The Device Driver

CM0506 – Small Embedded Systems

Dr Alun Moon

Department of Computer and Information Science

Lecture 4

Interpretation of Hardware Specifications

Typical hardware specifications include:

- Functional description
- Pinout specifications
- Operating voltages (Minimum, maximum, and typical)
- Timing Diagrams
- Protocol Diagrams
- Critical timing data

What is a device driver?

- A collection of software routines to perform I/O functions
- Interface software, called by the operating system or application code, to configure devices and perform I/O
- Software to *glue* the hardware and software together
- Separates policy from mechanism

A Device Driver

- Encapsulates the behaviour of a device
- Allows application developers to ignore low-level detail
- A consistent interface to a device or family of devices

Device Driver code

- Notoriously difficult to design and debug
- May be complex
- Requires a deep understanding of the hardware
- Low-level code – sometimes requires assembly language
- API (Application Programming Interface) requires careful design

Portability

- Device drivers provide a layer of abstraction to hardware I/O devices
- Higher levels of software can access devices in a uniform hardware-independent manner
- If designed well, device driver software can be ported.

Developing Device Drivers

- Read the hardware specification
- Re-read the specification, review in a group
- Specify an API and review this
- Design and develop code to provide the API and consistent with hardware specifications
- Test the API carefully – use instumentation, and simple, incremental, text harness software.

Typical Driver Functions

- Configure a device – initialise the hardware to a known state
- Turn a device on or off
- Assign interrupt handlers
- Read data from a device
- Write data to a device

API design

A good API should

- have clearly named functions and values
- do what is expected (principle of least surprise)
- hide unimportant implementation details
- follow conventional patterns, if appropriate (eg. `putc()`)

Case study

LED driver

function configure and switch LEDs on and off

hardware the LEDs are connected to a variety of pins and ports, with differing circuits

- different LEDs require different logic values to turn on

encapsulation API should isolate hardware dependencies

uniformity API should treat all LEDs consistently

LED Driver

Published API

The API is "published" through the header file (`led.h`) that programs `#include` to access the functions.

1. Provide labels (symbols) for LEDs
   - Physical names matching PCB and Schematics
   - Logical names, better descriptive names or functional names
2. Access LEDs via functions with symbolic names as arguments
3. Functions to:-
   - initialise IO to drive LEDs
   - turn LEDs on and off
   - interrogate LED state
   - toggle state of LED

LED Driver

led.h

```
enum LED {
        /* PCB names */
        LED1, LED2, LED3, LED4,
        /* logical names */
        left_green=LED1, right_green,
        left_blue,       right_blue,
};
```

Enumerated Types are useful here

- Automatically provide (arbitrary/independent) values
- Provide a type against which to check values and parameters
- Compiler has extra support for switch (more later)

LED Driver

led.h

```
void led_init(void);

void led_on    (enum LED name);
void led_off   (enum LED name);
void led_toggle(enum LED name);

int  led_state (enum LED name);
```

Function prototypes declare available functions that make up the API

- *Note* the use of enumerated types as parameters
- Compiler can check that one of the symbolic values is passed, or value is defined in the enumeration

LED Driver

Implementation extract

```
enum ledmasks {
        nil,
        led1pin=(1UL << 18), /* port 1 */
        led2pin=(1UL << 13), /* port 0 */
        led3pin=(1UL << 13), /* port 1 */
        led4pin=(1UL << 19), /* port 2 */
};
```

Internal values for convenience in writing code

- bit masks

LED Driver

Implementation extract

```
void led_on    (enum LED name){
  switch(name) {
    case LED1:
      LPC_GPIO1->CLR = led1pin;
      break;
    case LED2:
```

Switch statement to select action based on symbolic name

- Compiler can test to see if all cases are covered.

LED Driver

Implementation extract

```
case LED1:
  LPC_GPIO1->MASK = ~led1pin; /* mask (hide) not the led pin */
  LPC_GPIO1->PIN ^= led1pin;
  LPC_GPIO1->MASK = nil;      /* don't mask (hide) anything  */
  break;
```

- Make use of hardware features to make code simple
- may make code faster (less operations in software)
    - but measure don't assume

LED Driver

Implementation extract

```
int  led_state (enum LED name)
{
  int state = 0;
```

- local variables with default values

```
  LPC_GPIO1->MASK = ~led1pin;
  state = LPC_GPIO1->PIN;
  LPC_GPIO1->MASK = nil;
```

- set to values where known

```
  return state;
}
```

- C convention (ANSI C89)
  - return logic values as int
  - zero false, non-zero true

Buttons

```
#define <stdio.h>
```