Modular Software Development

- Modular programming breaks software problems in distinct and independent modules.
- Modular software development provides:
  - Functional abstraction to allow software reuse.
  - Complexity abstraction (i.e., divide and conquer).
  - Portability.
- A program module is a self-contained software task with clear entry and exit points.
- Can be a collection of subroutines or functions that in their entirety perform a well-defined set of tasks.

Stack Contents

```
movb ss,1,-sp ;push parameter onto stack
jsr sqrt   ;call sqrt subroutine
ins        ;save result
stb tt
```

Software Module

Diagram representing a software module with:
- Single entry point
- Local variables
- Module operations (calls to other modules)
- Global variables
- I/O ports
- Single exit point
- Decision structures
- Looping structures
Example Module in Assembly

```assembly
sqrt pshy
  tsy
  leas -4,sp ;allocate t,oldt,s16
  clrb
  ldab s8,y
  ...
  stab t,y
  dec cnt,y
  bne next
  done tys
  puly
  rts
```

Returning Multiple Parameters in Assembly 1

```assembly
module: ldab #1
  ldab #2
  lda #3
  ldab #4
  rts ;returns 4 parameters in 4 registers

*******calling sequence******
  jsr module
  leas -2,sp ;allocate space for results
  jsr module
  pula ;1st parameter from stack
  staa first
  pula ;2nd parameter from stack
  staa second
```

Returning Multiple Parameters in Assembly 2

```assembly
data1 equ 2
data2 equ 3

module movb #1,data1,sp ;1st parameter onto stack
module movb #2,data2,sp ;2nd parameter onto stack
  rts

*******calling sequence******
  leas -2,sp ;allocate space for results
  jsr module
  pula ;1st parameter from stack
  staa first
  pula ;2nd parameter from stack
  staa second
```

More Issues in Modular Software

- All exit points in an assembly routine must balance the stack and return parameters in the same way.
- Performing unnecessary I/O in a subroutine makes it harder to reuse at a later time.
- I/O devices must be considered global, and the number of modules that can access them should be restricted.
- Information hiding means to separate mechanism from policies (i.e., hiding the inner workings from the user).
Dividing a Software Task into Modules

- **Coupling** is influence one module’s behavior has another, and is typically caused by shared variables.
- When dividing into modules have these goals:
  - Make the software project easier to understand.
  - Increase the number of modules.
  - Decrease the interdependency (minimize coupling).
- Develop and connect modules in a hierarchical manner.
  - Top-down - “Write no software until every detail is specified.”
  - Bottom-up - “one brick at a time.”

Rules for Modular Software in Assembly

- The single entry point is at the top.
- The single exit point is at the bottom.
- Write structured programs.
- The registers must be saved.
- Use high-level languages when possible.
- Minimize conditional branching.

Simple Calling Graph

Layered Software Systems

- Software undergoes many changes as better hardware or algorithms become available.
- Layered software facilitates these changes.
- The top layer is the main program.
- The lowest layer, the **hardware abstraction layer**, includes all modules that access the I/O hardware.
- Each layer can only call modules in its layer or lower.
- A **gate** (also known as an application program interface (API)) is used to call from a higher-to a lower layer.
- The main advantage is that one layer can be replaced without affecting the other layers.
Layered Approach for Parallel Port

Three-Layer Software System (High Level)

```
org $E000
main:  lds  #$00FF
       bsr  Initialize ; simple call to a function
loop:   bsr  PrintInfo ; simple call to a function
        bra  loop
Initialize: ldaa #1 ; function code for InitPrinter
swi      ; call to middle level
rts
org $FFFF ; high level vector (reset)
fdb  main
```

Three-Layer Software System (Middle Level)

```
org $F400
swihandler: cmpa #1 ; function code for InitPrinter
            bne  notIP
            bsr  InitPrinter
            bra  swidone
notIP:    cmpa #2 ; function code for PrintInfo
            bne  notPI
            bsr  PrintInfo
            bra  swidone
notPI:    ; **** error
swidone:  rti
InitPrinter: ldaa #1 ; function code for InitIEEE
             ldx  $FF80 ; vector into low level
             jsr  0,x  ; call lower level
             rts
             org  $FFF6 ; middle level vector (SWI)
fdb  swihandler
```

Three-Layer Software System (Low Level)

```
org $F800
lowhandler: cmpa #1 ; function code for InitIEEE
            bne  notInit
            bsr  InitIEEE
            bra  lowdone
notInit:   cmpa #2 ; function code for SetDAV
            bne  notDAV
            bsr  SetDAV
            bra  lowdone
notDAV:    ; rest of the functions
lowdone:   rts
InitIEEE:  ; access to hardware
           rts
           org  $FF80 ; Lower level vector
           fdb  lowhandler
```
Layered Software Rules
1. A module may make simple call to modules in same layer.
2. A module may call a lower-level module only using gate.
3. A module may not directly access any function or variable in another layer (w/o going through a gate).
4. A module may not call a higher-level routine.
5. A module may not modify the vector address of another level’s handler(s).
6. (Optional) A module may not call farther than one level.
7. (Optional) All I/O hardware access is in lowest level.
8. (Optional) All user interface I/O is in highest level unless it is the purpose of the module to do such I/O.

Interrupts and Threads

Threads

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);
        OutChar(number%10);
    } else
        OutChar(number+'0'); }
```
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

- 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.

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)
Slide 25

**SCI Initialization**

- **BAUD**
- **SCCR1**
- **SCCR2**

```assembly
init ldaa #$33 ; 1200 baud
staa BAUD
lda #$00 ; mode
staa SCCR1
lda #$0C ; tie=rie=0,
staa SCCR2 ; te=re=1
rts
```

Slide 26

**SCI Input**

- **SCSR**
- **SCDR**

```assembly
InChar ldaa SCSR ; status
bita #$20 ; rdrf?
beq InChar
lda SCSR ; SCI data
rts
```

Slide 27

**SCI Output**

- **SCSR**
- **SCDR**

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

Slide 28

**Input Decimal Number**

- **InUDec clrb ; N=0
  InUDloop ber InChar ; Next input
  ber OutChar ; 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
clrb
sec ; error flag
InUDret rts
```

SCI Output Decimal Number

```
; Output unsigned byte to the SCI
; Inputs: Reg B= 0 to 255,
; Outputs: none
OutUDec clra ; Reg D=number
ldx #100
idiv ; X=num/100,
xgdex ; B=100’s digit
tba
adda #’0 ; A=100’s ascii
bsr OutChar
xgdex ; D=num
ldx #10
```

SCI Output String

```
; Output a string to the SCI
; Inputs: Reg X points to string
; Outputs: none
OutString ldaa 0,X
beq OSdone ; 0 at end
bsr OutChar
inx
bra OutString
OSdone rts
```

SCI Output Decimal Number (cont)

```
idiv ; X=num/10,
xgdex ; B=tens digit
tba
adda #’0 ; A=tens ascii
bsr OutChar
xgdex ; D=num
tba
adda #’0 ; A=ones ascii
bsr OutChar
rts
```
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, system 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
   movw 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.