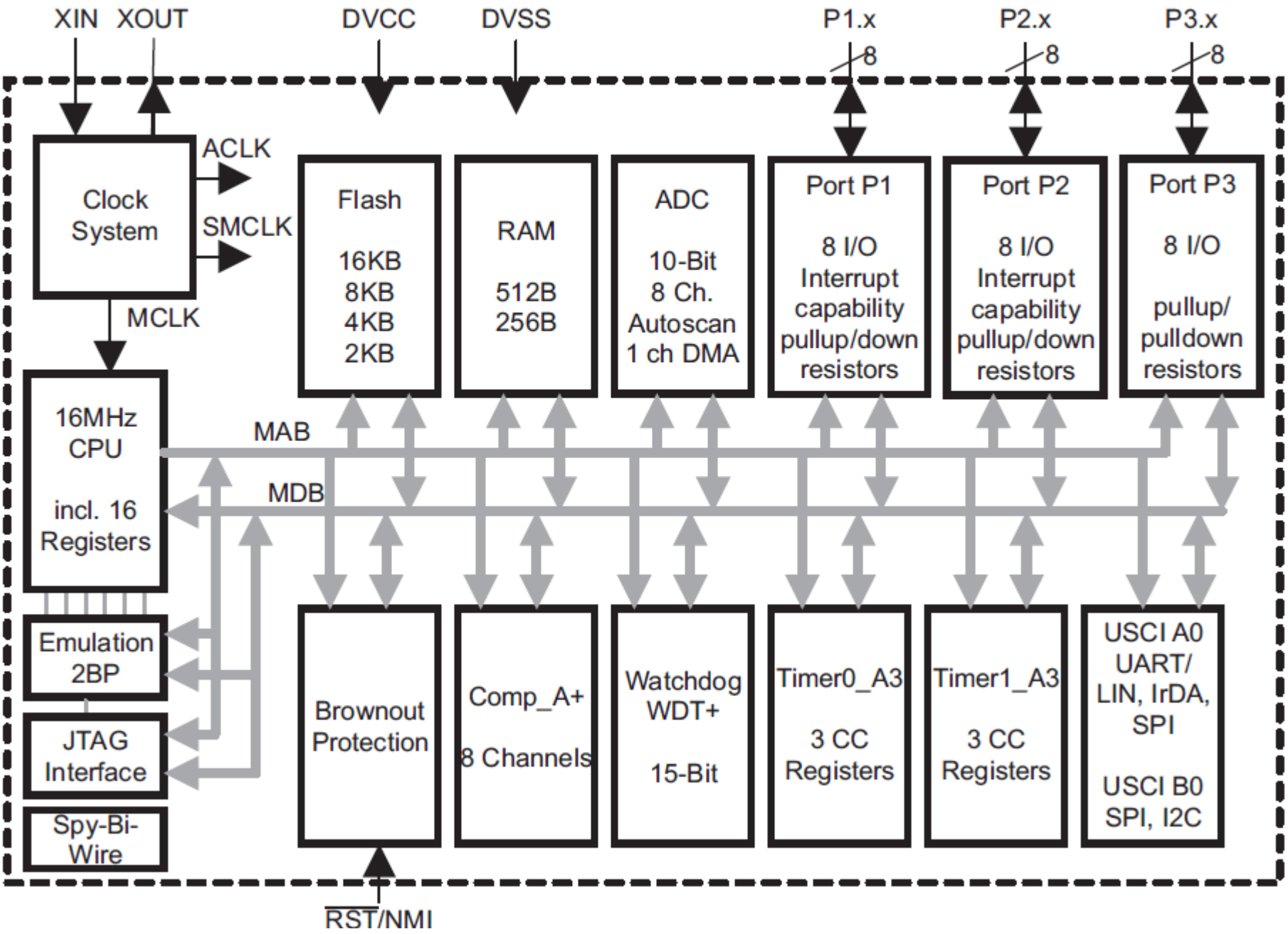
What is next?

- ***CHAPTER 5: Architecture of the MSP430 Processor (Page 119-175)***

Chapter 5: Architecture of the MSP430 Processor

1. **Central Processing Unit (pg 120-125)**
2. **Addressing Modes (pg125-131)**
3. **Constant Generator and Emulated Instructions (pg131-132)**
4. **Instruction Set(pg132-146)**
5. **Examples(pg146-153)**
6. **Reflections on the CPU and Instruction Set (pg153-157)**
7. **Resets (pg157-163)**
8. **Clock System (pg163- 175)**

Functional Block Diagram, MSP430G2x53



Central Processing Unit

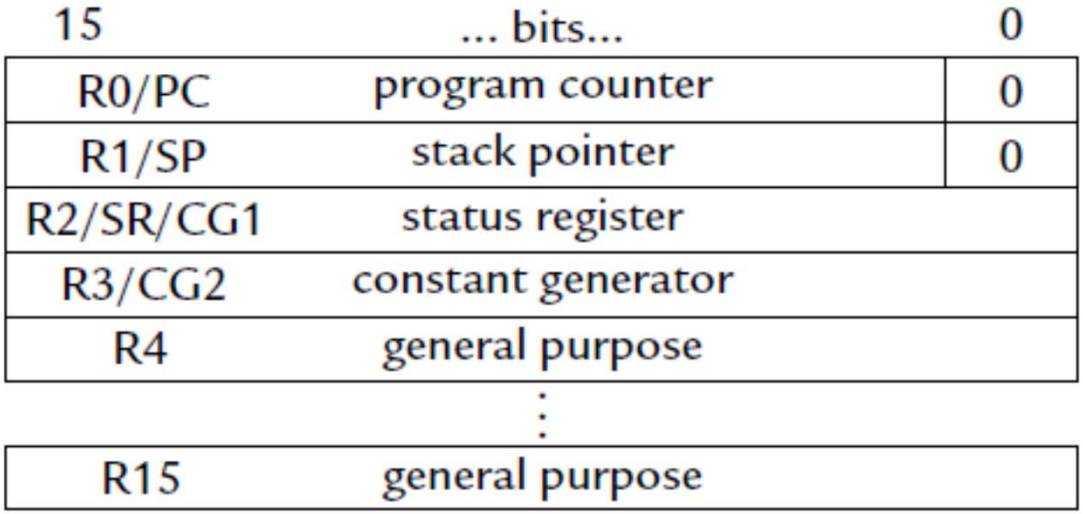


Figure 5.1: Registers in the CPU of the MSP430.

- **Program Counter (PC)**
- **Stack Pointer (SP)**
- **Status Register (SR)**
- **Constant Generators**
- **General-Purpose Registers**

Addressing Modes

- Just only for Assembly Programmers

Constant Generator and Emulated Instructions

- Just only for Assembly Programmers

Instruction Set

- **Movement Instructions**

  mov.w src ,dst *;// dst=src; //c -prg*
- **Arithmetic and Logic Instructions with Two Operands**

  add.w src ,dst // dst += src; *//c -prg*
- **Shift and Rotate Instructions**

  and.w src ,dst *; // dst &= src*
- **Flow of Control**
- **…**

Examples

add.w src ,dst *; add dst += src*

addc.w src ,dst *; add with carry dst += (src + C)*

adc.w dst *; add carry bit dst += C emulated*

sub.w src ,dst *; subtract dst -= src*

subc.w src ,dst *; subtract with borrow dst -= (src + ~C)*

Reflections on the CPU and Instruction Set

- **Simplicity**
- **Registers of the CPU**
- **Is the MSP430 a RISC—and Should It Be?**
- **Addressing Modes**

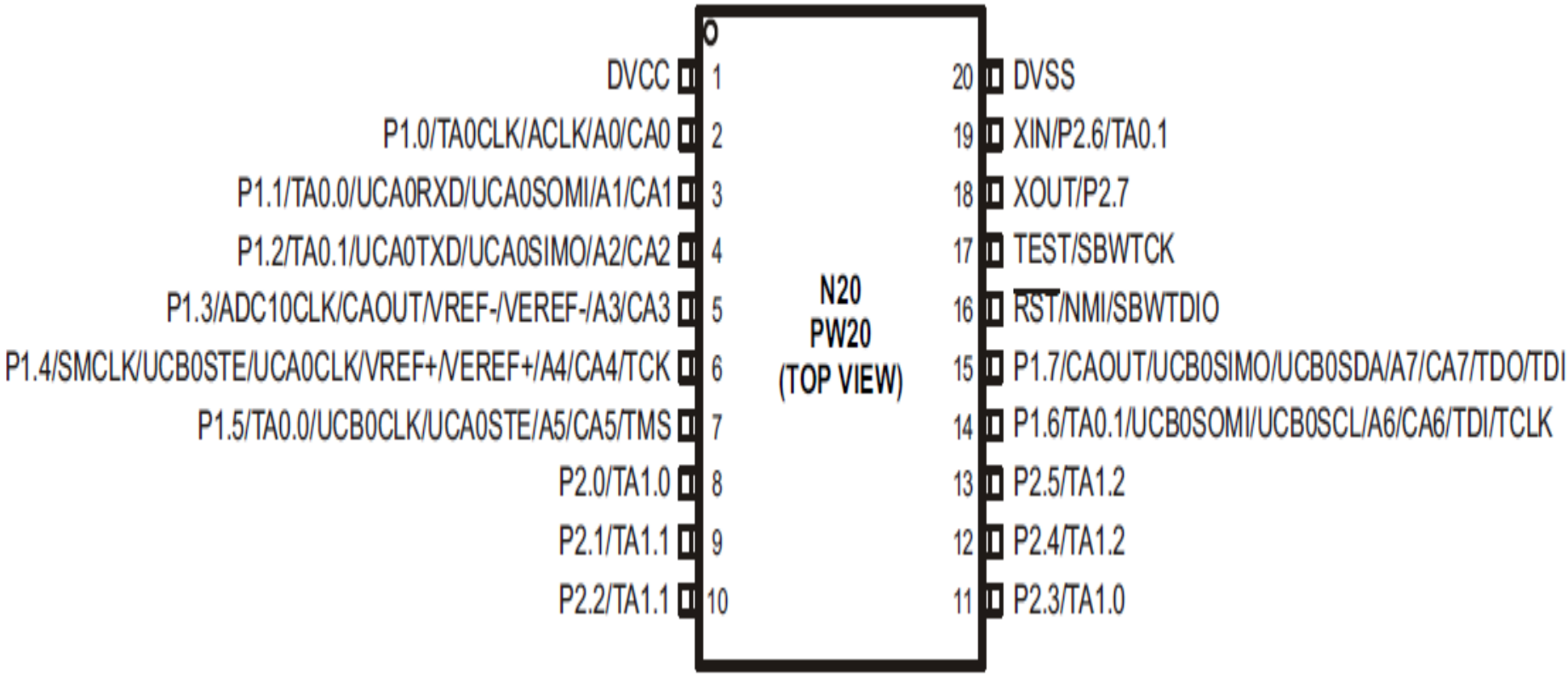
Resets

- A *reset* is a sequence of operations that puts the device into a well-defined state, from which the user's program may start
  - ***Power-on Reset (POR)***
    - The device is powered up. More generally, a POR is raised if the supply voltage drops to so low a value that the device may not work correctly: a brownout.
  - ***Power-up Clear (PUC)***
    - The watchdog timer overflows in watchdog mode.

Clock System

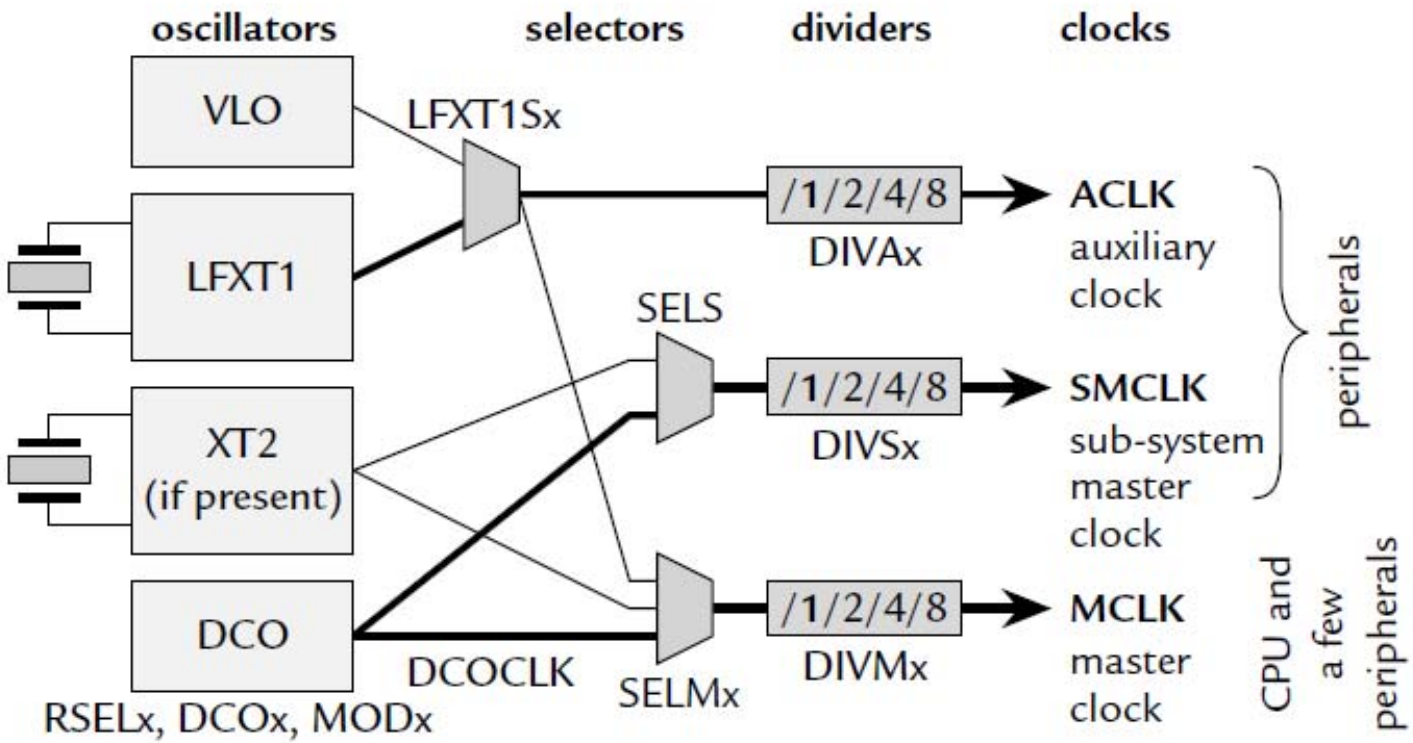
- **Master clock, MCLK** is used by the CPU and a few peripherals.
- **Sub-system master clock, SMCLK** is distributed to peripherals.
- **Auxiliary clock, ACLK** is also distributed to peripherals.

Device Pinout, MSP430G2x13 and MSP430G2x53, 20-Pin Devices, TSSOP and PDIP



NOTE: ADC10 is available on MSP430G2x53 devices only.

NOTE: The pulldown resistors of port P3 should be enabled by setting P3REN.x = 1.


oscillators
selectors
dividers
clocks
VLO
LFXT1Sx
LFXT1
/1/2/4/8
DIVAx
ACLK
auxiliary
clock
SELS
XT2
(if present)
/1/2/4/8
DIVSx
SMCLK
sub-system
master
clock
peripherals
DCO
DCOCLK
SELMx
/1/2/4/8
DIVMx
MCLK
master
clock
CPU and
a few
peripherals
RSELx, DCOx, MODx

Low- or high-frequency crystal oscillator, LFXT1: Available in all devices. An external clock signal can be used instead of a crystal if it is important to synchronize the MSP430 with other devices in the system.

High-frequency crystal oscillator, XT2: Similar to LFXT1 except that it is restricted to high frequencies.

Internal very low-power, low-frequency oscillator, VLO: Available in only the more recent MSP430F2xx devices. It provides an alternative to LFXT1 when the accuracy of a crystal is not needed.

Digitally controlled oscillator, DCO: Available in all devices and one of the highlights of the MSP430. It is basically a highly controllable *RC* oscillator that starts in less than1s in newer devices.

- **Control of the Clock Module through the Status Register**

- **CPUOFF** disables MCLK, which stops the CPU and any peripherals that use]MCLK.
- **SCG1** disables SMCLK and peripherals that use it.
- **SCG0** disables the DC generator for the DCO (disables the FLL in the MSP430x4xx family).
- **OSCOFF** disables VLO and LFXT1.

Quizzes

What are

Master clock (MCLK)?

Sub-system master clock(SMCLK)?

Auxiliary clock (ACLK)?

What is next?

- ***CHAPTER 6: Functions, Interrupts, and Low-Power Modes (Page 177-205)***

Embedded Systems

Hệ thống nhúng

Ts. Lê Mạnh Hải

Khoa CNTT,

ĐH Kỹ thuật Công nghệ TP HCM

Chapter 6: Functions, Interrupts, and Low-Power Modes

1. **Functions and Subroutines**
2. **What Happens when a Subroutine Is Called?**
3. **Storage for Local Variables**
4. **Passing Parameters**
5. **Example Mixing C and Assembly Language**
6. **Interrupts**
7. **What Happens when an Interrupt Is Requested?**
8. **Interrupt Service Routines**
9. **Issues Associated with Interrupts**
10. **Low-Power Modes of Operation**

Functions and Subroutines

- A well-structured program should be divided into separate modules—functions in C or subroutines in assembly language
- It is particularly important to understand the central role of the stack
- Interrupts are a major feature of most embedded software. They are vaguely like functions that are called by hardware rather than software.
- The final topic in this chapter is the range of low-power modes of operation. They are described here because the MSP430 needs an interrupt to wake it from a low-power mode.

Functions and Subroutines

- It makes programs easier to write and more reliable to test and maintain
- Functions can readily be reused and incorporated into libraries, provided that their documentation is clear

What Happens when a Subroutine Is Called?

- The address of the subroutine is then loaded into the PC and execution continues from there.
- At the end of the subroutine the ret instruction pops the return address off the stack into the PC so that execution resumes with the instruction following the call of the subroutine.
- The return operation is so straightforward that ret is emulated with a mov instruction

Storage for Local Variables

- CPU registers are simple and fast.
- A second approach is to use a fixed location in RAM,
- The third approach is to allocate variables on the stack and is generally used when a program has run out of CPU registers. This can be slow in older designs of processors, but the MSP430 can address variables on the stack with its general indexed and indirect modes. Of course it is still faster to use registers in the CPU when these are available.

Passing Parameters

- They are like functions but with the critical distinction that they are requested by hardware at unpredictable times rather than called by software in an orderly manner.

ExamplMixing C and Assembly Language

Interrupts

- Interrupts are commonly used for a range of applications:
  - Urgent tasks that must be executed promptly at higher priority than the main code. However, it is even faster to execute a task directly by hardware if this is possible.
  - Infrequent tasks, such as handling slow input from humans. This saves the overhead of regular polling.
  - Waking the CPU from sleep.
  - Calls to an operating system.

Interrupts Service Routine

- The code to handle an interrupt is called an *interrupt handler* or *interrupt service routine* (ISR).
- It looks superficially like a function but there are a few crucial modifications.
  - The feature that interrupts arise at unpredictable times means that an ISR must carry out its action and clean up thoroughly so that the main code can be resumed without error—it should not be able to tell that an interrupt occurred.

Interrupt Flags

- Each interrupt has a flag, which is raised (set) when the condition for the interrupt occurs. For example, Timer_A sets the TAIFG flag in the TACTL register when the counter TAR returns to 0.
- Most interrupts are *maskable*, which means that they are effective only if the general interrupt enable (GIE) bit is set in the status register (SR). They are ignored if GIE is clear.

Interrupt vector

- The MSP430 uses *vectored* interrupts, which means that the address of each ISR—its vector—is stored in a *vector table* at a defined address in memory.
- Each interrupt vector has a distinct priority, which is used to select which vector is taken if more than one interrupt is active when the vector is fetched. The priorities are fixed in hardware and cannot be changed by the user.

What Happens when an Interrupt Is Requested?

1. Any currently executing instruction is completed if the CPU was active when the interrupt was requested. MCLK is started if the CPU was off.
2. The PC, which points to the next instruction, is pushed onto the stack.
3. The SR is pushed onto the stack.
4. The interrupt with the highest priority is selected if multiple interrupts are waiting for service.
5. The interrupt request flag is cleared automatically for vectors that have a single source. Flags remain set for servicing by software if the vector has multiple sources, which applies to the example of TAIFG.
6. The SR is cleared, which has two effects.
7. The interrupt vector is loaded into the PC and the CPU starts to execute the interrupt service routine at that address.

latency

- The delay between an interrupt being requested and the start of the ISR is called the *latency*.
- If the CPU is already running it is given by the time to execute the current instruction, which might only just have started when the interrupt was requested, plus the six cycles needed to execute the launch sequence.

Interrupt Service Routines

- **Interrupt Service Routines in C**

```
void main (void)
{
WDTCTL = WDTPW|WDTHOLD; // Stop watchdog timer
P2OUT = ~LED1; // Preload LED1 on , LED2 off
P2DIR = LED1|LED2; // Set pins with LED1 ,2 to output
TACCR0 = 49999; // Upper limit of count for TAR
TACCTL0 = CCIE; // Enable interrupts on Compare 0
TACTL = MC_1|ID_3|TASSEL_2|TACLR; // Set up and start Timer A
// "Up to CCR0" mode , divide clock by 8, clock from SMCLK , clear timer
__enable _interrupt (); // Enable interrupts (intrinsic)
for (;;) { // Loop forever doing nothing
} // Interrupts do the work
}
// -------------------------------------------------------------------
// Interrupt service routine for Timer A channel 0
#pragma vector = TIMERA0_VECTOR
__interrupt void TA0_ISR (void)
{
P2OUT ^= LED1|LED2; // Toggle LEDs
}
```

Nonmaskable Interrupts

- There are a few small differences in the handling of nonmaskable interrupts Three modules can request a nonmaskable interrupt:
  - Oscillator fault, OFIFG.
  - Access violation to flash memory, ACCVIFG.
  - An active edge on the external RST/NMI pin if it has been configured for interrupts rather than reset.

Issues Associated with Interrupts

- ***Keep Interrupt Service Routines Short (why?)***
- ***Configure Interrupts Carefully***
- ***Define All Interrupt Vectors***
- ***The Shared Data Problem***

Low-Power Modes of Operation

- **Active mode:** CPU, all clocks, and enabled modules are active, $I \approx 300$uA. The MSP430 starts up in this mode, which must be used when the CPU is required. An interrupt automatically switches the device to active mode. The current can be reduced by running the MSP430 at the lowest supply voltage consistent with the frequeny of MCLK; *V*CC can be lowered to 1.8V for *f*DCO = 1MHz, giving $I \approx 200$uA.
- **LPM0:** CPU and MCLK are disabled, SMCLK and ACLK remain active, $I \approx 85$uA. This is used when the CPU is not required but some modules require a fast clock from SMCLK and the DCO.
- **LPM3:** CPU, MCLK, SMCLK, and DCO are disabled; only ACLK remains active; $I \approx 1$uA. This is the standard low-power mode when the device must wake itself at regular intervals and therefore needs a (slow) clock. It is also required if the MSP430 must maintain a real-time clock. The current can be reduced to about 0.5uAby using the VLO instead of an external crystal in a MSP430F2xx if *f*ACLK need not be accurate.
- **LPM4:** CPU and all clocks are disabled, $I \approx 0.1$uA. The device can be wakened only by an external signal. This is also called *RAM retention mode*.

_lowpower_mode_3 ();

Quizzes

Why Interrupt is important for embedded system?

What are flags?

What is GIE bit?

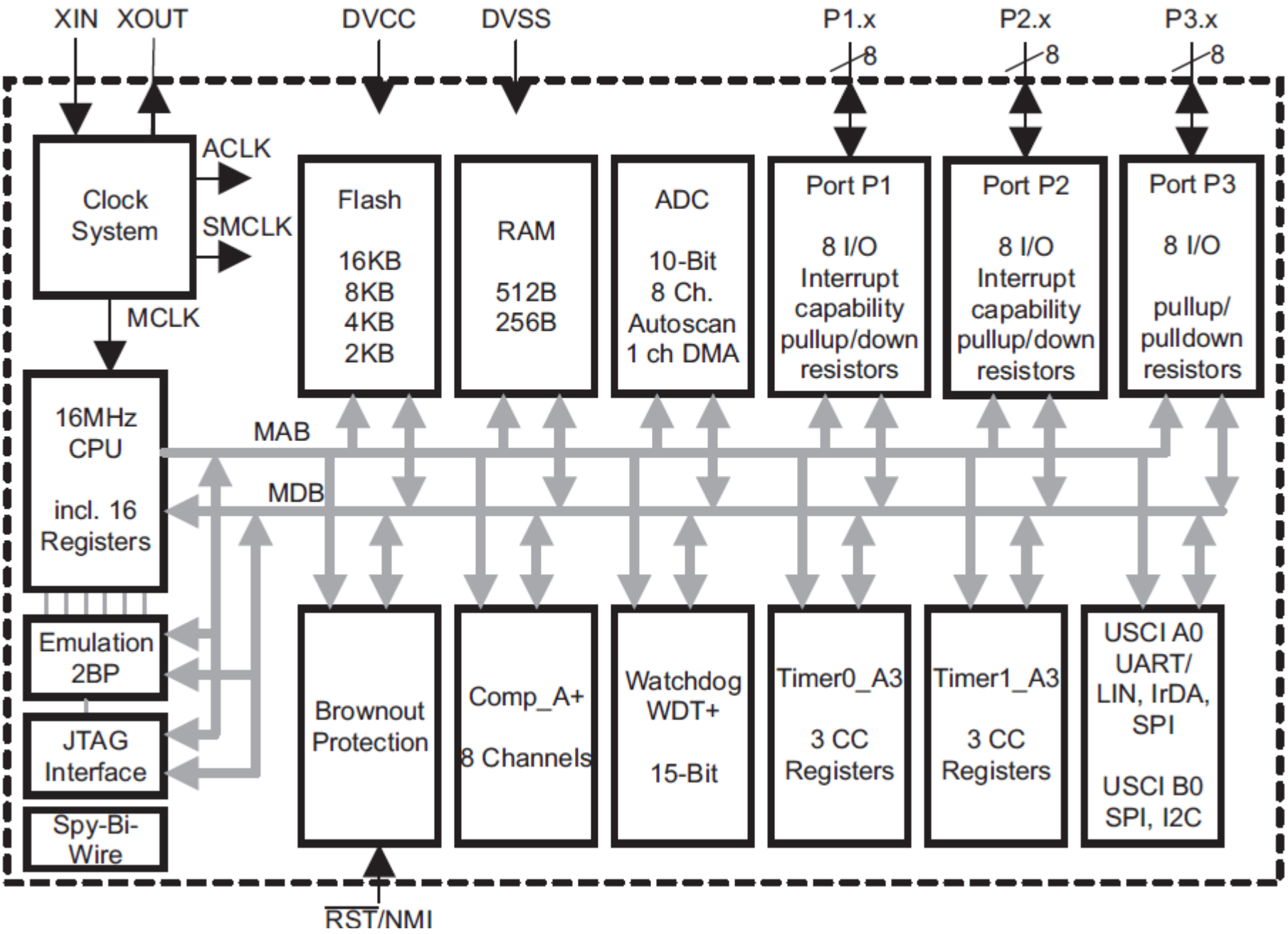
What is next?

- ***CHAPTER 7: Digital Input, Output, and Displays (Page 207-274)***

Chapter 7: Digital Input, Output, and Displays

1. Digital Input and Output: Parallel Ports (208-216)
2. Digital Inputs (216-225)
3. Switch Debounce (225-238)
4. Digital Outputs (238-243)
5. Interface between 3V and 5V Systems (243-247)
6. Driving Heavier Loads (247-252)
7. Liquid Crystal Displays (252-256)
8. Driving an LCD from an MSP430x4xx (256-264)
9. Simple Applications of the LCD (264-275)

Functional Block Diagram, MSP430G2x53

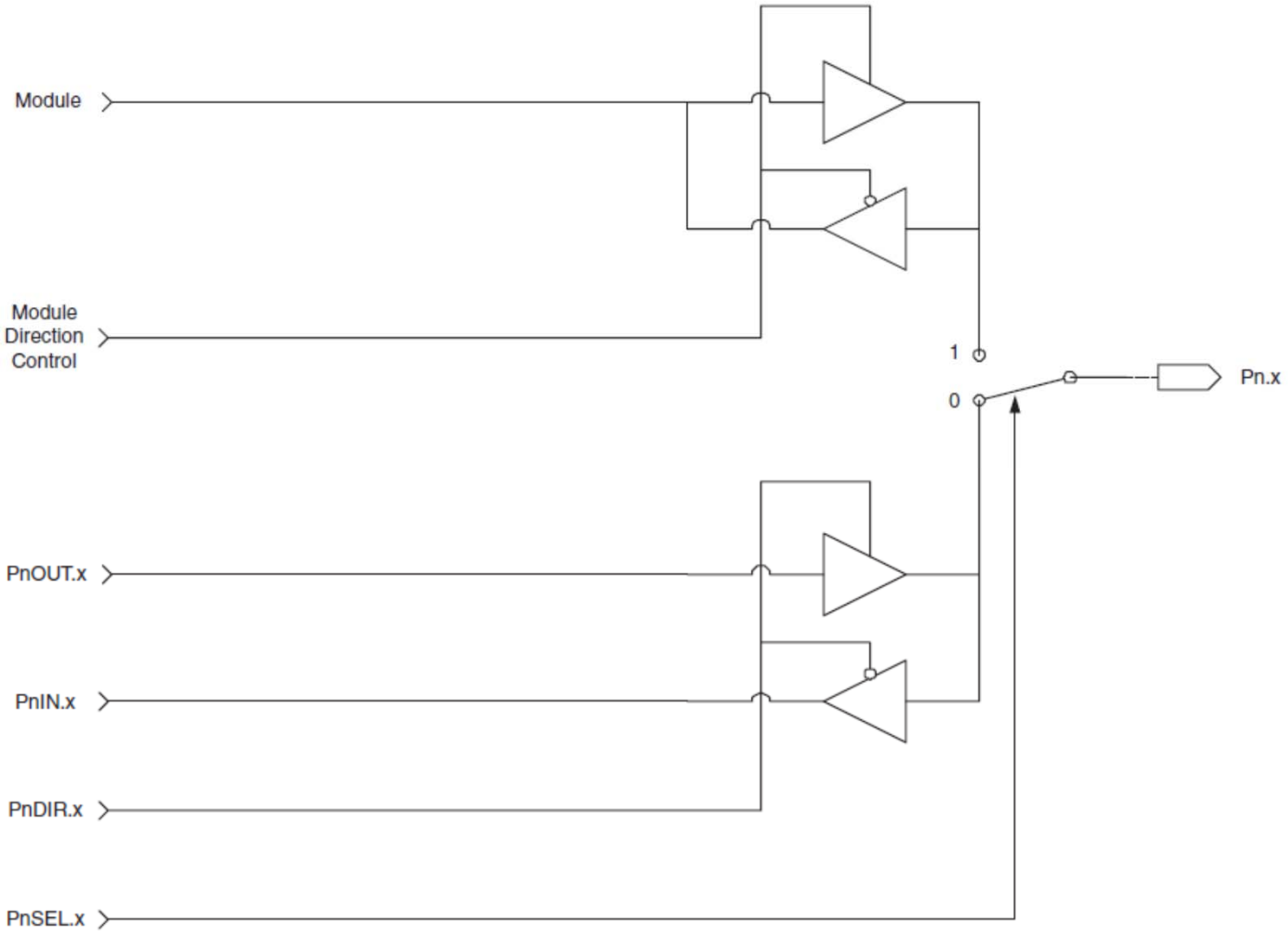


Digital Input and Output: Parallel Ports

- The most straightforward form of input and output is through the digital input/output ports using binary values (low or high, corresponding to 0 or 1).
- We already used these for driving LEDs and reading switches. In this section we look at their wider capabilities.

Digital Input and Output

- There are 10–80 input/output pins on different devices in the current portfolio of MSP430s;
- The F20xx has one complete 8-pin port and 2 pins on a second port, whilethe largest devices have ten full ports.
- Almost all pins can be used either for digital input/output or for other functions and their operation must be configured when the device starts up.
- This can be tricky. For example, pin P1.0 on the F2013 can be a digital input, digital output, input TACLK, output ACLK, or analog input A0+.


Module
Module
Direction
Control
1
0
Pn.x
PnOUT.x
PnIN.x
PnDIR.x
PnSEL.x

Digital Input and Output

- **Port P1 input, P1IN:** reading returns the logical values on the inputs if they are configured for digital input/output.
- **Port P1 output, P1OUT:** writing sends the value to be driven to each pin if it is configured as a digital output. If the pin is not currently an output, the value is stored in a buffer and appears on the pin if it is later switched to be an output. This register is not initialized and you should therefore write to P1OUT before configuring the pin for output.

Digital Input and Output

•**Port P1 direction, P1DIR:** clearing a bit to 0 configures a pin as an input, which is the default in most cases.Writing a 1 switches the pin to become an output. This is for digital input and output; the register works differently if other functions are selected using P1SEL.

•**Port P1 resistor enable, P1REN:** setting a bit to 1 activates a pull-up or pull-down resistor on a pin. Pull-ups are often used to connect a switch to an input as in the section "Read Input from a Switch" on page 80. The resistors are inactive by default (0). When the resistor is enabled (1), the corresponding bit of the P1OUT register selects whether the resistor pulls the input up to *V*CC (1) or down to *V*SS (0).

•**Port P1 selection, P1SEL:** selects either digital input/output (0, default) or an alternative function (1). Further registers may be needed to choose the particular function.

Digital Input and Output

- **Port P1 interrupt enable, P1IE:** enables interrupts when the value on an input pin changes. This feature is activated by setting appropriate bits of P1IE to 1. Interrupts are off (0) by default. The whole port shares a single interrupt vector although pins can be enabled individually.
- **Port P1 interrupt edge select, P1IES:** can generate interrupts either on a positive edge (0), when the input goes from low to high, or on a negative edge from high to low (1). This register is not initialized and should therefore be set up before interrupts are enabled.
- **Port P1 interrupt flag, P1IFG:** a bit is set when the selected transition has been detected on the input. In addition, an interrupt is requested if it has been enabled. These bits can also be set by software, which provides a mechanism for generating a software interrupt (SWI).

Circuit of an Input/Output Pin

- It is a lot easier to understand the peculiarities of input and output if you have a rough idea of the circuit.

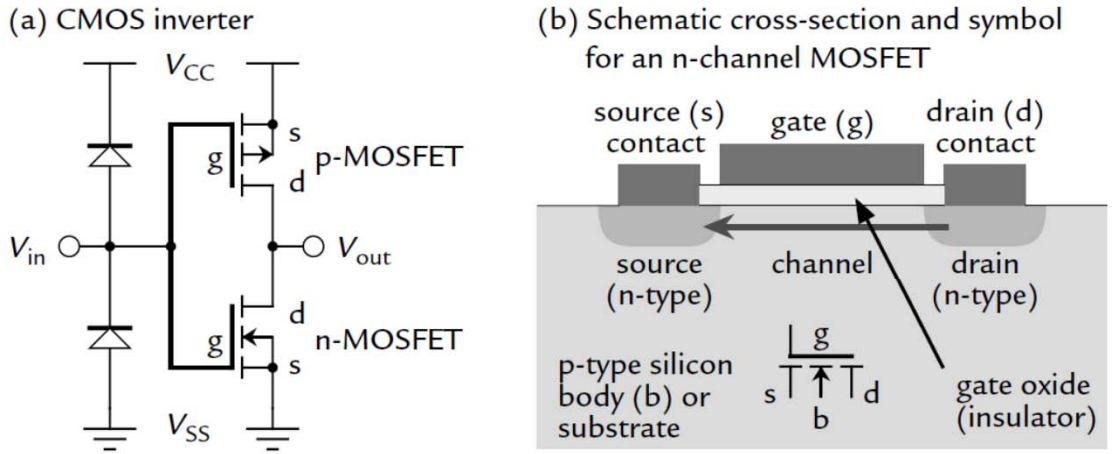


Figure 7.1: (a) Circuit of a simple CMOS inverter including the input protection diodes. (b) Schematic cross-section of an n-channel MOSFET.

Configuration of Unused Pins

- *Unused pins must never be left unconnected in their default state as inputs.*
- This follows a general rule that inputs to CMOS must never be left unconnected or “floating.” A surprising number of problems can be caused by floating inputs.
- The most trivial is that the input circuit draws an excessive current from the power supply

Configuration of Unused Pins

There are three ways of avoiding these problems:

1. Wire the unused pins externally to a well-defined voltage, either *V*SS or *V*CC, and configure them as inputs. The danger with this is that you might damage the MCU if the pins are accidentally configured as outputs.
2. Leave the pins unconnected externally but connect them internally to either *V*SS or *V*CC by using the pull-down or pull-up resistors. They are again configured as inputs. I prefer this approach but it is restricted to the MSP430F2xx family because the others lack internal pull resistors.
3. **Leave the pins unconnected and configure them as outputs.**

Digital Inputs

- **Interrupts on Digital Inputs**
  - Interrupts for port P1 are controlled by the registers P1IE and P1IES

Interrupt vector

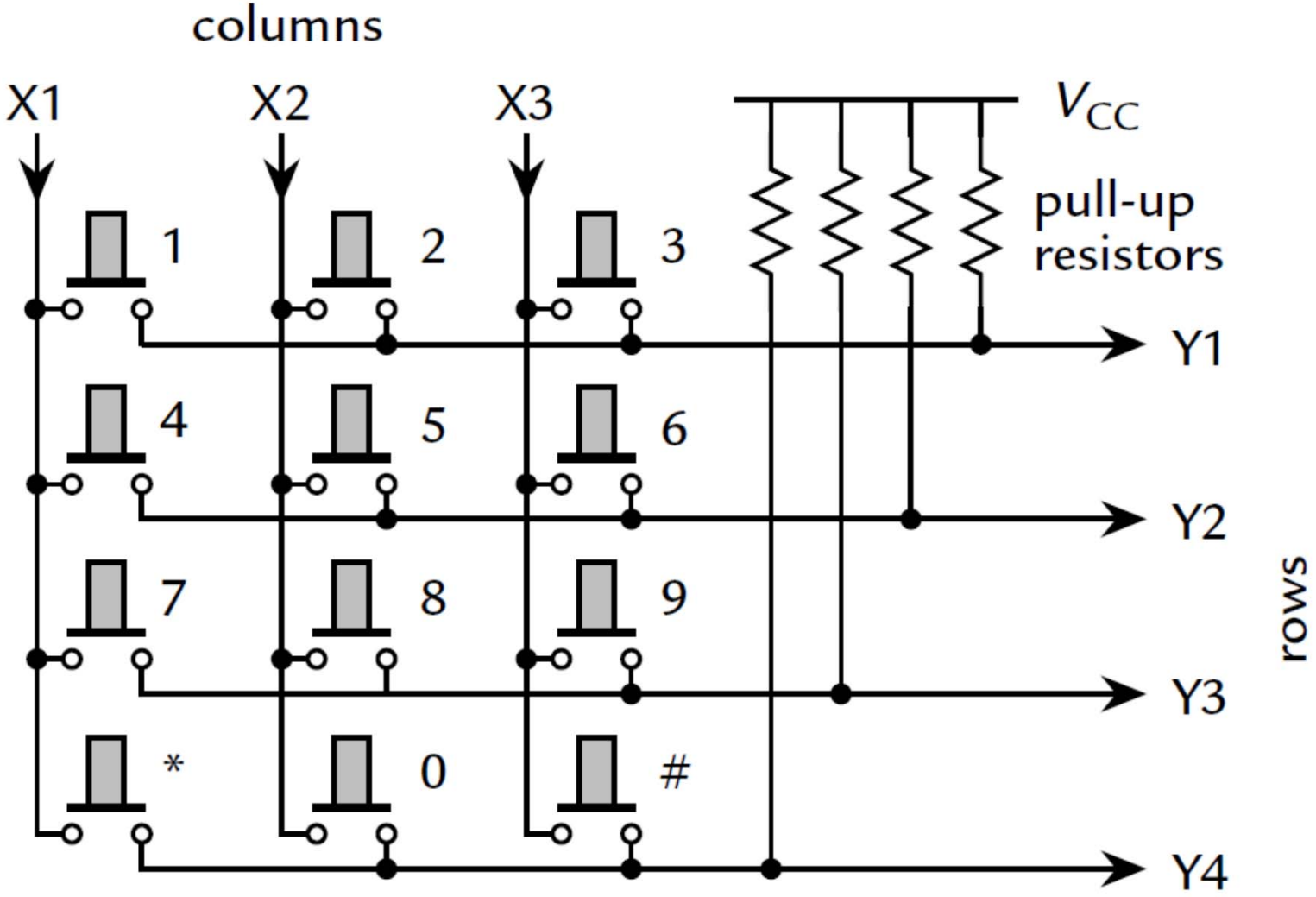
- There is a single vector for each port, so the user must check P1IFG to determine the bit that caused the interrupt.
- This bit must be cleared explicitly; it does not happen automatically as with interrupts that have a single source.

```
//The use of interrupts is illustrated in Listing 7.1, which is perhaps the ultimate
    development of the programs to light an LED when a button is pressed.
#include <io430x11x1.h> // Specific device
#include <intrinsics.h> // Intrinsic functions
void main (void)
{
WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
P2OUT_bit.P2OUT_3 = 1; // Preload LED1 off (active low!)
P2DIR_bit.P2DIR_3 = 1; // Set pin with LED1 to output
P2IE_bit.P2IE_1 = 1; // Enable interrupts on edge
P2IES_bit.P2IES_1 = 1; // Sensitive to negative edge (H->L)
do {
P2IFG = 0; // Clear any pending interrupts ...
} while (P2IFG != 0); // ... until none remain
for (;;) { // Loop forever (should not need)
__low_power_mode_4 (); // LPM4 with int'pts , all clocks off
} // (RAM retention mode)
}
```

```
#pragma vector = PORT2_VECTOR
__interrupt void PORT2_ISR (void)
{
P2OUT_bit.P2OUT_3 ^= 1; // Toggle LED
P2IES_bit.P2IES_1 ^= 1; // Toggle edge
   sensitivity
do {
P2IFG = 0; // Clear any pending interrupts ...
} while (P2IFG != 0); // ... until none remain
}
```

Multiplexed Inputs: Scanning a Matrix Keypad

- Many products require numerical input and provide a keypad for the user.
- These often have 12 keys, like a telephone, or more. An individual connection for each switch would use an exorbitant number of pins so they are usually arranged as a matrix instead.


columns
X1
X2
X3
V_{CC}
pull-up
resistors
1
2
3
4
5
6
7
8
9
*
0
#
Y1
Y2
Y3
Y4
rows

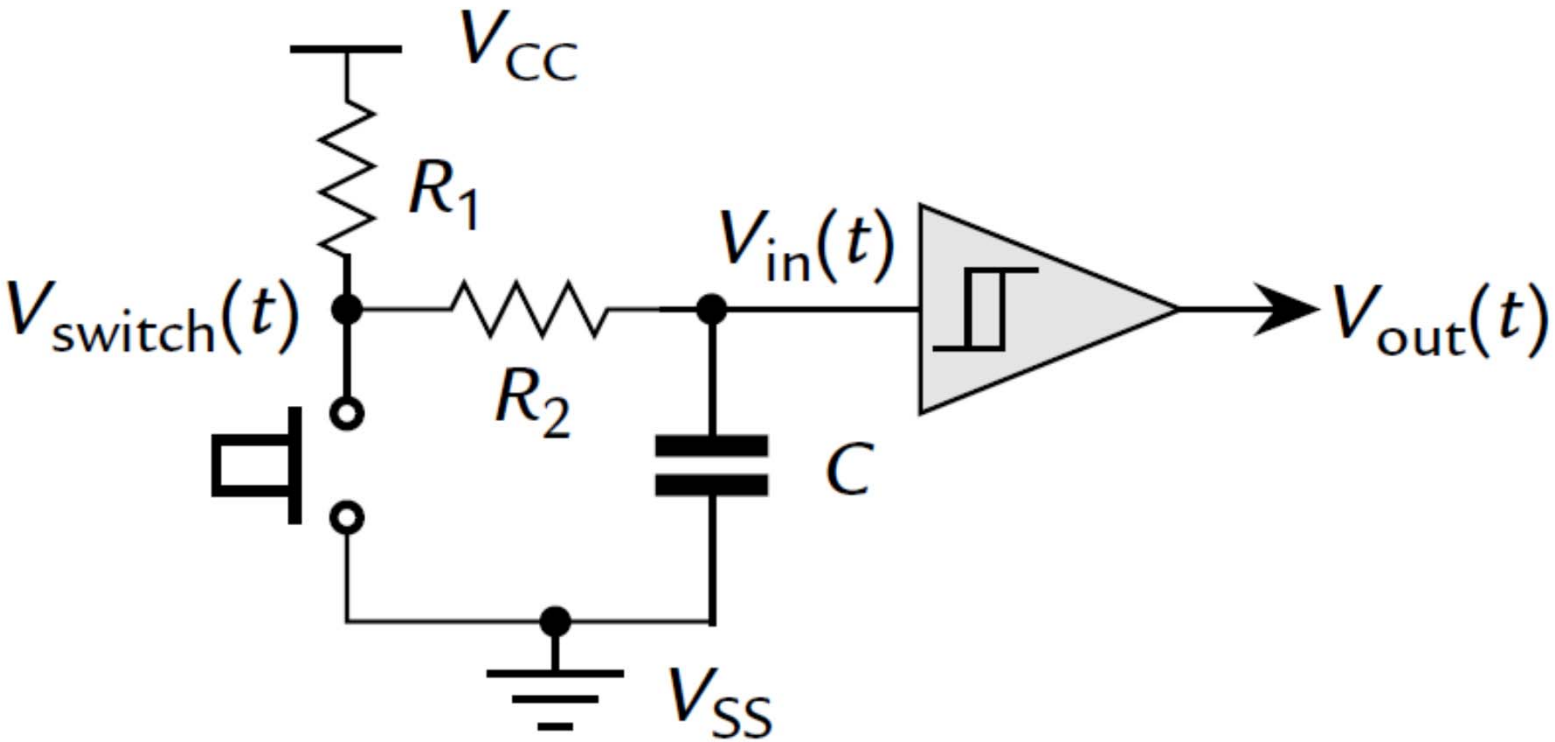
Scanning

1. Drive X1 low and the other columns X2 and X3 high. This makes the switches in column X1 active and the corresponding Y input goes low if a button is pressed. Thus we can detect the state of switches 1, 4, 7, or *. The switches in the other columns have no effect because both of their terminals are at *V*CC.
2. Drive X2 low and the other columns high to read the switches in column X2.
3. Repeat this for column X3.

Switch Debounce

- Older textbooks tell you that bounce is worse when a switch is closed than when it is opened and may last for around 50 ms.

Debouncing in Hardware



Debouncing in Software

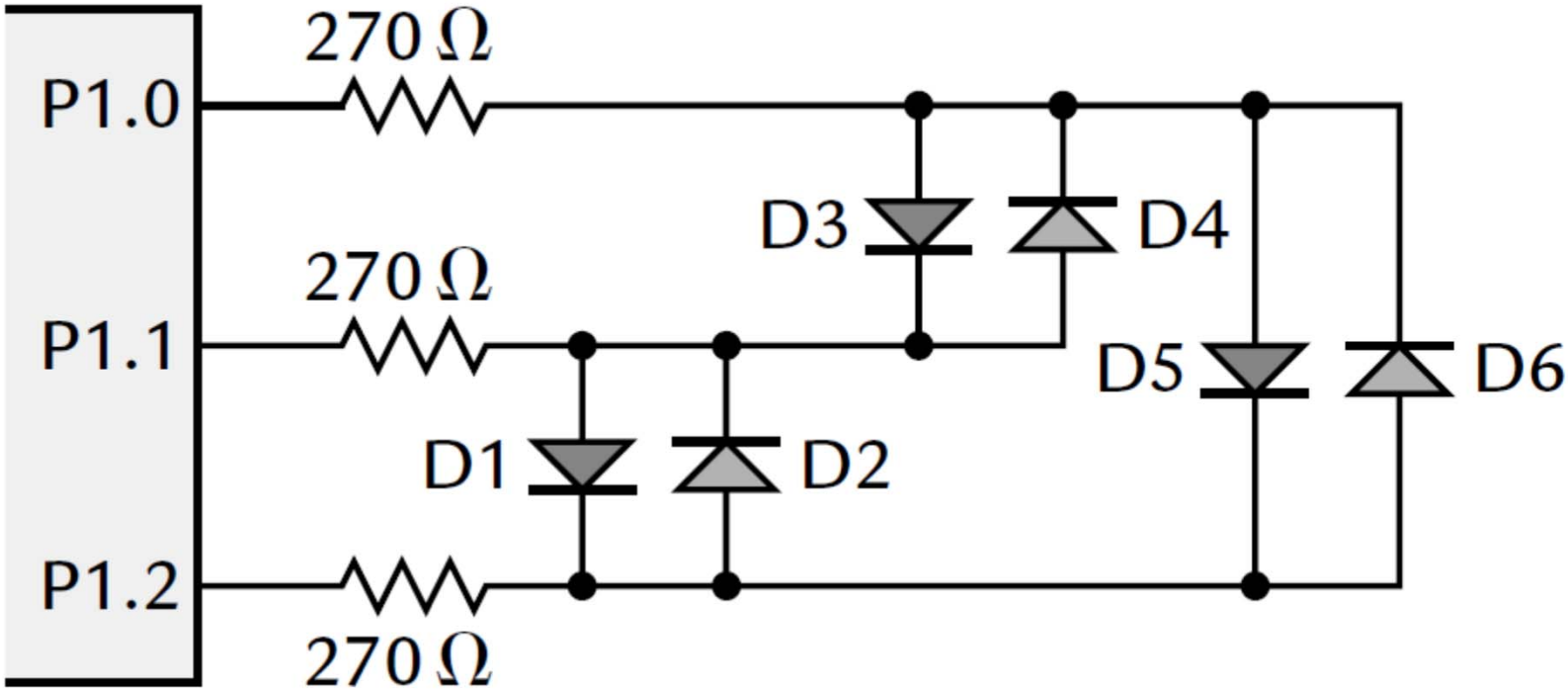
```
#include <io430x11x1.h> // Specific device
#include <intrinsics.h> // Intrinsic functions
#include <stdint.h> // Standard integer types
union { // Debounced state of P2IN
unsigned char DebP2IN; // Complete byte
struct {
unsigned char DebP2IN_0 : 1;
unsigned char DebP2IN_1 : 1;
unsigned char DebP2IN_2 : 1;
unsigned char DebP2IN_3 : 1;
unsigned char DebP2IN_4 : 1;
unsigned char DebP2IN_5 : 1;
unsigned char DebP2IN_6 : 1;
unsigned char DebP2IN_7 : 1;
} DebP2IN_bit; // Individual bits
};
#define RAWB1 P2IN_bit.P2IN_1
#define DEBB1 DebP2IN_bit.DebP2IN_1
#define LED1 P2OUT_bit.P2OUT_3
```

```
void main (void)
{
WDTCTL = WDTPW | WDTHOLD; // Stop watchdog timer
P2OUT_bit.P2OUT_3 = 1; // Preload LED1 off (active low)
P2DIR_bit.P2DIR_3 = 1; // Set pin with LED1 to output
DebP2IN = 0xFF; // Initial debounced state of port
TACCR0 = 160; // 160 counts at 32KHz = 5ms
TACCTL0 = CCIE; // Enable interrupts on Compare 0
TACTL = MC_1|TASSEL_1|TACLR; // Set up and start Timer A
// "Up to CCR0" mode , no clock division , clock from ACLK , clear
    timer
for (;;) { // Loop forever
__low_power_mode_3 (); // Enter LPM3 , only ACLK active
// Return to main function when a debounced transition has
    occurred
LED1 = DEBB1; // Update LED1 from debounced button
}
}
// -------------------------------------------------------------------
// Interrupt service routine for Timer A chan 0: no need to
```

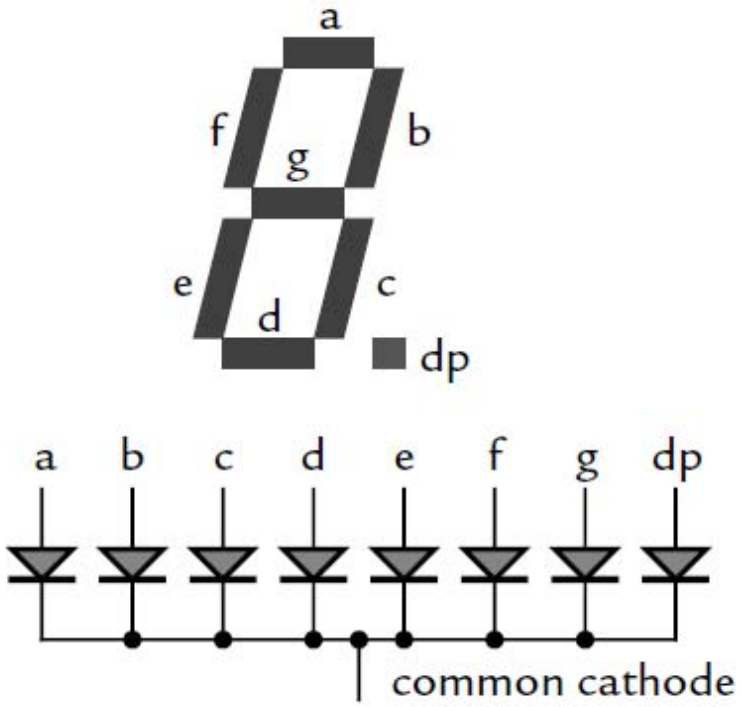
```
void main (void)
{
WDTCTL = WDTPW|WDTHOLD; // Stop watchdog timer
P2OUT = ~LED1; // Preload LED1 on , LED2 off
P2DIR = LED1|LED2; // Set pins with LED1 ,2 to output
TACCR0 = 49999; // Upper limit of count for TAR
TACCTL0 = CCIE; // Enable interrupts on Compare 0
TACTL = MC_1|ID_3|TASSEL_2|TACLR; // Set up and start Timer A
// "Up to CCR0" mode , divide clock by 8, clock from SMCLK , clear
    timer
__enable _interrupt (); // Enable interrupts (intrinsic)
for (;;) { // Loop forever doing nothing
} // Interrupts do the work
}
// -------------------------------------------------------------------
// Interrupt service routine for Timer A channel 0
```

```
#pragma vector = TIMERA0_VECTOR
__interrupt void TA0_ISR (void)
{
P2OUT ^= LED1|LED2; // Toggle LEDs
}
if (DEBB1 == 0) {
// Current debounced value low , looking for input to go high (release)
if (P21ShiftReg >= RELEASE_THRESHOLD) { // button released
DEBB1 = 1; // New debounced state high
__low_power_mode_off_on_exit(); // Wake main routine
}
} else {
// Current debounced value high , looking for input to go low (press)
if (P21ShiftReg <= PRESS_THRESHOLD) { // button pressed
DEBB1 = 0; // New debounced state low
__low_power_mode_off_on_exit(); // Wake main routine
}
}
}
```

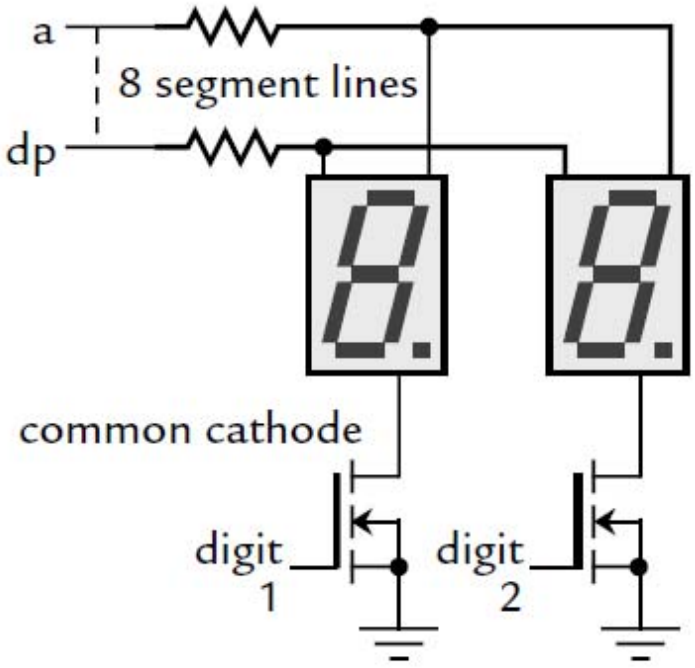
Digital Outputs



(a) individual seven-segment LED display



(b) multiplexed pair of displays



- http://www.youtube.com/watch?v=pOlhQ0n552Y&lr=1

Quizzes

Name some Registers for Port1?

What are P1REN?

How to scanning matrix keyboard?

What is next?

- ***CHAPTER 7: LCD***

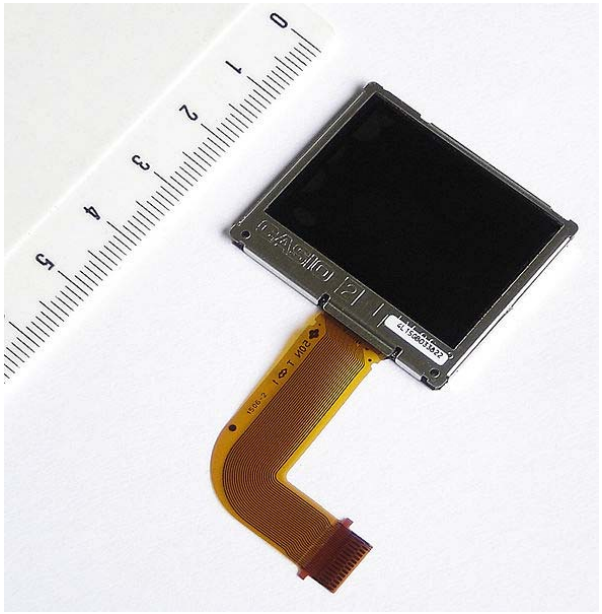
Liquid Crystal Displays

- A liquid crystal display (LCD) uses much less power than LEDs and is therefore a natural companion for the MSP430.
- LCDs fall into three classes:
  - **Segmented LCDs:** the simplest and can be driven directly by the MSP430x4xx family.These displays include the familiar seven-segment numerical displays found in watches,meters, and many other applications. (đồng hồ điện tử - chỉ hiển thị số)
  - **Character-based LCDs:** have a dot-matrix display, often with 1–4 rows of 8–72 characters. They can typically show a set of around 256 characters drawn from the ASCII characters, arrows, and a selection of other symbols. (hiển thị được chữ ABC..)
  - **Fully graphical LCDs:** found on every mobile (cell) phone (hiển thị hình ảnh)


Texas Instruments
on
ce/c
off


DEM16217
L-
L+
14


16
LCD-16x2B
WWW.GRAVITECH.US



Driving an LCD from an MSP430x4xx

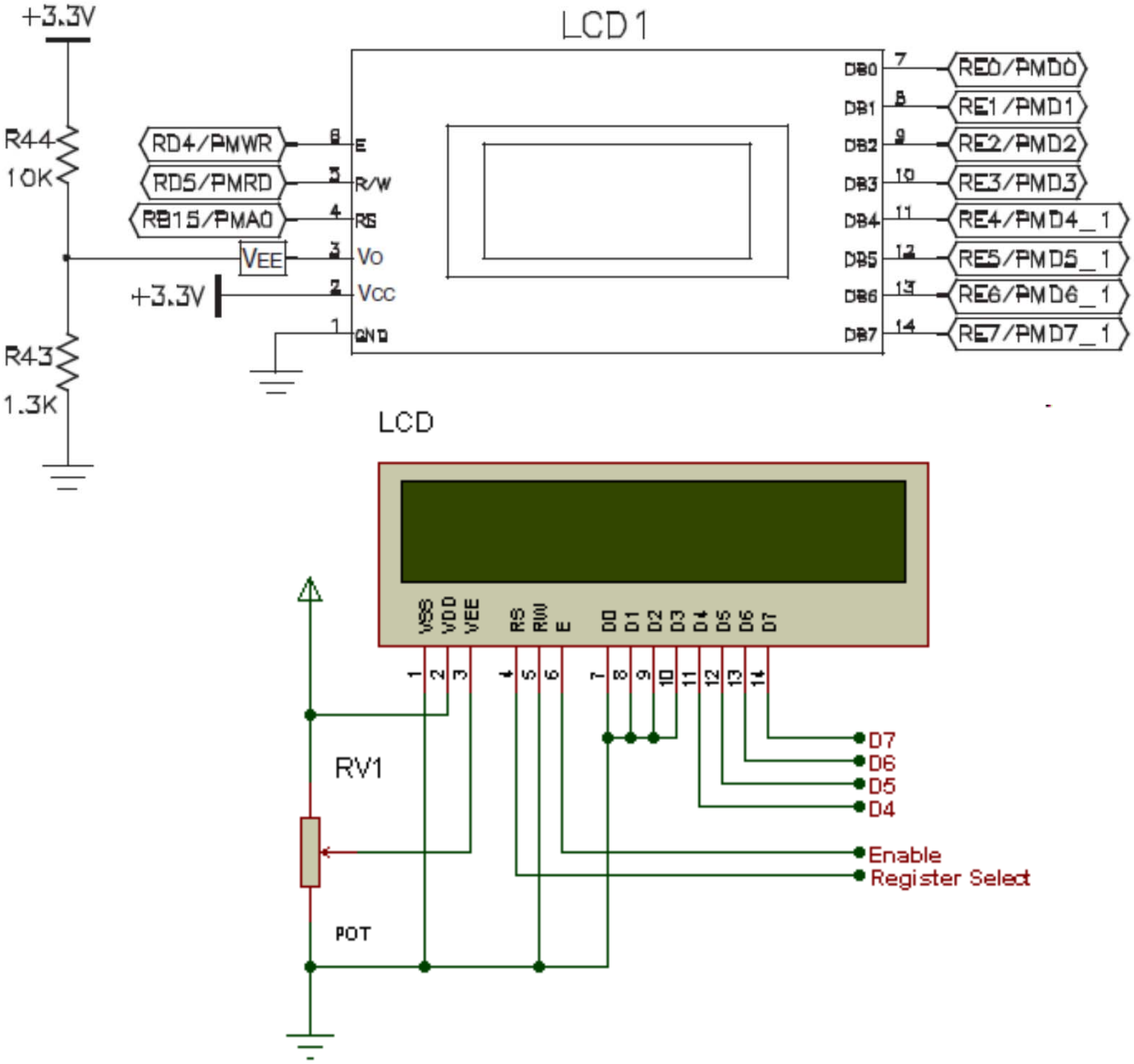
- All devices in the MSP430x4xx family contain an LCD controller and newer variants have an enhanced version called the LCD_A (for segmented LCD)
  - http://www.youtube.com/watch?v=M1wugVxp9t4&feature=related
- Character-based LCDs are usually incorporated into modules with a “Hitachi” interface.

Simple Applications of the LCD

- **A Very Simple Clock**

HD44780 LCD

- 2-rows by 16-character” display, a 3V alphanumeric LCD module (Tianma TM162JCAWG1) compatible with the
- Industry-standard HD44780 controllers


+3.3V
LCD1
R44
10K
R43
1.3K
RD4/PMWR
RD5/PMRD
RB15/PMA0
VEE
+3.3V
E
R/W
RS
Vo
Vcc
GND
DB0
DB1
DB2
DB3
DB4
DB5
DB6
DB7
RE0/PMD0
RE1/PMD1
RE2/PMD2
RE3/PMD3
RE4/PMD4_1
RE5/PMD5_1
RE6/PMD6_1
RE7/PMD7_1
LCD
VSS
VDD
VEE
RS
RW
E
D0
D1
D2
D3
D4
D5
D6
D7
RV1
POT
D7
D6
D5
D4
Enable
Register Select


Char. code
xxxx0000
xxxx0001
xxxx0010
xxxx0011
xxxx0100
xxxx0101
xxxx0110
xxxx0111
xxxx1000
xxxx1001
xxxx1010
xxxx1011
xxxx1100
xxxx1101
xxxx1110
xxxx1111

HD44780 controller compatibility

- The HD44780 compatibility means that the integrated controller contains just two registers separately addressable, one for
  - ASCII data and one for
  - commands, and the following standard set of commands can be used to set up and control the display

Instruction	Code										Description	Execution time
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Clear display	0	0	0	0	0	0	0	0	0	1	Clears display and returns cursor to the home position (address 0).	1.64ms
Cursor home	0	0	0	0	0	0	0	0	1	*	Returns cursor to home position (address 0). Also returns display being shifted to the original position. DDRAM contents remains unchanged.	1.64ms
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction (I/D), specifies to shift the display (S). These operations are performed during data read/write.	40us
Display On/Off control	0	0	0	0	0	0	1	D	C	B	Sets On/Off of all display (D), cursor On/Off (C) and blink of cursor position character (B).	40us
Cursor/display shift	0	0	0	0	0	1	S/C	R/L	*	*	Sets cursor-move or display-shift (S/C), shift direction (R/L). DDRAM contents remains unchanged.	40us

Instruction	Code										Description	Execution time
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Function set	0	0	0	0	1	DL	N	F	*	*	Sets interface data length (DL), number of display line (N) and character font(F).	40us
Set CGRAM address	0	0	0	1	CGRAM address						Sets the CGRAM address. CGRAM data is sent and received after this setting.	40us
Set DDRAM address	0	0	1	DDRAM address							Sets the DDRAM address. DDRAM data is sent and received after this setting.	40us
Read busy-flag and address counter	0	1	BF	CGRAM / DDRAM address							Reads Busy-flag (BF) indicating internal operation is being performed and reads CGRAM or DDRAM address counter contents (depending on previous instruction).	0us
Write to CGRAM or DDRAM	1	0	write data								Writes data to CGRAM or DDRAM.	40us
Read from CGRAM or DDRAM	1	1	read data								Reads data from CGRAM or DDRAM.	40us

Table 9-1. The HD44780 instruction set.

Bit name	Setting / Status	
I/D	0 = Decrement cursor position	1 = Increment cursor position
S	0 = No display shift	1 = Display shift
D	0 = Display off	1 = Display on
C	0 = Cursor off	1 = Cursor on
B	0 = Cursor blink off	1 = Cursor blink on
S/C	0 = Move cursor	1 = Shift display
R/L	0 = Shift left	1 = Shift right
DL	0 = 4-bit interface	1 = 8-bit interface
N	0 = 1/8 or 1/11 Duty (1 line)	1 = 1/16 Duty (2 lines)
F	0 = 5x7 dots	1 = 5x10 dots
BF	0 = Can accept instruction	1 = Internal operation in progress

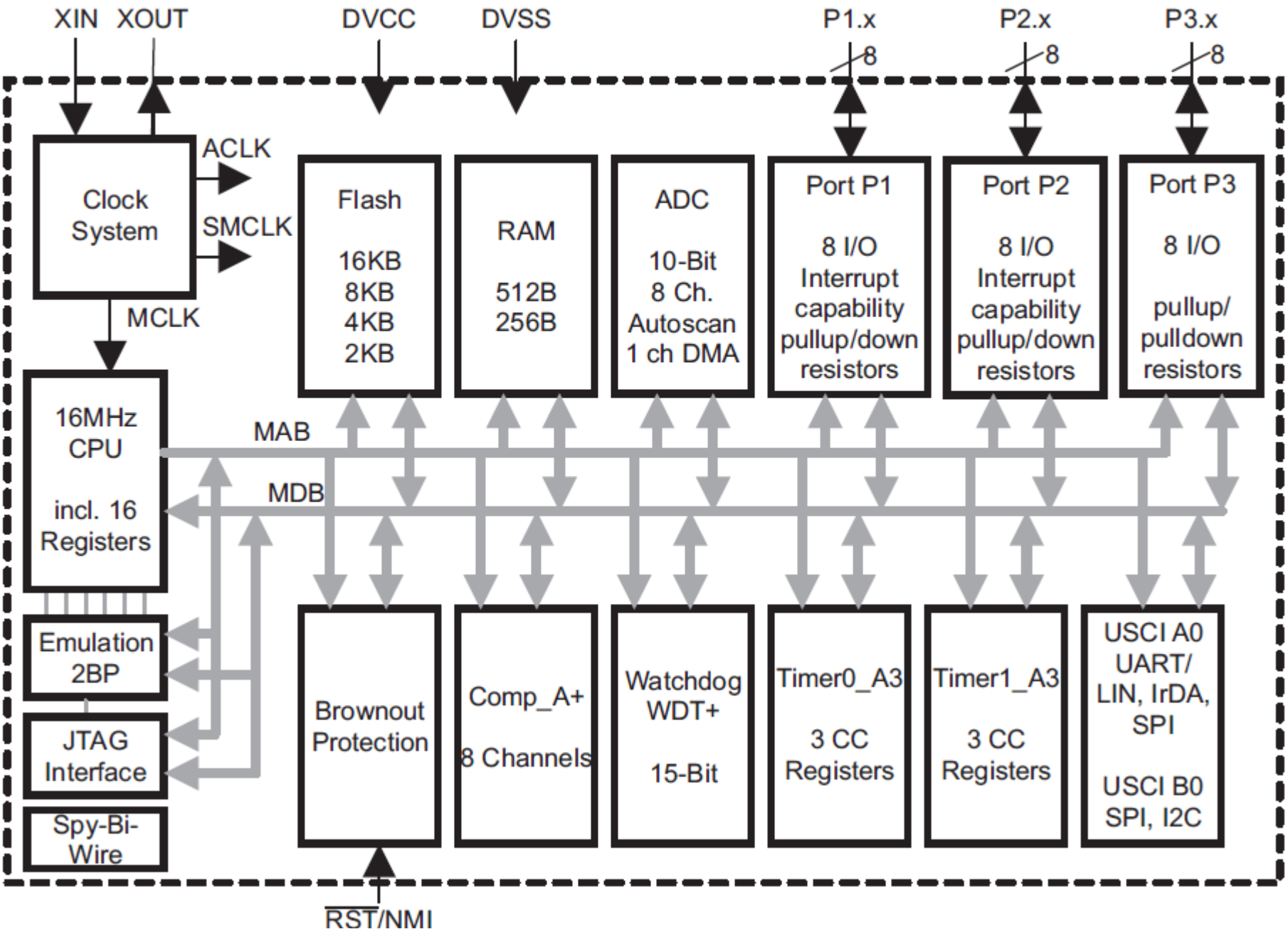
- **Demo on youtube**
- http://www.youtube.com/watch?v=M7dtBxrTIxA

- **Demo on Microcontroller Projects**
- **http://www.circuitvalley.com/2011/12/16x2-char-lcd-with-ti-msp430-launch-pad.html**

Chapter 8: Timers

1. Watchdog Timer
2. Basic Timer1
3. Timer_A
4. Measurement in the Capture Mode
5. Output in the Continuous Mode
6. Output in the Up Mode: Edge-Aligned Pulse-Width Modulation
7. Output in the Up/Down Mode: Centered Pulse-Width Modulation
8. Operation of Timer_A in the Sampling Mode
9. Timer_B
10. What Timer Where?
11. Setting the Real-Time Clock: State Machines

Functional Block Diagram, MSP430G2x53



Timers

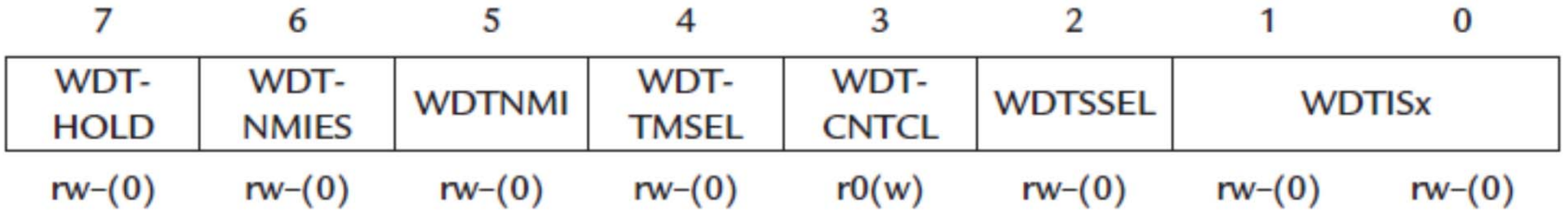
Watchdog timer: Included in all devices (newer ones have the enhanced watchdog timer+). Its main function is to protect the system against malfunctions but it can instead be used as an interval timer if this protection is not needed.

Timer_A: Provided in all devices. It typically has three channels and is much more versatile than the simpler timers just listed. Timer_A can handle external inputs and outputs directly to measure frequency, time-stamp inputs, and drive outputs at precisely specified times, either once or periodically

Timer_B: Included in larger devices of all families. It is similar to Timer_A with some extensions that make it more suitable for driving outputs such as pulse-width modulation.

Watchdog Timer

- The main purpose of the watchdog timer is to protect the system against failure of the software, such as the program becoming trapped in an unintended, infinite loop.
- Left to itself, the watchdog counts up and resets the MSP430 when it reaches its limit.
- The code must therefore keep clearing the counter before the limit is reached to prevent a reset.
- The operation of the watchdog is controlled by the 16-bit register WDTCTL



7	6	5	4	3	2	1	0
WDT-HOLD	WDT-NMIES	WDTNMI	WDT-TMSEL	WDT-CNTCL	WDTSSEL	WDTISx	
rw-(0)	rw-(0)	rw-(0)	rw-(0)	r0(w)	rw-(0)	rw-(0)	rw-(0)

Figure 8.1: The lower byte of the watchdog timer control register WDTCTL.

The watchdog is always active after the MSP430 has been reset. By default the clock is SMCLK, which is in turn derived from the DCO at about 1 MHz.
If the watchdog is left running, the counter must be repeatedly cleared to prevent it counting up as far as its limit. This is done by setting the WDTCNTCL bit in WDTCTL.
The task is often called *petting*, *feeding*, or *kicking the dog*.

```
// Watchdog config: active , ACLK /32768 -> 1s interval; clear counter
#define WDTCONFIG (WDTCNTCL|WDTSSEL)
// Include settings for _RST/NMI pin here as well
// -------------------------------------------------------------------
void main (void)
{
WDTCTL = WDTPW | WDTCONFIG; // Configure and clear watchdog
P2DIR = BIT3 | BIT4; // Set pins with LEDs to output
P2OUT = BIT3 | BIT4; // LEDs off (active low)
for (;;) { // Loop forever
LED2 = ~IFG1_bit.WDTIFG; // LED2 shows state of WDTIFG
if (B1 == 1) { // Button up
LED1 = 1; // LED1 off
} else { // Button down
WDTCTL = WDTPW | WDTCONFIG; // Feed/pet/kick/clear watchdog
LED1 = 0; // LED1 on
}
}
}
```

Watchdog as an Interval Timer

- The watchdog can be used as an interval timer if its protective function is not desired.
- Set the WDTTMSEL bit in WDTCTL for interval timer mode.
- The periods are the same as before and again WDTIFG is set when the timer reaches its limit, but no reset occurs.

Basic Timer1

Basic Timer1 is present in all MSP430xF4xx

Real-Time Clock

- A Real-Time Clock (RTC) module has been added to recent devices in the MSP430xFxx family.
- It counts seconds, minutes, hours, days, months, and years.

7	6	5 4	3 2	1	0
RTCBCD	RTCHOLD	RTCMODEx	RTCTEVx	RTCIE	RTCFG

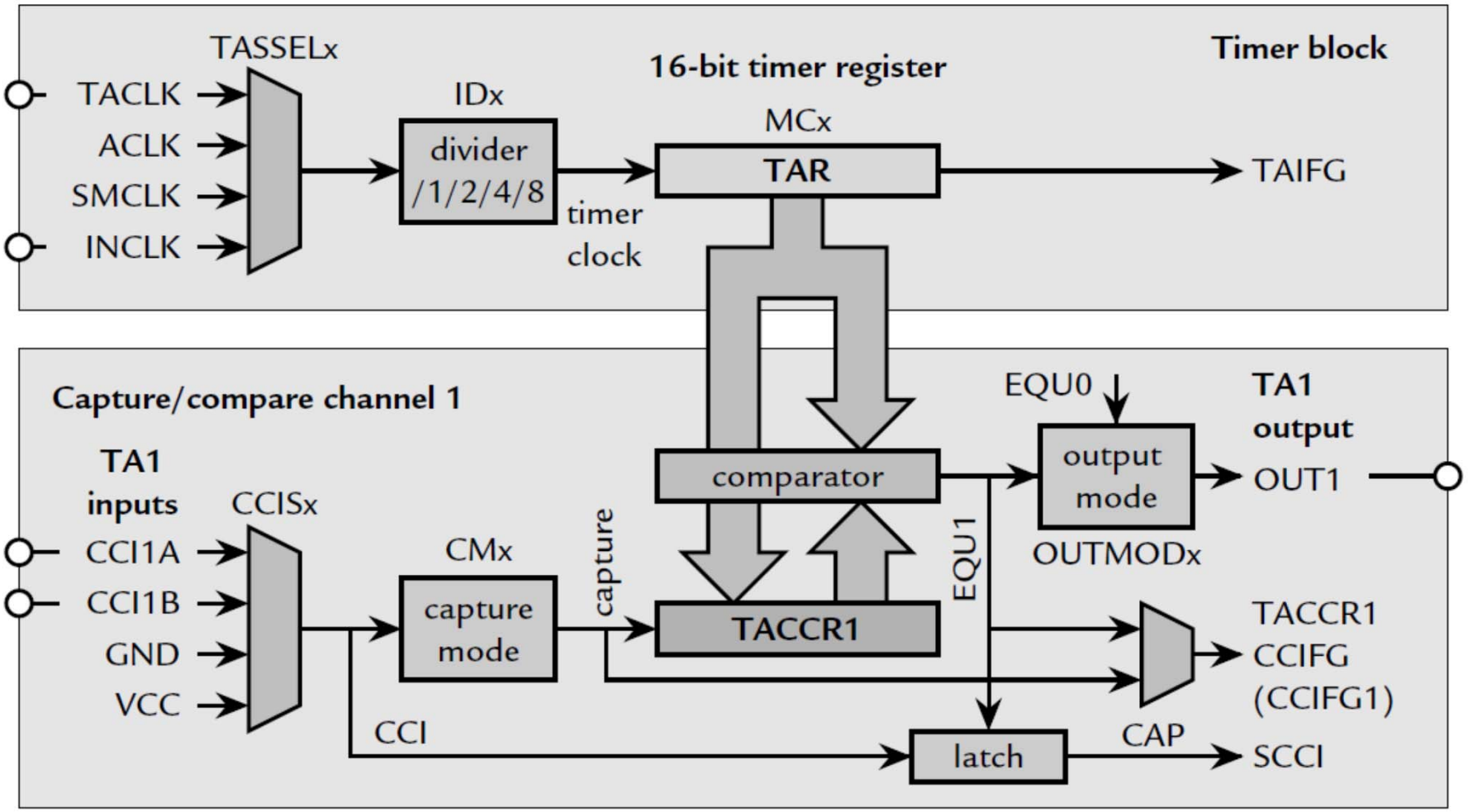
Figure 8.4: The Real-Time Clock control register RTCCTL.

Timer_A

- This is the most versatile, general-purpose timer in the MSP430 and is included in all

  devices
- **Timer block:** The core, based on the 16-bit register TAR. There is a choice of sources for the clock, whose frequency can be divided down (prescaled). The timer block has no output but a flag TAIFG is raised when the counter returns to 0.
- **Capture/compare channels:** In which most events occur, each of which is based on a register TACCRn. They all work in the same way with the important exception of TACCR0.

- **Capture** an input, which means recording the “time” (the value in TAR) at which the input changes in TACCRn; the input can be either external or internal from another peripheral or software.
- **Compare** the current value of TAR with the value stored in TACCRn and update an output when they match; the output can again be either external or internal.
- **Request an interrupt** by setting its flag TACCRn CCIFG on either of these events; this can be done even if no output signal is produced.
- **Sample** an input at a compare event; this special feature is particularly useful if Timer_A is used for serial communication in a device that lacks a dedicated interface.


TASSELx
TACLK
ACLK
SMCLK
INCLK
IDx
divider
/1/2/4/8
timer
clock
16-bit timer register
MCx
TAR
TAIFG
Timer block
Capture/compare channel 1
TA1
inputs
CCISx
CCI1A
CCI1B
GND
VCC
CMx
capture
mode
capture
comparator
TACCR1
EQU1
EQU0
output
mode
OUTMODx
TA1
output
OUT1
TACCR1
CCIFG
(CCIFG1)
CCI
latch
CAP
SCCI

- **Timer Block (**TASSELx bits)
  - **SMCLK** is internal and usually fast (megahertz).
  - **ACLK** is internal and usually slow, typically 32 KHz from a watch crystal but may be taken from the VLO in the MSP430F2xx family.
  - **TACLK** is external.
  - **INCLK** is also external, sometimes a separate pin but often it is connected through an inverter to the pin for TACLK so that INCLK = !TACLK.

Table 8.1: Resolution and period of Timer_A in the Continuous mode with different clocks and input dividers. Values have been rounded for clarity.

Input clock		Timer clock		Range of timer	
Source	Frequency	Divider	Resolution	Frequency	Period
SMCLK	16 MHz	1	$\frac{1}{16}$ µs	240 Hz	4 ms
SMCLK	1 MHz	1	1 µs	15 Hz	66 ms
SMCLK	1 MHz	8	8 µs	2 Hz	0.5 s
ACLK	32 KHz	1	31 µs	$\frac{1}{2}$ Hz	2 s
ACLK	32 KHz	8	240 µs	$\frac{1}{16}$ Hz	16 s

Four counter modes

- **Stop (MC = 0):** The timer is halted. All registers, including TAR, retain their values so that the timer can be restarted later where it left off.
- **Continuous (2):** The counter runs freely through its full range from 0x0000 to 0xFFFF,at which point it overflows and rolls over back to 0.
- **Up (1):** The counter counts from 0 up to the value in TACCR0, the capture/compareregister for channel 0. It returns to 0 on the next clock transition.
- **Up/Down (3):** The counter counts from 0 up to TACCR0, then down again to 0 andrepeats.

Capture/Compare Channels

- Timer_A has three channels in most MSP430s although channel 0 is lost to many applications because its register TACCR0 is needed to set the limit of counting in Up and Up/Down modes, as we have just seen.
- Each channel is controlled by a register TACCTLn.
- The central feature of each channel is its capture/compare register TACCRn.

Capture/Compare Channels

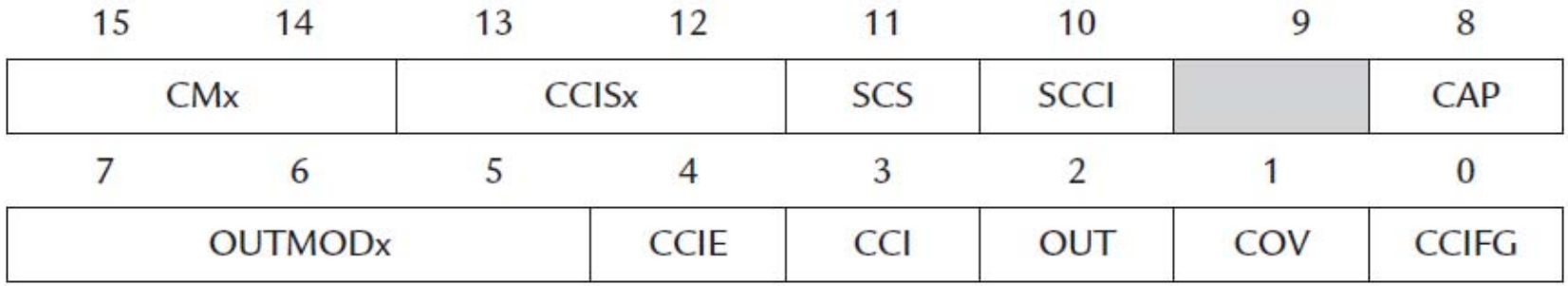


Figure 8.6: The Timer_A capture/compare control register TACCTLn.

Interrupts from Timer_A

- TACCR0 is privileged and has its own interrupt vector, TIMERA0_VECTOR. Its priority is higher than the other vector, TIMERA1_VECTOR, which is shared by the remaining capture/compare channels and the timer block.
- The MSP430 therefore provides an *interrupt vector register* TAIV to identify the source of the interrupt rapidly

```
#pragma vector = TIMERA1_VECTOR
__interrupt void TIMERA1_ISR (void) // ISR for TACCR1 CCIFG and
    TAIFG
{
// switch (TAIV) { // Standard switch
switch (__even_in_range(TAIV , 10)) { // Faster intrinsic fn
case 0: // No interrupt pending
break; // No action
case TAIV_CCIFG1: // Vector 2: CCIFG1
P1OUT_bit.P1OUT_0 = 0; // End of duty cycle: Turn off LED
break;
case TAIV_TAIFG: // Vector A: TAIFG , last value possible
P1OUT_bit.P1OUT_0 = 1; // Start of PWM cycle: Turn on LED
break;
default: // Should not be possible
for (;;) { // Disaster. Loop here forever
}
}
}
```

Application of Timers

- **Measurement of Time: Reaction Timer**
- **Measurement of Frequency: Comparison of SMCLK and ACLK**
- **Output in the Continuous Mode**
  - **Generation of Independent, Periodic Signals**
  - **A Single Pulse or Delay with a Precise Duration**
  - **Generation of a Precise Frequency**
  - **Output in the Up Mode: Edge-Aligned**
  - **Pulse-Width Modulation**

Embedded Systems

Hệ thống nhúng

Ts. Lê Mạnh Hải

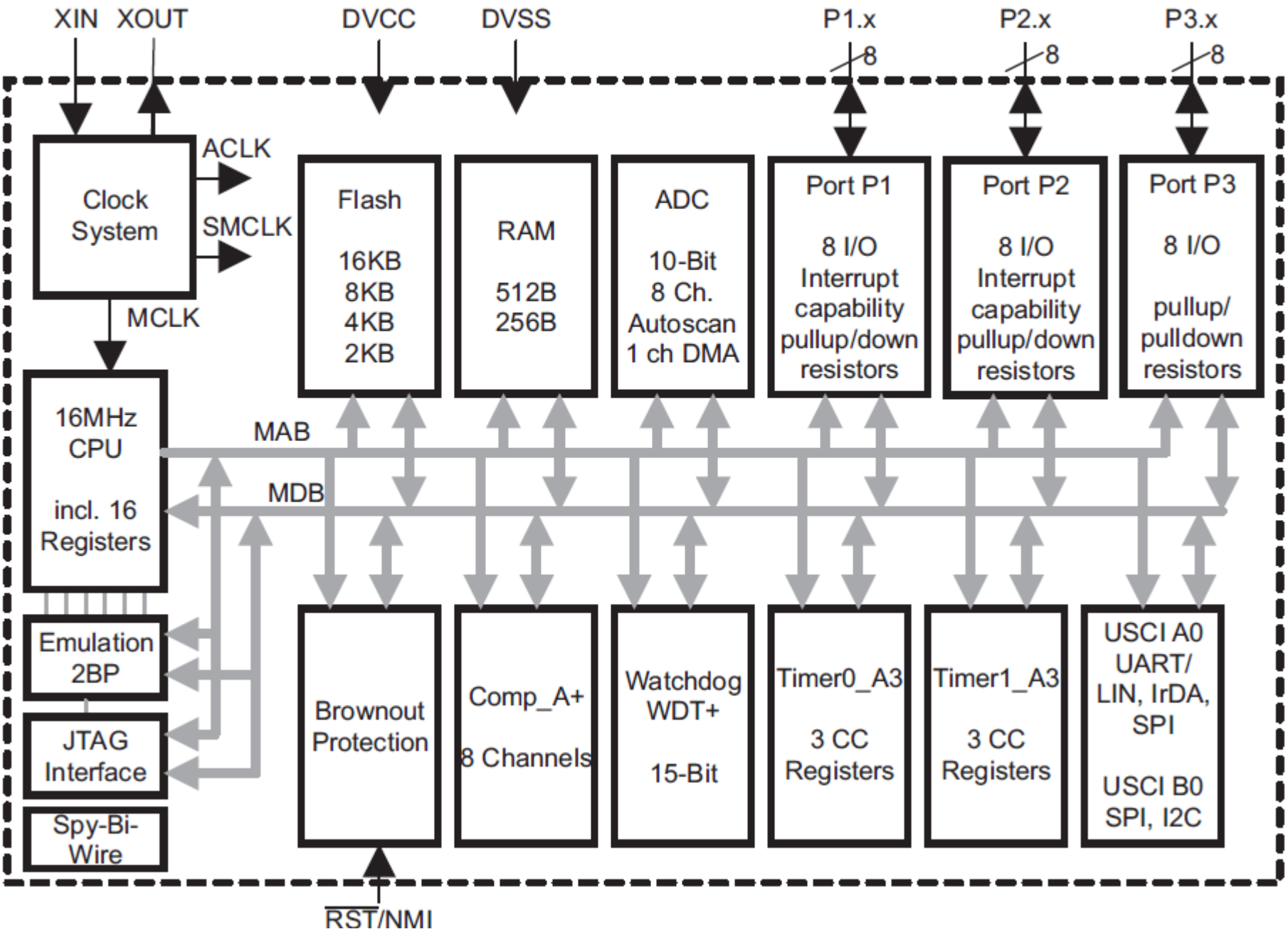
Khoa CNTT,

ĐH Kỹ thuật Công nghệ TP HCM

Chapter 10 : Communication

1. **Communication Peripherals in the MSP430**
2. **Serial Peripheral Interface**
3. **SPI with the USI**
4. **SPI with the USCI**
5. **Inter-integrated Circuit Bus**
6. **A Simple I²C Master with the USCI_B0**
7. **A Simple I²C Slave with the USI on a F2013**

Functional Block Diagram, MSP430G2x53



Communication Peripherals in the MSP430

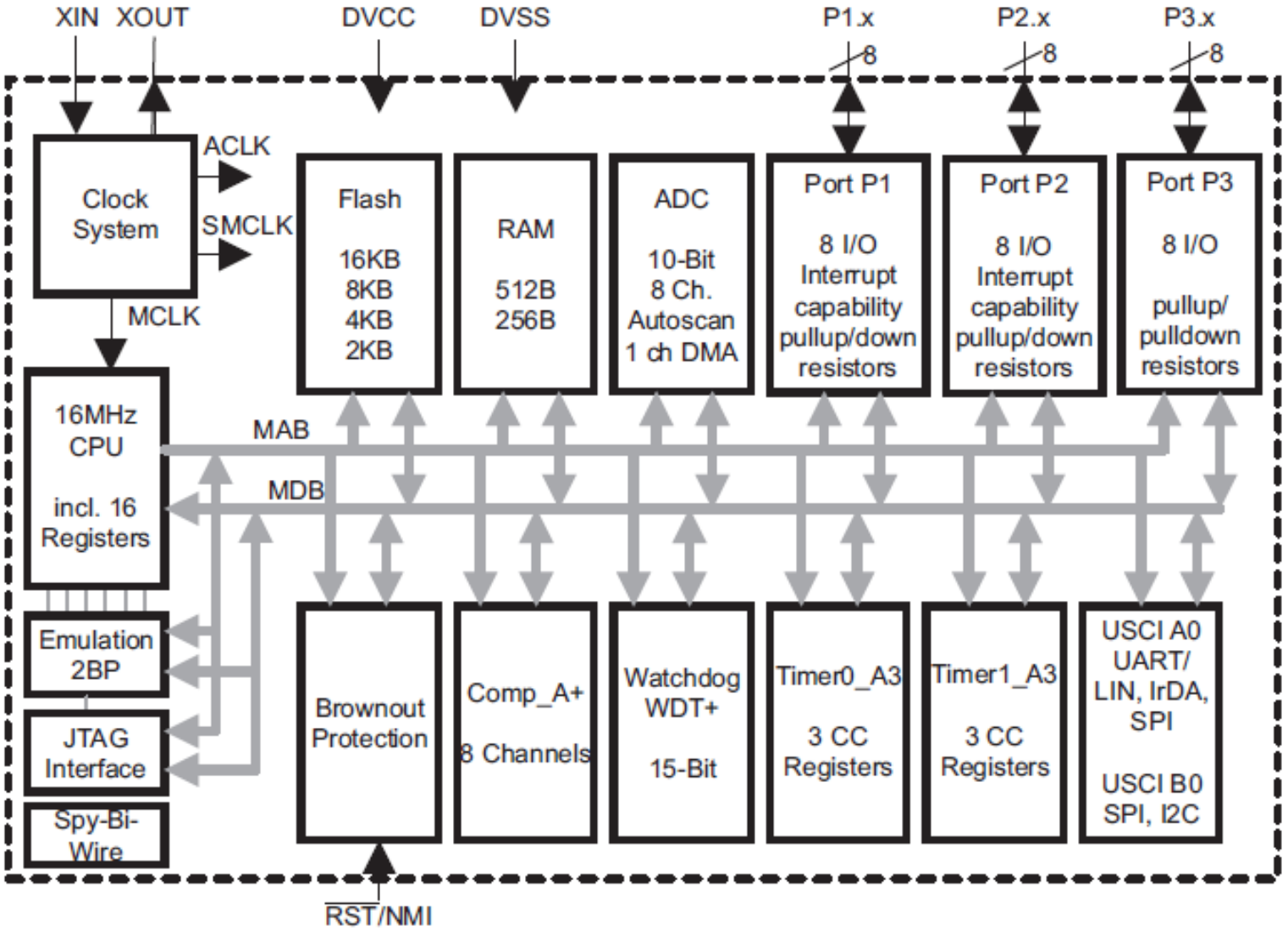
- The universal serial interface (USI) is a lightweight module, which is included in the small F20x2 and F20x3 devices. For a start, it handles only synchronous communication—SPI and I²C.
- **Universal Serial Communication Interface**

larger devices in the MSP430F2xx and MSP430F4xx families contain one or more *universal serial communication interface* (USCI) modules. The hardware handles almost all aspects of the communication, unlike the USI, so the software needs only to provide the data to transmit and store the received data in normal operation. Typically this requires only a couple of small interrupt service routines.

- **Universal Synchronous/Asynchronous Receiver/Transmitter**

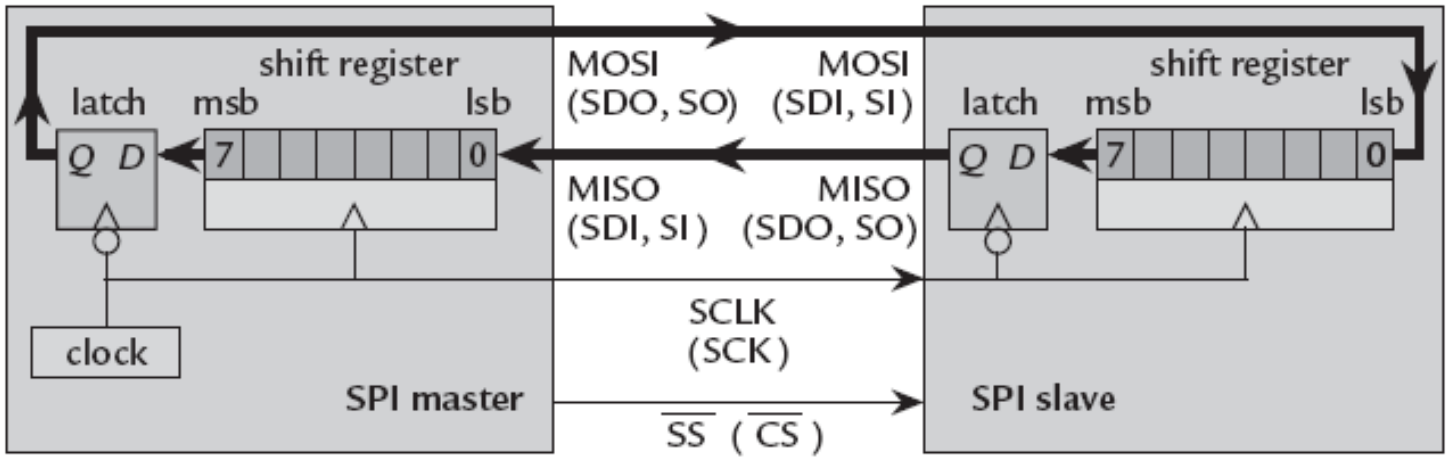
(USART) is an older module, which has been superseded by the USCI.

Functional Block Diagram, MSP430G2x53



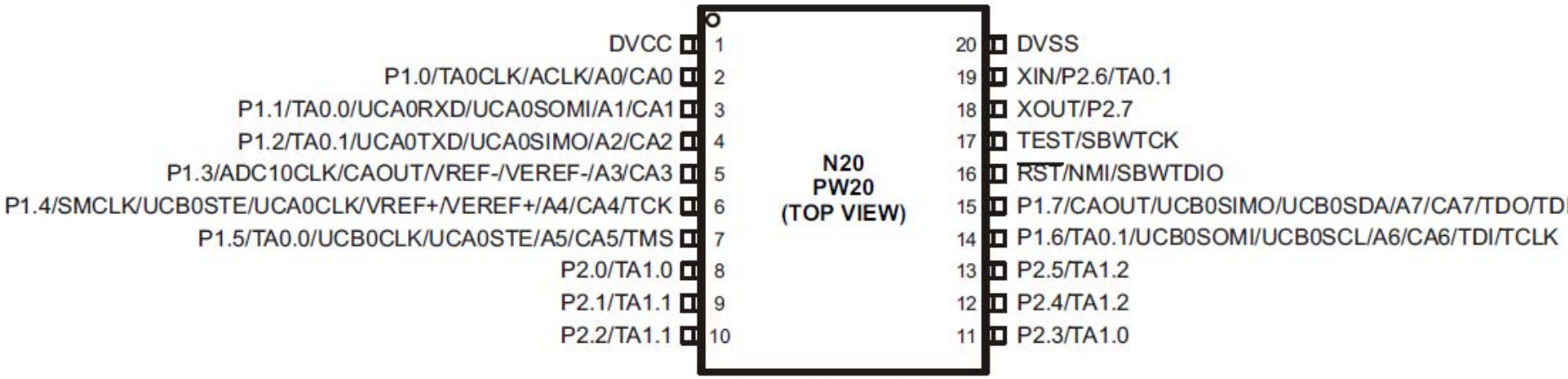
Serial Peripheral Interface

- The serial peripheral interface was introduced by Motorola and is the simplest synchronous communication protocol in general use.
- The only problem is that it is not a fixed standard like I²C. There are plenty of options within "standard" SPI and innumerable variations that go beyond this.



- The concept of SPI is based on two shift registers, one in each device, which are connected to form a loop.
- The registers usually hold 8 bits. Each device places a new bit on its output from the most significant bit (msb) of the shift register when the clock has a negative edge and reads its input into the lsb of the shift register on a positive edge of the clock.
- Thus a bit is transferred in each direction during each clock cycle.
- After eight cycles the contents of the shift registers have been exchanged and the transfer is complete.
- Transmission and reception are clearly inseparable.

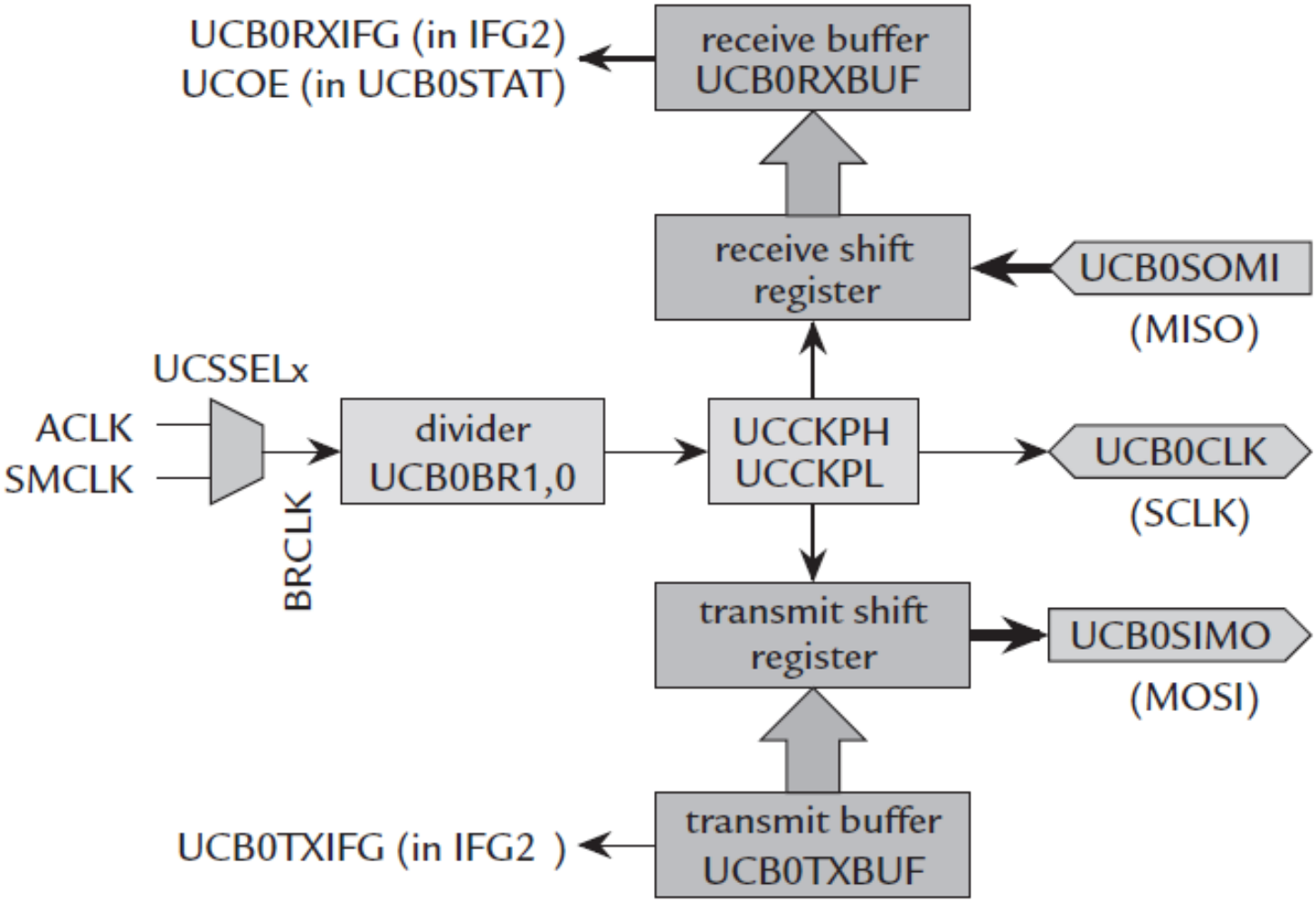
The main pins are labeled SOMI, SIMO, and CLK (2 USCI)



NOTE: ADC10 is available on MSP430G2x53 devices only.

NOTE: The pulldown resistors of port P3 should be enabled by setting P3REN.x = 1.

SPI block in USCI



SPI operation

There are separate shift registers for transmitting and receiving
Moreover, these registers are double-buffered and the user has no direct access to the shift registers themselves.
This means that a byte is moved from the receive shift register to RXBUF as soon as reception is complete, which leaves the shift register ready to accept the next transfer.
Similarly, a byte written to TXBUF remains in its buffer until the previous byte has been transmitted, at which point it is moved to the transmit shift register.
This relaxes considerably the constraints on handling interrupts in the USI, where the shift register must be read and updated rapidly between transfers. Although there are separate registers, reception and transmission are not independent because of the nature of SPI.

Read an examples C code for SPI communication

For Master

msp430g2xx3_uscia0_spi_09.c

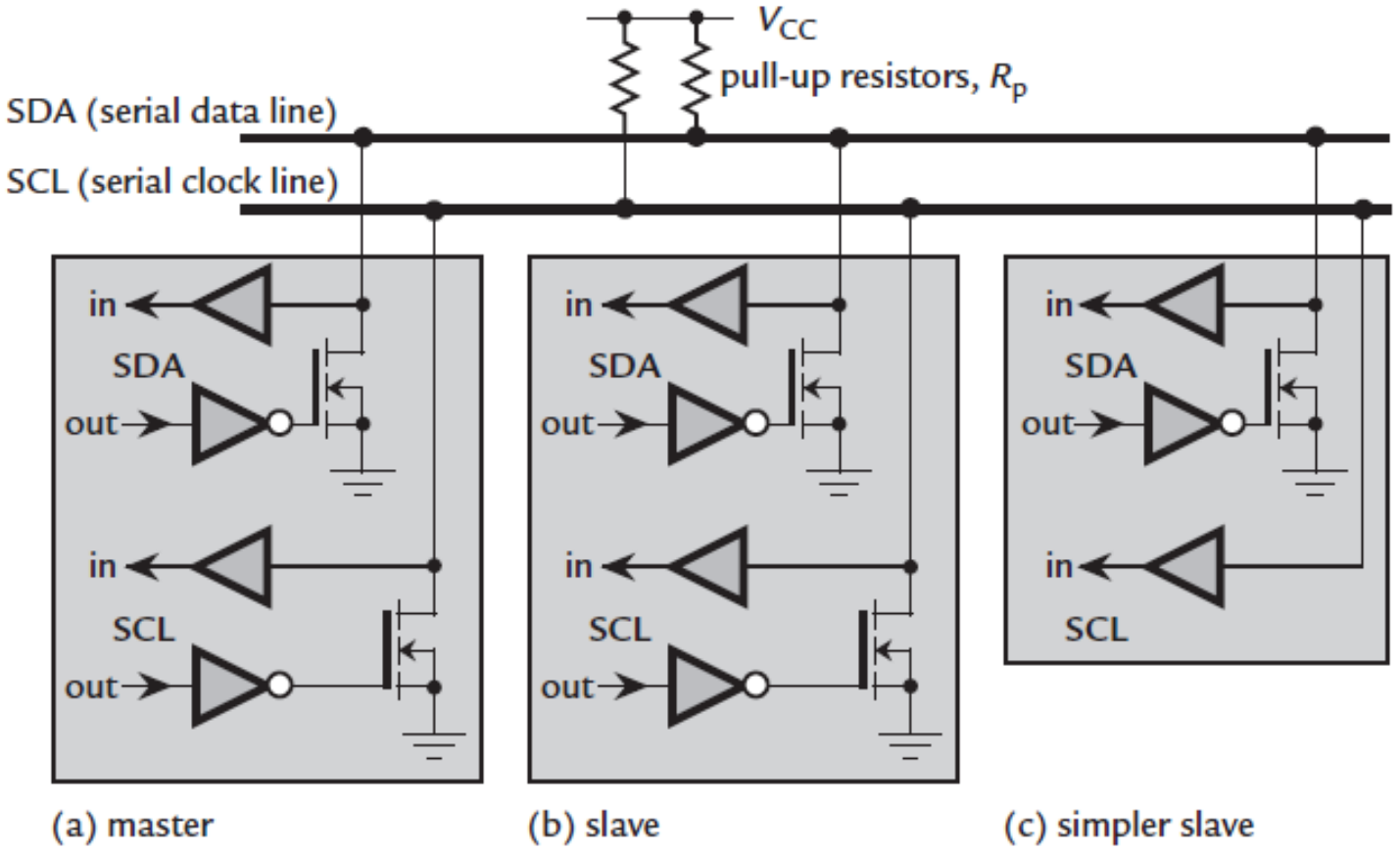
And for Slave

msp430g2xx3_uscia0_spi_10.c

Inter-integrated Circuit Bus

- The I²C bus was introduced by Philips (now NXP) Semiconductors. It was widely adopted and has become even more popular since its patents expired in 2006. It is a true bus, unlike SPI, with a specification and user manual that can be downloaded from NXP.
- Revision 03 of the user manual is document UM10204, dated June 19, 2007. It is clearly written and a lot easier to read than you might expect. The I²C bus uses only two, bidirectional lines:
- Serial data (SDA).
- Serial clock (SCL).

- Structure



- Operation: Read from page 534 -542

Ôn tập Hệ thống nhúng

1. Cấu trúc tổng thể của vi điều khiển
2. Sơ đồ khối của chíp TI MSP430G2553
3. Bộ nhớ MSP430G2553: Phân bổ vị trí bộ nhớ và ý nghĩa từng vùng nhớ
4. Cấu tạo CPU và ý nghĩa các thanh ghi trong CPU
5. Các loại xung nhịp (clock) và các chế độ hoạt động
6. Hàm và các bước thực hiện khi gọi một hàm
7. Khái niệm ngắt và chương trình phục vụ ngắt
8. Các bước thực thi khi thực hiện một ngắt.
9. Các chế độ công suất thấp.
10. Các cổng nhập xuất số (Digital Input and Output).
11. Quét ma trận bàn phím. Chống dội
12. Các loại LCD. Sơ đồ chân kết nối theo chuẩn HD44780
13. Các loại timer. Cấu trúc và hoạt động của WDT
14. Cấu trúc và các chế độ hoạt động của TimerA0
15. Kết nối Serial Peripheral Interface (SPI). Cấu trúc và hoạt động.
16. Kết nối Inter-integrated Circuit Bus (I2C).