Slide 1

ECE/CE 3720: Embedded System Design
(ECE 6960/2 and CS 6968)

Chris J. Myers

Lecture 3: Software Development

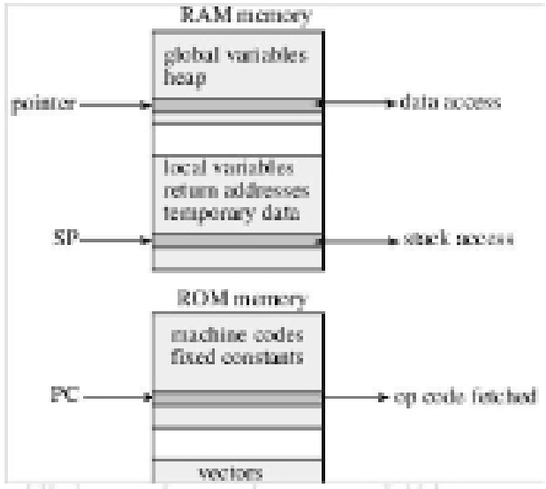
Slide 2

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.

Slide 3

Memory Allocation



Slide 4

Pseudo Instructions

`org`	`set location to ORiGin`
`fcc`	`Form Constant Character string`
`fcb`	`Form Constant Byte`
`fdb`	`Form Double Byte`
`equ`	`EQUate symbol to a value`
`rmb`	`Reserve Memory Bytes`

Slide 5

Memory Allocation Example

```
        org   $0000   ;RAM
cnt     rmb   1       ;global
        org   $B600   ;EEPROM
const   fcb   5       ;amount to add
        org   $E000   ;ROM
init    ldaa  #$FF
        staa  DDRC    ;outputs
        clr   cnt
        rts
main    lds   #$00FF  ;sp->RAM
        bsr   init
loop    ldaa  cnt
        staa  PORTC   ;output
        adda  const
        staa  cnt
        bra   loop
        org   $FFFE   ;ROM
        fdb   main    ;reset vector
```

Slide 6

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.

Slide 7

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.

Slide 8

Good Comments

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

```
BAD   I=I+4;   /* add 4 to I */
      Flag=0;  /* set Flag=0 */
GOOD  I=I+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 */
```

Slide 9

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).

Slide 10

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 *'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

Slide 11

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

Slide 12

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.