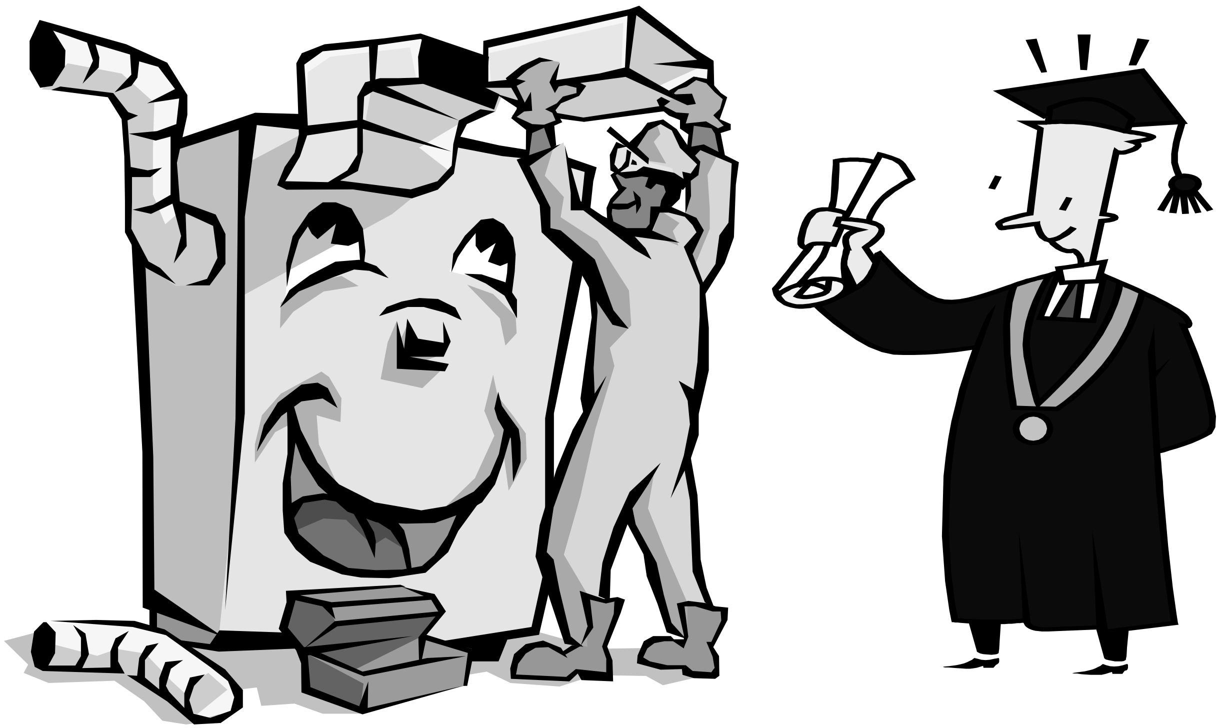
Top 25

Most Common Mistakes with Real-Time Software Development

Embedded Systems Conference
Boston, September 2006
Class ESC 401/421

Dave Stewart
Director of Software Engineering
InHand Electronics
Rockville, Maryland
dstewart@inhand.com
http://www.inhand.com

Why this presentation?



Novices and Experts
in both industry and university,
make the same mistakes over and over again.

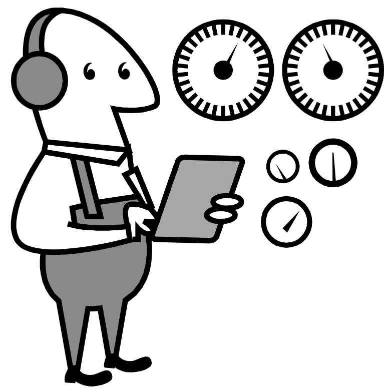
The Order

The order is subjective, based on personal observations when using the following criteria:

#1

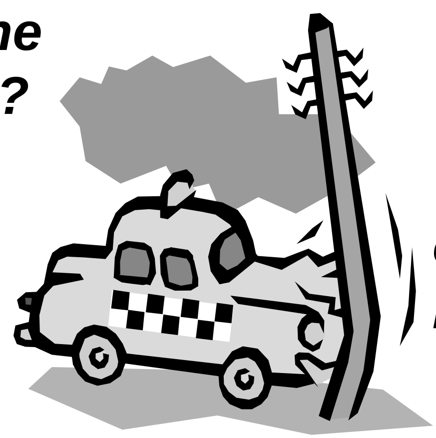
is highest on list

What is the bottom line regarding Time and Money?



How often is the mistake made?

Does the mistake increase complexity of the code?



What is the effect of the mistake on reliability?

12345678910

The Order is Not Really Important

What is important is that the mistake is on the list!

Correcting just ONE mistake can save thousands of dollars or significantly improve quality and robustness of software.

Correcting SEVERAL mistakes can lead to savings and improvements that are incalculable!

“My Problem is Different”

#25

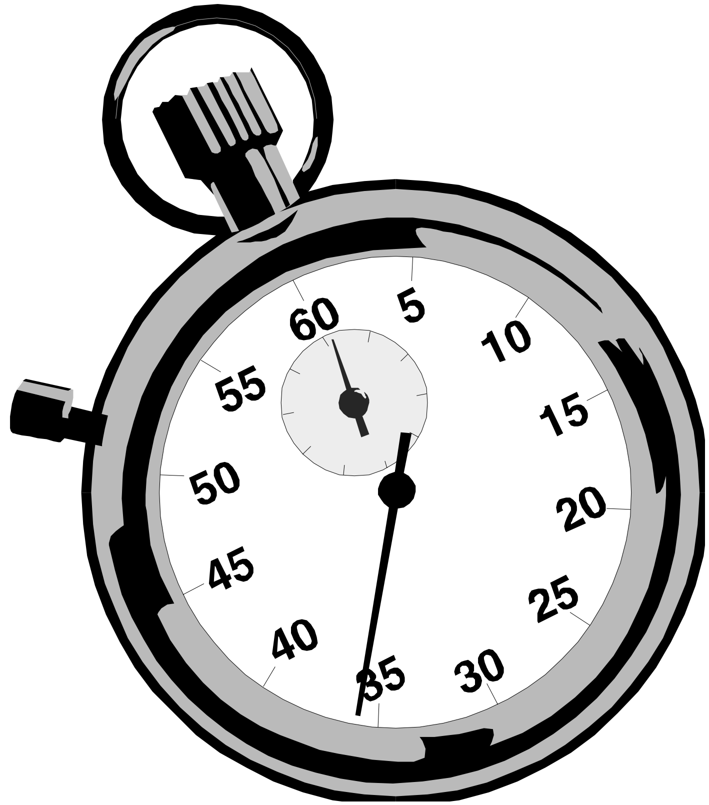
Learn from experience of others

Focus on similarities, not differences

Rarely, if ever, is entire problem different

Delays implemented as empty loops

#24



Use RTOS timing mechanisms

Build your own mechanism that automatically profiles CPU

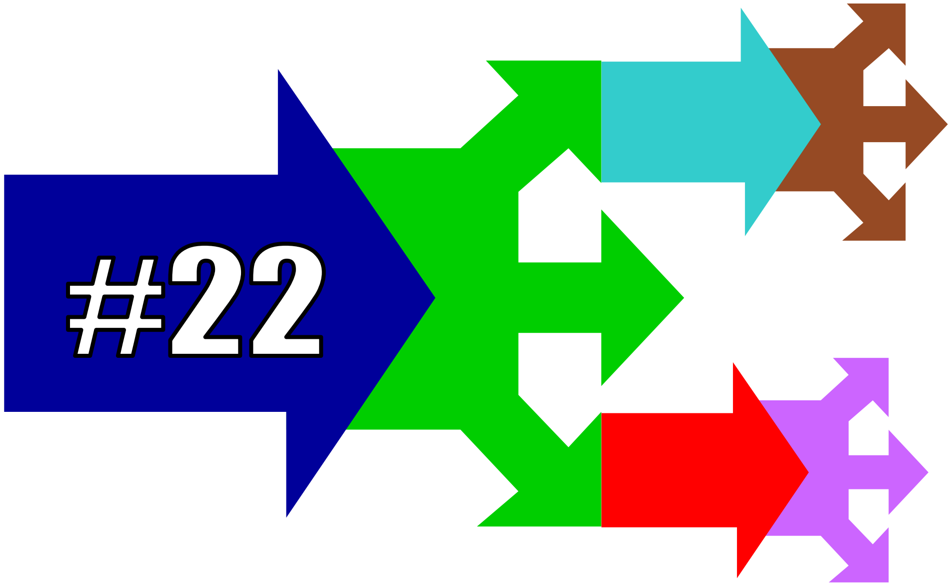
Poll the count-down value of a timer

#23

Tools choice driven by marketing hype, not by evaluation of technical needs

Select tools based on your own technical needs, not just because everybody else is using them.

Spending $2,000 for the right tool can save $100,000 in labor.



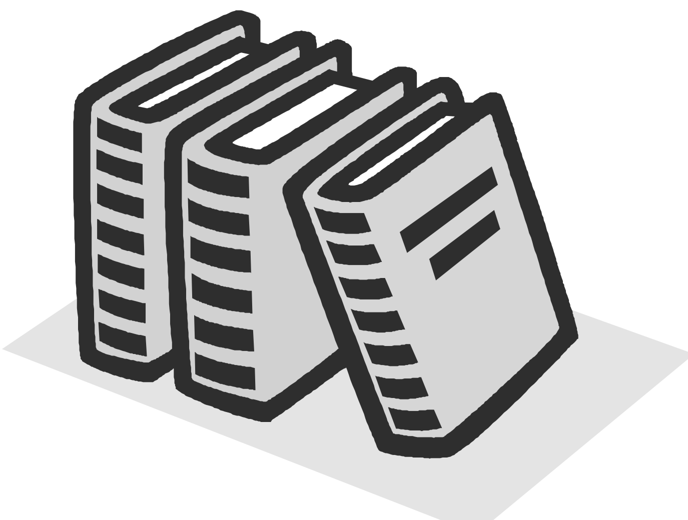
Large *if-then-else* and *case* statements
Usually a sign of implementation without design.

Instead, Design First!
Use Finite State Machines to reduce complexity.

Start implementation with documentation (the design document)

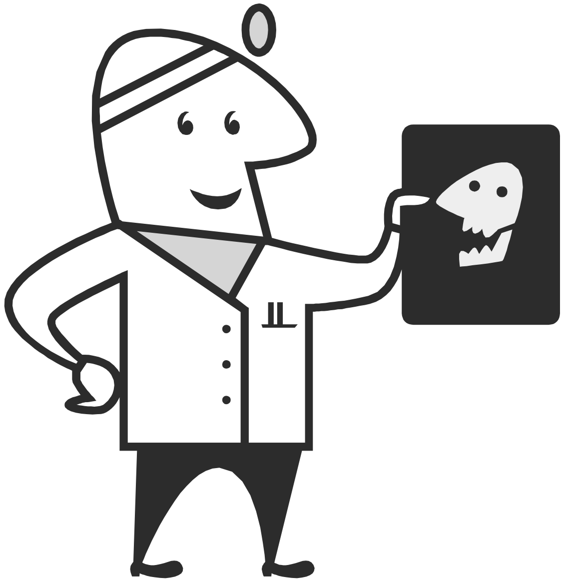
Revise documentation interactively; this serves as a sanity check to ensure that the code implements everything defined in it.

Document is written when functionality is fresh in programmer's mind.



#21

Documentation written *after* implementation



#20

Interactive and incomplete test programs

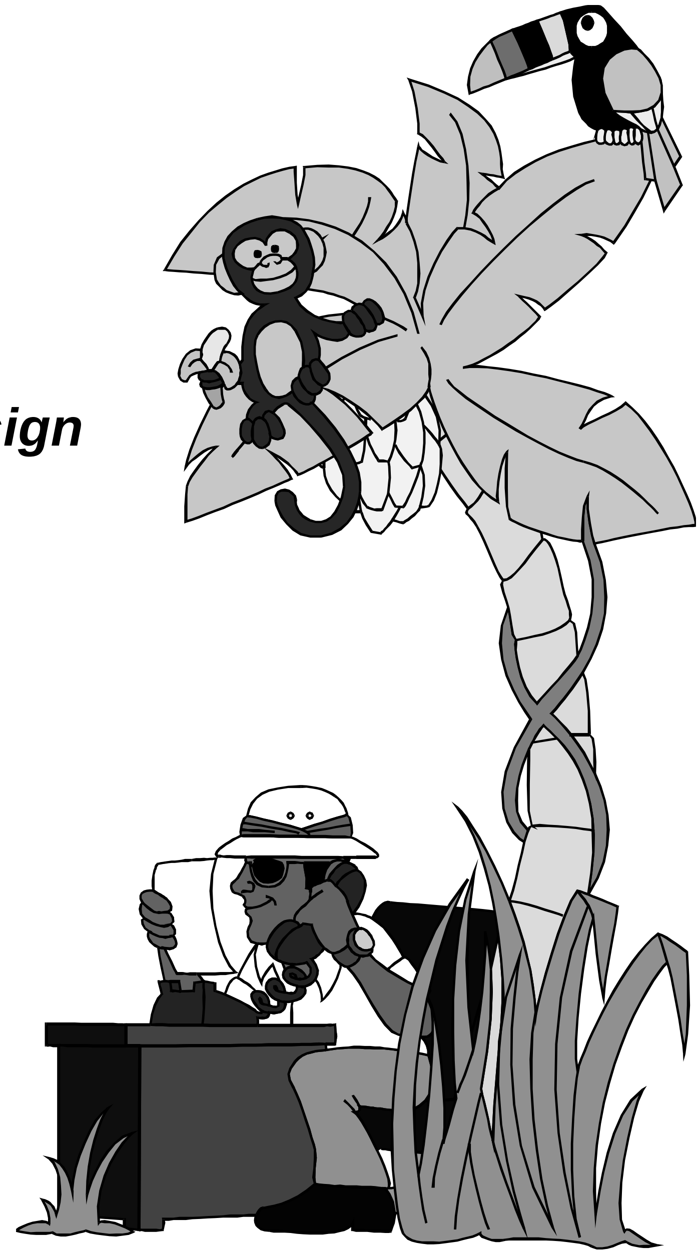
Instead:

- ***Create non-interactive test programs***
- ***Simulate input devices with known patterns***
- ***Always test the entire application all the time***
- ***Nightly extensive self-tests***

Software Engineers Don't Participate in Hardware Design

#19

Leads to over-designing the system
Instead, promote Hardware/Software Co-Design



#18

No Simulators of
Target Application

Using a simulator:

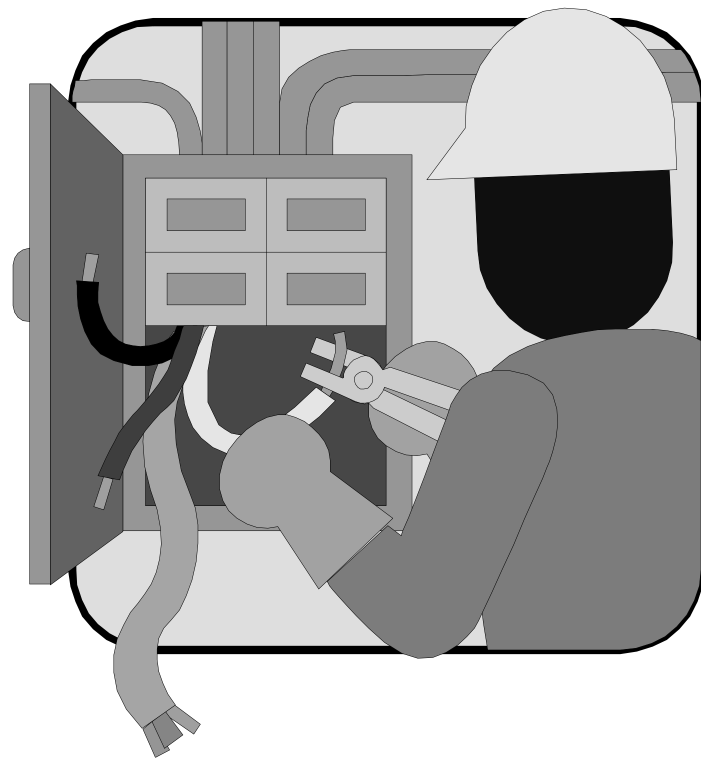
- ***Faster development***
- ***Better debugging tools***
- ***Multiple programmers***
- ***Customer feedback***
- ***Deeper understanding***
- ***Safer and cheaper!***

Error detection and handling is an after-thought, and implemented through trial and error

#17

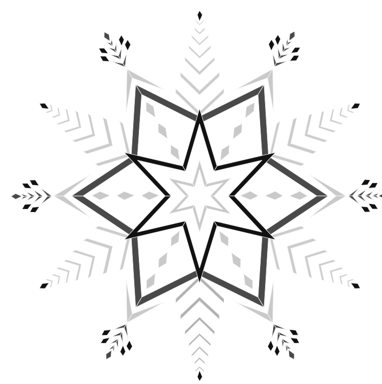
Treat errors as inputs, and error handling as a state

Error detection and handling must be specified and designed prior to implementation.



#16

Generalizations based on a single architecture

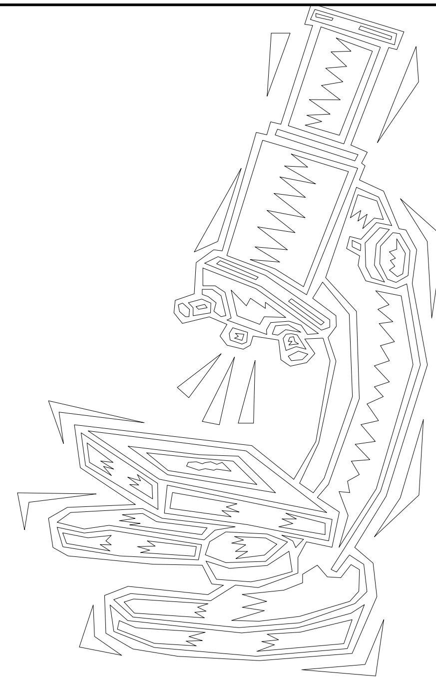


Develop code on multiple architectures simultaneously

Don’t generalize everything!

Create configurable modules for whatever is different between architectures

#15



Optimizing at the Wrong Time

3*x or x+x+x

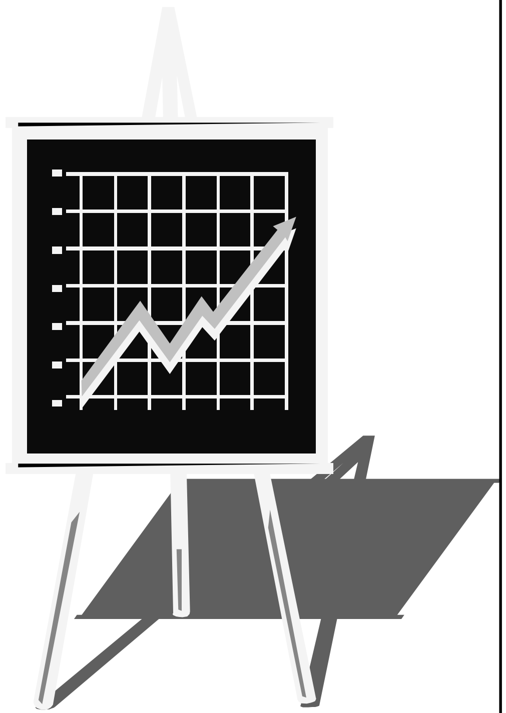
Do not perform fine-grain optimizations unless needed, and only during final stages of implementation

Measure performance after each optimization to ensure it is in fact an optimization

Do coarse-grain optimization during design phase

#15

Optimizing at the Wrong Time



To perform good coarse-grain optimization, must analyze hardware peculiarities before starting

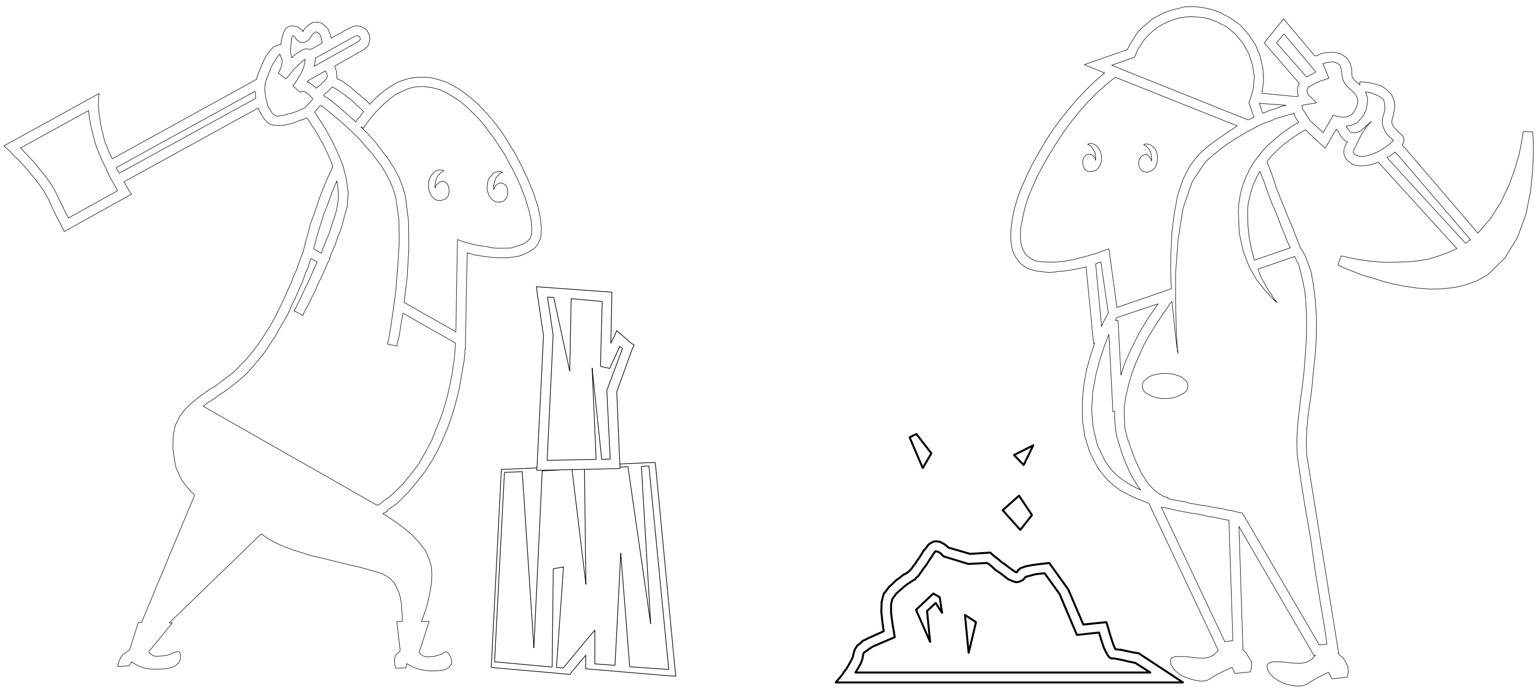
Profile CPU before writing programs for it, to identify and understand anomalies.

Better understanding of hardware peculiarities will lead to better designs.

On a 9 MHz Z180:

Byte+byte:	***7 usec***
16-bit+16-bit:	***12 usec***
32-bit+32-bit:	***28 usec***
float+float:	***137 usec***
float+byte:	***308 usec***

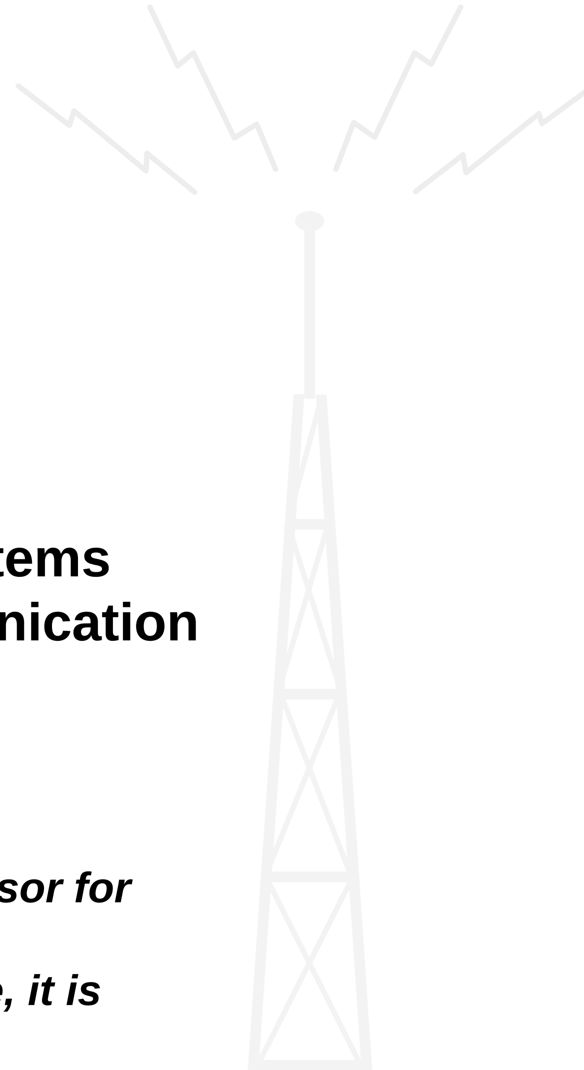
#14 Reusing code not designed for reuse



Don't waste time trying to use old code that was not designed for reuse. Instead, re-design it using proven techniques for software reuse.

Using blocking forms of message passing

#13



Problems:

- **Reduced real-time schedulable bound**
- **Significant overhead**
- **Results in lots of aperiodic servers**
- **Forces tight synchronization**
- **Potential for deadlock in closed-loop systems**
- **Additional complexity for 1:many communication**

Schedulable bound: The maximum utilization of the processor for which a task set is guaranteed to still meet all its timing constraints. Ideally, schedulable bound is 100%. In practice, it is lower than that.

Using blocking forms of message passing

#13

Solution:

Minimize inter-module communication and synchronization

Use a shared-memory based protocol, such as state variable communication, publish/subscribe, or non-blocking message passing.

If blocking is unavoidable, use proper synchronization techniques to prevent priority inversion and deadlock, such as priority ceiling protocol.

#12

No memory analysis during design

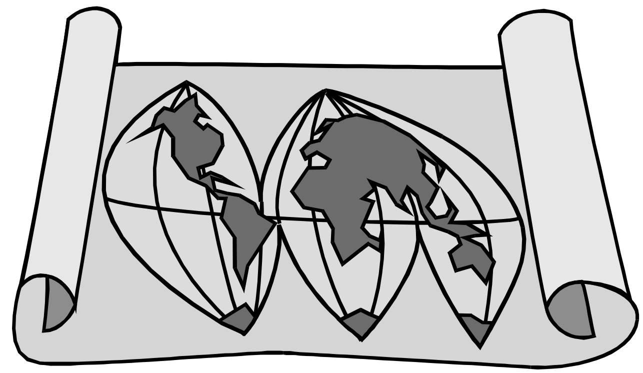
Compute memory usage during design phase.

Don't forget about memory used by string constants.

For code, estimate a budget for each module.

#11

Improper use of
Global Variables!



Problem -- reduces maintainability of software:
- ***Global variables (even static ones) are shared.***
- ***Limits expandability by preventing replication of modules.***
- ***Causes many inter-module dependencies.***

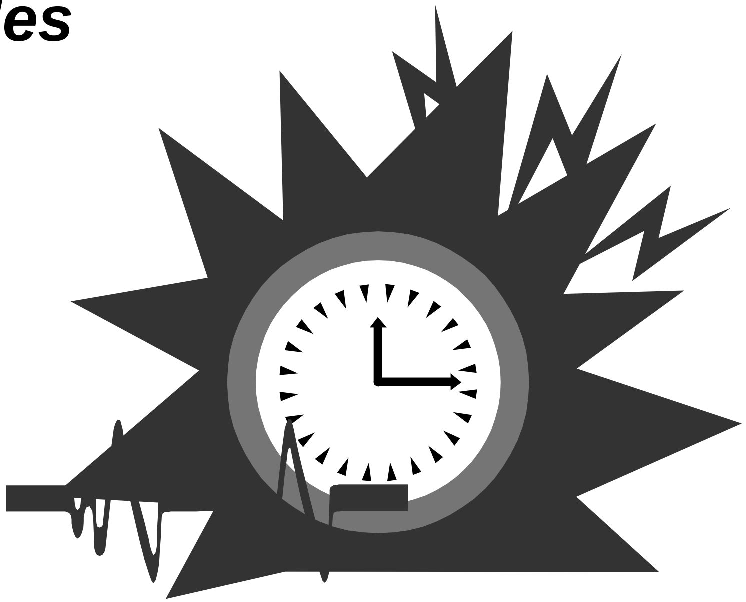
Solution -- eliminate (most) global variables:
- ***Use proper data abstraction and encapsulation***
- ***Use shared memory mechanisms to control access, such as State Variable Table, Publish/Subscribe, etc.***

Indiscriminate use of interrupts

#10

Interrupts are an enemy to real-time predictability:

- ***Always have high priority***
- ***Force a need for global variables***
- ***Cannot be scheduled***
- ***Difficult to analyze***
- ***Execute within wrong context***
- ***Operate in kernel space***
- ***Priority inversion***
- ***Difficult to debug***



Indiscriminate use of interrupts

#10

Instead, minimize use of interrupts whenever possible

Periodic polling threads are more desirable than interrupts because they are schedulable

Complex code should be replaced by a signal to an aperiodic server

Only use real-time analysis methods that take interrupt handling into account

Indiscriminate use of interrupts

#10

Myth: Interrupts save CPU time over processes
Reality: Not usually in real-time systems

Interrupts: 20 to 50 μsec per interrupt

Threads: 50 to 100 μsec per context switch

Non-preemptive processes: 10 to 30 μsec per switch

A real-time executive with non-preemptive periodic processes can sometimes provide more predictable results and better utilization than using interrupts.

Indiscriminate use of interrupts

#10

Myth: Interrupts save CPU time over processes
Reality: Not usually in real-time systems

Interrupts save a bit of overhead, but at the huge cost of reducing the schedulable bound and increasing the possibility of race conditions

Saving 10% overhead by using interrupts might reduce schedulable bound by 30% and increase overhead of using shared variables by 20%!

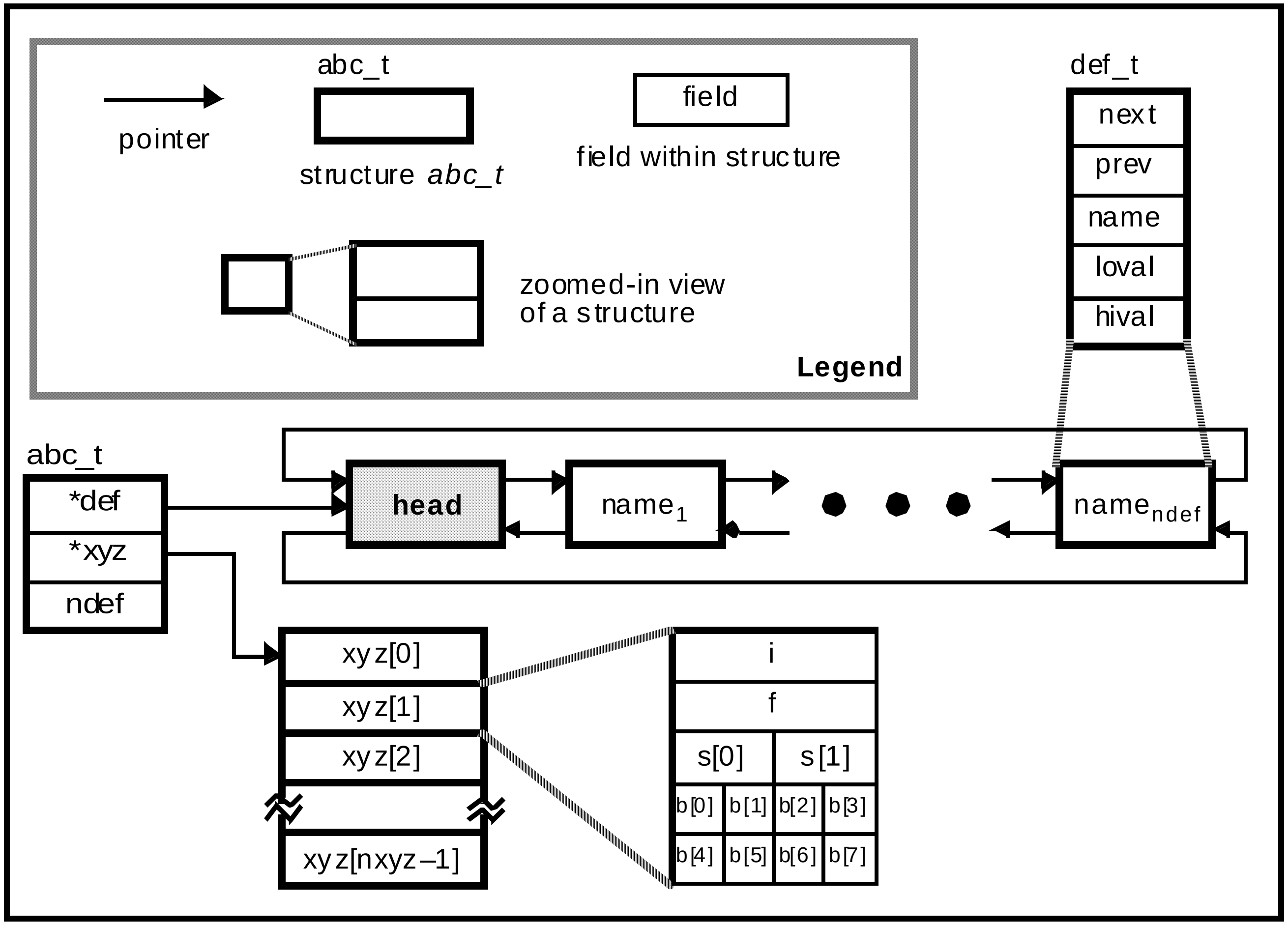
#9 Poor Software Design Diagrams

#9 No Software Design Diagrams

```
typedef struct _def_t {
   struct _def_t *next;
   struct _def_t *prev;
   char name[8];
   short loval;
   short hival;
} def_t;

typedef struct _xyz_t {
   int i;
   float f;
   short s[2];
   unsigned char b[8];
} xyz_t;

typedef struct _abc_t {
   def_t *def;
   xyz_t *xyz;
   short ndef;
} abc_t;
```



#9 Poor Software Design Diagrams

Architectural decomposition:
at least one diagram per level of decomposition

Detailed design:
at least one diagram per function or module

Process-flow

Data-flow

Finite-state machines

Dependency graphs

Data relationships

Sequence

#9 Poor Software Design Diagrams

How do we create good diagrams?

Create a legend for every diagram.

Every block, symbol, line, shading, color, and font type should be specified in legend.

Any deviation from legend shows an error in the design.

#8 “It’s just a glitch”

Note problem in your log book immediately!

Never assume that a problem has been fixed magically

Spend some time to try and fix the problem

#8 "It's just a glitch"

What are the most likely causes?

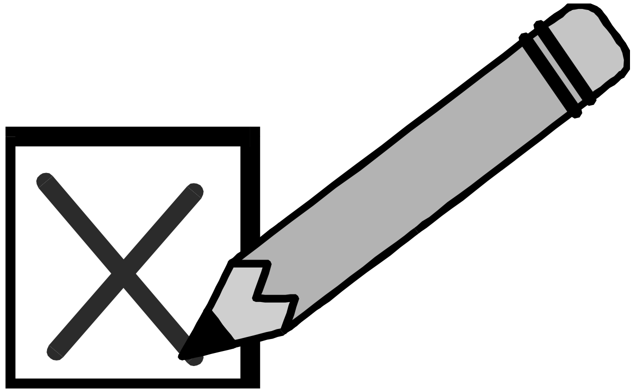
- ***Timing Error (race condition, priority inversion)***
- ***Memory Corruption***
- ***Deadlock***

#8 "It's just a glitch"

How do we pinpoint the problem?

(1) During design phase, take precautions:
- ***Formal code review***
- ***Minimize shared resources and memory***
- ***Minimize use of interrupts***
- ***Use deadlock-free IPC solutions***

(2) During testing and maintenance phases:
- ***Put sleep() commands within critical sections***
- ***Check for stack corruption***
- ***Incrementally add debug statements***
- ***Monitor progress on logic analyzer***



#7

The first right answer is the only answer

Every problem has at least 3 answers:
- ***The first answer***
- ***The opposite answer***
- ***A compromise between the first two answers***

Which is the best answer?

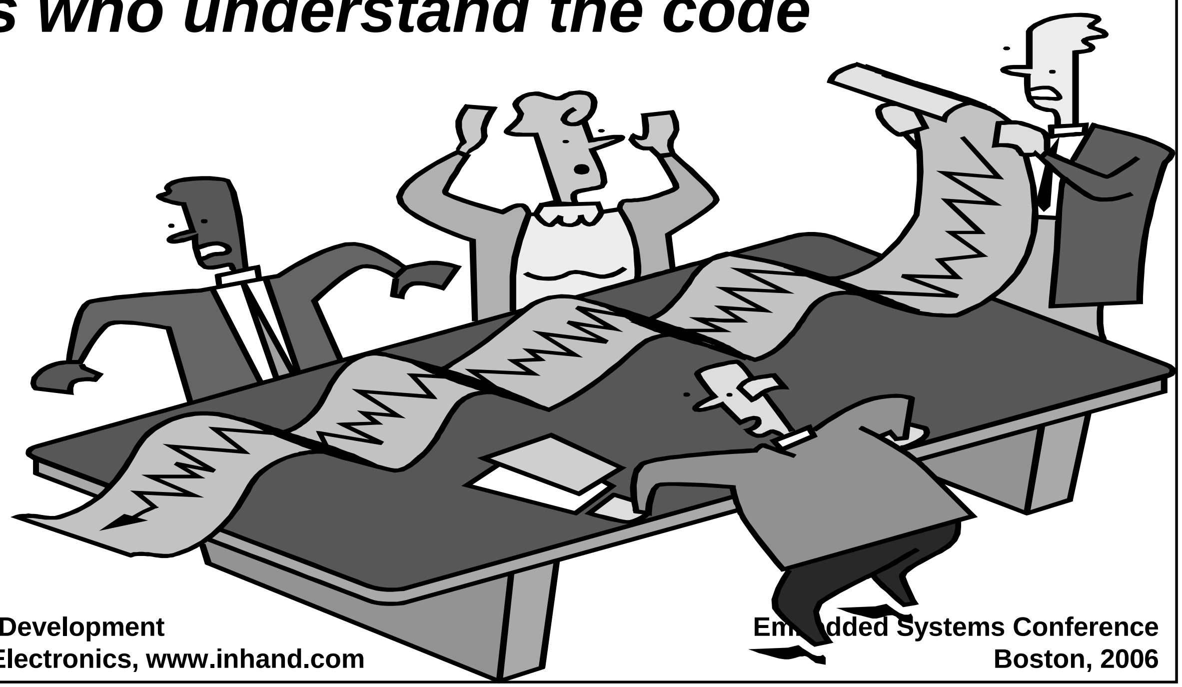
Learn to be more creative to find the other answers.

Code reviews are a proven way to improve quality and robustness

Studies have shown that more problems can get fixed in one day of code review than in a month of debugging

Reviews help eliminate messy code by forcing programmers to show their code to others

Reviews double as training sessions to increase number of employees who understand the code



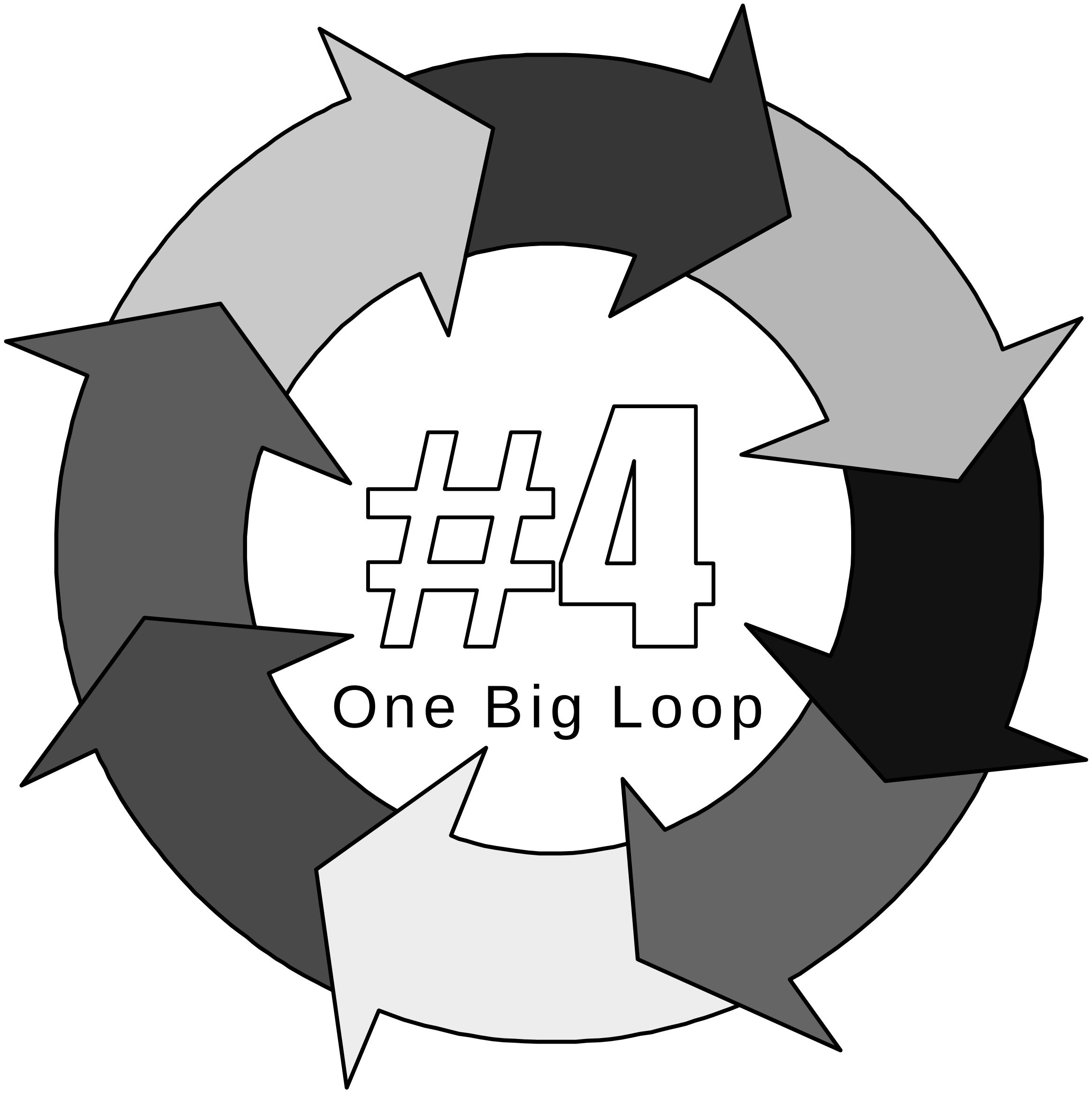
#6

No code reviews

"Nobody else here can help me" syndrome

#5

Learn by teaching others!



#4 One Big Loop

Use proper concurrent design techniques:

Non-preemptive: cyclic or multi-rate executive

Preemptive: real-time operating system

#4

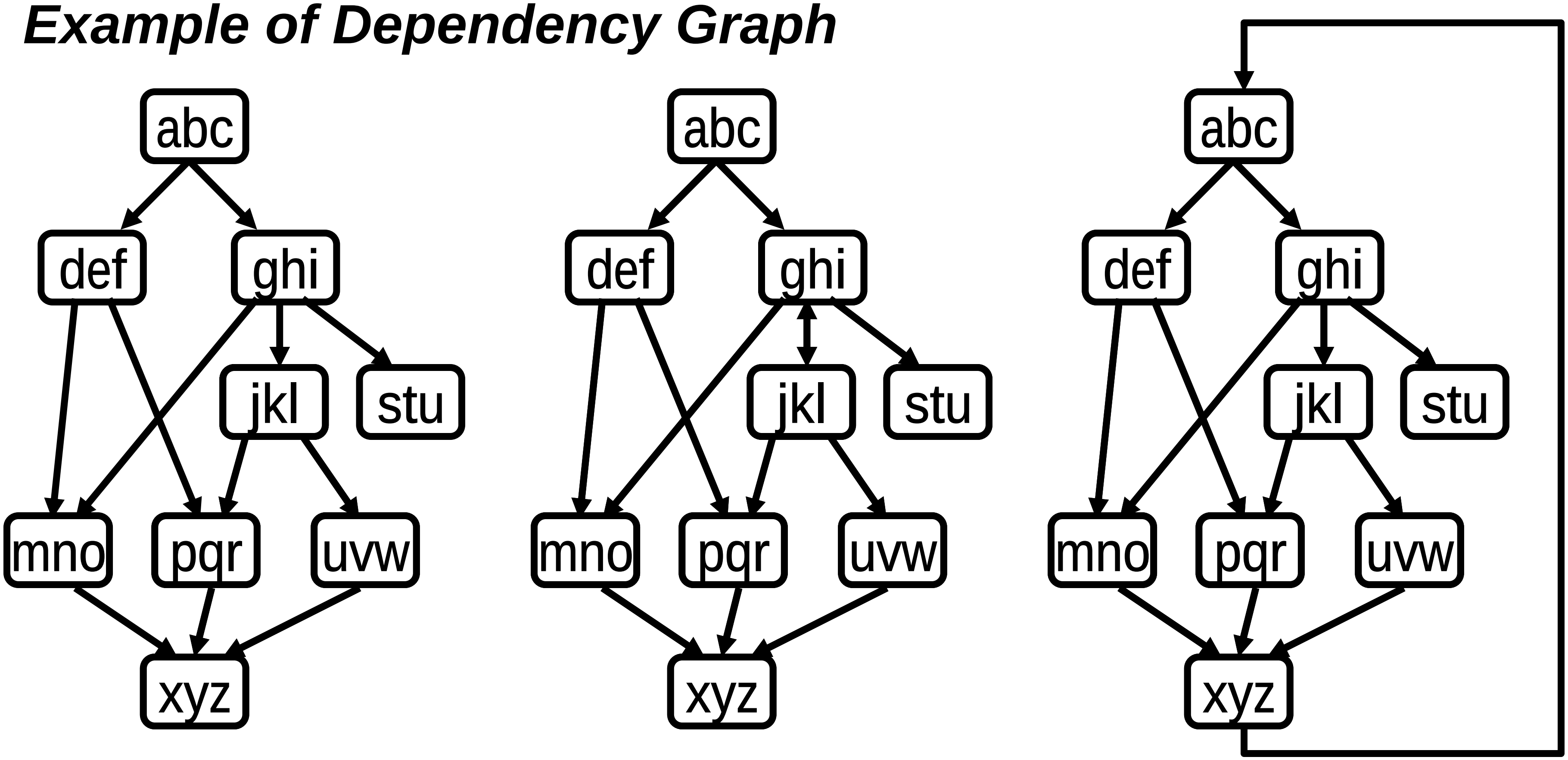
One Big Loop

Don't use interrupts to emulate multitasking

#3 Too many inter-module dependencies

#3 Too many inter-module dependencies

Example of Dependency Graph

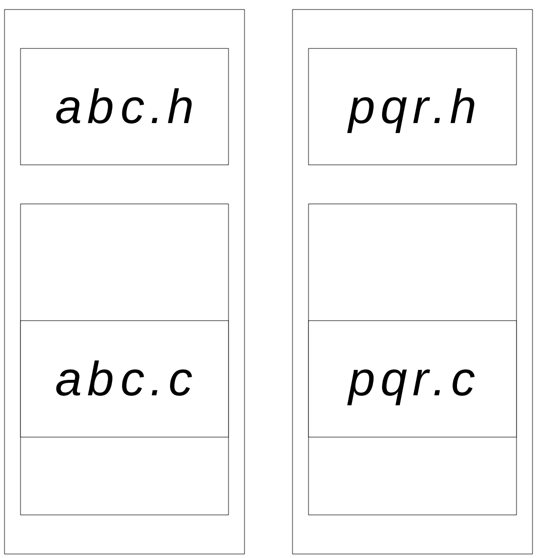


Minimize Circular Dependencies!

#3 Too many inter-module dependencies

#include “globals.h” problem

Follow fundamental Software Engineering concepts, especially:
· Data encapsulation and modularity
· Use abstract data types or objects



Hints for maximizing modularity:

Put code for module abc in file abc.c.

Only put definitions of anything exported from abc.c into file abc.h

#include only the .h files you need.

#2 No naming and style conventions!

Establish a set of conventions, and stick to them!

Use the conventions to help reader to quickly identify the origin and purpose of the symbol.

#1

No measurements of execution time!

#1

First, design your system so that the code is measurable!

Measure execution time as part of your standard testing. Do not only test the functionality of the code!

Learn both coarse-grain and fine-grain techniques to measure execution time.

Use coarse-grain measurements for analyzing real-time properties

Use fine-grain measurements for optimizing and fine-tuning

No measurements of execution time!

Top 25

Most Common Mistakes with Real-Time Software Development

Summary

Correcting just ONE mistake can save thousands of dollars or significantly improve quality and robustness of software.

Correcting SEVERAL mistakes can lead to savings and improvements that are incalculable!

Top 25

Most Common Mistakes with Real-Time Software Development

Dave Stewart
Director of Software Engineering
InHand Electronics
Rockville, Maryland
dstewart@inhand.com
http://www.inhand.com