ECE/CS 5780/6780: Embedded System Design

Chris J. Myers

Lecture 4: Software Design

Introduction

- Success of an embedded system project depends on both hardware and software.
- Real-time embedded systems are usually not very large, but are often quite complex.
- Needed software skills include: modular design, layered architecture, abstraction, and verification.
- Writing good software is an art that must be developed and cannot be added on at the end of a project.
- Good software with average hardware will always outperform average software with good hardware.

Golden Rule of Software Development

Write software for others as you wish they would write for you.

- Quantitative performance measurements:
  - *Dynamic efficiency* - number of CPU cycles required.
  - *Static efficiency* - number of memory bytes required.
  - Are given design constraints satisfied?
- Qualitative performance measurements:
  - Easy to debug (fix mistakes)
  - Easy to verify (prove correctness)
  - Easy to maintain (add features)
- Sacrificing clarity in favor of execution speed often results in software that runs fast but doesn't work and can't be changed.
- You are a good programmer if (1) you can understand your own code 12 months later and (2) others can change your code.

Software Maintenance

- Maintenance is the *most important* phase of development.
- Includes fixing bugs, adding features, optimization, porting to new hardware, configuring for new situations.
- Documentation should assist software maintenance.
- Most important documentation is in the code itself.

Good Comments

- Comments that simply restate the operation do not add to the overall understanding.

```
BAD   X=X+4;   /* add 4 to X */
      Flag=0;  /* set Flag=0 */
GOOD  X=X+4;   /* 4 is added to correct for the
               offset (mV) in the transducer */
      Flag=0;  /* means no key has been typed */
```

- When variable defined, should explain how used.

```
int SetPoint;  /* Desired temperature, 16-bit signed
               value with resolution of 0.5C,
               a range of -55C to +125C,
               a value of 25 means 12.5C */
```

- When constant defined, should explain what it means.

```
V=999; /* 999mV is the maximum possible voltage */
```

Assembly Language Style Issues

- Begins and ends with a line of *s
- States the purpose of the function
- Gives the I/O parameters, what they mean, and how they are passed
- Different phases of code delineated by a line of -'s

Client and Colleague Comments

- When a subroutine is defined, two types of comments needed:
  - *Client comments* explain how the function is to be used, how to pass parameters, and what errors and results are possible. (in header or start of subroutine)
  - *Colleague comments* explain how the function works (within the body of the function).

More on Client Comments

- Purpose of the module
- Input parameters
  - How passed (call by value, call by reference)
  - Appropriate range
  - Format (8 bit/16 bit, signed/unsigned, etc.)
- Output parameters
  - How passed (return by value, return by reference)
  - Format (8 bit/16 bit, signed/unsigned, etc.)
- Example inputs and outputs if appropriate
- Error conditions
- Example calling sequence
- Local variables and their significance

Self-Documenting Code

- Software written in a simple and obvious way such that its purpose and function are self-apparent.
- Use descriptive names for var, const, and functions.
- Formulate and organize into well-defined subproblems.
- Liberal use of `#define` and `equ` statements.

Use of `#define`

```
// An inappropriate use of #define.
#define size 10
short data[size];
void initialize(void){ short j
   for(j=0;j<10;j++)
      data[j]=0;
};
// An appropriate use of #define.
#define size 10
short data[size];
void initialize(void){ short j
   for(j=0;j<size;j++)
      data[j]=0;
};
```

Naming Convention

- Names should have meaning.
- Avoid ambiguities.
- Give hints about the type.
- Use the same name to refer to the same type of object.
- Use a prefix to identify public objects.
- Use upper and lower case to specify the scope of an object.
- Use capitalization to delimit words.

Naming Convention Examples

Type	Example
constants	PORTA
local variables	maxTemperature
private global variables	MaxTemperature
public global variables	DAC_MaxVoltage
private function	ClearTime
public function	Timer_ClearTime

Abstraction

- *Software abstraction* is when we define a complex problem with a set of basic abstract principles.
- Advantages of abstraction:
  - Faster to develop because some building blocks exist,
  - Easier to debug (prove correct) because it separates conceptual issues from implementation, and
  - Easier to change.
- *Finite state machine* (FSM) is a good abstraction.
- Consists of inputs, outputs, states, and state transitions.
- FSM software implementation is easy to understand, debug, and modify.

6812 Timer Details

- `TCNT` is a 16-bit unsigned counter that increments at a rate determined by `PR2`, `PR1`, and `PR0` in the `TSCR2` register.

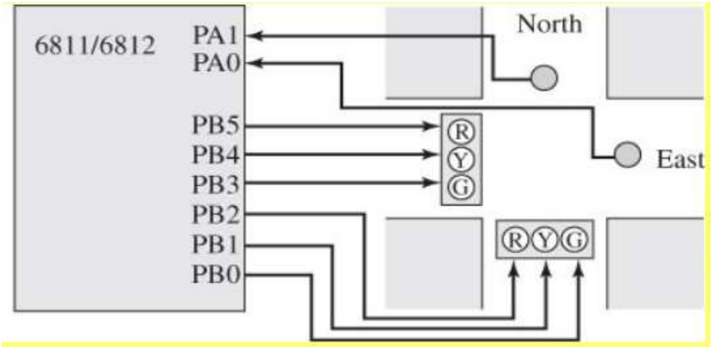
PR2	PR1	PR0	Divide by	TCNT Period	TCNT Frequency
0	0	0	1	250ns	4 MHz
0	0	1	2	500ns	2 MHz
0	1	0	4	1μs	1 MHz
0	1	1	8	2μs	500 kHz
1	0	0	16	4μs	250 kHz
1	0	1	32	8μs	125 kHz
1	1	0	64	16μs	62.5 kHz
1	1	1	128	32μs	31.25 kHz

- When `TCNT` overflows, `TOF` flag in the `TFLG2` register is set.
- Overflow causes an interrupt if the `TOI` bit in `TSCR2` is set.

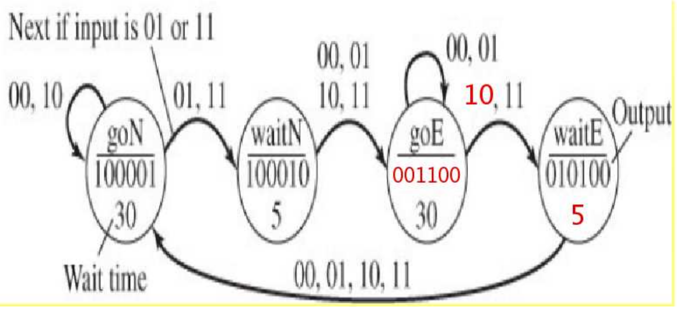
Time Delay

```
void Timer_Init(void){
  TSCR1 = 0x80; // enable TCNT
  TSCR2 = 0x04; // 1us TCNT
}
void Timer_Wait(unsigned short cycles){
unsigned short startTime = TCNT;
  while((TCNT-startTime) <= cycles){}
}
// 10000us equals 10ms
void Timer_Wait10ms(unsigned short delay){
unsigned short i;
  for(i=0; i<delay; i++){
    Timer_Wait(10000);  // wait 10ms
  }
}
```

Traffic Light Interface



Traffic Light Interface



C Implementation of a Moore FSM

```
const struct State {
  unsigned char Out;
  unsigned short Time;
  const struct State *Next[4];};
typedef const struct State STyp;
#define goN   &FSM[0]
#define waitN &FSM[1]
#define goE   &FSM[2]
#define waitE &FSM[3]
STyp FSM[4]={
 {0x21,3000,{goN,waitN,goN,waitN}},
 {0x22, 500,{goE,goE,goE,goE}},
 {0x0C,3000,{goE,goE,waitE,waitE}},
 {0x14, 500,{goN,goN,goN,goN}}};
```

C Implementation of a Moore FSM (cont)

```
void main(void){
STyp *Pt;  // state pointer
unsigned char Input;
  Timer_Init();
  DDRB = 0xFF;
  DDRA &= ~0x03;
  Pt = goN;
  while(1){
    PORTB = Pt->Out;
    Timer_Wait10ms(Pt->Time);
    Input = PORTA&0x03;
    Pt = Pt->Next[Input];
  }
}
```

Assembly Implementation of a Moore FSM

```
      org $4000 ; Put in ROM
OUT   equ 0    ;offset for output
WAIT  equ 1    ;offset for time
NEXT  equ 3    ;offset for next state
goN   fcb $21  ;North green, East red
      fdb 3000 ;30sec
      fdb goN,waitN,goN,waitN
waitN fcb $22  ;North yellow, East red
      fdb 500  ;5sec
      fdb goE,goE,goE,goE
goE   fcb $0C  ;North red, East green
      fdb 3000 ;30 sec
      fdb goE,goE,waitE,waitE
waitE fcb $14  ;North red, East yellow
      fdb 500  ;5sec
      fdb goN,goN,goN,goN
```

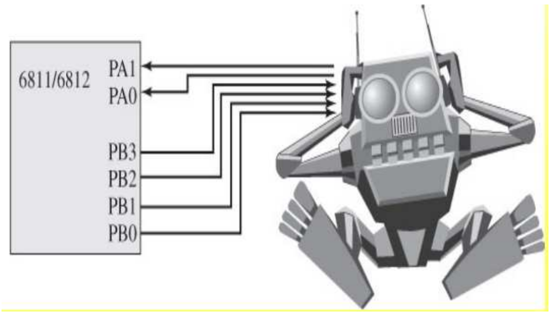
Assembly Implementation of a Moore FSM (cont)

```
Main lds  #$4000    ;stack init
     bsr  Timer_Init ;enable TCNT
     movb #$FF,DDRB ;PB5-0 are lights
     movb #$00,DDRA ;PA1-0 are sensors
     ldx  #goN      ;State pointer
```

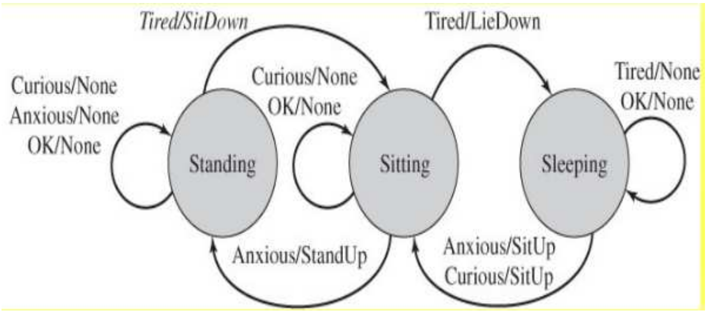
Assembly Implementation of a Moore FSM (cont)

```
FSM  ldab OUT,x
     stab PORTB     ;Output
     ldy  WAIT,x    ;Time delay
     bsr  Timer_Wait10ms
     ldab PORTA     ;Read input
     andb #$03      ;just bits 1,0
     lslb           ;2 bytes/address
     abx            ;add 0,2,4,6
     ldx  NEXT,x    ;Next state
     bra  FSM
     org  $FFFE
     fdb  Main       ;reset vector
```

Robot Interface



Mealy FSM for a Robot Controller



C Implementation of a Mealy FSM

```
// outputs defined as functions
const struct State{
  void (*CmdPt)[4](void);      // outputs
  const struct State *Next[4]; // Next
};
typedef const struct State StateType;
#define Standing &fsm[0]
#define Sitting  &fsm[1]
#define Sleeping &fsm[2]
void None(void){};
void SitDown(void){
  PORTB=0x08; PORTB=0;} // pulse on PB3
void StandUp(void){
  PORTB=0x04; PORTB=0;} // pulse on PB2
void LieDown(void){
  PORTB=0x02; PORTB=0;} // pulse on PB1
void SitUp(void) {
  PORTB=0x01; PORTB=0;} // pulse on PB0
```

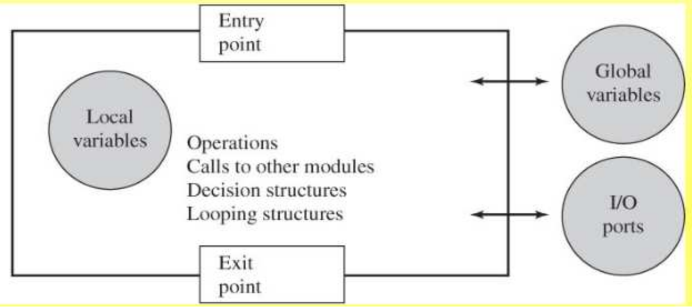
C Implementation of a Mealy FSM

```
StateType FSM[3]={
{{&None,&SitDown,&None,&None}, //Standing
  {Standing,Sitting,Standing,Standing}},
{{&None,&LieDown,&None,&StandUp},//Sitting
  {Sitting,Sleeping,Sitting,Standing }},
{{&None,&None,&SitUp,&SitUp},  //Sleeping
  {Sleeping,Sleeping,Sitting,Sitting}}};
void main(void){
StatePtr *Pt;  // Current State
unsigned char Input;
  DDRB = 0xFF;     // Output to robot
  DDRA &= ~0x03;   // Input from sensor
  Pt = Standing;   // Initial State
  while(1){
    Input = PORTA&0x03;    // Input=0-3
    (*Pt->CmdPt[Input])(); // function
    Pt = Pt->Next[Input];  // next state
  }}
```

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.

Software Modules



Global Variables

- *Global variable* is information shared by more than one module.
- Use globals to pass data between *main thread* and *interrupt thread*.
- There information is permanent and not deallocated.
- Can use absolute addressing to access their information.
- I/O ports and control registers are considered global variables.

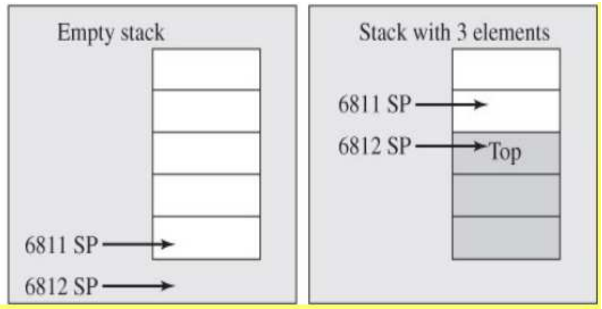
Local Variables

- *Local variable* is temporary information used by only one module.
- Typically allocated, used, and deallocated.
- Information is not permanent.
- Stored on stack or in registers because:
  - Dynamic allocation/release allows for memory reuse.
  - Limited scope provides data protection.
  - Since interrupt saves registers and uses own stack, code is *reentrant*.
  - Code is relocatable.
  - Number of variables only limited by stack size.

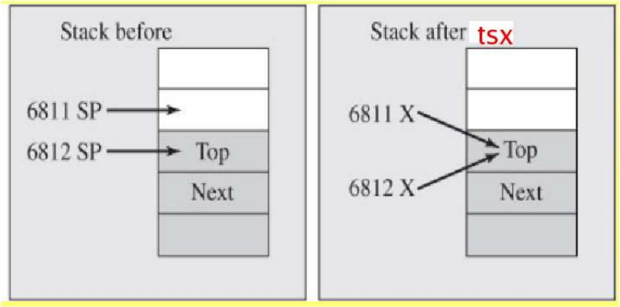
Two Local 16-bit Variables: Approach One

```
;unsigned short calc(void){ unsigned short sum,n;
;  sum = 0;
;  for(n=100;n>0;n--){
;    sum=sum+n;
;  }
;  return sum;
;}
; *****binding phase***********
sum  set  0  16-bit number
n    set  2  16-bit number
; *******allocation phase *****
calc pshx   ;save old Reg X
     pshx   ;allocate n
     pshx   ;allocate sum
     tsx    ;stack frame pointer
```

Stack



Stack After `tsx` Instruction



Two Local 16-bit Variables: Approach One (cont)

```
; ********access phase ********
     ldd  #0
     std  sum,x  ;sum=0
     ldd  #100
     std  n,x    ;n=100
loop ldd  n,x    ;RegD=n
     addd sum,x  ;RegD=sum+n
     std  sum,x  ;sum=sum+n
     ldd  n,x    ;n=n-1
     subd #1
     std  n,x
     bne  loop
; ******deallocation phase ***
     pulx       ;DIFFERENT THAN BOOK
     pulx       ;DIFFERENT THAN BOOK
     pulx       ;restore old X
     rts        ;RegD=sum ; 6812 only
```

Two Local 16-bit Variables: Approach Two

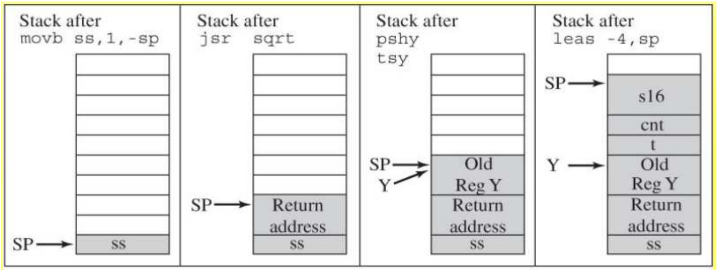
```
; *****binding phase************
sum  set  -4  16-bit number
n    set  -2  16-bit number
; *******allocation phase ******
calc pshx       ;save old Reg X
     tsx        ;stack frame pointer
     leas -4,sp ;allocate n,sum
```

Two Local 16-bit Variables: Approach Two (cont)

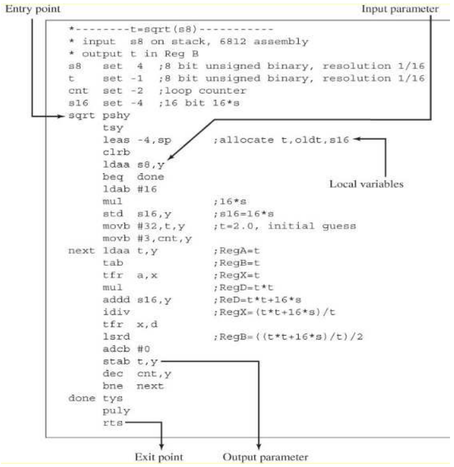
```
; ********access phase *********
     movw #0,sum,x  ;sum=0
     movw #100,n,x  ;n=100
loop ldd  n,x    ;RegD=I
     addd sum,x  ;RegD=sum+n
     std  sum,x  ;sum=sum+n
     ldd  n,x    ;n=n-1
     subd #1
     std  n,x
     bne  loop
; *****deallocation phase *****
     txs        ;deallocation
     pulx       ;restore old X
     rts        ;RegD=sum
```

Stack Contents

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

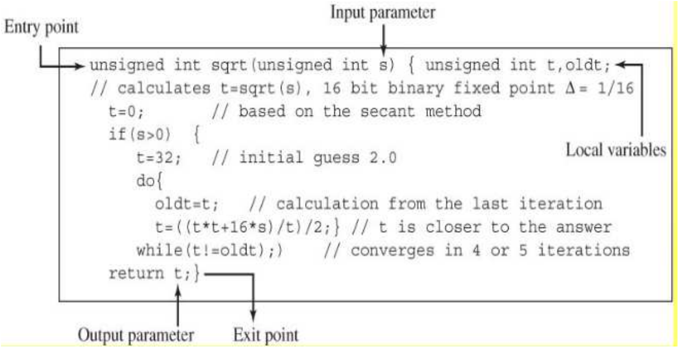


Example Module in Assembly



```
*--------t=sqrt(s8)-----------
* input  s8 on stack, 6812 assembly
* output t in Reg B
s8   set  4  ;8 bit unsigned binary, resolution 1/16
t    set -1  ;8 bit unsigned binary, resolution 1/16
cnt  set -2  ;loop counter
s16  set -4  ;16 bit 16*s
sqrt pshy
     tsy
     leas -4,sp        ;allocate t,oldt,s16
     clrb
     ldaa s8,y
     beq  done
     ldab #16
     mul               ;16*s
     std  s16,y        ;s16=16*s
     movb #32,t,y      ;t=2.0, initial guess
     movb #3,cnt,y
next ldaa t,y          ;RegA=t
     tab               ;RegB=t
     tfr  a,x          ;RegX=t
     mul               ;RegD=t*t
     addd s16,y        ;ReD=t*t+16*s
     idiv              ;RegX=(t*t+16*s)/t
     tfr  x,d
     lsrd              ;RegB=((t*t+16*s)/t)/2
     adcb #0
     stab t,y
     dec  cnt,y
     bne  next
done tys
     puly
     rts
```

Example Module in C



```
unsigned int sqrt(unsigned int s) { unsigned int t,oldt;
// calculates t=sqrt(s), 16 bit binary fixed point Δ= 1/16
  t=0;      // based on the secant method
  if(s>0) {
     t=32;  // initial guess 2.0
     do{
       oldt=t;  // calculation from the last iteration
       t=((t*t+16*s)/t)/2;} // t is closer to the answer
     while(t!=oldt);)    // converges in 4 or 5 iterations
  return t;}
```

Returning Multiple Parameters in Assembly 1

```
module: ldaa #1
        ldab #2
        ldx  #3
        ldy  #4
        rts     ;returns 4 parameters in 4 registers
********calling sequence******
        jsr module
* Reg A,B,X,Y have four results
```

Returning Multiple Parameters in Assembly 2

```
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
```

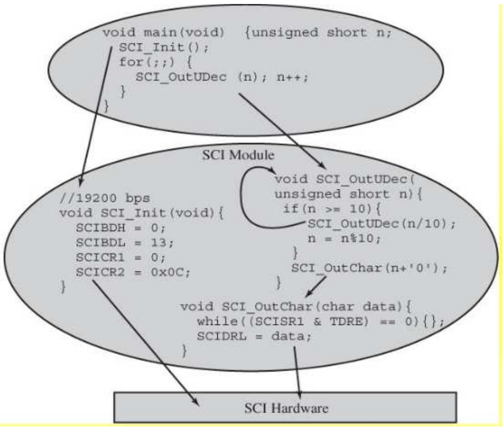
More Issues in Modular Software

- 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 on 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."

Simple Calling Graph



```
void main(void)  {unsigned short n;
  SCI_Init();
  for(;;) {
    SCI_OutUDec (n); n++;
  }
}
```

```
//19200 bps
void SCI_Init(void){
  SCIBDH = 0;
  SCIBDL = 13;
  SCICR1 = 0;
  SCICR2 = 0x0C;
}
```

```
void SCI_OutUDec(
unsigned short n){
 if(n >= 10){
   SCI_OutUDec(n/10);
   n = n%10;
 }
 SCI_OutChar(n+'0');
}
```

```
void SCI_OutChar(char data){
  while((SCISR1 & TDRE) == 0){};
  SCIDRL = data;
}
```

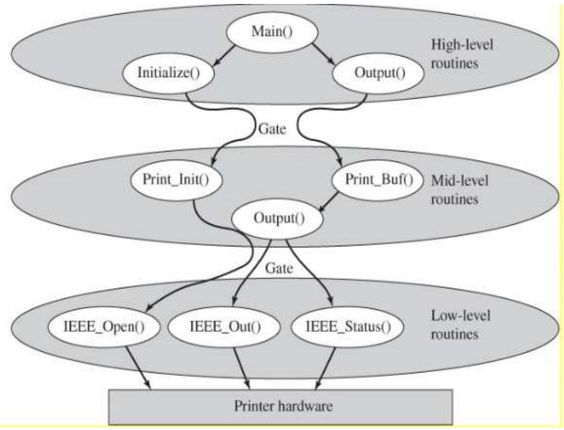
Rules for Modular Software in Assembly

- 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.

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 a Parallel Port



Layered Software Rules

- A module may make simple call to modules in same layer.
- A module may call a lower-level module only using gate.
- A module may not directly access any function or variable in another layer (w/o going through a gate).
- A module may not call a higher-level routine.
- A module may not modify the vector address of another level's handler(s).
- (Optional) A module may not call farther than one level.
- (Optional) All I/O hardware access is in lowest level.
- (Optional) All user interface I/O is in highest level unless it is the purpose of the module to do such I/O.

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:
  - New hardware to be installed.
  - New algorithms to be implemented.
    - Synchronization with gadfly, interrupts, or DMA.
    - Error detection and recovery methods.
    - Enhancements like automatic data compression.
  - Higher-level features to be built on top of the low level
    - Operating system features like blocking semaphores.
    - Additional features like function keys.

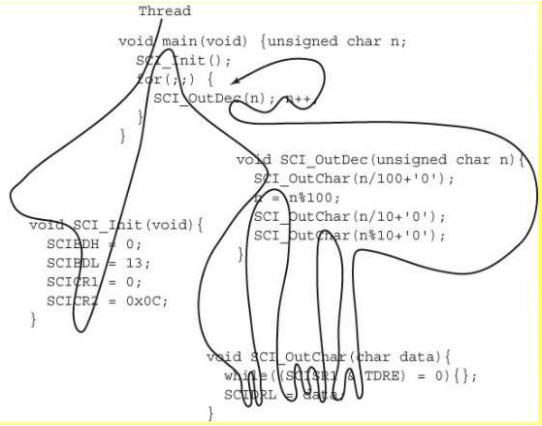
Device Driver Software

- Data structures: global (private)
  `bool OpenFlag //True if SCI has been initialized.`
- Initialization routines (public, called by client once)
  `void SCI_Init(unsigned short baudRate); //Initialize SCI`
- Regular I/O calls (public, called by client to perform I/O)
  `char SCI_InChar(void); //Wait for new SCI input character`
  `char SCI_OutChar(void); //Transmit character out SCI port`
- Support software (private)
  `void SCIHandler(void) //SCI interrupt handler`

Encapsulated Objects Using Standard C

- Choose function names to reflect the module in which they are defined.
- Example:
  `LCD_Clear()` (C)
  `LCD.clear()` (C++)
- Only put public function declarations in header files.
- Example (`Timer.H`):
  `void Timer_Init(void);`
  `void Timer_Wait10ms(unsigned short delay);`
  Since the function `wait(unsigned short cycles)` is not in the header file, it is a private function.

Threads



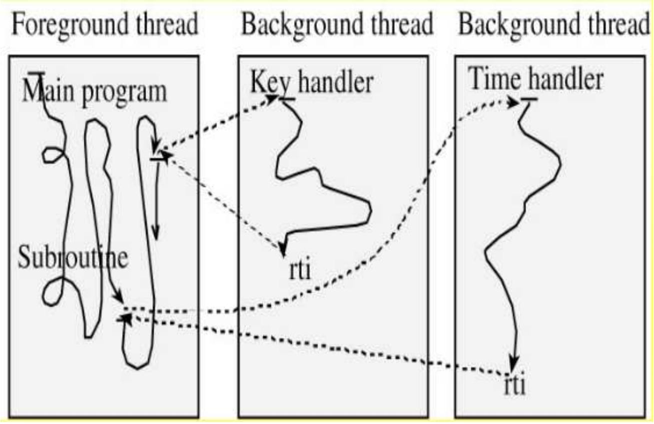
```
void main(void) {unsigned char n;
  SCI_Init();
  for(;;) {
    SCI_OutDec(n); n++;
  }
}

void SCI_Init(void){
  SCIBDH = 0;
  SCIBDL = 13;
  SCICR1 = 0;
  SCICR2 = 0x0C;
}

void SCI_OutDec(unsigned char n){
  SCI_OutChar(n/100+'0');
  n = n%100;
  SCI_OutChar(n/10+'0');
  SCI_OutChar(n%10+'0');
}

void SCI_OutChar(char data){
  while((SCISR1 & TDRE) = 0){};
  SCIDRL = data;
}
```

Interrupts and Threads

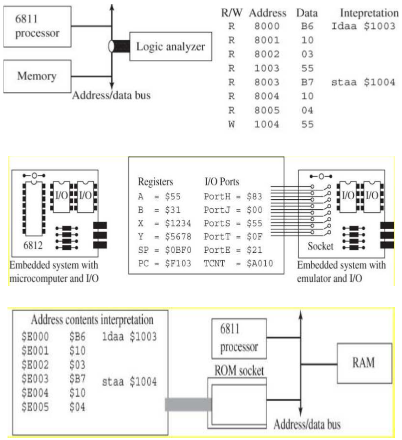


Recursion

- 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.

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

Debugging Tools



R/W	Address	Data	Intepretation
R	8000	B6	ldaa $1003
R	8001	10	
R	8002	03	
R	1003	55	
R	8003	B7	staa $1004
R	8004	10	
R	8005	04	
W	1004	55	

Registers	I/O Ports
A = $55	PortH = $83
B = $31	PortJ = $00
X = $1234	PortS = $55
Y = $5678	PortT = $0F
SP = $0BF0	PortE = $21
PC = $F103	TCNT = $A010

Address contents interpretation

Address	contents	interpretation
$E000	$B6	ldaa $1003
$E001	$10	
$E002	$03	
$E003	$B7	staa $1004
$E004	$10	
$E005	$04	

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:
  - Place all print statements in a unique column.
  - Define instruments with specific pattern in their name.
  - Define all instruments to test a run-time global flag.
  - 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:
  - Single stepping or tracing.
  - Breakpoints without filtering.
  - Conditional breakpoints.
  - Instrumentation: print statements.
  - Instrumentation: dump into array without filtering.
  - Instrumentation: dump into array with filtering.
  - Monitor using fast displays.

Instrumentation Dump Without Filtering

```
// global variables in RAM
#define size 20
unsigned char buffer[size][2];
unsigned int cnt=0;
// dump happy and sad
void Save(void){
  if(cnt<size){
    buffer[cnt][0] = happy;
    buffer[cnt][1] = sad;
    cnt++;
  }
}
```

Instrumentation Dump With Filter

```
// dump happy and sad
void Save(void){
  if(sad>100){
    if(cnt<size){
      buffer[cnt][0] = happy;
      buffer[cnt][1] = sad;
      cnt++;
    }
  }
}
```

An LED Monitor

```
void Toggle(void){
  PORTB ^= 0x40; // flip LED
}
```

Performance (Dynamic) Debugging

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

```
before  rmb  2      ;TCNT value before the call
elasped 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     elasped     ;execute time in cycles
```

  - Instrumentation: output port.

Instrumentation Output Port

```
Set bset PORTB,#$40
    rts
Clr bclr PORTB,#$40
    rts

loop jsr  Set
     jsr  Calculate   ; function under test
     jsr  Clr
     bra  loop
```

Performance (Dynamic) Debugging

```
; Assembly listing from TExaS of the sqrt subroutine.
$F019                               org  *  ;reset cycle counter
$F019 35              [ 2](  0)sqrt pshy
$F01A B776            [ 1](  2)     tsy
$F01C 1B9C            [ 2](  3)     leas -4,sp      ;allocate t,oldt,s16
$F01E C7              [ 1](  5)     clrb
$F01F A644            [ 3](  6)     ldaa s8,y
$F021 2723            [ 3](  9)     beq  done
$F023 C610            [ 1]( 12)     ldab #16
$F025 12              [ 3]( 13)     mul             ;16*s
$F026 6C5C            [ 2]( 16)     std  s16,y      ;s16=16*s
$F028 18085F20        [ 4]( 18)     movb #32,t,y    ;t=2.0, initial guess
$F02C 18085E03        [ 4]( 22)     movb #3,cnt,y
$F030 A65F            [ 3]( 26)next ldaa t,y        ;RegA=t
$F032 180E            [ 2]( 29)     tab             ;RegB=t
$F034 B705            [ 1]( 31)     tfr  a,x        ;RegX=t
$F036 12              [ 3]( 32)     mul             ;RegD=t*t
$F037 E35C            [ 3]( 35)     addd s16,y      ;RegD=t*t+16*s
$F039 1810            [12]( 38)     idiv            ;RegX=(t*t+16*s)/t
$F03B B754            [ 1]( 50)     tfr  x,d
$F03D 49              [ 1]( 51)     lsrd            ;RegB=((t*t+16*s)/t)/2
$F03E C900            [ 1]( 52)     adcb #0
$F040 6B5F            [ 2]( 53)     stab t,y
$F042 635E            [ 3]( 55)     dec  cnt,y
$F044 26EA            [ 3]( 58)     bne  next
$F046 B767            [ 1]( 61)done tys
$F048 31              [ 3]( 62)     puly
$F049 3D              [ 5]( 65)     rts
$F04A 183E            [16]( 70)     stop
```

Empirical Measurement of Dynamic Efficiency

```
unsigned short before,elasped;
void main(void){
   ss=100;
   before=TCNT;
   tt=sqrt(ss);
   elasped=TCNT-before;
}
```

Another Empirical Measurement of Dynamic Efficiency

```
void main(void){
   DDRB=0xFF;  // PB7 is connected to a scope
   ss=100;
   while(1){
     PORTB |= 0x80;  // set PB7 high
     tt=sqrt(ss);
     PORTB &= ~0x80; // clear PB7 low
   }
}
```

Profiling

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

A Time/Position Profile Dumping into a Data Array

```
unsigned short time[100];
unsigned short place[100];
unsigned short n;
void profile(unsigned short p){
  time[n]=TCNT; // record current time
  place[n]=p;
  n++; }
unsigned short sqrt(unsigned short s){ unsigned short t,oldt;
profile(0);
  t=0;       // based on the secant method
  if(s>0) {
profile(1);
     t=32;   // initial guess 2.0
     do{
profile(2);
       oldt=t;  // calculation from the last iteration
       t=((t*t+16*s)/t)/2;} // t is closer to the answer
     while(t!=oldt);}    // converges in 4 or 5 iterations
profile(3);
  return t;}
```

A Time/Position Profile Using Two Output Bits

```
unsigned int sqrt(unsigned int s){ unsigned int t,oldt;
PORTB=0;
  t=0;       // based on the secant method
  if(s>0) {
PORTB=1;
     t=32;   // initial guess 2.0
     do{
PORTB=2;
       oldt=t;  // calculation from the last iteration
       t=((t*t+16*s)/t)/2;} // t is closer to the answer
     while(t!=oldt);}    // converges in 4 or 5 iterations
PORTB=3;
  return t;}
```

The Power of 10: Rules for Developing Safety-Critical Code

Gerald Holzmann / NASA/ JPL Laboratory for Reliable Software

1. Do not use goto, setjmp, longjmp, direct or indirect recursion.
2. Give all loops a fixed upper bound.
3. Do not use dynamic memory allocation after initialization.
4. No function should be larger than can fit on a sheet of paper.
5. There should be at least two assertions per function.
6. Declare all data objects at smallest possible level of scope.
7. Each calling function must check return value of nonvoid functions, and each called function should check validity of all parameters.
8. Use of preprocessor should be restricted to inclusion of header files and simple macro definitions.
9. No more than one level of pointer dereferencing and shouldn't be hidden.
10. All code must compile with no warnings.