Lecture #5

# Intro. to Engineering Process and Design Techniques

**18-348 Embedded System Engineering**

**Philip Koopman**

**Wednesday, 27-Jan-2016**

Electrical & Computer ENGINEERING

Carnegie Mellon

## Real embedded projects aren’t just about the CPU!

- http://www.youtube.com/watch_popup?v=PKaNyvOgMA0&vq=medium#t=52
- **Digital Hardware**
- **Software**

… but also …

- Mechanical
- Electrical
- Fluids
- Environmental control
- Food Safety
- Security
- Lots of engineering considerations

- *What happens if a software defect causes temperature problems?*

[Moobella06]

# Where Are We Now?

- **Where we've been:**
  - Hardware & assembly language
- **Where we're going today:**
  - Is there more to embedded systems than just slapping together hardware and hacking out the code?
- **Where we're going next:**
  - Embedded C and language use
  - Embedded programming techniques
  - Memory bus
  - Economics / general optimization
  - Serial ports
  - Debug & Test
  - Exam #1

# Preview

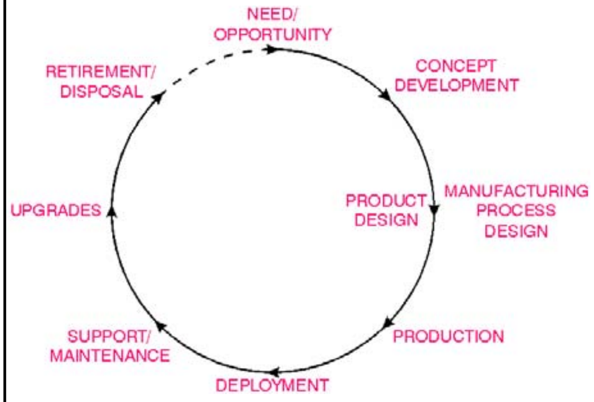
- **Engineering projects have phases**
  - Marketing, product definition, requirements, architecture, design, implementation, test, V&V, support, evolution
- **Requirements**
  - Shall vs. should
  - Keep an eye on these in projects
- **Design**
  - Flowcharts
  - Statecharts
  - Sequence Diagrams
- **Implementation**
  - Basic coding style
  - Good & Bad practices

# Embedded System Engineers Need Perspective

- **How does what I do fit into the bigger picture?**
  - What outside constraints do I have to meet (e.g., limited battery life)
  - How can I exploit the non-computer aspects of the system?
- **How can I contribute toward the product, not just my piece?**
  - Ask how your company makes money from the product
  - Advocate a decent engineering process (what this lecture is about)
- **Important embedded skills perspective:**
  - Knowing how to solder doesn't make you a hardware engineer
  - Knowing how to write lines of code doesn't make you a software engineer
  - Knowing how to do both isn't enough to be an embedded systems engineer

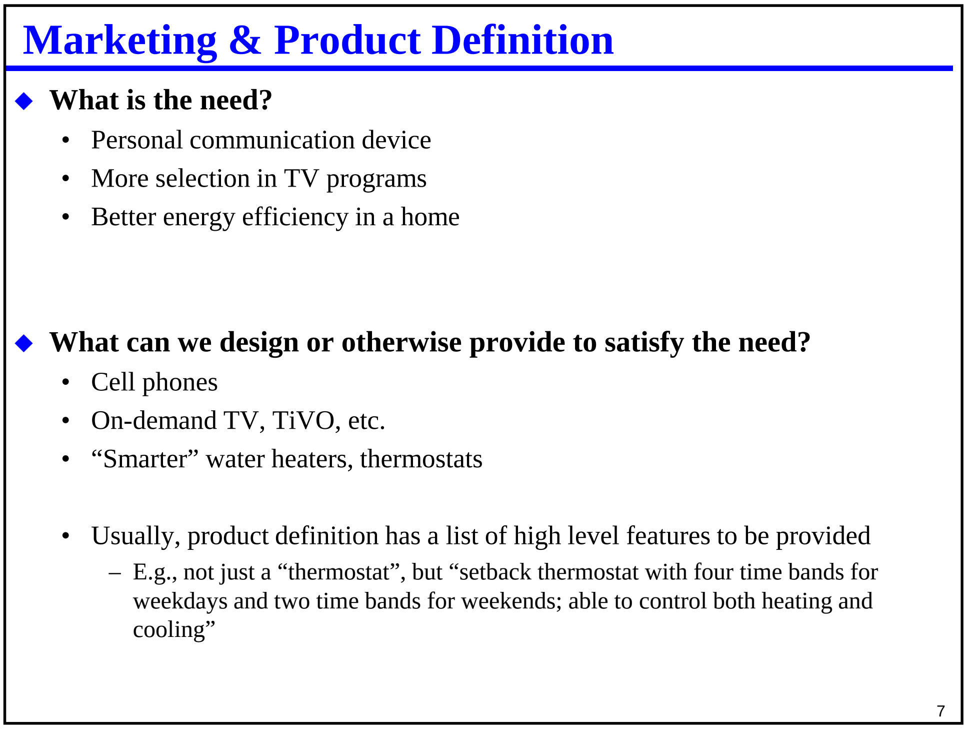
# Typical project life cycle:

**General System Life Cycle**



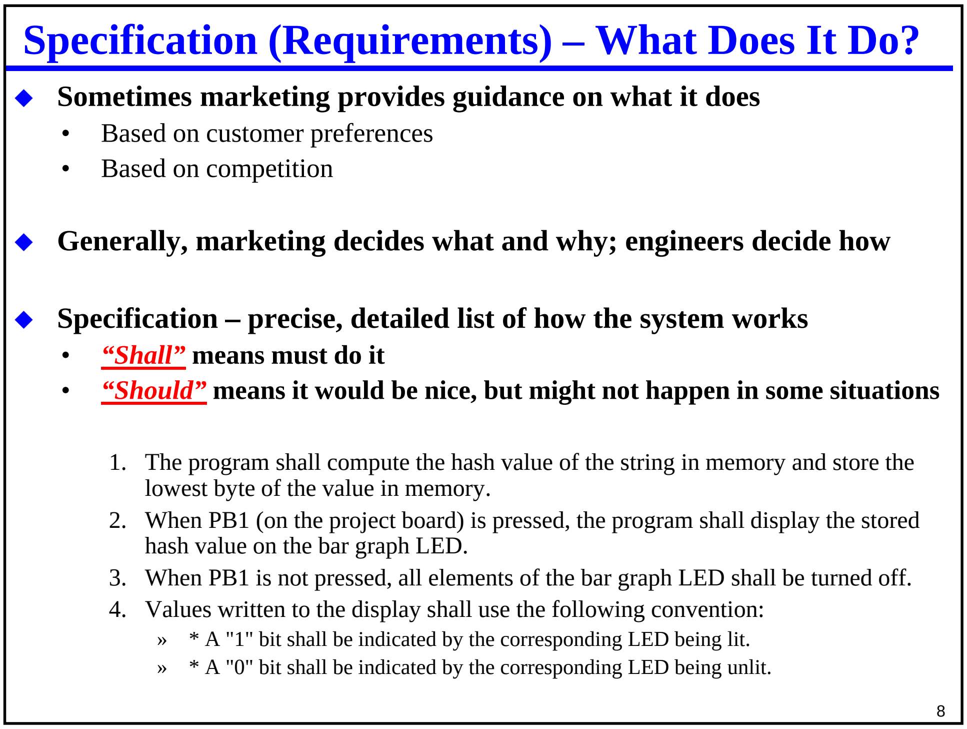
**Engineering phases:**

- **Marketing**
- **Product definition**
- **Requirements**
- **Architecture**
- **Design**
- **Implementation**
- **Test**
- **Verification & Validation**
- **Support**
- **Evolution**



## Marketing & Product Definition

- **What is the need?**
  - Personal communication device
  - More selection in TV programs
  - Better energy efficiency in a home
- **What can we design or otherwise provide to satisfy the need?**
  - Cell phones
  - On-demand TV, TiVO, etc.
  - “Smarter” water heaters, thermostats
  - Usually, product definition has a list of high level features to be provided
    - E.g., not just a “thermostat”, but “setback thermostat with four time bands for weekdays and two time bands for weekends; able to control both heating and cooling”



## Specification (Requirements) – What Does It Do?

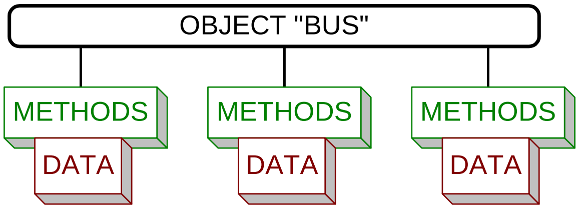
- **Sometimes marketing provides guidance on what it does**
  - Based on customer preferences
  - Based on competition
- **Generally, marketing decides what and why; engineers decide how**
- **Specification – precise, detailed list of how the system works**
  - ***“Shall”*** **means must do it**
  - ***“Should”*** **means it would be nice, but might not happen in some situations**

1. The program shall compute the hash value of the string in memory and store the lowest byte of the value in memory.
2. When PB1 (on the project board) is pressed, the program shall display the stored hash value on the bar graph LED.
3. When PB1 is not pressed, all elements of the bar graph LED shall be turned off.
4. Values written to the display shall use the following convention:
   - » * A "1" bit shall be indicated by the corresponding LED being lit.
   - » * A "0" bit shall be indicated by the corresponding LED being unlit.

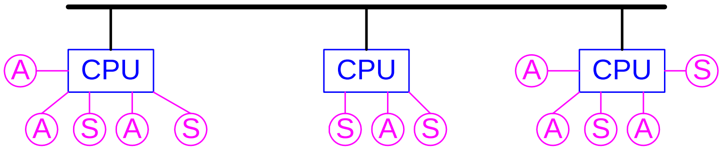
# Architecture – How Do The Pieces Fit Together?

- **Architectures are all about "boxes and arrows"**
  - Boxes are the pieces
  - Arrows are how they fit together

- **Software architecture**



- **Hardware architecture**



# Architecture Examples From [Valvano]

**Figure 1.9**
A data flow graph showing how signals pass through a motor controller.

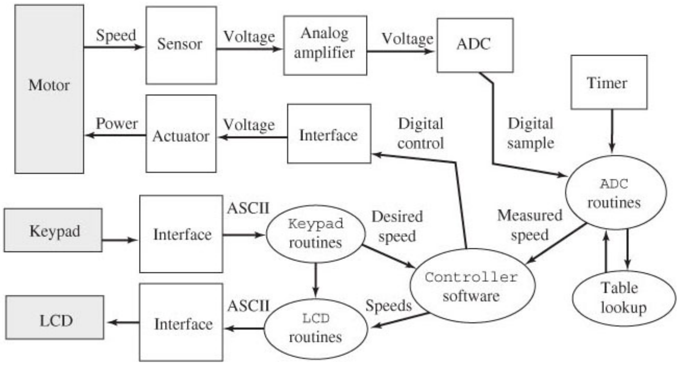
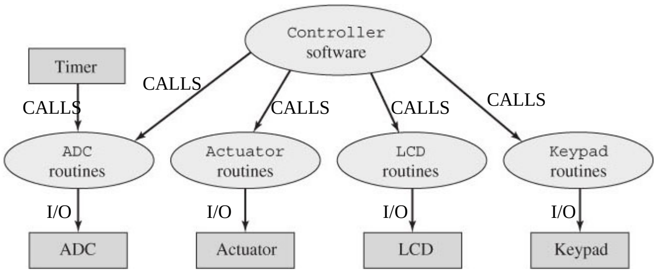


**Figure 1.10**
A call flow graph for a motor controller.



## Design – Working Out High Level Details

**(Is “High Level Details” an oxymoron? – Not in computer abstractions!)**

- **Hardware: “design” level is usually schematics**
  - Which devices are connected and how
  - BUT NOT: how wires are routed on printed circuit board
  - BUT NOT: not package selection and placement
  - Maybe you can synthesis implementation from there; depends on tool chain
- **Software: “design” is high level description**
  - Pseudocode, algorithms
  - Flow charts, state charts, most UML diagrams, …
  - **BUT NOT: actual lines of code (either C or assembly language)**
- **The point of design is to hide messy details so you can do the hard stuff**
  - Circuit board routing doesn’t (usually) affect how registers are connected
  - Software design shouldn’t worry about the name of a variable used as the index in a switch statement

## Implementation – The Gory Details

- **Hardware**
  - Component placement
  - Circuit board routing
  - Decoupling capacitors
  - Connector locations
  - Etc.
- **Software**
  - Source code
  - Header files
  - Etc.

# Test – Executing Code To Check Its Operation

- **How many of you write perfect code all the time?**
  - If you play around with code a few minutes, was that perfect testing?

- **Testing is one way to gain confidence the code is correct**
  - Involves actual execution of code (or executing with a CPU simulator)
  - Testing includes:
    - The program you are testing
    - Support framework to provide inputs/outputs
    - The workload (data inputs, etc.) you are testing it with
    - A set of expected outputs – passing the test means program performs to expectations

- **You all do some form of testing**
  - Most engineers don’t do very good testing without some training (later lecture to cover the basics)
  - Some engineers are “born testers” – they are good at breaking things!

# Verification and Validation

- **Testing is not the only way to know you got it right**
  - Verification – did you produce what you planned to produce?
    - e.g., does the implementation actually implement what the design says?
  - Validation – does it actually do the Right Thing?
    - e.g., does the implementation make the customer happy?

- **Simple verification and validation techniques**
  - Have someone grade your work (e.g., TAs grading labs)
    - But generally this doesn’t happen in the Real World
    - There is usually no “solution sheet” in the Real World either!
  - Have a buddy look over your stuff (“peer reviews”)
  - Have an outsider look over your stuff (“external reviews”)
  - Have a testing agency exercise your stuff (“FAA flight certification”)
    - This is close to grading – but they don’t have a true answer sheet
    - They expect you to give them an answer sheet and then convince them it is right

# Version Control

- **What happens if you need to go back to an old version?**
  - The new version doesn't work, and you don't remember what you changed
  - The new version doesn't work, and you can't figure out why
  - Your computer crashed and you need to restore *some* recent version
  - Someone messed up the code base before quitting
  - There is a bug in an old deployed version and you need to test small fixes
  - …
- **Use some sort of version control – *always***
  - Simplest versions:  copy directories to temp directory once in a while
  - Better:  keep old versions around for a long time  (Prog.c, Prog.save1.c, Prog.save2.c, …)   or do this at directory level
  - Best: use a version control system
    - SVN
    - Sourcesafe
    - …. lots more
    - For this course (and others) don't use one that makes your code publicly visible to avoid problems with someone else copying your stuff

# Lifecycle Support

- **What happens after you ship version 1.0?**
  - Version 1.1, Emergency Bug Patch 1.1.035, and Version 2.0 happen
  - You get calls at 3 AM because there is a bug in version 1.7.3.2
    - … but you can't even remember what is in that particular version!
    - … and you can't find, or even recreate the source code for it!
  - You get calls at your new job from desperate people at your old job
  - …
- **Good engineering is more than making it work.**
  **You need to make sure:**
  - Other people can understand it
    - For that matter, you can understand it a few years later!
  - You/others can modify it, both in small and large ways
  - You can demonstrate your part of the design isn't where the bug is (if true)
    - It is always your fault unless you can demonstrate otherwise
  - It isn't brittle to changes in operating conditions, technology, users, …

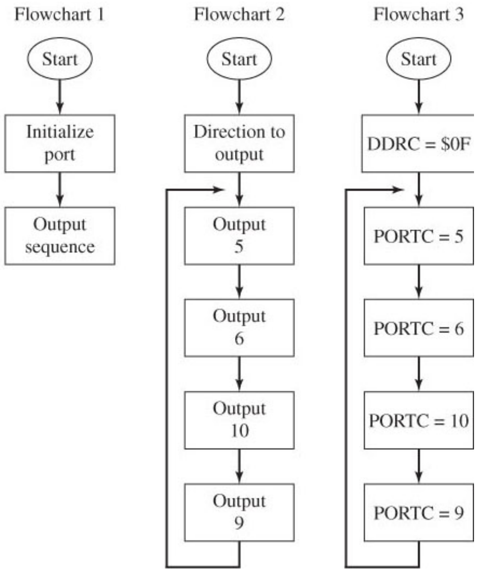
# DESIGN

- **Would you start building a house without a floorplan?**
- **Would you just start bending sheet metal without a drawing?**
- **Would you start fabricating a chip without a layout?**
- **Would you just start soldering or protoboarding without a schematic?**
- **Would you just start writing code without a software design?**

**The answer to all these questions is (or should be) NO!**

# Flowchart Basics

**Figure 1.47**
Software design for the LED output system using flowcharts.



## Flowchart Pro/Con

- **Pro:**
  - Good at describing a lot of classical software…
    … especially if its job is to execute a sequence of steps in mostly linear order
  - Everyone seems to know how to create one
  - Better than nothing
- **Con:**
  - Easy to get caught in trap of one line of code per box – pretty much useless
    - Each box should be a high level operation, not just a line of code
  - Subroutine calls are the only way to manage complexity – but not enough
  - Usually get out of date with software, because aren't that useful for maintenance
- **But, still can be pretty useful for some situations**
  - For object-oriented systems, generally use UML sequence diagrams instead

## Exercise – Flow Chart For Doing An 18-348 Lab

# Statechart Basics

- **Statecharts are a software finite state machine diagram**
  - “Bubbles” are states of a state machine
    - While in each box, perform some action
  - “Arrows” between bubbles are guarded state transitions
    - Take an arrow of the “guard” condition is true
  - Has a “reset” or initialization state
  - Can be implemented via switch statements in software

# Example Statechart

This is a controller for a multi-speed motor or other similar application

- Inputs: SPDBUTTON and ONOFF
- Outputs: Speed = {Stop, Slow, Med, Fast}
- State names (arbitrary labels): {OFF, SLOW, MEDIUM, FAST}
- System Reset is to state s1

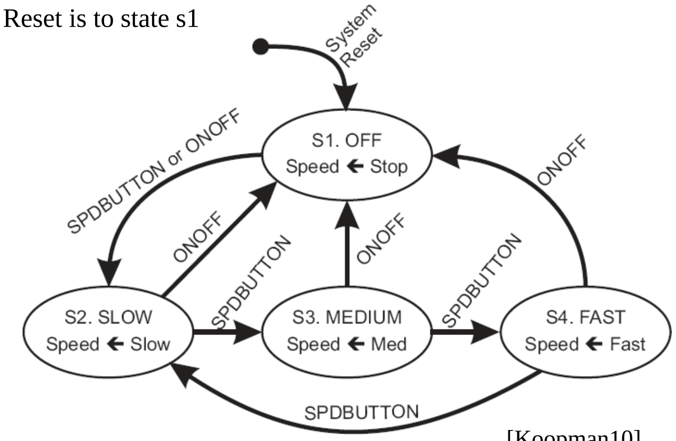


Figure 13.1. An example statechart.

[Koopman10]

# Example Statechart Implementation – 1

```
enum CurrState
{OFF, SLOW, MEDIUM, FAST}; // define states

#define SpdOff  0   // define speed constant values
#define SpdSlow 10
#define SpdMed  15
#define SpdFast 25

CurrState = OFF;  // initialize state machine to OFF

while (1) // do forever
{
  switch (CurrState) {
  case OFF:     // State S1
    speed(SpdOff);         // Take action in state

    // Test arc guards and take transitions
    if (SpdButton() == TRUE || OnOffButton() == TRUE)
    {CurrState = SLOW;}
    break;   // go to end of switch statement

  case SLOW:      // State S2
    speed(SpdSlow);     // take action
    if (SpdButton() == TRUE)    {CurrState = MEDIUM;}
    if (OnOffButton() == TRUE) {CurrState = OFF;}
    break;
```

# Example Statechart Implementation – 2

```
    case MEDIUM:    // State S3
      speed(SpdMed);      // take action
      if (SpdButton() == TRUE)    {CurrState = FAST;}
      if (OnOffButton() == TRUE) {CurrState = OFF;}
      break;

    case FAST:      // State S4
      speed(SpdFast);      // take action
      if (SpdButton() == TRUE)    {CurrState = SLOW;}
      if (OnOffButton() == TRUE) {CurrState = OFF;}
      break;

    default:        // Error – invalid state
      error(“invalid state!”); // should never get here
    }
  }
```

## Statechart Pro/Con

- **Pro:**
  - If you had 18-240, you already know how to do these!
    - They are the software version of FSM state diagrams
  - Many embedded systems have a lot of modes; great for that
    - What common embedded systems have modes?
  - Forcing designers to look for states generally improves designs
    - Lots of duplicative nested “if” statements usually means it should have been designed as a state machine with a “switch” statement instead
- **Con:**
  - Not every system is reducible to states
  - Not good at representing flows of control (long lists of steps)
- **Other considerations**
  - Use a switch statement to convert to code, with integer state numbers

## Exercise: State Chart For Traffic Light (one direction)

## On Clarity in Requirements

- **A wife asks her software engineer husband, “Could you please go shopping for me and buy one carton of milk? And if they have eggs, get six.”**

- **A short time later the husband comes back with 6 cartons of milk and no eggs.  The wife asks him, “Why did you buy six cartons of milk?!”**

- **He replied, “They had eggs.”**

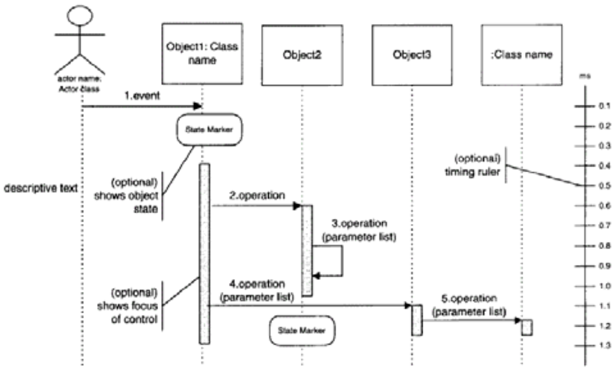
**www.ganssle.com/jokes.htm.**

## Sequence Diagram Basics

- **Sequence diagrams show the interactions between components**
  - Each component is a box at the top of the diagram
  - Time extends downward from the component
  - Arcs go between timelines to show messages/method calls/etc.

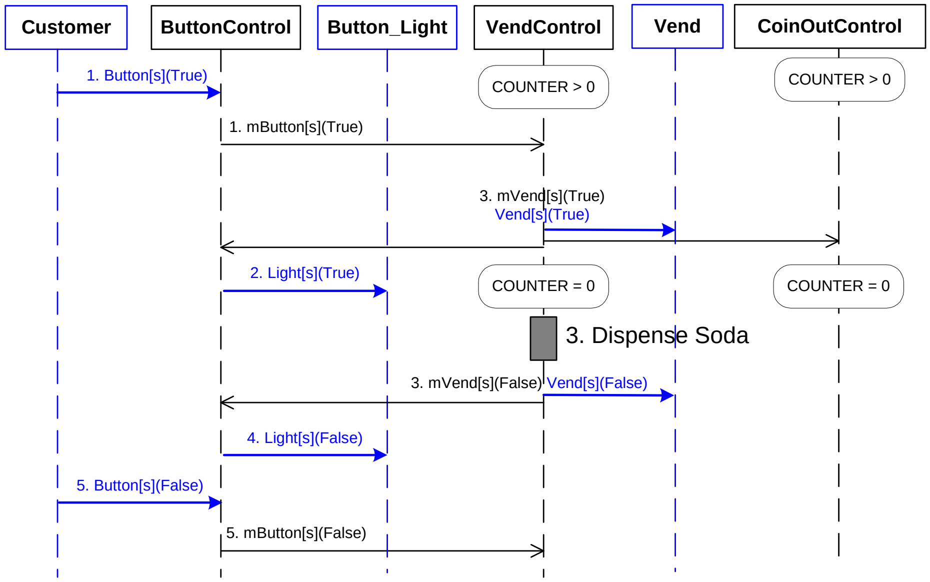
**Sequence Diagram**

Shows a sequenced set of messages illustrating a specific example of object interaction.



**Sequence diagrams** have two dimensions.  The vertical dimension usually represents time, the horizontal represents different objects.  (These may be reversed.)

# Example Sequence Diagram



# Sequence Diagram Pro/Con

- **Pro:**
  - Well suited to object-oriented designs (one box per object)
  - Lets you emphasize interactions rather than in-line order of steps
  - Very useful for distributed embedded systems (18-649 course)

- **Con:**
  - Only works if you have organized code into multiple objects

(Learn more about sequence diagrams & state charts in 18-649)

# Exercise: Sequence Diagram For Getting Voice Mail

# Good & Bad Design Practices

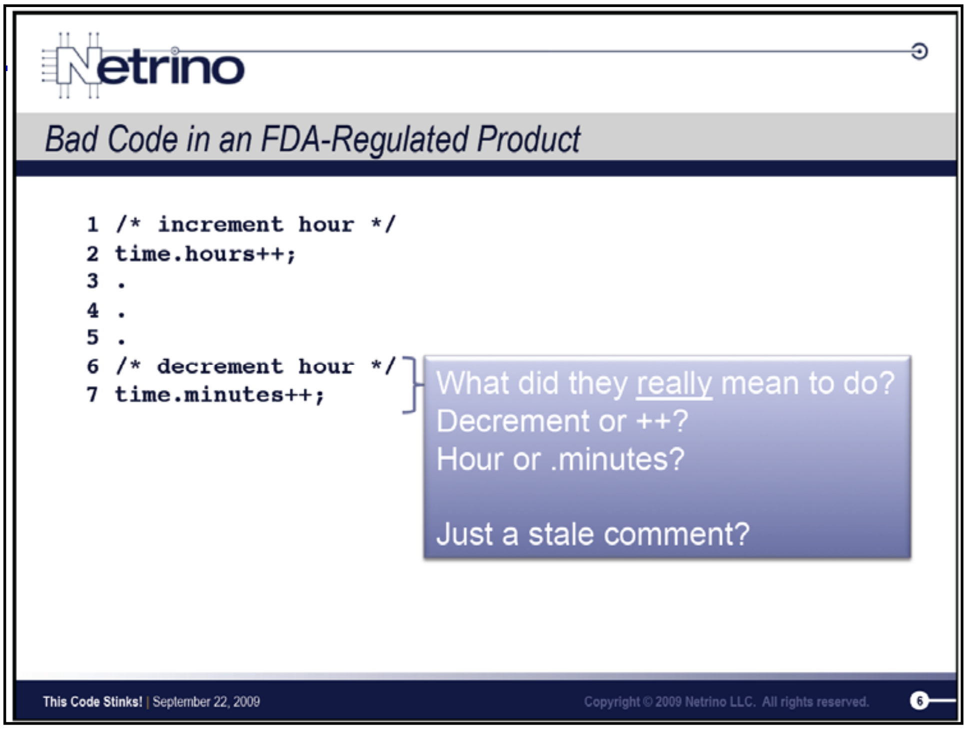
- **Some GOOD practices**
  - Make tradeoffs and decisions at a high level – before writing code
  - Keep design documents in synch with implementation
  - Get design documents reviewed by someone else before spending time on implementation
- **Some BAD practices**
  - Starting writing design documents after the implementation is completed (design documents as “documentation” rather than “design steps”)
  - Overly detailed designs
    - One line of code per box is bad
  - Ignoring design and jumping right to implementation

# IMPLEMENTATION

- **How long does software last?**
  - Many people act as if their software will be thrown away next week
  - But cars typically live 10-15 years after sale
  - Houses last up to 100 years
  - Example: SAGE air defense system
    - Started 1954
    - Deployed 1963 – vacuum tube hardware (500,000 lines of code)
    - In operation until 1983 – (IBM PC came out in 1982)

[http://history.acusd.edu/gen/20th/sage.html]

  - Example: MS-DOS. Released 1981. Still runs as command line for Windows.
  - 1AESS telephone switch: 1976 through 2008+ (maybe some still working)



Netrino

## Bad Code in an FDA-Regulated Product

```
/* increment hour */
time.hours++;
.
.
.
/* decrement hour */
time.minutes++;
```

What did they really mean to do?
Decrement or ++?
Hour or .minutes?

Just a stale comment?

## Why Do You Need Coding Style Sheets?

- **So others can understand your software quickly and accurately**
  - Your design colleagues
  - YOU – when you look at it years later
  - People who have to maintain and expand it
  - Your successors, when you move on but the software stays behind

- **How can you make your code understandable?**
  1. Document the architecture and design, not just the code
  2. Use a consistent style of programming, with good practices

- **This is a partial step toward hard-to-get wrong code**
  - Coding rules can avoid the dark, scary corners of languages
  - Avoid static analysis problems (compiler warnings)

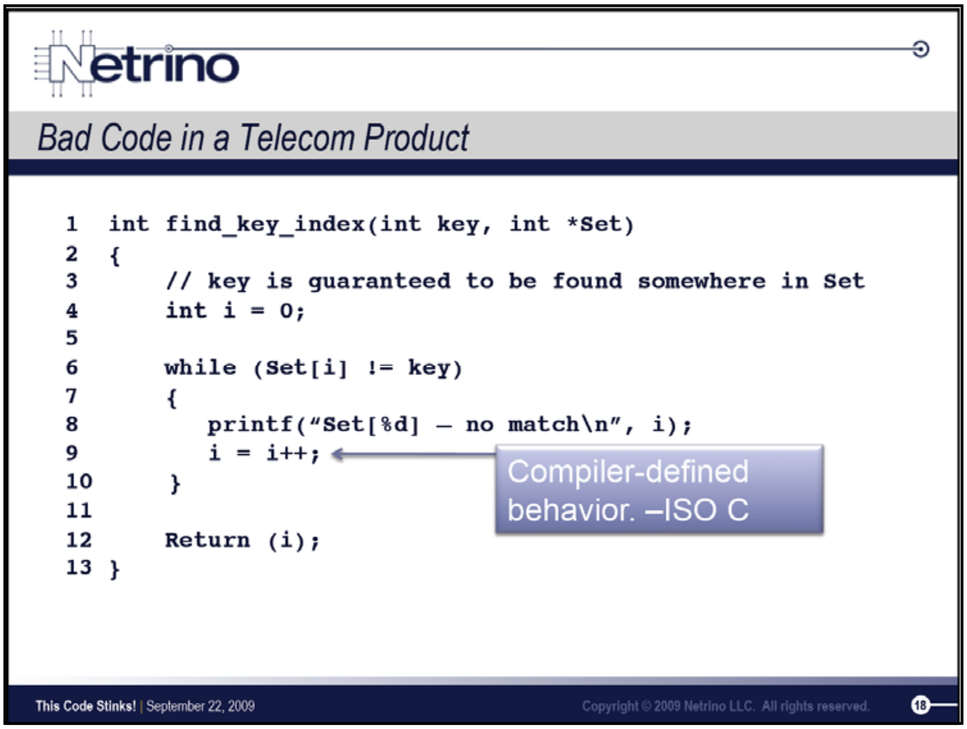
## 18-348 Code Style Overview

**Every file shall have:**

- **Title block**
  - Programmer's name and revision history of software
  - Summary of externally visible items (e.g., variables visible to other modules)
  - Tool chain specification (e.g., codewarrior version and target chip)
- **Global variable and constant definitions**
  - RAM variables
  - ROM constant values
- **"Main" routine**
  - Must initialize stack pointer, interrupts, and so on
- **Major routines/procedures/methods (one page each, max)**
- **Support routines**

# Important Practices

- **Avoid “magic numbers”**
  - If a value is used repeatedly or could possibly change, use an EQU value
    - or #define
    - or C++ const
- **Almost every line of code should have a comment**
  - Describe why something is happening and end goal…
    … not just what the instruction is doing to the machine
  - **BAD:** CLC ; Clear Carry Bit
  - **GOOD:** CLC ; Set return status flag (cy bit) to False
  - Also include higher level comments about what is happening
- **Code should compile “clean”**
  - No warnings at all – otherwise you will miss a new warning if it gets lost in all the “false alarm” warnings



Netrino

## Bad Code in a Telecom Product

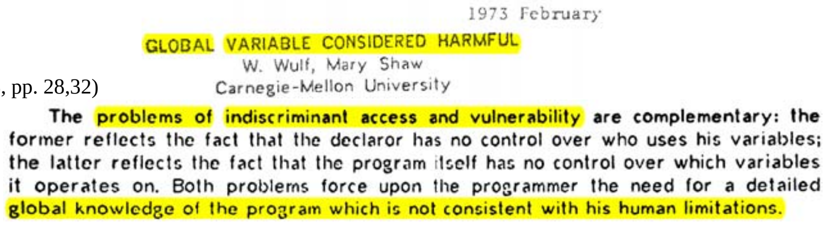
```
int find_key_index(int key, int *Set)
{
    // key is guaranteed to be found somewhere in Set
    int i = 0;

    while (Set[i] != key)
    {
        printf(“Set[%d] – no match\n”, i);
        i = i++;
    }

    Return (i);
}
```

Compiler-defined behavior. –ISO C

This Code Stinks! | September 22, 2009

# Avoid Global Variables In Real Code

- **Global variables can be read/written from any module in the system**
  - In contrast, local variables can only be seen from a particular software module
- **Excessive use of globals tends to compromise modularity**
  - Changes to code in one place affect other parts of code via the globals
  - **In other words: Global Variables are Considered Harmful**



1973 February

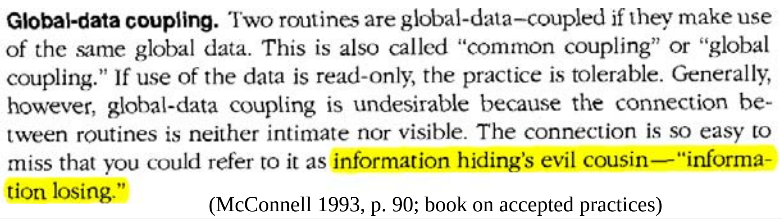
GLOBAL VARIABLE CONSIDERED HARMFUL

W. Wulf, Mary Shaw

Carnegie-Mellon University

The problems of indiscriminant access and vulnerability are complementary: the former reflects the fact that the declaror has no control over who uses his variables; the latter reflects the fact that the program itself has no control over which variables it operates on. Both problems force upon the programmer the need for a detailed global knowledge of the program which is not consistent with his human limitations.

(Wulf 1973, pp. 28,32)



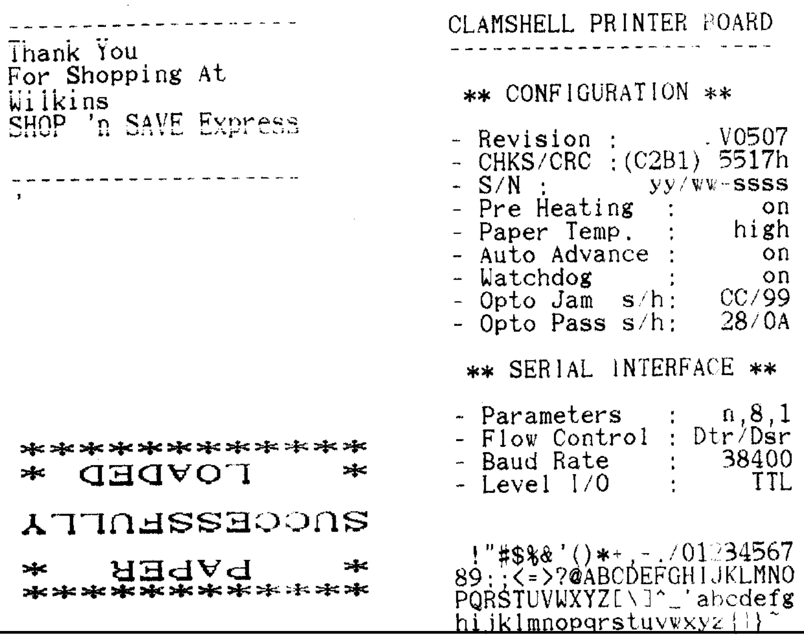
**Global-data coupling.** Two routines are global-data–coupled if they make use of the same global data. This is also called "common coupling" or "global coupling." If use of the data is read-only, the practice is tolerable. Generally, however, global-data coupling is undesirable because the connection between routines is neither intimate nor visible. The connection is so easy to miss that you could refer to it as information hiding's evil cousin—"information losing."

(McConnell 1993, p. 90; book on accepted practices)

# Other Good Ideas For Product Development

- **Version number**
  - Store the version number of the software in ROM
  - What if a PROM label comes off? You won't know which version it is
  - Also, ROM-based version number makes diagnosis easier
- **Make variables distinct in the first 6 characters**
  - Some assemblers think FOOBAR1 and FOOBAR2 are the same – because they just look at length and first 6 characters!!
- **Set "unused" resources to something safe**
  - E.g., unused ROM should be set to a "halt" instruction and not random junk
  - E.g., unused interrupt vectors should go to an error recovery routine
- **Uniform naming conventions across project(s)**
- **Copyright and proprietary information**
  - Is this module a trade secret?

## Gas Pump Startup Printout Example (Aug 2009)



```
--------------------
Thank You
For Shopping At
Wilkins
SHOP 'n SAVE Express

--------------------
'

*************
*  LOADED    *

SUCCESSFULLY

*   PAPER    *
*************
```

```
CLAMSHELL PRINTER BOARD
------------------ ----

 ** CONFIGURATION **

- Revision :         .V0507
- CHKS/CRC :(C2B1) 5517h
- S/N :         yy/ww-ssss
- Pre Heating   :       on
- Paper Temp.   :     high
- Auto Advance  :       on
- Watchdog      :       on
- Opto Jam  s/h:    CC/99
- Opto Pass s/h:    28/0A

 ** SERIAL INTERFACE **

- Parameters    :    n,8,1
- Flow Control  :  Dtr/Dsr
- Baud Rate     :    38400
- Level I/O     :      TTL

 !"#$%&'()*+,-./01234567
89:;<=>?@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_'abcdefg
hijklmnopqrstuvwxyz{|}~
```

## Embedded Design Review Checklist

- **Areas based on doing many design reviews in industry**
  - Function – does the code do the right thing and not the wrong thing?
  - Style – is the code structured in a way that makes bugs less likely?
  - Architecture – is code modular, cleanly nested, and without complex dependencies?
  - Exception handling – does the code fall over if something goes worng?
  - Timing – are real time deadlines and concurrency handled appropriately?
  - Validation and Test – is test coverage high?
  - Hardware – are timing, power, and other hardware problems addressed?

Source: http://betterembsw.blogspot.com/2011/11/embedded-system-code-review-checklist.html
(See the detailed checklist in these handouts)

- **We'll talk more about design reviews in a later lecture**
  - This checklist is for information, not a course requirement
- **A good way to use a checklist this complex:**
  - Have 3-4 reviewers in a joint session
  - Assign different sections to each reviewer for primary responsibility

# Conclusions

- **Engineering projects have phases**
  - Marketing, product definition, requirements, architecture, design, implementation, test, V&V, support, evolution
- **Requirements**
  - Shall vs. should
  - Keep an eye on these in projects
- **Design**
  - Flowcharts
  - Statecharts
  - Sequence Diagrams
- **Implementation**
  - Basic coding style
  - Good & Bad practices
  - Read the course coding style sheet before recitation!
  - Read the style sheet attached to this lecture and understand what it is talking about
    - Not required to follow this style sheet for course projects, but might be a good idea

# Lab Skills

- **State chart**
  - Create a state chart for a moderately complex system
- **Implementation**
  - Assembly language implementation of a state machine
  - Follow course coding style for all labs after this point

# Embedded System Code Review Checklist

Gautam Khattak & Philip Koopman

**July 2012 Version 1.01**

## Recommended Usage:

- Assign each section below to a specific reviewer, giving two or three sections to each reviewer.
- Ensure that each question has been considered for every piece of code.
- Review 100-400 lines of code per 1-2 hour review session. Do the review in person.

## FUNCTION

- ☐ F-1. Does the code match the design and the system requirements?
- ☐ F-2. Does the code do what it should be doing?
- ☐ F-3. Does the code do anything it should not be doing?
- ☐ F-4. Can the code be made simpler while still doing what it needs to do?
- ☐ F-5. Are available building blocks used when appropriate? (algorithms, data structures, types, templates, libraries, RTOS functions)
- ☐ F-6. Does the code use good patterns and abstractions? (e.g., state charts, no copy-and paste)
- ☐ F-7. Can this function be written with a single point of exit? (no returns in middle of function)
- ☐ F-8. Are all variables initialized before use?
- ☐ F-9. Are there unused variables?
- ☐ F-10. Is each function doing only one thing? (Does it make sense to break it down into smaller modules that each do something different?)

## STYLE

- ☐ S-1. Does the code follow the style guide for this project?
- ☐ S-2. Is the header information for each file and each function descriptive enough?
- ☐ S-3. Is there an appropriate amount of comments? (frequency, location, and level of detail)
- ☐ S-4. Is the code well structured? (typographically and functionally)
- ☐ S-5. Are the variable and function names descriptive and consistent in style?
- ☐ S-6. Are "magic numbers" avoided? (use named constants rather than numbers)
- ☐ S-7. Is there any “dead code” (commented out code or unreachable code) that should be removed?
- ☐ S-8. Is it possible to remove any of the assembly language code, if present?
- ☐ S-9. Is the code too tricky? (Did you have to think hard to understand what it does?)
- ☐ S-10. Did you have to ask the author what the code does? (code should be self-explanatory)

## ARCHITECTURE

- ☐ A-1. Is the function too long? (e.g., longer than fits on one printed page)
- ☐ A-2. Can this code be reused? Should it be reusing something else?
- ☐ A-3. Is there **minimal** use of global variables? Do all variables have minimum scope?
- ☐ A-4. Are classes and functions that are doing related things grouped appropriately? (cohesion)
- ☐ A-5. Is the code portable? (especially variable sizes, e.g., “int32” instead of “long”)
- ☐ A-6. Are specific types used when possible? (e.g., “unsigned” and typedef, not just "int")
- ☐ A-7. Are there any if/else structures nested more than two deep? (consecutive “else if” is OK)
- ☐ A-8. Are there nested switch or case statements? (they should never be nested)

## EXCEPTION HANDLING

□ E-1. Are input parameters checked for proper values (sanity checking)?
□ E-2. Are error return codes/exceptions generated and passed back up to the calling function?
□ E-3. Are error return codes/exceptions handled by the calling function?
□ E-4. Are null pointers and negative numbers handled properly?
□ E-5. Do switch statements have a default clause used for error detection?
□ E-6. Are arrays checked for out of range indexing? Are pointers similarly checked?
□ E-7. Is garbage collection being done properly, especially for errors/exceptions?
□ E-8. Is there a chance of mathematical overflow/underflow?
□ E-9. Are error conditions checked and logged? Are the error messages/codes meaningful?
□ E-10. Would an error handling structure such as try/catch be useful? (depends upon language)

## TIMING

□ T-1. Is the worst case timing bounded? (no unbounded loops, no recursion)
□ T-2. Are there any race conditions? (especially multi-byte variables modified by an interrupt)
□ T-3. Is appropriate code thread safe and reentrant?
□ T-4. Are there any long-running ISRs? (no loops inside ISRs; should be half-page of code)
□ T-5. Are interrupts masked for more than a few clocks?
□ T-6. Is priority inversion avoided or handled by the RTOS?
□ T-7. Is the watchdog timer turned on? Is the watchdog kicked only if ***every*** task is executing?
□ T-8. Has code readability been sacrificed for unnecessary optimization?

## VALIDATION & TEST

□ V-1. Is the code easy to test? (how many paths are there through the code?)
□ V-2. Do unit tests have 100% branch coverage? (code should be written to make this easy)
□ V-3. Are the compilation and/or lint checks 100% warning-free? (are warnings enabled?)
□ V-4. Is special attention given to corner cases, boundaries, and negative test cases?
□ V-5. Does the code provide convenient ways to inject faulty conditions for testing?
□ V-6. Are all interfaces tested, including all exceptions?
□ V-7. Has the worst case resource use been validated? (stack space, memory allocation)
□ V-8. Are run-time assertions being used? Are assertion violations logged?
□ V-9. Is there commented out code (for testing) that should be removed?

## HARDWARE

□ H-1. Do I/O operations put the hardware in correct state?
□ H-2. Are min/max timing requirements met for the hardware interface?
□ H-3. Are you sure that multi-byte hardware registers can't change during read/write?
□ H-4. Does the software ensure that the system resets to a well defined hardware system state?
□ H-5. Have brownout and power loss been handled?
□ H-6. Is the system correctly configured for entering/leaving sleep mode (e.g. timers)?
□ H-7. Have unused interrupt vectors been directed to an error handler?
□ H-8. Has care been taken to avoid EEPROM corruption? (e.g., power loss during write)