Assembler is the program that converts assembly language into object code that can be executed by the processor.

Addressing modes are key to reading and writing assembly language.

Condition codes and subsequent branches are also very important.

For efficiency and convenience, HCS12 provides extensive math operations wider than 8 bits.

Otherwise, math on the HCS12 is totally standard.

Assembly language permits very basic, very serious mistakes such as failing to save the right registers on a function call, creating mismatched stack frames, improper use of condition code registers, etc.
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.
What does good software look like?

Quantitative performance measurements:

*Dynamic efficiency* - number of CPU cycles & power required.
*Static efficiency* - number of RAM/ROM 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 an extremely important phase of development. Includes fixing bugs, adding features, optimization, porting to new hardware, configuring for new situations. Documentation should assist software maintenance. Documentation resides in and outside of the software.
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 */
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 \texttt{define} and \texttt{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;
};
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.
<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>

There are many conventions like this one. Often, companies will use some common set of conventions, or develop their own.
A software abstraction factors common functionality out of diverse examples.

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.
void Timer_Init(void){
    TSCR1 = 0x80; // enable TCNT
    TSCR2 = 0x02; // 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
Moore FSM & State Table

Input vector \(\langle N, E\rangle\)
Output vector \(\langle RE, YE, GE, RN, YN, GN\rangle\)

<table>
<thead>
<tr>
<th></th>
<th>No cars</th>
<th>Car E</th>
<th>Car N</th>
<th>Car N,E</th>
</tr>
</thead>
<tbody>
<tr>
<td>goN</td>
<td>goN</td>
<td>waitN</td>
<td>goN</td>
<td>waitN</td>
</tr>
<tr>
<td>waitN</td>
<td>goE</td>
<td>goE</td>
<td>goE</td>
<td>goE</td>
</tr>
<tr>
<td>goE</td>
<td>goE</td>
<td>goE</td>
<td>waitE</td>
<td>waitE</td>
</tr>
<tr>
<td>waitE</td>
<td>goN</td>
<td>goN</td>
<td>goN</td>
<td>goN</td>
</tr>
</tbody>
</table>
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}}
};
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  lds  #$4000 ;stack init
       bsr  Timer_Init ;enable TCNT
       movb  #$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  ldam  OUT,x
       stab  PORTB
       lday  WAIT,x
       bsr  Timer_Wait10ms
       ldam  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
## 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></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>waitN</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>goE</td>
<td>0816</td>
<td>0C</td>
<td>out</td>
</tr>
</tbody>
</table>
```plaintext
ldx  #goN

FSM
ldab OUT,x
stab PORTB
ldy WAIT,x
bsr Timer_Wait10ms
ldab PORTA
andb #$03
lslb
abx
ldx  NEXT,x
bra  FSM

<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>waitN</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>goE</td>
<td>0816</td>
<td>0C</td>
<td>out</td>
</tr>
</tbody>
</table>
```

```
| RegX   | XX XX |
| RegY   | XX XX |
| AccB   | XX    |
```

```
ldx #goN

FSM  ldab OUT,x

stab PORTB

ldy WAIT,x

bsr Timer_Wait10ms

ldab PORTA

andb #$03

lslb

abx

ldx NEXT,x

bra FSM

<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>waitN</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>goE</td>
<td>0816</td>
<td>0C</td>
<td>out</td>
</tr>
</tbody>
</table>

| RegX   | 08 00   |
| RegY   | XX XX   |
| AccB   | XX      |
Code Execution

```
ldx #goN

FSM
ldab OUT,x ;0800+0
stab PORTB
ldy WAIT,x
bsr Timer_Wait10ms
ldab PORTA
andb #$03
lslb
abx
ldx NEXT,x
bra FSM
```

<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>waitN</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>goE</td>
<td>0816</td>
<td>0C</td>
<td>out</td>
</tr>
</tbody>
</table>

| RegX | 08 00 |
| RegY | XX XX |
| AccB | 21    |
### Code Execution

**FSM**

```
ldx #goN
ldab OUT,x
stab PORTB
ldy WAIT,x ;0800+1
bsr Timer_Wait10ms
ldab PORTA
andb #$03
lslb
abx
ldx NEXT,x
bra FSM
```

<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>waitN</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>goE</td>
<td>0816</td>
<td>0C</td>
<td>out</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RegX</th>
<th>08 00</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegY</td>
<td>0B B8</td>
</tr>
<tr>
<td>AccB</td>
<td>21</td>
</tr>
</tbody>
</table>
Code Execution

```assembly
ldx #goN

FSM
ldab OUT,x
stab PORTB
ldy WAIT,x
bsr Timer.Wait10ms
ldab PORTA
andb #$03
lslb
abx
ldx NEXT,x
bra FSM

<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>waitN</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>goE</td>
<td>0816</td>
<td>0C</td>
<td>out</td>
</tr>
</tbody>
</table>

RegX  08 00
RegY  0B B8
AccB  81
```
ldx  #goN
sta PORTB
ldy  WAIT,x
bsr Timer_Wait10ms
ldab PORTA
andb  #$03
lslb
abx
ldx  NEXT,x
bra  FSM

<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>waitN</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>goE</td>
<td>0816</td>
<td>0C</td>
<td>out</td>
</tr>
</tbody>
</table>

RegX   08 00
RegY   0B B8
AccB   01
ldx #goN

FSM

ldab OUT,x
stab PORTB
ldy WAIT,x
bsr Timer_Wait10ms
ldab PORTA
andb #$03
lslb
abx
ldx NEXT,x
bra FSM

<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>waitN</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>goE</td>
<td>0816</td>
<td>0C</td>
<td>out</td>
</tr>
</tbody>
</table>

RegX  08 00
RegY  0B B8
AccB  02
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>

| RegX | 08 02  |
| RegY | 0B B8  |
| AccB | 02     |
Code Execution

```assembly
ldx  #goN
FSM  ldab  OUT,x
     stab  PORTB
ldy  WAIT,x
bsr  Timer_Wait10ms
ldab  PORTA
andb  #$03
lslb
abx
ldx  NEXT,x ;0802+3
bra  FSM

<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>waitN</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>goE</td>
<td>0816</td>
<td>0C</td>
<td>out</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>RegX</th>
<th>08 0B</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegY</td>
<td>0B B8</td>
</tr>
<tr>
<td>AccB</td>
<td>02</td>
</tr>
</tbody>
</table>
Robot Interface
Mealy FSM for a Robot Controller
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 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
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 programming breaks software problems into distinct and independent modules.

Modular software development provides:

- Functional abstraction to allow software reuse.
- Complexity abstraction (i.e., divide and conquer).
- Portability.
Software Modules

- Local variables
  - Operations
  - Calls to other modules
  - Decision structures
  - Looping structures

- Entry point

- Exit point

- Global variables

- I/O ports
Global variable is information shared by more than one module. Use global variables to pass data between \textit{main()} and \textit{interrupts}.

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 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.
;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
; *******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

; ******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
; ********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

```
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
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0800</td>
<td>XXXX</td>
</tr>
<tr>
<td>0802</td>
<td>XXXX</td>
</tr>
<tr>
<td>0804</td>
<td>XXXX</td>
</tr>
<tr>
<td>0806</td>
<td>XXXX</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>0806</td>
</tr>
<tr>
<td>RegX</td>
<td>FFFF</td>
</tr>
<tr>
<td>AccD</td>
<td>XXXX</td>
</tr>
</tbody>
</table>
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
  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

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

| 0800 | XXXX |
| 0802 | XXXX |
| 0804 | FFFF |
| 0806 | XXXX |
| SP   | 0804 |
| RegX | 0804 |
| AccD | XXXX |
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
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

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
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

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
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

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

<table>
<thead>
<tr>
<th>SP</th>
<th>0800</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegX</td>
<td>0804</td>
</tr>
<tr>
<td>AccD</td>
<td>0064</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0800</td>
<td>0064</td>
</tr>
<tr>
<td>0802</td>
<td>0064</td>
</tr>
<tr>
<td>0804</td>
<td>FFFF</td>
</tr>
<tr>
<td>0806</td>
<td>XXXX</td>
</tr>
</tbody>
</table>
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
   addd sum,x
   std sum,x
   ldd n,x ;0804-2
   subd #1
   std n,x
   bne loop
   ldd sum,x
   txs
   pulx

<table>
<thead>
<tr>
<th>SP</th>
<th>0800</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegX</td>
<td>0804</td>
</tr>
<tr>
<td>AccD</td>
<td>0064</td>
</tr>
</tbody>
</table>
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
      addd sum,x
      std sum,x
      ldd n,x
      subd #1
      std n,x
      bne loop
      ldd loop
      txs
      pulx

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0800</td>
<td>0064</td>
</tr>
<tr>
<td>0802</td>
<td>0064</td>
</tr>
<tr>
<td>0804</td>
<td>FFFF</td>
</tr>
<tr>
<td>0806</td>
<td>XXXX</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>0800</td>
</tr>
<tr>
<td>RegX</td>
<td>0804</td>
</tr>
<tr>
<td>AccD</td>
<td>0063</td>
</tr>
</tbody>
</table>
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
     addd sum,x
     std sum,x
     ldd n,x
     subd #1
     std n,x ;0804-2
     bne loop
     ldd sum,x
     txs
     pulx
```

```
<table>
<thead>
<tr>
<th>SP</th>
<th>0800</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegX</td>
<td>0804</td>
</tr>
<tr>
<td>AccD</td>
<td>0063</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0800 0064</td>
<td></td>
</tr>
<tr>
<td>0802 0063</td>
<td></td>
</tr>
<tr>
<td>0804 FFFF</td>
<td></td>
</tr>
<tr>
<td>0806 XXXX</td>
<td></td>
</tr>
</tbody>
</table>
```
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
addd sum,x
std sum,x
ldd n,x
subd #1
std n,x
bne loop
ldd sum,x
txs
pulx

<table>
<thead>
<tr>
<th>SP</th>
<th>0800</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegX</td>
<td>0804</td>
</tr>
<tr>
<td>AccD</td>
<td>0000</td>
</tr>
</tbody>
</table>

0800 13BA
0802 0000
0804 FFFF
0806 XXXX
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 ;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
  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

**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
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
data1    equ 2
data2    equ 3
module:  movb #1,data1,sp ;1st parameter onto stack
        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
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.”
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 showing a layered approach with functions such as Main(), Initialize(), Output(), Print_Init(), Print_Buf(), IEEE_Open(), IEEE_Out(), IEEE_Status(), and Printer hardware.
A module may make calls 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.
(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.
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. Good low-level device drivers allow:

- New hardware to be installed.
- New algorithms to be implemented.
- Synchronization with completion flags / interrupts
- Error detection and recovery methods.

Higher-level features to be built on top of the low level

- Operating system features like blocking semaphores.
- Driver features like automatic data compression
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
Choose function names to reflect the module in which they are defined.

Example:
In C: LCD_Clear()
In C++: LCD.clear()

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.
A program segment is *reentrant* if it can be concurrently executed by two (or more) threads, or by main() plus one (or more) interrupts.

Example of a function you would expect to be reentrant?

Example of a function you would expect to not be reentrant?

Rules for reentrant functions:

- Must not call a non-reentrant function
- Must not touch global variables w/o proper locking
Coding guidelines that cannot be checked by a smart compiler are less effective.

Too many coding guidelines aren’t effective because they are not remembered or enforceable.

Following list of guidelines is by Gerard Holzmann — leader of NASA/JPL’s Laboratory for Reliable Software.
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: 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 are like run-on sentences: they need to be rewritten.
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: 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: 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: 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.
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 with Hardware

<table>
<thead>
<tr>
<th>R/W</th>
<th>Address</th>
<th>Data</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>8000</td>
<td>B6</td>
<td>ldaa $1003</td>
</tr>
<tr>
<td>R</td>
<td>8001</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>8002</td>
<td>03</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>1003</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>8003</td>
<td>B7</td>
<td>staa $1004</td>
</tr>
<tr>
<td>R</td>
<td>8004</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>8005</td>
<td>04</td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>1004</td>
<td>55</td>
<td></td>
</tr>
</tbody>
</table>

Registers
- \( A = 55 \)  
- \( B = 31 \)  
- \( X = 1234 \)  
- \( Y = 5678 \)  
- \( SP = 0BF0 \)  
- \( PC = F103 \)  

I/O Ports
- PortH = 83  
- PortJ = 00  
- PortS = 55  
- PortT = 0F  
- PortR = 21  
- TCNT = A010

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

Embedded system with microcomputer and I/O

Embedded system with emulator and I/O

6811 processor

ROM socket

RAM
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 debugging means: checking if the code implements the correct computation.

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

```c
unsigned short before, elapsed;
void main(void){
    ss=100;
    before=TCNT;
    tt=sqrt(ss);
    elapsed=TCNT-before;
}
```
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 754 [ 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