Reasons for Object-Oriented in PC Domain

1. We need a common user interface on top of a multitude of similar but not identical computers.
2. The hardware platform makes a fundamental advancement every 6 months.
3. Many companies act in concert to produce a product.
4. The newer software must run on the older computers.
5. The older software must run on the newer computers.
6. Hardware/software configuration may change at run time.

Object-Oriented in the Embedded Space

1. It is useful for embedded software to run on multiple microcomputers (i.e., code reuse).
2. Embedded microcomputers have a much longer lifetime than an x86 microprocessor.
3. Usually single company develops hardware and software.
4. Hardware and software are upgraded together.
5. Hardware and software are upgraded together.
6. Configurations are usually well-defined at compile time.
   • Still object-oriented does help portability and in providing a good hardware abstraction layer.

Threads

(See Figure 2.13)
Interrupts and Threads

(Slide 5)

Basic Concepts of Device Drivers

- A device driver consists of software routines that provide the functionality of an I/O device.
- Includes interface routines and low-level routines for configuring the device and performing actual I/O.
- Separation of policy and mechanism is very important.
- Interface may include routines to open, read, and write files, but should not care what device the files reside on.
- Require a good hardware abstraction layer (HAL).
- Low-level device drivers normally found in basic I/O system (BIOS) ROM and have direct access to hardware.

Recursion

- A program segment is reentrant if it can be concurrently executed by two (or more) threads.
- A recursive program is one that calls itself.
- When we draw a calling graph, a circle is formed.
- Recursive subroutines must be reentrant.
- Often easy to prove correct and use less permanent memory, but use more stack space and are slower.

```c
void OutUDec(unsigned int number){
    if (number>=10){
        OutUDec(number/10);
        OutUDec(number%10);
    }
    else
        OutChar(number+’0’);
}
```

(Slide 6)

Low-Level Device Drivers

- Low-level device drivers normally found in basic I/O system (BIOS) ROM and have direct access to hardware.
- Good low-level device drivers allow:
  1. New hardware to be installed.
  2. New algorithms to be implemented.
     (a) Synchronization with gadfly, interrupts, or DMA.
     (b) Error detection and recovery methods.
     (c) Enhancements like automatic data compression.
  3. Higher-level features to be built on top of the low level
     (a) Operating system features like blocking semaphores.
     (b) Additional features like function keys.

(Slide 7)

Basic Concepts of Device Drivers

- A device driver consists of software routines that provide the functionality of an I/O device.
- Includes interface routines and low-level routines for configuring the device and performing actual I/O.
- Separation of policy and mechanism is very important.
- Interface may include routines to open, read, and write files, but should not care what device the files reside on.
- Require a good hardware abstraction layer (HAL).
- Low-level device drivers normally found in basic I/O system (BIOS) ROM and have direct access to hardware.

(Slide 8)
**Device Driver Software**

- Data structures: global (protected)
  - `OpenFlag` Boolean that is true if keyboard port is open
- Initialization routines (public, called by client once)
  - `KeyOpen` Initialize the keyboard port
- Regular I/O calls (public, called by client to perform I/O)
  - `KeyIn` Input an ASCII character from the keyboard port
- Support software (protected)
  - `KeyHan` An interrupt service handler

---

**Serial Communication Interface (SCI)**

(See Figure 2.12)

---

**SCI Initialization**

```assembly
init
  ldaa #$33 ;1200 baud
  staa BAUD

ldaa #$00 ;mode
  staa SCCR1

ldaa #$0C ;tie=rie=0,
  staa SCCR2 ;te=re=1
rts
```

---

**SCI Input**

```assembly
InChar
  ldaa SCSR ;status
  bita #$20 ;rdrf?
  beq InChar

  ldaa SCDR ;SCI data
  rts
```
**SCI Output**

```
OutChar ldab SCSR ;status
bitb #$80 ;tdre?
beq OutChar
staa SCDR ;output
rts
```

**Input Decimal Number**

```
; Input a byte from the SCI
; Inputs: none
; Outputs: Reg B 0 to 255
; C=1 if error

DIGIT rmb 1 ;global
InUDec clrb ;N=0
InUDloop bsr OutChar ;Next input
 bsr InUDret ;Echo
 cmpa #13 ;done if cr
 beq InUDret ;with C=0
 cmpa #0
 blo InUDerr ;error?
 cmpa #9
 bhI InUDerr ;error?
```

**Input Decimal Number (cont)**

```
anda #$0F ;0-9 digit
staa DIGIT
ldaa #10
mul
tsta ;overflow?
bne InUDerr
addb DIGIT ;N=10+N+DIGIT
bra InUDloop
InUDerr ldaa #7
bsr OutChar
clr
sec ;error flag
InUDret rts
```

**SCI Output String**

```
; Output a string to the SCI
; Inputs: Reg X points to string
; String ends with 0
; Outputs: none

OutString ldaa 0,X
 bsr OutChar
 inx
 bra OutString
OSdone rts
```
SCI Output Decimal Number

; Output unsigned byte to the SCI
; Inputs: Reg B= 0 to 255,
; print as 3 digit ascii
; Outputs: none

OutUDec clr A ;Reg D=number
ldx #100
idiv ;X=num/100,
xgdx ;B=100s digit
  tba
  adda #’0 ;A=100’s ascii
  bsr OutCh
xgdx ;D=num
ldx #10

SCI Output Decimal Number (cont)

idiv ;X=num/10,
xgdx ;B=tens digit
  tba
  adda #’0 ;A=tens ascii
  bsr OutCh
  xgdx ;D=num
  tba
  adda #’0 ;A=ones ascii
  bsr OutCh
rts

Debugging Tools

(See Figure 2.15 through 2.17)

Debugging Theory

- The debugging process is defined as testing, stabilizing, localizing, and correcting errors.
- Research in program monitoring and debugging has not kept pace with developments in other areas of software.
- In embedded systems, debugging is further complicated by concurrency and real-time requirements.
- Although monitoring and debugging tools exist, many still use manual methods such as print statements.
- Print statements are highly intrusive especially in a real-time system because they can take too much time.
Debugging Instruments

- A *debugging instrument* is code that is added to a program for the purpose of debugging.
- A print statement is a common example.
- When adding print statements, use one of the following:
  1. Place all print statements in a unique column.
  2. Define instruments with specific pattern in their name.
  3. Define all instruments to test a run-time global flag.
  4. Use conditional compilation (assembly) to turn on/off.

Functional (Static) Debugging

- *Functional debugging* is verification of I/O parameters.
- Inputs are supplied, systems is run, outputs are checked.
- There exist many functional debugging methods:
  1. Single stepping or tracing.
  2. Breakpoints without filtering.
  3. Conditional breakpoints.
  4. Instrumentation: print statements.
  5. Instrumentation: dump into array without filtering.
  6. Instrumentation: dump into array with filtering.
  7. Monitor using fast displays.

Performance (Dynamic) Debugging

- *Performance debugging* is verification of timing behavior.
- System is run and dynamic behaviors of I/O checked.
  1. Count bus cycles using the assembly listing.
  2. Instrumentation: measuring with a counter.

```
before rmb 2 ; TCNT value before the call
elapsed rmb 2 ; # of cycles to execute sqrt
mov TCNT, before
movb ss,1,-sp ; push parameter on stack
jsr sqrt ; call sqrt module
ins
stab tt ; save result
ldd TCNT ; TCNT value after the call
subd before
std elapsed ; execute time in cycles
```
  3. Instrumentation: output port.

Profiling

- *Profiling* collects time history of strategic variables.
  1. Use a software dump to study execution pattern.
  2. Use an output port.
- When multiple threads are running can use these techniques to determine the thread activity.