Embedded Systems Fundamentals

ENGD2103

Dr M A Oliver

michael.oliver@dmu.ac.uk

Lecture 4: Bare Metal and Introduction to Concurrency

Contents

This lecture will include two topics:-

- Bare metal:
  - Using register-level access to drive I/O pins.
- Concurrency
  - Getting the processor to do multiple tasks ‘at once’
  - Taking our first steps……

Bare Metal

To perform digital I/O access, we are familiar with the following Arduino library functions:-

```
pinMode()
digitalWrite()
digitalRead()
```

- These were designed for the Arduino.
- Using these is straightforward, but restrictive.
- Using these functions is not a transferrable skill.

Bare Metal

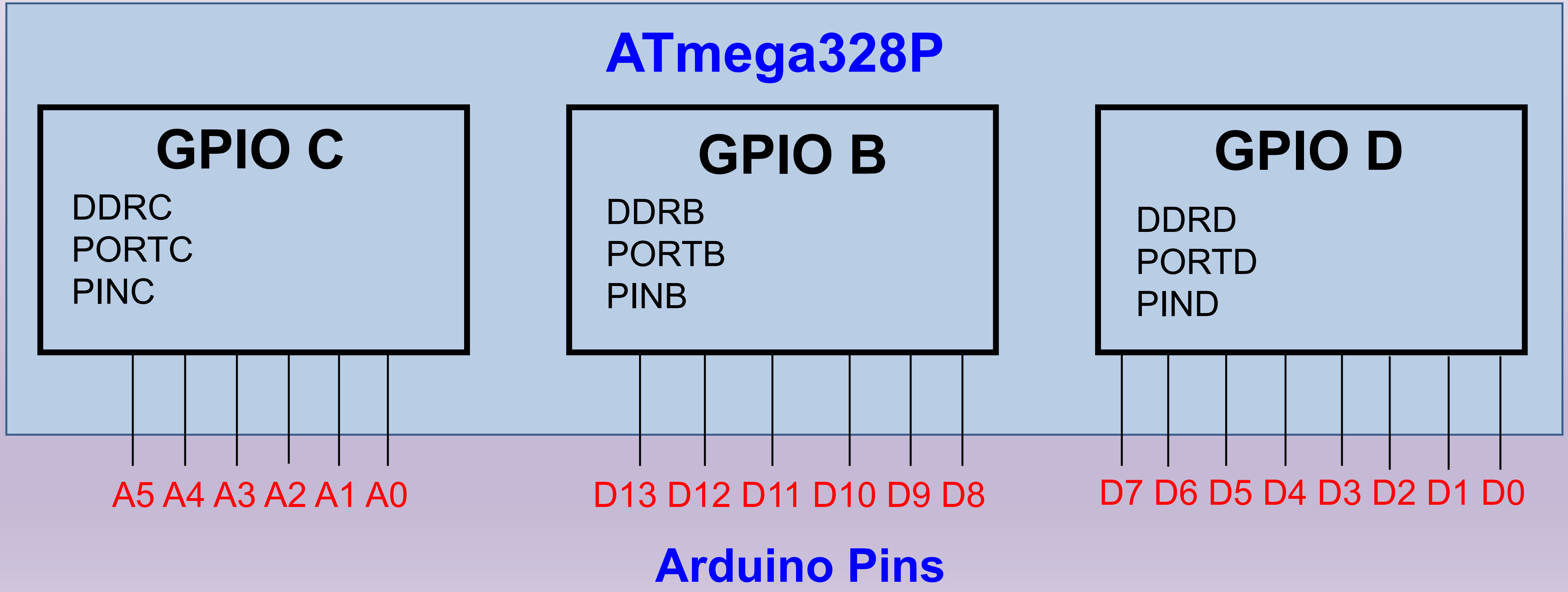
To perform digital I/O access with bare metal, register-level access is required.

- Reading and writing to a register is like accessing memory.
- The main difference is that writing to certain bits in a register can activate / deactivate subsystems within the microcontroller
- Reading certain bits in a register can provide status information about subsystems within the microcontroller.
- Read the ATmega328P datasheet!

Principles that will be learned here can be applied to most microcontrollers……

Bare Metal

- ATmega328P contains 3 ‘ports’ that drive the I/O lines
  - Port B, Port C, Port D. There is no Port A.



Bare Metal – GPIO D

The three registers associated with GPIO D

DDRD

Data direction register D

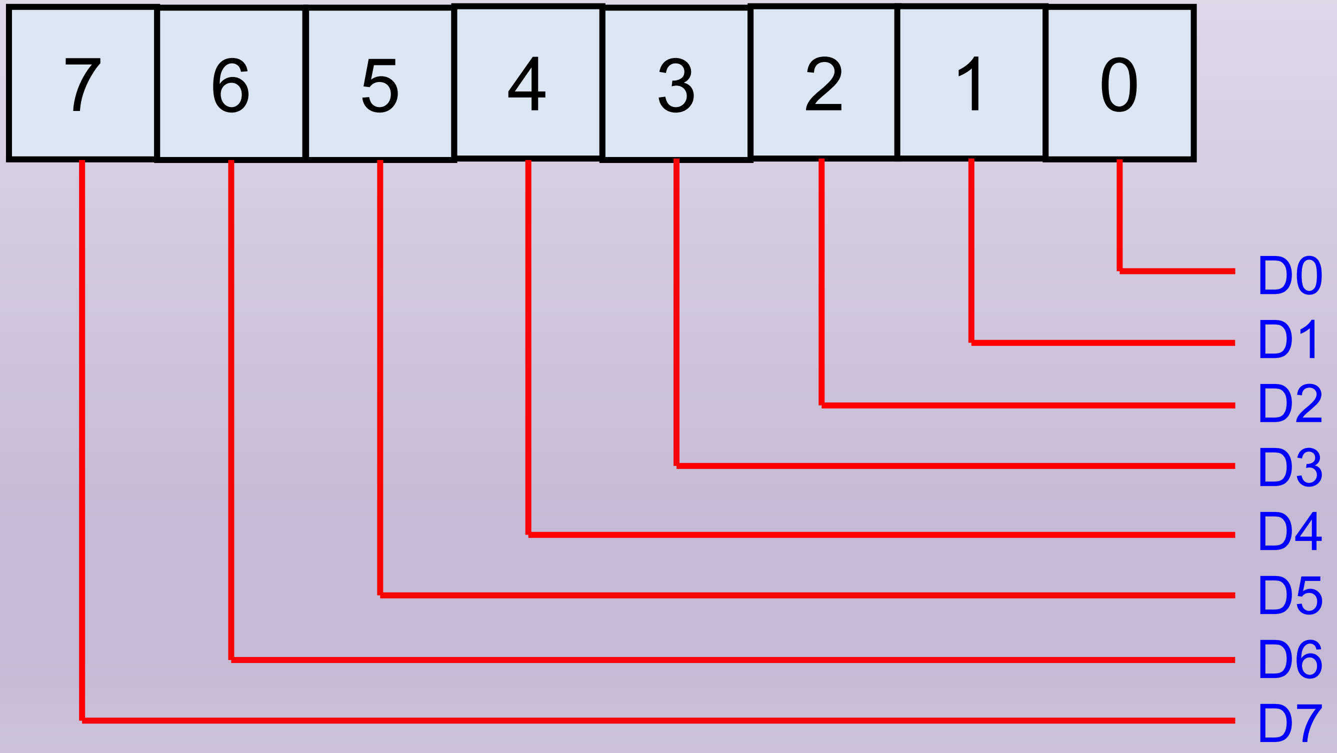
PORTD

Output register D

PIND

Input register D

For each register, each bit controls a digital I/O line



Bare Metal – GPIO B

The three registers associated with GPIO B

DDRB

Data direction register B

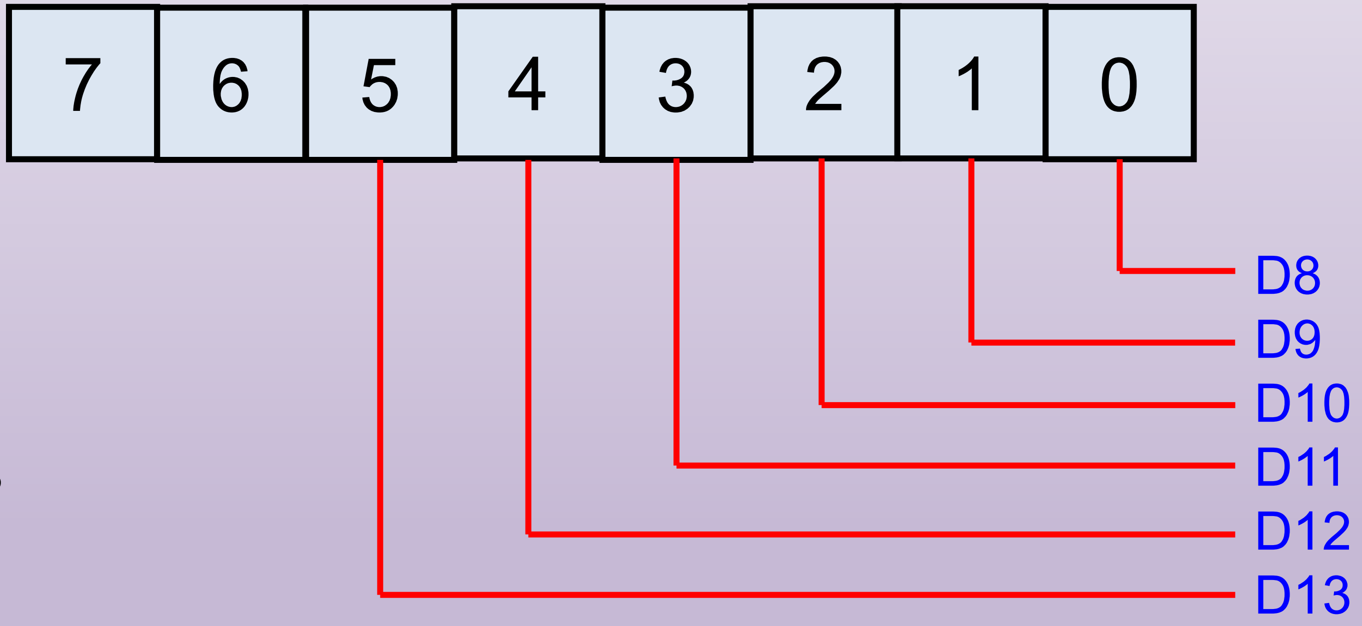
PORTB

Output register B

PINB

Input register B

For each register, most bits control a digital I/O line



Bare Metal – GPIO C

The three registers associated with GPIO C

DDRC

Data direction register C

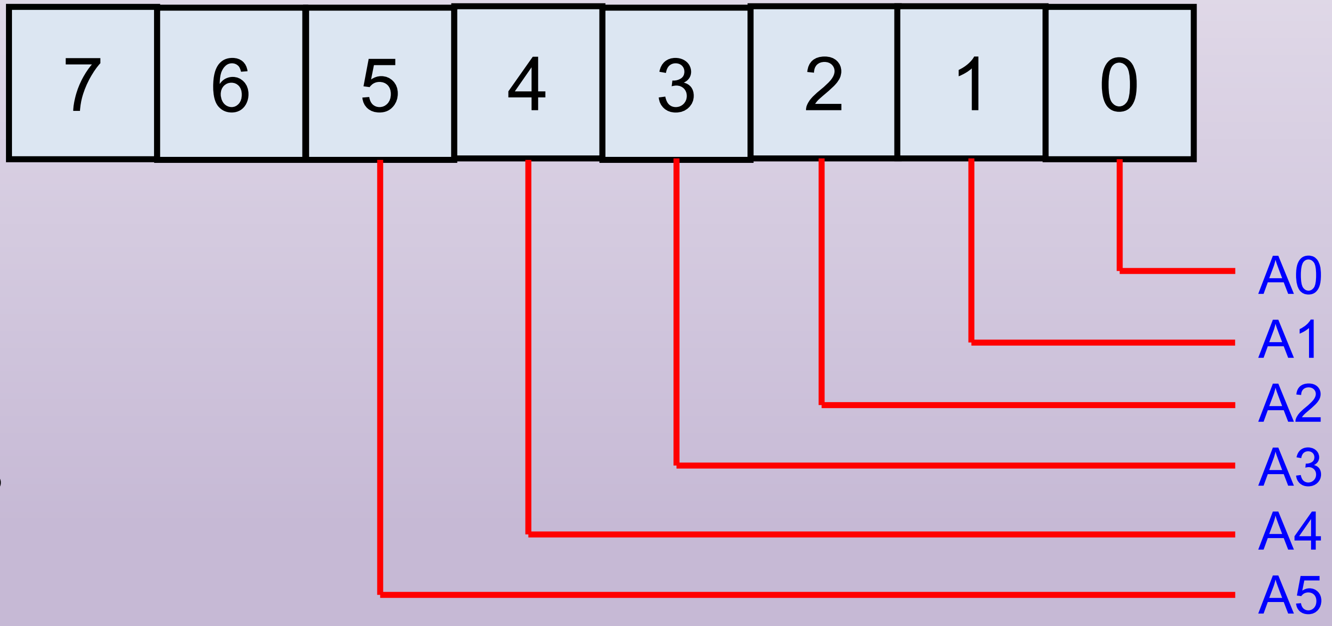
PORTC

Output register C

PINC

Input register C

For each register, most bits control a digital I/O line



Bare Metal
Data Direction Registers

On the ATmega328P, these registers are denoted **DDRx**, where x could be B, C or D

Setting a bit high in a DDRx register makes the corresponding line an OUTPUT

Setting a bit low in a DDRx register makes the corresponding pin an INPUT

Bare Metal
Data Output Registers

On the ATmega328P, these registers are denoted **PORTx**, where x could be B, C or D

- Setting a bit high in a PORTx register causes the corresponding line to go HIGH
- Setting a bit low in a PORTx register causes the corresponding line to go LOW

The previous clauses assume the port pin has been configured as an output. Suppose it is configured as an INPUT:-

  - Setting a bit high in a PORTx register makes the line an INPUT_PULLUP.
  - Setting a bit low in a PORTx register essentially makes the line an INPUT

Bare Metal
Data Input Registers

On the ATmega328P, these registers are denoted **PINx**, where x could be B, C or D

These are read-only and give the states of all inputs associated with that port.

If a particular bit needs analyzing, bit-wise AND-ing is required to mask-off the bits that are not of interest.

Bare Metal
Example: LED Output

Suppose an LED exists on digital pin 2.

- This corresponds to GPIO D, pin 2.

Firstly define the pin:-

```
#define LED           B00000100        // Bit 2 is set
```

To make this an output, in `setup()` use:-

```
DDRD |= LED;
```

Then define the following macros to control the LED:-

```
#define LED_ON        PORTD |= LED
#define LED_OFF       PORTD &= ~LED
```

Bare Metal Example: Switch Input

Suppose a switch exists on pin A0 of the Arduino Nano.

- This corresponds to GPIO C, pin 0.

Firstly define the pin:-

```
#define SW1          B00000001      // Bit 0 is set
```

To make this an input with pullup, in `setup()` use:-

```
DDRC &= ~SW1;        // PC.0 becomes an input
PORTC |= SW1;        // with pullup.
```

Then define the following macros to read the status of the switch:-

```
#define SW1_RELEASED  (PINC & SW1)    // True on release
#define SW1_PRESSED   !SW1_RELEASED   // True on press
```

Bare Metal Considerations

Consider a **`digitalWrite()`** function call. What happens?

- The `digitalWrite()` function is ‘called’.
- The arguments (Arduino pin and value) are checked.
- The port and port-pins are determined
- The PORTx register is written
- The function ‘returns’ to the calling program

Now consider a bare-metal register write. What happens?

- The PORTx register is written

Clearly there is a lot of overhead with `digitalWrite()`. The bare-metal approach is significantly faster.

Bare Metal Considerations

With a HAL, converting from Arduino libraries to bare-metal is straightforward.

A definition such as

```
#define GREEN_LED           4
#define HAL_greenLedOn      digitalWrite(GREEN_LED, HIGH)
```

Now becomes

```
#define GREEN_LED           B00010000
#define HAL_greenLedOn      PORTD |= GREEN_LED
```

Bare Metal Considerations

It is possible to access multiple bits in a single bare metal operation. For example, to switch on all 6 traffic lights LEDs (on pins 7-2)

```
DDRD = B11111100;
```

- Significantly faster than 6 digitalWrite() calls!

Most microcontrollers have data direction registers, output registers and input registers. This approach is now a transferrable skill.

Register-level access can also be used to create your own `millis()` function, or your own Wire library – but this goes beyond the scope of the module.

Introduction to Concurrency

- Suppose there is an LED blink program:-

```
void loop()
{
    HAL_ledOn;          // Turn the LED on
    delay(500);         // Wait for 500 ms
    HAL_ledOff;         // Turn the LED off
    delay(200);         // Wait for 200ms
}
```

- How do you get the microcontroller to perform other tasks?
- Potential to add code below the 4 lines. However need to wait 700ms for the process to complete.
- The Arduino `delay()` function is a blocking delay.
  - Processor can do nothing until the delay() function has completed execution.
- How can we get the microcontroller to perform multiple tasks concurrently?

Introduction to Concurrency

- The microcontroller can only execute one line of C-code at a time.
- As `delay()` is blocking we need to forsake this.
  - If is executing, nothing else can be executed.
- The good news is the Arduino **`millis()`** function
  - This returns the number of milliseconds since the microcontroller was last reset.
  - The returned value is of type `unsigned long`
  - `unsigned long` is 32-bits wide representing values between 0 and 2^{32} (4,294,967,296).
  - `millis()` returns values representing times up to 4,294,967,296 ms, i.e. 49.7 days!

Introduction to Concurrency

- By taking advantage of `millis()` and looking at time differences, it is possible to create self-contained "code modules" that perform tasks at prescribed time intervals.
- For each code module a global variable can be declared, for example:

```
bool init_module0_clock;
```

The "code module" can be initialized in setup() using

```
void setup(){
  init_module0_clock = true;
}
```

“Code Module”, runs inside `loop()`

```
{ // module 0
   static unsigned long module_time, module_delay, debounce_count;
   static bool module_doStep;
   static unsigned char state; // state variable for module 0

   if (init_module0_clock) {
     module_delay = 500;
     module_time = millis();
     module_doStep = false;
     init_module0_clock = false;
     state=0;
   }
   else {
     unsigned long m = millis();
     if ( (m - module_time) > module_delay ) {
       module_time = m;
       module_doStep = true;
     }
     else module_doStep = false;
   }

   if (module_doStep) {
     // Do your task here
   }
 }
```

Summary

- Covered bare-metal coding for GPIO
  - Transferrable skill; the principles learned here can be applied to most microcontrollers.
- Introduced concurrency
  - Pros:
    - Works well
    - Easy to implement
  - Cons:
    - Unstructured, messy, ugly code with multiple code modules
    - Difficult to maintain.
  - Next time, we will look at improving the structure.