How to structure embedded software

Setting the scene

Aim is to understand the basic structure of embedded software

Embedded software is often divided into 3 layers

- the top layer is the application itself
- the middle layer contains the *device drivers* for the component parts of the application
- the lowest layer is the target platform - called the hardware abstraction layer (HAL)

The three layers

application layer

- wash programmes
- cancel button stops wash

device layer

- door
- buttons
- motor

target hardware layer

- parallel ports
- serial ports
- network interface

Target hardware layer

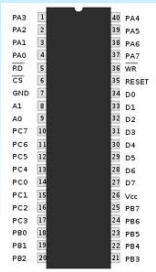
Target hardware

A **port** is the physical connection between the microprocessor and external device

A port may provide

- a serial interface, where a byte is transmitted as a sequence of bits
- a parallel interface, where separate pins represent each bit of a byte
- ethernet, graphics, video, etc

Peripheral interface devices



Generally targets have a number of *ports* that perform I/O and a control register to configure the device

8255 Programmable Peripheral Interface		
00	port A	8 bits
01	port B	8 bits
10	port C	8 bits
11	control register	8 bits

For the 8255 parallel device, the control register can configure the ports as input or output, unidirectional handshake, or bidirectional (with handshake). Bits in the control register are used to implement the handshaking.

How does a CPU talk to a port?

Two types of addressing scheme are used

- **Port mapped I/O** (PMIO) peripherals are attached to a separate, I/O dedicated address space
- **Memory mapped I/O** (MMIO) peripherals are located in the main address space of the processor

Nearly all modern systems use MMIO

PMIO addressing

PMIO ports and registers are accessible using

- assembly-language instructions
- compilers targeted at that specific CPU

UART

port 2	0x08001003
reg 2	0x08001002
port 1	0x08001001
reg 1	0x08001000

PMIO

```
int main()
{
  unsigned char val;
  val = input(0x08001001);
  output(val, 0x08001001);
}
```

MMIO addressing

UART

port 2	0x08001003
reg 2	0x08001002
port 1	0x08001001
reg 1	0x08001000

Ports and registers look like ordinary variables stored at a specific memory location

MMIO

```
int main()
{
  unsigned char val;
  volatile unsigned char *uartp = (unsigned char*) 0x08001000; // C style cast
  // volatile unsigned char *uartp = reinterpret_cast<unsigned char*>(0x08001000); // C++
  *uartp = 0; // write to register 1
  val = *uartp; // read register 1 – NOTE that it may not be possible to read registers
}
```


Defining ports in C/C++

`volatile` specifies that the actual port is always read, not a register or cached value

convert an integer constant to an address holding an `unsigned char`

```
volatile unsigned char *uartp = (unsigned char *) 0x08001000;
```

match the type to the number of port bits - for the ARM compiler this is `char` for 8, `short` for 16 and `int` for 32

we are just interested in the bit pattern, so use `unsigned`

Modelling multiple ports

- the peripheral interface could be represented as a `struct` or, much better, as a `class`
- the keyword `volatile` is applied to the members

```
// Parallel port example has three consecutive memory locations, each a byte
struct ParPort
{
  volatile unsigned char reg;   // used to configure port1 and port2
  volatile unsigned char port1; // assume this port is for input only
  volatile unsigned char port2; // assume this port is for output only
};

int main() {
  // define a pointer to the parallel port
  struct ParPort *PPort = (struct ParPort *) 0x08001000;

  // access the parallel port
  PPort->reg = 0x4A;   // configure the ports
  unsigned char read_value = PPort->port1;   // read a value from the port
}
```


Accessing ports

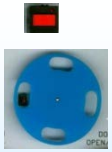
Individual parallel ports will normally be configured as input or output

- you can't always read an output port to see the last values you send there
- so, you may need to keep a copy of the outputs you send to a port

```
unsigned char Port2;  // assume this is an output port

Port2 = 0x2E;  // work out the value to send to the port
PPort->port2 = Port2;  // update the port while keeping a copy
```

Device layer


Many devices, single port

Often, a number of devices will be connected to a single port

For input ports, we need to be able to read from the port and extract the value of the specific device

switch4	switch3	switch2	switch1	switch0	sensor1	sensor0	button

For output ports, we need to be able to write to a specific device without affecting the other connected devices

motor 2 clkwise	motor 2 on	motor 1 clkwise	motor 1 on	LED3	LED2	LED1	LED0

Reading from input ports

```
struct ParPort
{
  volatile unsigned char reg;
  volatile unsigned char port1;
  volatile unsigned char port2;
};
int main()
{
  // define a pointer to the parallel port
  struct ParPort *PPort = (struct ParPort *) 0x08001000;
  // get the input port value
  unsigned char Port1;   // assume this is an input port
  Port1 = PPort->port1;  // read the input port
  // extract the button and switch values
  unsigned char Button = Port1 & 0x01;
  unsigned char Switch = Port1 & 0xF8;
}
```

port 1

switch4	switch3	switch2	switch1	switch0	sensor1	sensor0	button
7	6	5	4	3	2	1	0

Writing to output ports

```
// Parallel port structure as before
int main() {
  // define a pointer to the parallel port
  struct ParPort *PPort = (struct ParPort *) 0x08001000;
  // copy of the output port value
  unsigned char Port2 = 0x08;  // start with only LED3 on
  PPort->port2 = Port2;  // send to port
  // turn LED1 on
  Port2 = Port2 | 0x02;  // LED1 on, leaving other outputs unchanged
  PPort->port2 = Port2;  // send to port
  // turn motor 1 on, turning clockwise
  Port2 = Port2 | 0x30;  // motor on and clockwise, leaving other outputs unchanged
  PPort->port2 = Port2;  // send to port
  // turn LED3 off
  Port2 = Port2 & ~0x08; // LED3 off, leaving other outputs unchanged
  PPort->port2 = Port2;  // send to port
}
```

port 2

motor 2 clkwise	motor 2 on	motor 1 clkwise	motor 1 on	LED3	LED2	LED1	LED0
7	6	5	4	3	2	1	0

Comments on the code

The above code mixes the hardware and the target layer

➢this will mean the code has to be changed if *either* the target *or* the application components are changed

The code for the layers should be separated, so as to

•make as few function calls between the layers as possible (low coupling)

•keep all the code for a layer together (high cohesion)

Device layer

The device layer has *drivers* that hide the complexity of interfacing to an application's components

- the device driver is typically accessed using function calls
- once operational and properly tested, the details of the implementation of this code can be ignored

To access devices connected to a PC (say a USB stick), the programmer uses the OS drivers

In embedded systems, an RTOS may be used, or device drivers could be specifically written

Benefits of device drivers

Modularization into a set of device drivers makes the software structure easier to understand

Only one module (a specific device driver) ever interacts with a specific application hardware component

If a hardware component is changed only its device driver needs to be modified

If a layered structure is used, if the target is changed, the device driver often does not need to be re-written

Requirements of device drivers

They may have an initialization routine (constructor in C++) to set the component to a known state

The defined API (the set of functions to access components) should be simple and easy to understand

They should include variables that reflect the state of the hardware

2 layer example

hardware layer

```
struct ParPort
{
  volatile unsigned char reg1;
  volatile unsigned char port1;
  volatile unsigned char reg2;
  volatile unsigned char port2;
};
// define a pointer to the parallel port
struct ParPort *PPort = (struct ParPort *) 0x08001000;
```

mapping

```
// port 1 mappings
#define BUTTON 0x01
#define SENSOR0 0x02
#define SENSOR1 0x04
#define SWITCH0 0x08
#define SWITCH1 0x10
#define SWITCH2 0x20
#define SWITCH3 0x40
#define SWITCH4 0x80
```

device layer

```
class Switches {
  public:
    Switches(unsigned char);
    bool GetSwitch();
    void ReadSwitch();
  private:
    unsigned char port_map;
    bool value;
};
// initialize the port number
Switches::Switches(unsigned char map) {
  port_map = map;
}
// return the switch value
bool Switches::GetSwitch() {
  return value;
}
// get the switch value
void Switches::ReadSwitch() {
  if (PPort->port1 & map) {
    value = true;
  }
  else {
    value = false;
  }
}
```


Comments on the code

Think about improvements to the above code

- The HAL should be defined using a `class` rather than a `struct`
- The HAL class should have member functions to access the port's members - if the target is changed then a new HAL can be written *with the same interface*
- The device driver code needs to have greater independency from the HAL - it should call member functions and not access public members

Application layer

Application code

The complexity is hidden in the application layer and device layer code

The application code that uses the device driver simply needs to create a `Switches` object and call its functions

```
int main()
{
  Switches Switch0(SWITCH0);

  // read the switch value from the port
  Switch0.ReadSwitch();

  // show the switch values
  cout << "The switch value is : " << Switch0.GetSwitch() << endl;
}
```

Building on the application code

Assuming there are device drives for all components…

```
WashingMachine:WashingMachine()
{
  Switches Start(SWITCH0);
  Switches Spin(SWITCH1);
  Switches Cancel(SWITCH3);
  Motors Drum(MOTOR);
  ...

  // start the wash if the user has pressed the start button
  if (Start.Read()) Drum.On();
  ...
}
```


Summary

There are three layers to the structure of most embedded software

The code in each layer is written in such a way that it is independent of other layers

In this way, a change to the target, to the devices or to the application can be performed with minimal affect on the software in other layers