02: Digital I/O, instructions and programs, hardware abstraction

Microcontrollers

Stefan Huber
`www.sthu.org`

Dept. for Information Technologies and Digitalisation
FH Salzburg

Winter 2024

Digital I/O

Ports

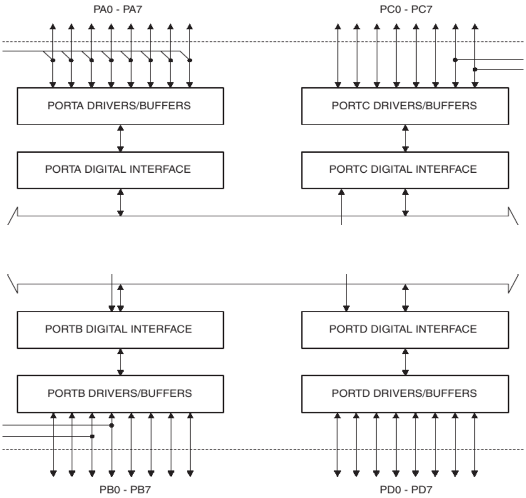
Digital I/O is a basic feature of a microcontroller:

- The ATmega32 has ports A–D with 8 pins each.
- They can be used to read or write logical 1 or 0 on each individually.

Ports often have alternate functions. For the ATmega32:

- Port A: A/D converter
- Port B: SPI, etc.
- Port C: JTAG, two-wire serial, etc.
- Port D: USART, ext. interrupts, etc.

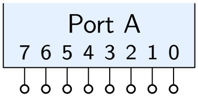
The Raspberry Pi has up to 6 alternative functions for a pin.



Digital I/O basics

Digital I/O of each of port x is controlled by three registers[1]:

- **DDRx** Data Direction Register: A bit 1 means output, a 0 means input.
- **PORTx** Port Register: A bit 1 sets output voltage to logical 1, and otherwise 0 (if pin is configured as output).[2]
- **PINx** Port Input Register: A bit 1 means that the pin's voltage reads as logical 1, and otherwise 0.



DDRA:	1 0 0 0 0 0 1 1	0x83:	pin 0, 1 and 7 are output, all others input
PORTA:	1 0 0 0 0 0 0 1	0x81:	pin 0 and 7 drives high, pin 1 drives low
PINA:	1 0 1 1 0 0 0 1	0xb1:	sense high at pin 0, 4, 5, and 7, all others low

1 See [ATmega32, p. 49].

2 And PORTx used to configure pull-up resistors for input pins, see later.

Digital I/O demo

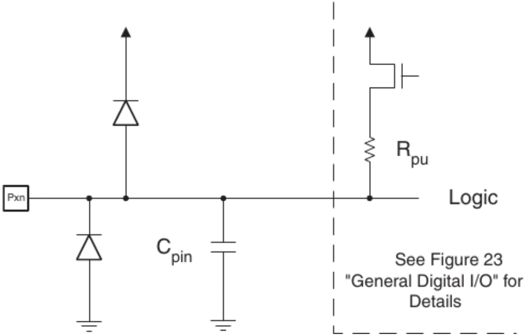
Using bit operations, we read, write and flip bits in control registers.

```c
#include <stdbool.h>
#include <avr/io.h>

int main() {
    /* On port B, set pins 0..1 to output and pins 2..7 to input. */
    DDRB = 0x03;

    /* Change the output pins 0..1 to high on port B. */
    PORTB |= 0x03;
    /* Change the output pin 0 to low on port B. */
    PORTB &= ~0x01;
    /* Flip pin 1 on port B (high to low, low to high). */
    PORTB ^= 0x02;

    /* Read level of pin 5 on port B. */
    bool pin5 = PINB & (1 << 5);
}
```

Protection diodes to Vcc and Gnd.

Configurable pull-up resistor of 20 kΩ to 50 kΩ:

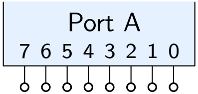
- Without pull-up resistor an input pin is floating if level is not driven.
  Hence, pin is prone to noise, e.g., when using mechanical switches.
- With a pull-up resistor the potential is pulled to Vcc.
  Hence, if pin is not driven (e.g. not connected) then we read a logical one.
  But if pin is driven to ground potential then we have power consumption at the pull-up resistor: The pull-up resistor acts as a load to the driving potential of the pin.



Configuring pull-up resistors

For input pins the PORTx register configures the pull-up resistor:

- A bit 0 means without pull-up resistor, a bit 1 means with pull-up resistor.



DDRA: 1 0 0 0 0 0 1 1 0x83: pin 0, 1 and 7 are output, all others input

PORTA: 0 0 1 1 0 0 0 0 0x30: pin 4 and 5 with pull-up, pins 2, 3, and 6 without

```c
#include <avr/io.h>

int main() {
    /* On port B, set all pins to input. */
    DDRB = 0x00;
    /* Activate pull-up resistor for pins 0..3 (and deactivate for 4..7). */
    PORTB = 0x0f;
}
```

Read-modify-write

Assume we want to *change* Pin 3 of Port B to output.

- In assembler there are instructions `SBI`, `CBI` to set or clear a bit atomically in one cycle.
- In C we use a read-modify-write access.
  - This is not atomic! In fact, `DDRB` may have been altered *between* read and write, e.g., by an interrupt.

```
/* Change bit 3 of DDRB to 1. Does not happen in one cycle. */
DDRB = DDRB | (1 << 3);
/* In a shorter notation. */
DDRB |= (1 << 3);
/* There is a preprocessor definition for Port-B-Pin-3. */
DDRB |= (1 << PB3);
```

Code style

Prefer makro `PB3` over 3, because `PB3` tells you mean a pin, not just a number.

Datatypes in C

Standard arithmetic data types[3] for (signed) integers in C and their minimum size are

Type	char	short	int	long	long long
Min. size (bytes)	1	2	2	4	8

The actual size of the above data type is *not* defined by the C programming language. However, there are common data models:

Model	char	short	int	long	long long	void*	
IP16	1	2	2	4	8	2	avr-gcc[4], MS-DOS
ILP32	1	2	4	4	8	4	typical 32-bit OS
LLP64	1	2	4	4	8	8	64-bit Windows
LP64	1	2	4	8	8	8	typical 64-bit UNIX-like OS

3 Since C99 there is a datatype for boolean values, too.

4 See [AVR-GCC-wiki] for details.

New data types in C99

The C99 standard adds `inttypes.h` as header file with platform independent integer data types:

Size in bytes	signed	unsigned
1	`int8_t`	`uint8_t`
2	`int16_t`	`uint16_t`
4	`int32_t`	`uint32_t`
8	`int64_t`	`uint64_t`

The C99 standard also adds a header file `stdbool.h` with a genuine boolean datatype `bool`.

```
/* A boolean is either false (0) or true (1). Tertium non datur! */
bool x = 2;
assert(x == true);
assert(x == 1);
assert(x != 2);
```

Code style

It is good practice to be explicit on the language standard, e.g., compiling with `gcc -std=c99 -pedantic`.

Bit handling in C

```c
void bitdemo() {
    uint8_t x, y;


    x = 0xa5;

    y = 1 << 6;


    y = x & (1 << 6);


    x |= (1 << 3);

    x &= ~(1 << 2);


    y = !!y;
}
```

Bit handling in C

```c
void bitdemo() {
    uint8_t x, y;

    /* x is binary 1010 0101. (A common test pattern.) */
    x = 0xa5;
    /* y is 0000 0001 shifted left by 6, which is 0100 0000. */
    y = 1 << 6;
    /* y is true if bit 6 of x is set. That is, if bit-6 of x is set then y is
     * (1 << 6), otherwise 0. */
    y = x & (1 << 6);

    /* Set bit 3 of x. That is, bitwise or of x with 0000 1000. */
    x |= (1 << 3);
    /* Clear bit 2 of x. That is, bitwise and of x with 1111 1011. */
    x &= ~(1 << 2);

    /* Double logicial negation: Turns *any* true into 1 and leaves false as 0. */
    y = !!y;
}
```

Bit handling with C macros

```c
/** Returns a word with only bit-th bit set. Mind the parentheses! */
#define BIT(bit) (1ull << (bit))
/** Raise bit-th bit in word. */
#define BIT_SET(word, bit) ((word) |= BIT(bit))
/** Clear bit-th bit in word. */
#define BIT_CLR(word, bit) ((word) &= ~BIT(bit))
/** Returns BIT(bit) if bit-th bit of word is set and 0 otherwise. */
#define MASK_BIT(word, bit) ((word) & BIT(bit))
/** Returns 1 if bit-th bit of word is set and 0 otherwise. */
#define BIT_IS_SET(word, bit) (!!MASK_BIT(word, bit))

void bitdemo() {
    uint8_t x=0xa5, y;
    /* y is 0100 0000. */
    y = BIT(6);
    /* y is true if bit 6 of x is set. */
    y = MASK_BIT(x, 6);
    /* Set bit 3 of x. */
    BIT_SET(x, 3);
    /* Clear bit 2 of x. */
    BIT_CLR(x, 2);
}
```

Instructions and programs

Instruction Set

The AVR CPU knows 131 instructions in five groups:

- Arithmetic and logical
- Branch
- Data transfer
- Bit and bit-test
- MCU control

Mnemonics	Operands	Description	Operation	Flags	#Clocks
ARITHMETIC AND LOGIC INSTRUCTIONS					
ADD	Rd, Rr	Add two Registers	Rd ← Rd + Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Add with Carry two Registers	Rd ← Rd + Rr + C	Z,C,N,V,H	1
ADIW	Rdl,K	Add Immediate to Word	Rdh:Rdl ← Rdh:Rdl + K	Z,C,N,V,S	2
SUB	Rd, Rr	Subtract two Registers	Rd ← Rd - Rr	Z,C,N,V,H	1
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd - K	Z,C,N,V,H	1
SBC	Rd, Rr	Subtract with Carry two Registers	Rd ← Rd - Rr - C	Z,C,N,V,H	1
SBCI	Rd, K	Subtract with Carry Constant from Reg.	Rd ← Rd - K - C	Z,C,N,V,H	1

Figure: See [ATmega32, p. 329].

There are two architectural styles for the instruction set: RISC and CISC

Instruction Set Architectures: CISC versus RISC

History in instruction set design:

- Hardware design was mature, but compilers were immature.
- Hence, make assembler programming easier by having powerful, complex instructions.
- Control units used to be hard-wired, and got more and more complex.
- At some point it was realized that control unit became a little “CPU” by itself: complex instructions formed by micro instructions executed by the control unit.
- For Intel processors, since Pentium Pro (1995), we can even update the microcode and so “patch the processor”.[5]

- **CISC**: Complex Instruction Set Computer
  The history described above
- **RISC**: Reduced Instruction Set Computer
  The counter movement towards simpler, cleaner instructions

5 The micro operations of the Intel microcode are themselves of RISC style.

CISC

- Instructions often take many cycles.
- Different instructions are encoded by codes of different lengths.
- Typically a register-memory architecture:
  - ALU operations can operate on memory directly
  - Complex memory addressing modes, e.g., array access on instruction level

```
% uname -om
x86_64 GNU/Linux
% objdump --disassemble /bin/ls
  16c8c:        c3                      retq
  16c8d:        0f 1f 00                nopl   (%rax)
  16c90:        c3                      retq
  16c91:        66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
% objdump --disassemble /usr/lib32/libm.so.6            # An x86 binary rather than x86_
```

RISC

- Counter-movement to simple and hard-wired instructions.
  - Focus: Typically a program uses only few instructions most of the time (80/20 rule).
  - Complex instructions are substituted by a couple of simple ones.
- Each instruction takes one or a few cycles only and is encoded by a fixed size.
- Typically a load/store architecture:
  - ALU operations operate on registers only rather than directly in memory.
  - Hence, RISC computers often have many registers.

```
% uname -om
armv7l GNU/Linux
% objdump --disassemble /bin/ls
  24ee8:      e12fff1e      bx      lr
  24eec:      e59f300c      ldr     r3, [pc, #12]
  24ef0:      e3a01000      mov     r1, #0
  24ef4:      e08f3003      add     r3, pc, r3
  24ef8:      e5932000      ldr     r2, [r3]
```

Delay

A common machine instruction to all processors is `NOP`:

- No operation. Do nothing for a single cycle.
- Why does a `NOP` take a single cycle? Recall the CPU timing slide of last lecture.

```c
#include <inttypes.h>
#include <avr/io.h>
#include <avr/cpufunc.h>

uint8_t readback(uint8_t x) {
    PORTB = x;
    /* We need to wait one cycle until we can read back PINB. See fig. 25 of
     * ATmega32 data sheet. */
    _NOP();
    return PINB;
}
```

Delay

Waiting for a specific time requires a specific number of NOPs. A helper function hides that from us.

```
#define F_CPU 8000000
#include <util/delay.h>

void toggle_portb_forever() {
    while (1) {
        PORTB = ~PORTB;
        /* There is also a _delay_us(). */
        _delay_ms(1000);
    }
}
```

- It needs to know the CPU clock rate in Hz via the preprocessor definition `F_CPU`.
- It assumes that compiler optimizations are not turned off.

Code style

Do not `#define F_CPU` in the source code, but pass it as compiler flag, e.g., `avr-gcc -DF_CPU=8000000`. Hence, set this option in your Makefile or in your project configuration.

Programming

Modify-compile-run on a general-purpose OS:

- The compiler outputs a binary that can be executed by the OS.

For a microcontroller:

- The development machine typically has a different architecture. It runs a cross-compiler to produce output for a target architecture.
- Programming hardware – like the Atmel JTAGICE3 – takes a hex file, connects to the microcontroller, writes the program into the Flash memory, and then the microcontroller resets to execute the new program.

Life cycle of a program

Microcontroller programs typically do not terminate.

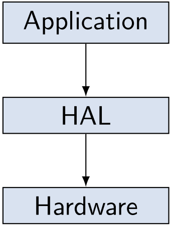
- Each program typically has two phases:
  - First some initialization phase
  - Then a loop of cyclic work
- Unexpected stops are prohibitive in most control tasks.
  - Unexpected C++ exceptions, out of memory situations, floating-point exceptions, invalid memory access, et cetera must not happen or must be dealt with gracefully!

Hardware Abstraction

Hardware Abstraction Layer

In a clean software architecture it is easy to make changes.[6]

- In embedded systems, hardware is diverse, and therefore might change.
  - Things that change: pin numbering schemes, offset addresses, timing details, ...
- Changing hardware should be easy in the software architecture of embedded systems.
- Hence, we add abstraction of hardware, by a hardware abstraction layer (HAL).
  - Gives a three-layered architecture pattern



Uses abstractions, independent from concrete hardware,
e.g., draws a line

Provides device abstractions, hides details of hardware,
e.g., provides a graphics API

The concrete hardware, which might be replaced,
e.g., implements display driver

[6] Compare with the *Liskov substitution* principle in OOP, which is the L in SOLID.

HAL: LED example

Example LED:

- An abstract LED can be turned on, turned off, toggled and one can read the state.
- Hardware details are hidden: Setting port pin to output mode, maintaining or reading state when toggeling, et cetra.

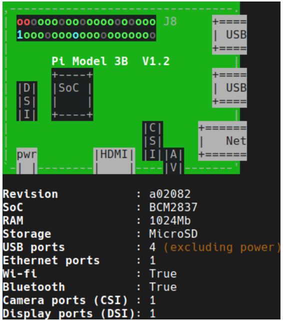
Concrete drivers in hardware layer:

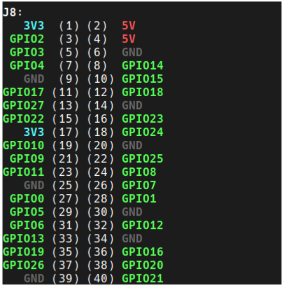
- A physical LED connected to a port of the ATmega32
- A LED bar connected via a communication interface

However, the HAL presents an abstract LED to the application. The application does not depend on the concrete driver.

The Raspberry Pi provides so-called GPIO pins which can be used for digital I/O and much more.

- The command `pinout` on Raspbian gives us a visual representation:





The GPIO pin numbering changed with hardware revisions of the BCM SoC.

Wiring Pi comes with a tool `gpio` for debugging.

- It shows the pin numbering and levels, can modify pins, output PWM signals, et cetera.

```
$ gpio blink 23      # Let GPIO 13 (wiring pi pin 23) blink
$ gpio readall
 [...]
 |   6 |  22 | GPIO.22 |   IN | 1 | 31 || 32 | 0 | IN   | GPIO.26 | 26  | 12  |
 |  13 |  23 | GPIO.23 |  OUT | 0 | 33 || 34 |   |      | 0v      |     |     |
 |  19 |  24 | GPIO.24 |   IN | 0 | 35 || 36 | 0 | IN   | GPIO.27 | 27  | 16  |
 |  26 |  25 | GPIO.25 |   IN | 0 | 37 || 38 | 0 | IN   | GPIO.28 | 28  | 20  |
 |     |     |      0v |      |   | 39 || 40 | 0 | IN   | GPIO.29 | 29  | 21  |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+---Pi 3B--+---+------+---------+-----+-----+
```

Wiring Pi hides these details by defining its own number scheme that hides changes in hardware.[7]

- The Wiring Pi number scheme leaves physical positions untouched, where as the BCM numbering scheme may change.

[7] *Wiring Pi Pins*. URL: http://wiringpi.com/pins/

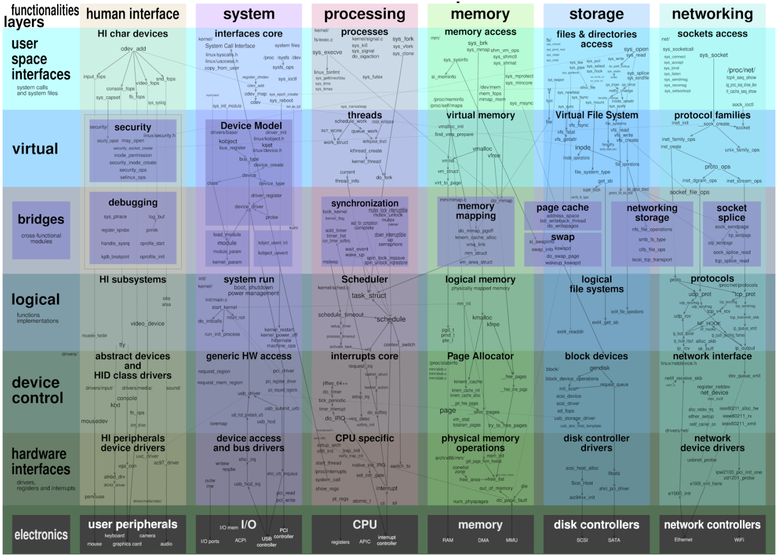


Figure: The Linux kernel map. Source: `https://www.makelinux.net/kernel_map/`

References I

[ATmega32] *ATmega32: 8-bit AVR Microcontroller with 32KBytes In-System Programmable Flash*. Atmel Corporation. Feb. 2011.

[AVR-GCC-wiki] *AVR GCC*. URL: https://gcc.gnu.org/wiki/avr-gcc.

[ISO18037] *Programming languages – C – extensions to support embedded processors*. Standard ISO/IEC TR 18037:2008. International Organization for Standardization, June 2008. URL: https://www.iso.org/standard/51126.html.

[wiringpi] *Wiring Pi Reference*. URL: http://wiringpi.com/reference/.

[wiringpi-pins] *Wiring Pi Pins*. URL: http://wiringpi.com/pins/.

Programming languages

Choice of the programming language:

- Limited amount of memory, special-purpose peripherals, programming close to hardware and direct access to registers or memory.
- Dynamic memory allocation is often prohibitive, in particular for real-time systems.
- Still, there are projects like MicroPython for microcontrollers.

Assembly:

- Rarely used for development anymore, but still for debugging.
- Direct control over the sequence of machine instructions and timing.
- When compiler is not available or to emit certain machine instructions.

C:

- The typical choice for hardware-related and embedded software development.
- Some microcontrollers require non-standard dialects of C. Many manufacturers ship their own IDE and/or own compiler.
- There is an embedded C standard [ISO18037], which adds, e.g., fixed-point arithmetic.

Blink demo with Wiring Pi

```c
#include <stdlib.h>
#include <unistd.h>
#include <wiringPi.h>

int main() {
    /* WiringPi requires some setup. */
    wiringPiSetup();
    /* Make Wiring Pi pin 23 (GPIO 13 on model 3B) an output pin. */
    pinMode(23, OUTPUT);

    digitalWrite(23, HIGH);
    usleep(200000);
    digitalWrite(23, LOW);
    return EXIT_SUCCESS;
}
```

Documentation:

- *Wiring Pi Reference.* URL: http://wiringpi.com/reference/