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.
Memory Allocation Example

```assembly
org $0000   ;RAM
rmb $0000  ;global
org $B600   ;EEPROM
fcb $01     ;amount to add
org $E000    ;ROM
ld #B600    ;inputs
clrc        ;outputs
ld #B600    ;inputs
add #B600   ;outputs
br al       ; reset vector
```

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 *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 */`
Good Comments (cont)
- When a subroutine defined, two types of comments:
  - *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).

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 & organize into well-defined subproblems.
- Liberal use of `#define` and `equ` statements.
- Assembly language style issues:
  - Begins and ends with a line of `*`
  - 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 `-`

Software Documentation
- 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

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.
- An FSM software implementation is easy to understand, debug, and modify.
Mealy FSM in C

```c
const struct State
{ unsigned char Time; /* Time to wait in each state */
  unsigned char Out[2]; /* Output if input=0,1 */
  const struct State* Next[2]; /* Next state if inp=0,1*/
} StateType fsm[4]={
  {100, {0,0}, {SB,SD}},
  {100, {0,8}, {SC,SA}},
  {15, {0,0}, {SB,SD}},
  {15, {8,8}, {SC,SA}}
};
```

Mealy FSM in C (cont)

```c
typedef const struct State StateType;
#define SA &fsm[0]
#define SB &fsm[1]
#define SC &fsm[2]
#define SD &fsm[3]

void Wait(unsigned int delay) {
  int Endt;
  Endt=TCNT+delay; /* Time (125ns cycles) to wait */
  while((Endt-(int)TCNT)>0) { /* wait */
  }
}

void main(void) {
  StatePtr *Pt; /* Current State */
  unsigned char Input;
  Pt=SA; /* Initial State */
  DDRC=0x08; /* PortC bit3 is output */
  while(1) {
    Wait(Pt->Time); /* Time to wait in this state */
    Input=PORTC>>7; /* Input=0 or 1 */
    PORTC=Pt->Out[Input]; /* Perform output */
    Pt=Pt->Next[Input]; /* Move to the next state */
  }
}
```

Mealy FSM in Assembly

```assembly
org $B600
PutinEEPROMsoitcanbechanged

* Finite State Machine

Time equ 0 Index for time to wait in this state
Out0 equ 1 Index for output pattern if input=0
Out1 equ 2 Index for output pattern if input=1
Next0 equ 3 Index for next state if input=0
Next1 equ 5 Index for next state if input=1
IS fdb SA Initial state
SA fcb 100 Time to wait
fdb 0,0 Outputs for inputs 0,1
fdb SB Next state if Input=0
fdb SD Next state if Input=1
```
Mealy FSM in Assembly (cont)

SB  fcb 100   Time to wait  
    fcb 0,8  Outputs for inputs 0,1  
    fdb SC  Next state if Input=0  
    fdb SA  Next state if Input=1  

SC  fcb 15   Time to wait  
    fcb 0,0  Outputs for inputs 0,1  
    fdb SB  Next state if Input=0  
    fdb SD  Next state if Input=1  

SD  fcb 15   Time to wait  
    fcb 8,8  Outputs for inputs 0,1  
    fdb SC  Next state if Input=0  
    fdb SA  Next state if Input=1  

Slide 17

Mealy FSM in Assembly (cont)

is1  ldaa Out1,X  Get desired output from structure  
     staa $1003  Set PC3=output  
     ldx Next1,X  Input is 1  
     bra LL  

is0  ldaa Out0,X  Get desired output from structure  
     staa $1003  Set PC3=output  
     ldx Next0,X  Input is 0  
     bra LL  Infinite loop  
     org $FFFE  reset vector  

Slide 19

Moore FSM

Slide 20