Module 4. MSP430 Architecture & Intro to Digital I/O

Topics

- Getting to know the MSP430 Hardware
- Start of Digital I/O

Last Time

- Memory Organization
- Floating point format

Getting to know the MSP430 Hardware

In a programming course, typically you focus on just the code:

- Learn a high level programming language and some algorithms
- Use a "computer" from a high level

A typical program does three things:

A few common points for all kinds of software:

- Use constructs like loops, conditional, algorithms like search, sort, data structures
- Software: write logic and syntax correctly and it will just work
- Library functions for I/O

... but what's inside the "computer"? How does it work? When you are writing in a high level programming language, do you care?

In contrast, developing software for embedded **requires** much more in depth knowledge about the microprocessor that is the target of your program. For instance, it's important to know:

A general software hierarchy

We can think of the software components in a system and the way they interact with the hardware as a hierarchy or *software stack*:

A general computing device (PC) might have a software hierarchy like this:

Applications
Operating System (Includes user interface)
System Software: Interface to hardware
Hardware (CPU, Memory, Peripherals)

On an embedded system, this stack gets “squashed”:

Application
System software / OS
Hardware (CPU, Memory, Peripherals)

- Application is closely integrated with the hardware layer
- Little or no operating system–usually only runs one task or a set of tasks
- Often little or no "wrapping" of functionality
- On larger systems, you may use a Real Time Operating System (RTOS) that provides some basic support for multitasking…

A general microprocessor hardware architecture

In general, any microprocessor system has the following components:

CPU (Central Processing Unit)

The "core" of the computer

Memory

Stores information

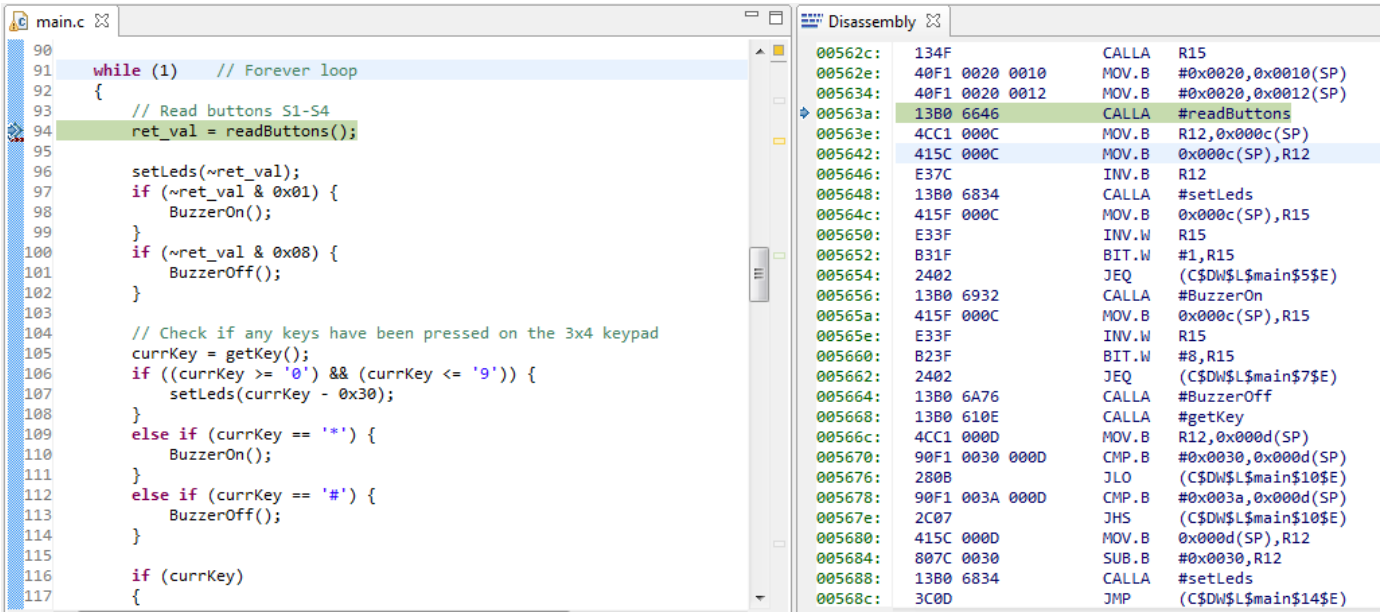
Peripherals

How does the CPU work?

The CPU executes *machine code*, which are low-level instructions directly run by the hardware. Machine code is a binary format seen by the CPU.

- Instructions perform very specific tasks
- Instruction set (ISA) is different for every CPU type (MSP430, ARM, x86, …)

- **Compiler is responsible for figuring out how to build all programs using these instructions!**



We will never write in assembly in this class. However, it is important that you understand that these instructions exist!

CPU instructions operate on…

- **Internal Registers**: 16 general purpose registers (R0-R15)
  - Storage locations inside the CPU used for recent instructions
    All registers are 16-bits wide (except R0 and R1, which are 20 bits)
  - Can be accessed very quickly (one clock cycle)
  - Some registers control program execution
    (R0 = Program counter, R1 = Stack pointer, R2 = Status register)
- **Memory**: Instructions read from and write to memory
  - Load and store data from the outside world using the memory bus!

What goes in memory?

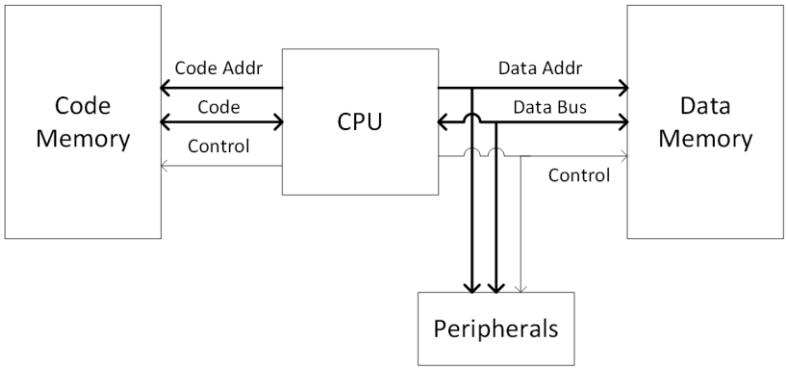
Remember that memory doesn't just store your variables–it stores the program's code as well!

- The CPU needs to load both code **and** data from memory

There are two generic types of memory architectures used by microprocessors and microcontroller systems:

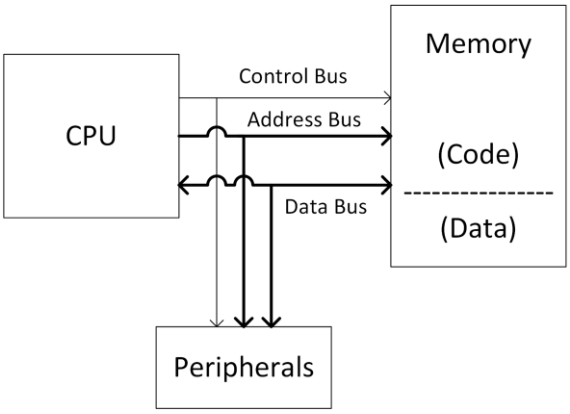
- **Von Neumann** Architecture (~1952)
- **Harvard** Architecture (~1944)

Harvard Architecture: Separate memory address spaces for code and data



Benefits: Instruction fetch and data read happen in parallel
Drawbacks: Separate instruction and data buses
In this form, the Harvard architecture is used today by highly-pipelined systems like DSP chips.

Von Neumann Architecture: Single memory address spaces for code and data



Benefits: Single address and data buses (simpler to interface)
Drawbacks: Implicit bottleneck since we have the same pipeline for code and data

The MSP430 Architecture

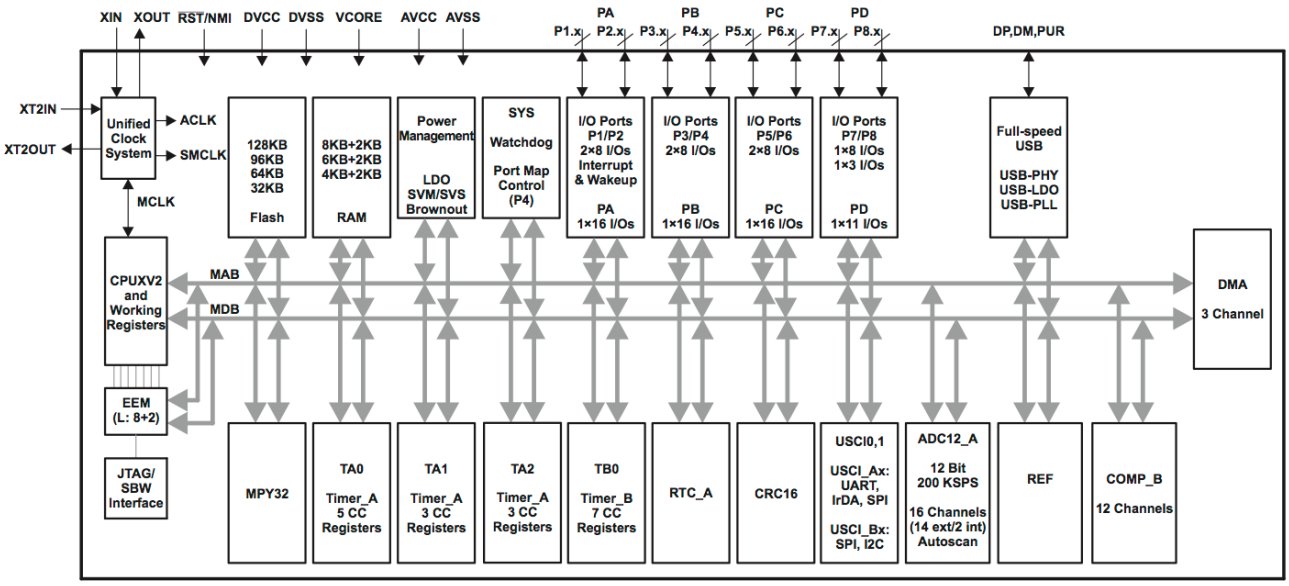
The MSP430 is a family of microcontrollers–there are hundreds of versions of this CPU with various configurations of memory and peripherals!

- You can think of it as a type of *System on a Chip* (SoC)

In our labs, we use the MSP430F5529

- 128KB of flash memory: Used for code storage
- 8 KB of RAM (+ 2KB RAM for USB controller): Used for data storage
- Lots of peripherals
  - 32 bit multiplier
  - Timers, comparator, USB controller
  - Much, much more!

How much more? Here's a block diagram:



Note the lines connecting all of the peripherals: this is the memory bus!

MAB: Memory Address Bus

MDB: Memory Data Bus

MSP430 Memory Organization

Memory: A group of sequential locations where binary data is stored

- On the MSP430, each memory location holds one byte
- Each byte has a unique address which the CPU uses to access it
- Multibyte data is stored in _____ Endian!

Two types of memory: Volatile and Non-Volatile

RAM (Random Access Memory)

- Our MSP430 has 8KB of RAM + 2KB for USB
- RAM is **volatile**, meaning that it loses its state when the chip is not powered
- Used as data memory
- Accessed via read and write instructions

Flash

- Used primarily for code memory
- Flash is **non-volatile**, meaning that its state persists even if the chip is not powered
- CPU fetches code from flash automatically
- Accessed via program control, but more difficult than RAM
    - Write time >> Read time
    - Writes must occur in large segments (512 bytes)

How are programs stored in memory?

When a program is compiled, the linker arranges different portions into various memory *segments*, which are stored in different contiguous memory regions. The most important segments are:

- The stack (`.stack`): Stores local variables and context information on each function call
- Constant data (`.data`, `.bss`): Stores global variables and other constant data (strings, lookup tables, etc.
- Text (`.text`): Compiled code for your program (code you write + libraries)
- Heap: Dynamically allocated memory (avoid using this!)

When compiling, the linker reads a script called a *command file*, which maps each section to a memory device. Usually, most code is stored in flash, while most data goes in RAM, though it may be necessary to adjust these requirements. Why?

Why should we avoid dynamically allocated memory?

Memory architecture and layout

The MSP430 is a 16-bit microcontroller, meaning that:

- The data bus is 16 bits wide
- Internal CPU registers are 16-bits

Note: MSP430 '5xxx and '6xxx families use a **20 bit address bus** to allow access to at most 1MB of memory.

However, memory isn't just one big block….

Mapping Memory

In practice, the “memory space” is *mapped* across the different types of memory and hardware devices connected to the CPU.

- This includes the different types of physical memory (RAM, flash), as well as *hardware peripherals*
- The mappings of which components use which addresses is based on the physical wiring in the IC (we know the mappings based on the header files)

The Memory Map (found in MSP430F5529 datasheet)

Table 5. Memory Organization(1)

		MSP430F5522 MSP430F5521 MSP430F5513	MSP430F5525 MSP430F5524 MSP430F5515 MSP430F5514	MSP430F5527 MSP430F5526 MSP430F5517	MSP430F5529 MSP430F5528 MSP430F5519
Memory (flash) Main: interrupt vector	Total Size	32 KB 00FFFFh–00FF80h	64 KB 00FFFFh–00FF80h	96 KB 00FFFFh–00FF80h	128 KB 00FFFFh–00FF80h
Main: code memory	Bank D	N/A	N/A	N/A	32 KB 0243FFh–01C400h
	Bank C	N/A	N/A	32 KB 01C3FFh–014400h	32 KB 01C3FFh–014400h
	Bank B	15 KB 00FFFFh–00C400h	32 KB 0143FFh–00C400h	32 KB 0143FFh–00C400h	32 KB 0143FFh–00C400h
	Bank A	17 KB 00C3FFh–008000h	32 KB 00C3FFh–004400h	32 KB 00C3FFh–004400h	32 KB 00C3FFh–004400h
RAM	Sector 3	2 KB(2) 0043FFh–003C00h	N/A	N/A	2 KB 0043FFh–003C00h
	Sector 2	2 KB(3) 003BFFh–003400h	N/A	2 KB 003BFFh–003400h	2 KB 003BFFh–003400h
	Sector 1	2 KB 0033FFh–002C00h	2 KB 0033FFh–002C00h	2 KB 0033FFh–002C00h	2 KB 0033FFh–002C00h
	Sector 0	2 KB 002BFFh–002400h	2 KB 002BFFh–002400h	2 KB 002BFFh–002400h	2 KB 002BFFh–002400h
USB RAM(4)	Sector 7	2 KB 0023FFh–001C00h	2 KB 0023FFh–001C00h	2 KB 0023FFh–001C00h	2 KB 0023FFh–001C00h
Information memory (flash)	Info A	128 B 0019FFh–001980h	128 B 0019FFh–001980h	128 B 0019FFh–001980h	128 B 0019FFh–001980h
	Info B	128 B 00197Fh–001900h	128 B 00197Fh–001900h	128 B 00197Fh–001900h	128 B 00197Fh–001900h
	Info C	128 B 0018FFh–001880h	128 B 0018FFh–001880h	128 B 0018FFh–001880h	128 B 0018FFh–001880h
	Info D	128 B 00187Fh–001800h	128 B 00187Fh–001800h	128 B 00187Fh–001800h	128 B 00187Fh–001800h
Bootstrap loader (BSL) memory (flash)	BSL 3	512 B 0017FFh–001600h	512 B 0017FFh–001600h	512 B 0017FFh–001600h	512 B 0017FFh–001600h
	BSL 2	512 B 0015FFh–001400h	512 B 0015FFh–001400h	512 B 0015FFh–001400h	512 B 0015FFh–001400h
	BSL 1	512 B 0013FFh–001200h	512 B 0013FFh–001200h	512 B 0013FFh–001200h	512 B 0013FFh–001200h
	BSL 0	512 B 0011FFh–001000h	512 B 0011FFh–001000h	512 B 0011FFh–001000h	512 B 0011FFh–001000h
Peripherals	Size	4 KB 000FFFh–0h	4 KB 000FFFh–0h	4 KB 000FFFh–0h	4 KB 000FFFh–0h

(1) N/A = Not available
(2) MSP430F5522 only
(3) MSP430F5522, MSP430F5521 only
(4) USB RAM can be used as general purpose RAM when not used for USB operation.

What can we learn from this?

- RAM starts at 0x2400, implemented in 2KB "Banks"
- Flash uses the address range 0x4400 to 0xFFFF
  - Code is written to flash starting from this address
- What about addresses 0x0010-0x0fff?

So what's the deal with addresses 0010h-0FFFh again?

These addresses are assigned to *peripherals:*

- Each peripheral has its own registers that are *mapped* as part of the memory that the CPU can access
- CPU can read or write data to peripherals just like any other memory address

This is how you make the CPU do I/O!

Input and Output

Consider this C code for a general-purpose system:

```
#include <stdio.h>

void main()
{
   char inKey = '-';  // declare variable named inKey
                        // and initialize to ASCII '-'

   while (inKey != 'X');
   {
         /* get character from keyboard */
         inKey = getchar();

         /* display character entered on screen */
         putchar(inKey);
   }
}
```

What is really happening here?

`getchar()` and `putchar()` are functions from the C standard library (part of stdio.h)

- Library for these functions is part of OS, and linked into code during build process
- These functions have always been part of the standard library because general purpose systems have always needed to use this type of I/O (eg. keyboard, screen, …)

Example: When a key is pressed, several layers below our little application, a byte has been placed on the microprocessor's data bus from a *port* connected to the keyboard:

Digital I/O: The Basics

Why do we use Digital I/O anyway?

Digital I/O is a method of directly inputting our outputting logic levels to the pins of the MSP430 Package.

You can use this functionality to implement almost anything!

- Simple devices: Buttons and LEDs
- Control signals for complex peripherals
- … and more!

Fun Facts about Digital I/O

- Eight independent, individually-configurable *ports*
- Ports 1-7 each have 8 configurable *pins*, and are thus 8 bits wide; Port 8 is 3 bits wide
- Each pin of each port can be configured individually as **input** or **output**
    - What makes something an input or an output? Inputs are devices from which you *read* a state, outputs require you to *write* a state to it.
- Ports 1 and 2 can generate *interrupts* on certain events, which are control signals that can be accepted or ignored by the MSP430
    - We will discuss these soon!
- Each port is controlled by **six** single-byte registers
- All the I/O port registers are *memory-mapped*, meaning that each register associated with a digital I/O port has a unique address in memory
    - How do you know what the addresses are? These are defined in msp430.h and msp430f5529.cmd. In these files, each register is given a specific name.
- Many more fun facts can be found in the User's Guide!

Digital I/O Registers (Part 1)

Each Digital I/O port has six registers to control its features. We will start by discussing three of them:

Direction Register (PxDIR)

Sets port pins as Input or Output
Set to 1 = Output
Set to 0 = Input

Input Register (PxIN)

Output Register (PxOUT)

The other registers are:

- **Function Select Register (PxSEL)**
- **Drive Strength (PxDS)**
- **Pull-up/Pull-Down Resistor Enable (PxREN)**

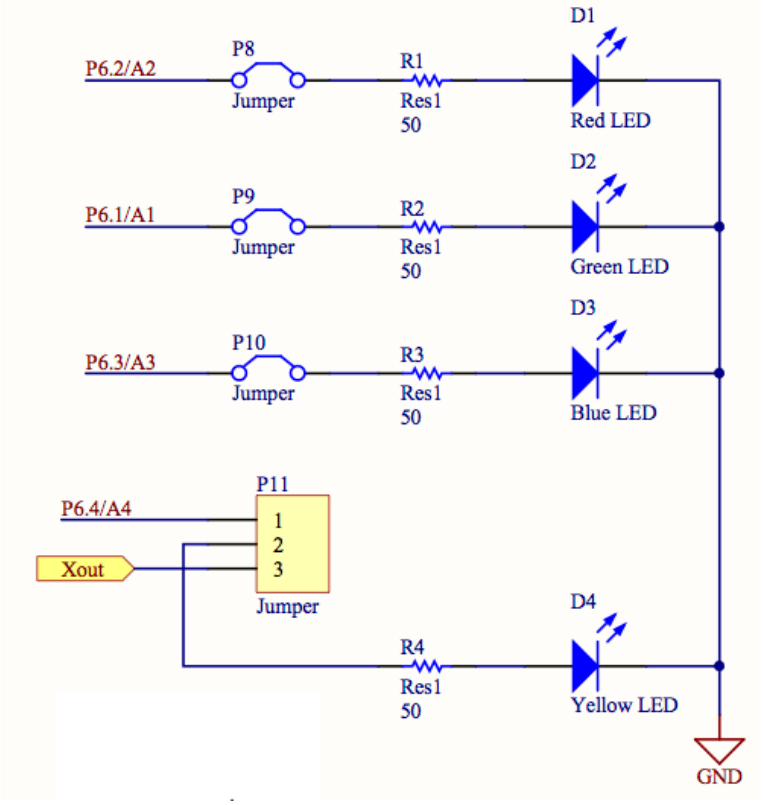
We will discuss these (using examples) later.

Conceptually, once you know which registers to use, using Digital I/O is pretty simple–all you need to do is read or write the desired values to the registers.

Digital I/O Concepts: Input or Output?

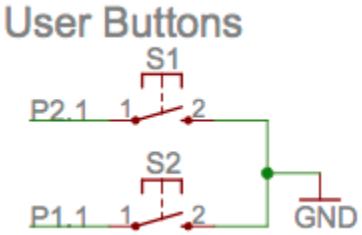
How do you know if something is an input or an output?

Consider the LEDs on our board. Are they inputs or outputs? What logic level lights the LED?



Example: Buttons

Consider the buttons on the Launchpad board:



Are the buttons inputs or outputs? **Inputs! We "read" values from them.**

Consider S1. What logic level indicates the button is pressed? What about when it is unpressed?