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 & power 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.

Administrivia

- How is Lab 1?
- Don’t forget Lab 2 has a pre-lab assignment.
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.

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).

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 */

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 `equiv` statements.

Use of `#define`

```c
// 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

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>constants</td>
<td>PORTA</td>
</tr>
<tr>
<td>local variables</td>
<td>maxTemperature</td>
</tr>
<tr>
<td>private global variables</td>
<td>MaxTemperature</td>
</tr>
<tr>
<td>public global variables</td>
<td>DAC_MaxVoltage</td>
</tr>
<tr>
<td>private function</td>
<td>ClearTime</td>
</tr>
<tr>
<td>public function</td>
<td>Timer_ClearTime</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>PR2</th>
<th>PR1</th>
<th>PR0</th>
<th>Divide by</th>
<th>TCNT Period</th>
<th>TCNT Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>250ns</td>
<td>4 MHz</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>500ns</td>
<td>2 MHz</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>1µs</td>
<td>1 MHz</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>2µs</td>
<td>500 kHz</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>4µs</td>
<td>250 kHz</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>32</td>
<td>8µs</td>
<td>125 kHz</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>64</td>
<td>16µs</td>
<td>62.5 kHz</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>128</td>
<td>32µs</td>
<td>31.25 kHz</td>
</tr>
</tbody>
</table>

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

Traffic Light Interface
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 for the Traffic Light Controller

org $800
OUT equ 0 ;offset for output
WAIT equ 1 ;offset for time (8 bits+OUT)
NEXT equ 3 ;offset for next state (16 bits+WAIT)
goN fcb $21 ;East red, north green
  fdb 3000 ;30 second delay
  fdb goN,waitN,goN,waitN
waitN fcb $22 ;East red, north yellow
  fdb 500 ;5 second delay
  fdb goE,goE,goE,goE
goE fcb $0C ;East green, north red
  fdb 3000 ;30 second delay
  fdb goE,goE,waitE,waitE
waitE fcb $14 ;East yellow, north red
  fdb 500 ;5 second delay
  fdb goN,goN,goN,goN
Assembly for the Traffic Light Controller

Main 1ds #$4000 ;stack init
bsr Timer_Init ;enable TCNT
movb #$FF,DDRb ;PORTB5-0 set to output to lights
movb #$00,DDRa ;PORTA1-0 set to input from sensors
ldx #goN ;Initialize state pointer (register X)

FSM 1dab OUT,x
stb PORTb
ldy WAIT,x
bsr Timer_Wait10ms
1dab PORTa
andb #$03 ;Keep the bottom two bits
lslb ;Multiply by two b/c addresses are 2 bytes
abx ;add 0,2,4,6
ldx NEXT,x
bra FSM

Code Execution

ldx #goN

FSM 1dab OUT,x
stb PORTb
ldy WAIT,x
bsr Timer_Wait10ms
1dab PORTa
andb #$03
lslb
abx
ldx NEXT,x
bra FSM

Memory Map

<table>
<thead>
<tr>
<th>State</th>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>goN</td>
<td>0800</td>
<td>21</td>
<td>out</td>
</tr>
<tr>
<td>OUT</td>
<td>0800</td>
<td>0</td>
<td>wait</td>
</tr>
<tr>
<td>WAIT</td>
<td>0801</td>
<td>0B B8</td>
<td>wait</td>
</tr>
<tr>
<td>NEXT</td>
<td>0802</td>
<td>0</td>
<td>ns0</td>
</tr>
<tr>
<td>goN</td>
<td>0803</td>
<td>0</td>
<td>ns1</td>
</tr>
<tr>
<td>fdb</td>
<td>0804</td>
<td>3000</td>
<td></td>
</tr>
<tr>
<td>waitN</td>
<td>0805</td>
<td>0</td>
<td>ns2</td>
</tr>
<tr>
<td>fdb</td>
<td>0806</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>goE</td>
<td>0807</td>
<td></td>
<td></td>
</tr>
<tr>
<td>waitN</td>
<td>0808</td>
<td>0</td>
<td>ns3</td>
</tr>
<tr>
<td>fdb</td>
<td>0809</td>
<td></td>
<td></td>
</tr>
<tr>
<td>goE</td>
<td>0810</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>fdb</td>
<td>0811</td>
<td></td>
<td></td>
</tr>
<tr>
<td>goE</td>
<td>0812</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>fdb</td>
<td>0813</td>
<td></td>
<td></td>
</tr>
<tr>
<td>goE</td>
<td>0814</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>fdb</td>
<td>0815</td>
<td></td>
<td></td>
</tr>
<tr>
<td>goE</td>
<td>0816</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Code Execution

ldx #goN

FSM 1dab OUT,x
stb PORTb
ldy WAIT,x
bsr Timer_Wait10ms
1dab PORTa
andb #$03
lslb
abx
ldx NEXT,x
bra FSM

State | Address | Value | Comment |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>goN</td>
<td>0800</td>
<td>21</td>
<td>out</td>
</tr>
<tr>
<td>OUT</td>
<td>0801</td>
<td>0B B8</td>
<td>wait</td>
</tr>
<tr>
<td>WAIT</td>
<td>0802</td>
<td>0</td>
<td>ns0</td>
</tr>
<tr>
<td>NEXT</td>
<td>0803</td>
<td>0</td>
<td>ns1</td>
</tr>
<tr>
<td>goN</td>
<td>0804</td>
<td>3000</td>
<td></td>
</tr>
<tr>
<td>fdb</td>
<td>0805</td>
<td>0</td>
<td>ns2</td>
</tr>
<tr>
<td>fdb</td>
<td>0806</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>goE</td>
<td>0807</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>fdb</td>
<td>0808</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>goE</td>
<td>0809</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>fdb</td>
<td>0810</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>goE</td>
<td>0811</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>fdb</td>
<td>0812</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>goE</td>
<td>0813</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>fdb</td>
<td>0814</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>goE</td>
<td>0815</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>fdb</td>
<td>0816</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
### Code Execution

<table>
<thead>
<tr>
<th>State</th>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>goN</td>
<td>0800</td>
<td>21</td>
<td>out</td>
</tr>
<tr>
<td></td>
<td>0801</td>
<td>0B B8</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td>0803</td>
<td>08 00</td>
<td>ns0</td>
</tr>
<tr>
<td></td>
<td>0805</td>
<td>08 0B</td>
<td>ns1</td>
</tr>
<tr>
<td></td>
<td>0807</td>
<td>08 00</td>
<td>ns2</td>
</tr>
<tr>
<td></td>
<td>0809</td>
<td>08 0B</td>
<td>ns3</td>
</tr>
<tr>
<td></td>
<td>080B</td>
<td>22</td>
<td>out</td>
</tr>
<tr>
<td></td>
<td>080C</td>
<td>01 F4</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td>080E</td>
<td>08 16</td>
<td>ns0</td>
</tr>
<tr>
<td></td>
<td>0810</td>
<td>08 16</td>
<td>ns1</td>
</tr>
<tr>
<td></td>
<td>0812</td>
<td>08 16</td>
<td>ns2</td>
</tr>
<tr>
<td></td>
<td>0814</td>
<td>08 16</td>
<td>ns3</td>
</tr>
<tr>
<td></td>
<td>0816</td>
<td>0C</td>
<td>out</td>
</tr>
</tbody>
</table>

### Code Execution

<table>
<thead>
<tr>
<th>State</th>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>goN</td>
<td>0800</td>
<td>21</td>
<td>out</td>
</tr>
<tr>
<td></td>
<td>0801</td>
<td>0B B8</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td>0803</td>
<td>08 00</td>
<td>ns0</td>
</tr>
<tr>
<td></td>
<td>0805</td>
<td>08 0B</td>
<td>ns1</td>
</tr>
<tr>
<td></td>
<td>0807</td>
<td>08 00</td>
<td>ns2</td>
</tr>
<tr>
<td></td>
<td>0809</td>
<td>08 0B</td>
<td>ns3</td>
</tr>
<tr>
<td></td>
<td>080B</td>
<td>22</td>
<td>out</td>
</tr>
<tr>
<td></td>
<td>080C</td>
<td>01 F4</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td>080E</td>
<td>08 16</td>
<td>ns0</td>
</tr>
<tr>
<td></td>
<td>0810</td>
<td>08 16</td>
<td>ns1</td>
</tr>
<tr>
<td></td>
<td>0812</td>
<td>08 16</td>
<td>ns2</td>
</tr>
<tr>
<td></td>
<td>0814</td>
<td>08 16</td>
<td>ns3</td>
</tr>
<tr>
<td></td>
<td>0816</td>
<td>0C</td>
<td>out</td>
</tr>
</tbody>
</table>

### Code Execution

<table>
<thead>
<tr>
<th>State</th>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>goN</td>
<td>0800</td>
<td>21</td>
<td>out</td>
</tr>
<tr>
<td></td>
<td>0801</td>
<td>0B B8</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td>0803</td>
<td>08 00</td>
<td>ns0</td>
</tr>
<tr>
<td></td>
<td>0805</td>
<td>08 0B</td>
<td>ns1</td>
</tr>
<tr>
<td></td>
<td>0807</td>
<td>08 00</td>
<td>ns2</td>
</tr>
<tr>
<td></td>
<td>0809</td>
<td>08 0B</td>
<td>ns3</td>
</tr>
<tr>
<td></td>
<td>080B</td>
<td>22</td>
<td>out</td>
</tr>
<tr>
<td></td>
<td>080C</td>
<td>01 F4</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td>0812</td>
<td>08 16</td>
<td>ns2</td>
</tr>
<tr>
<td></td>
<td>0814</td>
<td>08 16</td>
<td>ns3</td>
</tr>
<tr>
<td></td>
<td>0816</td>
<td>0C</td>
<td>out</td>
</tr>
</tbody>
</table>

### Code Execution

<table>
<thead>
<tr>
<th>State</th>
<th>Address</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>goN</td>
<td>0800</td>
<td>21</td>
<td>out</td>
</tr>
<tr>
<td></td>
<td>0801</td>
<td>0B B8</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td>0803</td>
<td>08 00</td>
<td>ns0</td>
</tr>
<tr>
<td></td>
<td>0805</td>
<td>08 0B</td>
<td>ns1</td>
</tr>
<tr>
<td></td>
<td>0807</td>
<td>08 00</td>
<td>ns2</td>
</tr>
<tr>
<td></td>
<td>0809</td>
<td>08 0B</td>
<td>ns3</td>
</tr>
<tr>
<td></td>
<td>080B</td>
<td>22</td>
<td>out</td>
</tr>
<tr>
<td></td>
<td>080C</td>
<td>01 F4</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td>0812</td>
<td>08 16</td>
<td>ns2</td>
</tr>
<tr>
<td></td>
<td>0814</td>
<td>08 16</td>
<td>ns3</td>
</tr>
<tr>
<td></td>
<td>0816</td>
<td>0C</td>
<td>out</td>
</tr>
</tbody>
</table>

### Robot Interface

![Robot Interface Diagram]

---

Scott R. Little (Lecture 4: Software Design) ECE/CS 5780/6780 29 / 96

Scott R. Little (Lecture 4: Software Design) ECE/CS 5780/6780 30 / 96

Scott R. Little (Lecture 4: Software Design) ECE/CS 5780/6780 32 / 96
Mealy FSM for a Robot Controller

C Implementation of a Mealy FSM

```c
// 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
```

Mealy FSM Example

- Similar to Moore FSM except that the output depends on both input and current state.
- This results in the two “tables” in the assembly code.
- Both the output value and next state value must be looked up for a given input.

```c
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.

Global Variables

- Global variable is information shared by more than one module.
- Use global variables to pass data between main thread and interrupt thread.
- Their 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 may still be reentrant.
  - Code is relocatable.
  - Number of variables only limited by stack size.
Two Local 16-bit Variables: Approach One

```c
#define sum set 0 16-bit number
#define n set 2 16-bit number

unsigned short calc(void)
{
    unsigned short sum, n;
    sum = 0;
    for (n = 100; n > 0; n--)
    {
        sum = sum + n;
    }
    return sum;
}
```

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

```c
;******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 ***
    ldd sum, x ; RegD = sum
    pulx ; deallocate sum
    pulx ; deallocate n
    pulx ; restore old X
    rts
```

Two Local 16-bit Variables: Approach Two

```c
#define sum set -4 16-bit number
#define n set -2 16-bit number

unsigned short calc(void)
{
    unsigned short sum, n;
    sum = 0;
    for (n = 100; n > 0; n--)
    {
        sum = sum + n;
    }
    return sum;
}
```

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

```c
;******access phase ******
    movw #0, sum, x ; sum = 0
    movw #100, 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 ***
    ldd sum, x ; RegD = sum
    txs ; deallocation
    pulx ; restore old X
    rts
```
### Local variable allocation/deallocation

```assembly
sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x
movw #100,n,x
loop
  ldd n,x
  addd sum,x
  std sum,x
  ldd n,x
  subd #1
  std n,x
  bne loop
  ldd sum,x
txs
pulx
```

### Local variable allocation/deallocation

```assembly
sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x
movw #100,n,x
loop
  ldd n,x
  addd sum,x
  std sum,x
  ldd n,x
  subd #1
  std n,x
  bne loop
  ldd sum,x
txs
pulx
```

### Local variable allocation/deallocation

```assembly
sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x
movw #100,n,x
loop
  ldd n,x
  addd sum,x
  std sum,x
  ldd n,x
  subd #1
  std n,x
  bne loop
  ldd sum,x
txs
pulx
```

### Local variable allocation/deallocation

```assembly
sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x
movw #100,n,x
loop
  ldd n,x
  addd sum,x
  std sum,x
  ldd n,x
  subd #1
  std n,x
  bne loop
  ldd sum,x
txs
pulx
```
Local variable allocation/deallocation

```assembly
sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x ;0804-4
movw #100,n,x
loop ldd n,x
add sum,x
std sum,x
ldd n,x
subd #1
std n,x
bne loop
ldd sum,x
txs
pulx
```

Local variable allocation/deallocation

```assembly
sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x
movw #100,n,x ;0804-2
loop ldd n,x
add sum,x
std sum,x
ldd n,x
subd #1
std n,x
bne loop
ldd sum,x
txs
pulx
```

Local variable allocation/deallocation

```assembly
sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x
movw #100,n,x
loop ldd n,x
add sum,x
std sum,x
ldd n,x
subd #1
std n,x
bne loop
ldd sum,x
txs
pulx
```

Local variable allocation/deallocation

```assembly
sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x
movw #100,n,x
loop ldd n,x
add sum,x
std sum,x
ldd n,x
subd #1
std n,x
bne loop
ldd sum,x
txs
pulx
```
Local variable allocation/deallocation

sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x
movw #100,n,x

loop ldd n,x
add sum,x
std sum,x ;0804-4

0800 0064
0802 0064
0804 FFFF
0806 XXXX

sp
0800
RegX 0804
AccD 0064

local variable allocation/deallocation

sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x
movw #100,n,x

loop ldd n,x
add sum,x
std sum,x

0800 0064
0802 0064
0804 FFFF
0806 XXXX

sp
0800
RegX 0804
AccD 0064

local variable allocation/deallocation

sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x
movw #100,n,x

loop ldd n,x
add sum,x
std sum,x

0800 0064
0802 0064
0804 FFFF
0806 XXXX

sp
0800
RegX 0804
AccD 0064

local variable allocation/deallocation

sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x
movw #100,n,x

loop ldd n,x
add sum,x
std sum,x

0800 0064
0802 0064
0804 FFFF
0806 XXXX

sp
0800
RegX 0804
AccD 0064

local variable allocation/deallocation

sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x
movw #100,n,x

loop ldd n,x
add sum,x
std sum,x

0800 0064
0802 0064
0804 FFFF
0806 XXXX

sp
0800
RegX 0804
AccD 0064

local variable allocation/deallocation

sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x
movw #100,n,x

loop ldd n,x
add sum,x
std sum,x

0800 0064
0802 0064
0804 FFFF
0806 XXXX

sp
0800
RegX 0804
AccD 0064

local variable allocation/deallocation

sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x
movw #100,n,x

loop ldd n,x
add sum,x
std sum,x

0800 0064
0802 0064
0804 FFFF
0806 XXXX

sp
0800
RegX 0804
AccD 0064
Local variable allocation/deallocation

sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x
movw #100,n,x
loop ldd n,x
add sum,x
std sum,x
ldd n,x
subd #1
std n,x
bne loop
ldd sum,x
txs
pulx

Local variable allocation/deallocation

sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x
movw #100,n,x
loop ldd n,x
add sum,x
std sum,x
ldd n,x
subd #1
std n,x
bne loop
ldd sum,x
;0804-4
txs
pulx

Local variable allocation/deallocation

sum set -4
n set -2
calc pshx
tsx
leas -4,sp
movw #0,sum,x
movw #100,n,x
loop ldd n,x
add sum,x
std sum,x
ldd n,x
subd #1
std n,x
bne loop
ldd sum,x
txs
pulx
Returning Multiple Parameters in Assembly 1

module: ldax #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.”
**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**

![Diagram of a 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.

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.
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'); }
```

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 Tools

- Debugging instruments 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

```c
// 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

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

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.
    - unsigned short before,elapsed;
    - void main(void){
      ss=100;
      before=TCNT;
      tt=sqrt(ss);
      elapsed=TCNT-before;
    }
```
Instrumentation Output Port

Set bset PORTB,#$40

Clr bclr PORTB,#$40

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

Performance (Dynamic) Debugging

; Assembly listing from TENAs of the sqrt subroutine.
\#019    org * ; reset cycle counter
\#019 35  [2] {0} reset push
\#01A 77B6 [1] 2) tay
\#01C 1B0C [2] 3) leas -4,sp ; allocate t, oldt, st16
\#01E C7 [1] 5) clrb
\#01F 4644 [3] 6) leaa a8,y
\#021 2723 [3] 9) beq done
\#023 C610 [1] 12) lab #16
\#025 12 [3] 13) mul ; i6 +
\#026 65C2 [2] 16) std a16,y ; s16 = i6 a
\#028 16099FF20 [4] 18) movb #32, t,y ; '0.0', initial guess
\#02C 18098E30 [4] 22) movb #3,cnt,y
\#030 465F [3] 26) next leaa t,y ; Reg4 a
\#032 180E [2] 29) tab ; Reg4 b
\#034 B76E [1] 31) tfr a.x ; Reg4 c
\#036 12 [3] 32) mul ; Reg4 d
\#037 E3EC [3] 35) addd a16,y ; Reg0 = t+t+i6 a
\#039 1810 [12] 38) idiv ; Reg0 = (t+t+i6 a)/t
\#03B 8754 [1] 50) tfr x.d
\#03D 49 [1] 51) lard ; Reg0 = (t+t+i6 a)/t/2
\#03E C900 [1] 52) addc #0
\#040 63B2 [2] 53) tab t,y
\#042 635E [3] 55) dec cnt,y
\#044 26EA [3] 58) bne next
\#046 8767 [1] 61) done tys
\#049 30 [3] 65) rts
\#04A 183E [16] 70) stop

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

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.
Correct code: Who do you believe?

- pg. 128 of your textbook - "Recursive algorithms are often easy to prove correct."
- Gerard J. Holzman "The Power of Ten" - Eliminating recursion can help prove boundedness of code.

Introduction

- Coding guidelines that cannot be checked by a tool are less effective.
- Too many coding guidelines aren’t effective because they are not remembered or enforceable.
- The cost of restrictive guidelines may pay off with code that is more correct.

Rule 1

Rule: Restrict all code to very simple control flow constructs – do not use goto statements, setjmp or longjmp constructs, and direct or indirect recursion.

- Simple control translates into easier code verification and often improved clarity.
- Without recursion the function call graph is acyclic which directly aids in proving boundedness of the code.
- This rule doesn’t require a single return point for a function although this often simplifies control flow.

Rule 2

Rule: All loops must have a fixed upper-bound. It must be trivially possible for a checking tool to prove statically that a preset upper-bound on the number of iterations of a loop cannot be exceeded. If the loop-bound cannot be proven statically, the rule is considered violated.

- The absence of recursion and presence of loop bounds prevents runaway code.
- Functions intended to be nonterminating must be proved to not terminate.
- Some functions don’t have an obvious upper bound (i.e. traversing a linked list), so an artificial bound should be set and checked via an assert.
Rule 3

Rule: Do not use dynamic memory allocation after initialization.
- Memory allocation code is unpredictable from a time standpoint and therefore impractical for time critical code.
- Many errors are introduced by improper dynamic memory allocation.
- Without dynamic memory allocation the stack is used for dynamic structures and without recursion bounds can be proved on stack size.

Rule 4

Rule: No function should be longer than what can be printed on a single sheet of paper in a standard reference format with one line per statement and one line per declaration. Typically, this means no more than 60 lines of code per function.
- Long functions often indicate poor code structure.

Rule 5

Rule: The assertion density should average to a minimum of two assertions per function. Assertions are used to check for anomalous conditions that should never happen in real-life executions. Assertions must always be side-effect free and should be defined as Boolean tests. When an assertion fails, an explicit recovery action must be taken.
- Use of assertions is recommended as part of a strong defensive coding strategy.
- Assertions can be used to check pre- and post-conditions of functions, parameter values, return values, and loop invariants.
- Assertions can be disabled in performance critical code because they are side-effect free.

Rule 6

Rule: Data objects must be declared at the smallest possible level of scope.
- Variable will not be modified in unexpected places if they are not in scope.
- It can be easier to debug a problem if the scope of the variable is smaller.
Rule 7

Rule: The return value of non-void functions must be checked by each calling function, and the validity of parameters must be checked in each function.

- If the response to the error would be no different to the response to the success then there is no point in checking the value.
- Useless checks can be indicated by casting the return value to (void).

Rule 8

Rule: The use of the preprocessor must be limited to the inclusion of header files and simple macro definitions. Token pasting, variable argument lists, and recursive macro calls are not allowed. All macros must expand into complete syntactic units. The use of conditional compilation directives is often also dubious but cannot always be avoided. Each use of a conditional compilation directive should be flagged by a tool-based checker and justified in the code.

- Conditional compilation directives can result in an exponentially growing number of code versions.

Rule 9

Rule: The use of pointers should be restricted. Specifically, no more than one level of dereferencing is allowed. Pointer dereference operations may not be hidden in macro definitions or inside typedef declarations. Function pointers are not permitted.

- Pointers are easily misused even by experienced programmers.
- Function pointers can severely limit the utility of static code checkers.

Rule 10

Rule: All code must be compiled, from the first day of development, with all compiler warnings enabled at the compiler’s most pedantic setting. All code must compile with these settings without any warnings. All code must be checked daily with at least one, but preferably more than one, state-of-the-art static code analyzer and should pass the analyses with zero warnings.

- This rule should be followed even in the case when the warning is invalid.
- Code that confuses the compiler or checker enough to result in an invalid warning should be rewritten for clarity.
- Static checkers should be required for any serious coding project.