Lecture #10

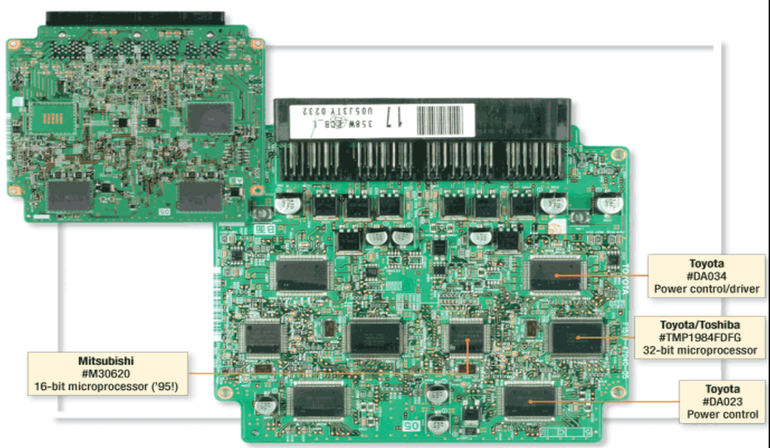
Test & Debug

18-348 Embedded System Engineering
Philip Koopman
Monday, 15-Feb-2016

Electrical & Computer ENGINEERING

Carnegie Mellon

Toyota Prius Skid Control Module



- **Features:**
  - Prevents wheel slip on ice or wet roads [Carey07]
  - Anti-lock braking – reduces braking power to prevent lock-up (safety critical?)
  - Wheel-by-wheel braking to prevent wheel slip in turns
  - 32-bit CPU (probably for numerics); (only Japanese spec. sheets on web)
  - 16-bit CPU (probably for state charts; etc.); 10KB RAM; 64-128KB ROM

Is Skid Control Safety Critical?

- **In July 1999, General Motors has to recall 3.5 million vehicles because of an anti-lock braking software defect.**
  - Stopping distances were extended by 15- 20 meters. Federal investigators received reports of 2,111 crashes and 293 injuries.
  - http://autopedia.com/html/Recall_GM072199.html
- **If we assume it costs $100 per recall**
  - » ($100 is on the low side, and assumes pure software update fix, no HW)
  - This is $350 Million for a recall
  - Pentium FDIV bug was about $475M in 1994
    - So, more or less the same economic loss
    - Which one got more exposure on the news?

Preview

- **Debugging tools**
  - How single-step debuggers work
  - Other debugging tools
- **Debugging thoughts**
  - Common bugs
  - Common ways to look for bugs
- **Testing**
  - Types of testing
  - Coverage

Roald Amundsen

First explorer to the South Pole

"Adventure is just bad planning."

Victory awaits him who has everything in order—luck, people call it. Defeat is certain for him who has neglected to take the necessary precautions in time; this is called bad luck.
—*from The South Pole, by Roald Amundsen*

https://en.wikipedia.org/wiki/Roald_Amundsen

Testing != Debugging

- **Two related goals:**
  - Are there any software defects? (if so, where)
  - We think the software is good to go – are we confident no defects remain?
- **Testing:**
  - Executing a program to see if it performs as expected
- **Debugging:**
  - Given a symptom of software failure, locate and correct the defect
- **Unpleasant truths:**
  - Testing tells you how buggy your software is, ***NOT*** where ***ALL*** the bugs are
  - Taking bug-ridden software and removing bugs doesn't make it good software
    - Number of bugs remaining is can be proportional to number found
    - That means …. **Less Buggy is not Bug-Free!**
  - That being said, it's still useful know how to remove bugs

Debugging Tools: Simulators

- **If you can, the most flexible debugging is use a simulator**
  - Simulate execution of processor
  - Allows total flexibility for debugging
- **BUT, some serious limitations**
  - Cycle accurate simulators can be very complex and slow to execute
    - Especially for high-end CPUs
  - Have to simulate I/O and real-world hardware
  - Can't control hardware at real-world speeds
- **In practice simulators are OK for small bits of code**
  - But aren't very helpful for system integration debugging
  - So … need strategies that use real hardware for "emulation"
    - Simulation: software A pretends to be system hardware B
    - Emulation: hardware A pretends to be system hardware B

But, This Is Real Time Software

- **Some things can only be debugged at full speed**
  - What if you're controlling real mechanical devices? Can't single step!
- **Sometimes just need to passively monitor**
  - How can we tell what happened over 100 million instructions?
    - Single stepping with manual observation isn't realistic
  - Real time trace:
    keep record of program counter values to see how flow went through branches
    - Use software tools to look through trace to find anomalies
- **Sometimes need to run at full speed until a problem occurs**
  - Can just run until some condition is satisfied … but how do we specify that?
  - Breakpoints – condition is executing at a specific address
  - Watchpoints – condition is accessing a specific memory address for data

SWI — Software Interrupt — SWI

Operation:

$(SP) - \$0002 \Rightarrow SP; RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) - \$0002 \Rightarrow SP; Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) - \$0002 \Rightarrow SP; X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) - \$0002 \Rightarrow SP; B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) - \$0001 \Rightarrow SP; CCR \Rightarrow (M_{(SP)})$
$1 \Rightarrow I$
$(SWI\ Vector) \Rightarrow PC$

Subroutine call plus automatic push of D, X, Y, CCR

Jump to a predefined address

Description:

Causes an interrupt without an external interrupt service request. Uses the address of the next instruction after SWI as a return address. Stacks the return address, index registers Y and X, accumulators B and A, and the CCR, decrementing the SP before each item is stacked. The I mask bit is then set, the PC is loaded with the SWI vector, and instruction execution resumes at that location. SWI is not affected by the I mask bit. Refer to Chapter 7 Exception Processing for more information.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	1	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
SWI	INH	3F	VSPSSPSsP[1]	VSPSSPSsP[1]

1. The CPU also uses the SWI processing sequence for hardware interrupts and unimplemented opcode traps. A variation

How Breakpoint Debugging Works

- **Simple version (all in RAM)**
  - Debugger inserts SWI at a “break point” and saves byte that was there
  - Program runs until it hits the SWI
  - SWI causes a subroutine call to the debugger
  - Debugger restores byte to original value
  - When debugging done, subtract one from return address, clean stack, restart program

Address	Program in RAM Memory	Address	Program with Breakpoint	Stack After SWI	
$40F5	ADC	$40F5	ADC	RetLo	$CF0
$40F6	#$42	$40F6	#$42	RetHi	$CEF
$40F7	BCC	$40F7	**SWI**	Ylo	$CEE
$40F8	*+$8	$40F8	*+$8	Yhi	$CED
$40F9	DEX	$40F9	DEX	Xlo	$CEC
$40FA	…	$40FA	…	Xhi	$CEB
				A	$CEA
				B	$CE9
			SP→	CCR	$CE8

BreakpointSave: $40F7 | BCC

Implementing Breakpoint Debugging

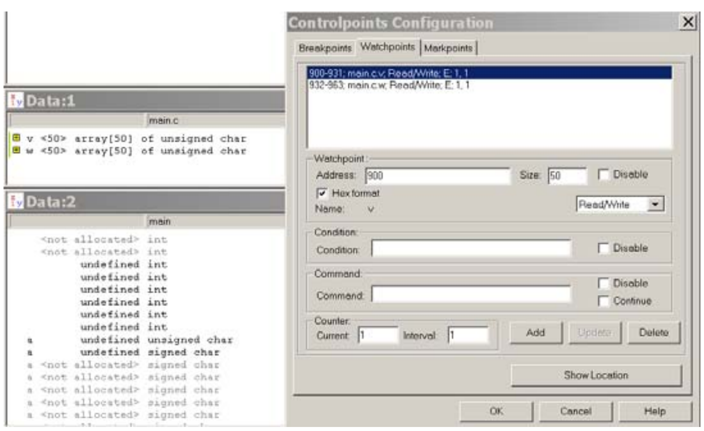
- **What if program is in flash memory?**
  - Could re-flash sector each time SWI is added or removed
    - Wear-out probably isn't an issue since it is a manual process
  - Could have a hardware address comparator that fakes an SWI

- **Software support on top for:**
  - Examining register values, memory, etc.
  - Single step (put breakpoint after next instruction)
  - Set breakpoint(s) (put SWIs at multiple user-defined locations)
  - Run to breakpoint
  - Step over (put breakpoint after the next JSR instruction)
  - etc.

- **You've been doing this in the lab all along…**
  - BDM (Background Debug Module) – serial network & HW support for debug
  - DBG (Debug Module) – address comparators (see watchpoints later)
    - Option of either faking an SWI or entering BDM on address match

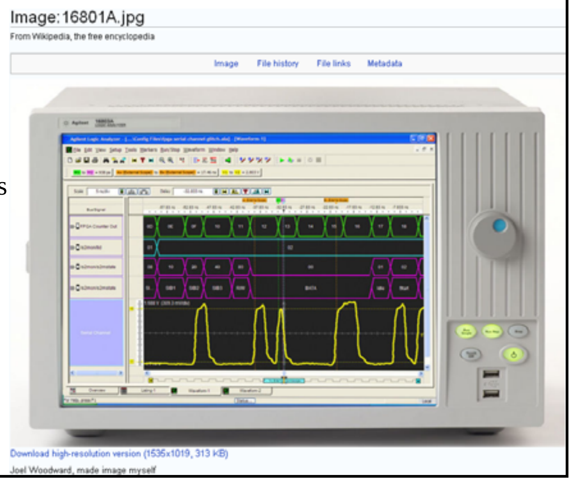
Watchpoints

- Stop program any time a memory location is referenced
  - Or, some other condition (e.g., stop if variable zz == 73)
- In a simulator it's easy (but slow) … in real hardware it's more complicated



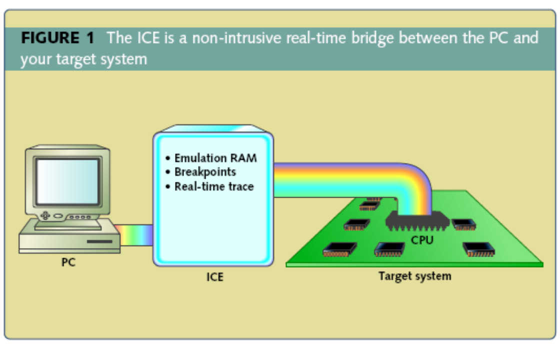
Classical Tool – Logic Analyzer

- **Put probes on each address and data bus pin**
  - Record logic values in real time – lots and lots of probes connect to many pins
  - Can capture real-time traces
  - For fancy devices, even include disassembly capability for instruction fetches
- **Problem – what if address and data don't show up on external pins?**
  - Problem for cached CPUs
  - Problem for microcontrollers with reduced pinouts (like ours)
  - Logic analyzers can be very expensive
  - But, if you can afford one and use one – they are pretty cool!



Classical Tool: ICE – In Circuit Emulator

- **Idea: provide complete chip emulation**
  - Hardware plugs into CPU/MCU socket in real hardware
  - Can run at full CPU speed
  - Provides complete access to CPU internals (e.g., can single-step)
  - PC has extra hardware support for watchpoints, real-time trace, etc
  - Less common than they used to be, in part because of on-chip debug support



[Gannsle01]

On-Chip Debugging Support (On-Chip HW)

- **Course CPU can't use pure software (SWI) approach**
  - If nothing else, wears out flash memory by rewriting SWI instructions!
- **Course CPU provides on-chip support similar to ICE**
  - Without the external hardware!
- **More info in the processor data sheet:**
  - "BDM" Background Debug Module
    - See Chapter 6 of processor data sheet
    - Allows reading and writing CPU memory & registers
  - Debug Module (DBGV1)
    - See Chapter 7 of processor data sheet
    - Hardware support for flexible breakpoints

BDM – implements breakpoint features

- **Serial interface (to save pins) to allow access to hardware**

Table 6-6. Firmware Commands

[Freescale]

Command[1]	Opcode (hex)	Data	Description
READ_NEXT	62	16-bit data out	Increment X by 2 (X = X + 2), then read word X points to.
READ_PC	63	16-bit data out	Read program counter.
READ_D	64	16-bit data out	Read D accumulator.
READ_X	65	16-bit data out	Read X index register.
READ_Y	66	16-bit data out	Read Y index register.
READ_SP	67	16-bit data out	Read stack pointer.
WRITE_NEXT	42	16-bit data in	Increment X by 2 (X = X + 2), then write word to location pointed to by X.
WRITE_PC	43	16-bit data in	Write program counter.
WRITE_D	44	16-bit data in	Write D accumulator.
WRITE_X	45	16-bit data in	Write X index register.
WRITE_Y	46	16-bit data in	Write Y index register.
WRITE_SP	47	16-bit data in	Write stack pointer.
GO	08	None	Go to user program. If enabled, ACK will occur when leaving active background mode.
GO_UNTIL[2]	0C	None	Go to user program. If enabled, ACK will occur upon returning to active background mode.
TRACE1	10	None	Execute one user instruction then return to active BDM. If enabled, ACK will occur upon returning to active background mode.
TAGGO	18	None	Enable tagging and go to user program. There is no ACK pulse related to this command.

1. If enabled, ACK will occur when data is ready for transmission for all BDM READ commands and will occur after the write is

Debug Module – On-Chip Hardware Support

- **Implements watchpoint; tracing; profiling**

[Freescale]

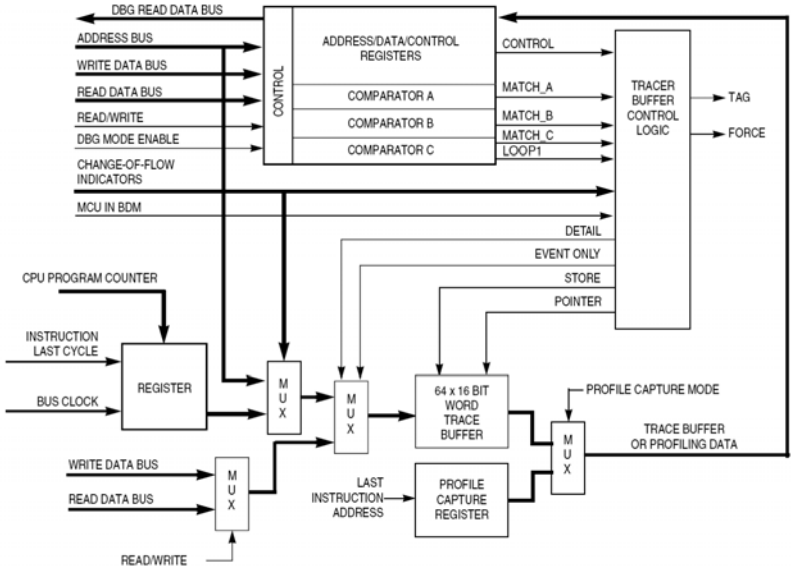
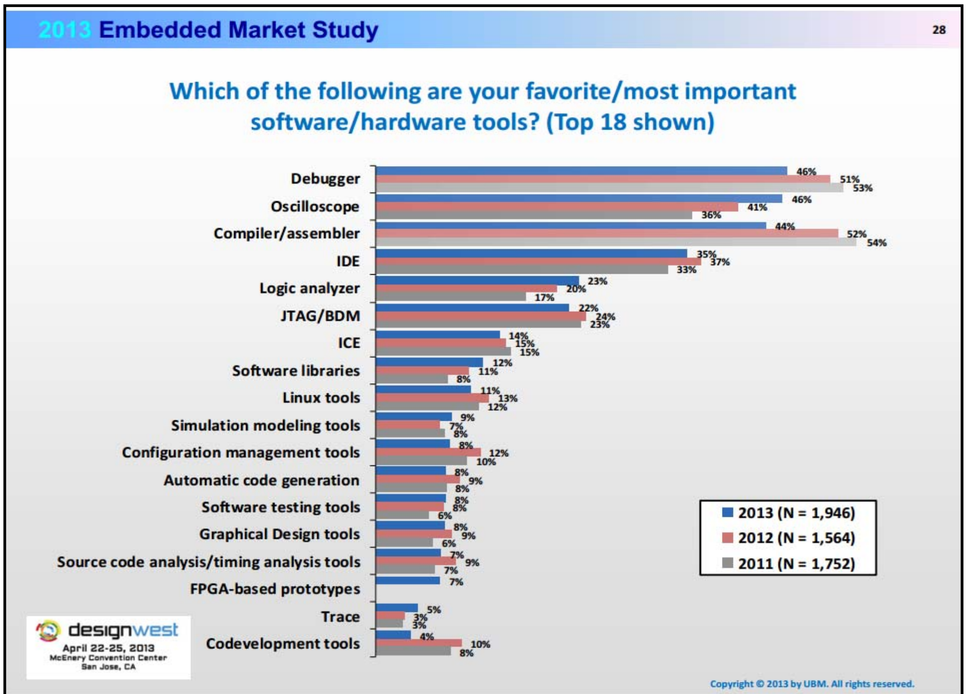
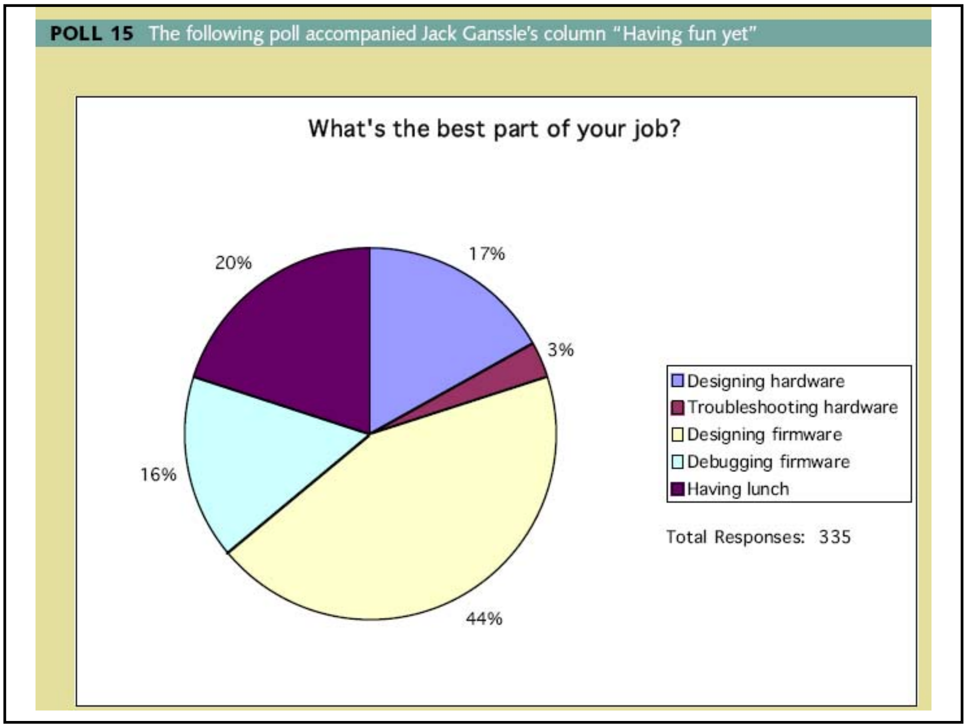


Figure 7-2. DBG Block Diagram in DBG Mode





Common Bug – Memory Leak

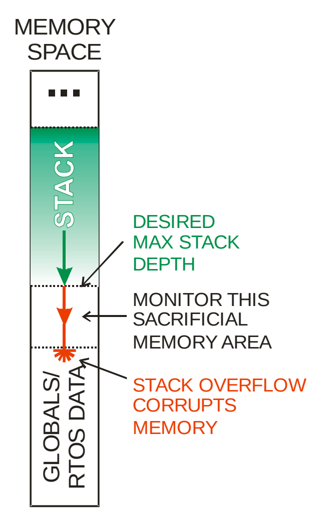
- **In C, what happens if you malloc() but don't free… and put that inside an infinite loop?**
  - You eventually run out of memory!
- **Mitigation:**
  - Outlaw malloc
    - This is common in high-criticality embedded systems
    - Alternative: only allow malloc during system init; then turn it off in main loop
  - Code review for malloc/free pairs
  - Instrument free memory and see if it goes down over time

Common Bug – Wild Memory Pointer

- **Null pointers**
  - Null pointers are sometimes used as a "fault" flag
- **Invalid pointers**
  - Usually due to going passed defined limits of array
  - (similarly, array index too large or negative)
- **Mitigation:**
  - Put these on design review checklists
  - Including out-of-bounds and null checks (speed penalty)
  - Checking only in "debug" compiles ("#ifdef DEBUG …. check … #endif")
  - Put a watch point on address location zero to catch null pointer accesses
  - Use memory bug finding tools (e.g., Purify; Bounds Checker)

Common Bug – Stack Overflow

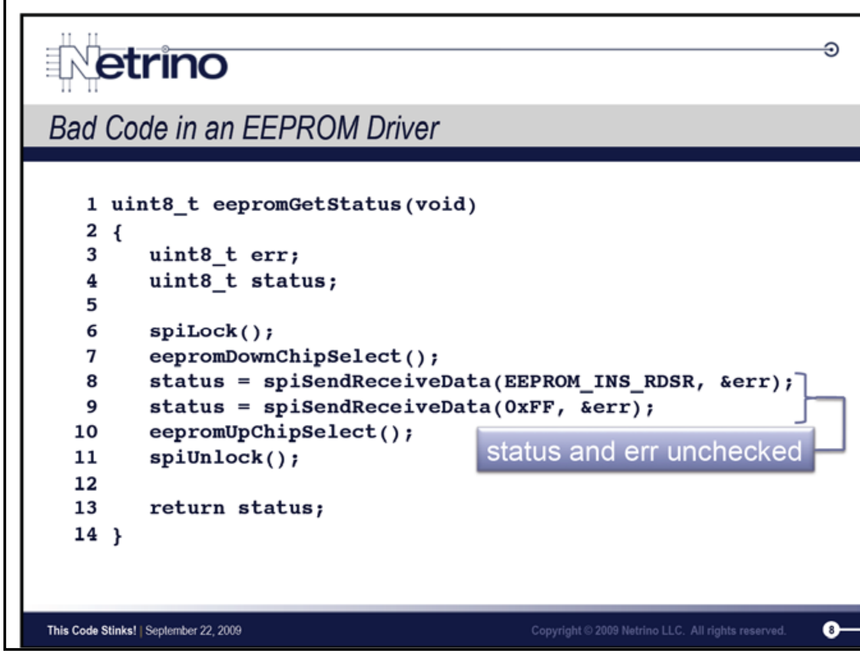
- **Stack grows bigger than available RAM**
  - Especially a problem on microcontrollers with fixed stack size (e.g., 32 deep stack)
- **Mitigation:**
  - Outlaw recursion
  - Use a static stack analysis tool (will tell you maximum stack depth)
    - e.g., "stacktool"
  - Use dynamic stack high watermark technique
    - Initialize stack memory to known value (e.g., $AA)
    - Run program
    - See how many bytes were changed to not be $AA
    - BUT, only gives stack depth of the test run, not necessarily worst case!



Common Bug – Timing Problems

- **Basic problem – program works sometimes**
  - When it works depends on what is going on
  - Usually problems due to multiple competing tasks
- **We'll cover these in concurrency lectures**

Common Bug – Fail To Check Return Codes



Bad Code in an EEPROM Driver

```
uint8_t eepromGetStatus(void)
{
    uint8_t err;
    uint8_t status;

    spiLock();
    eepromDownChipSelect();
    status = spiSendReceiveData(EEPROM_INS_RDSR, &err);
    status = spiSendReceiveData(0xFF, &err);
    eepromUpChipSelect();
    spiUnlock();

    return status;
}
```

If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would **destroy civilization**

– Gerald Weinberg

Debugging is like **alien abduction**.
Large amounts of time disappear, for which you have no explanation.

– Unknown

Always code as if the guy who ends up maintaining your code will be a **violent psychopath** who knows where you live.

– Damian Conway

Hofstadter's Law:
It always takes **longer than you expect**, even when you take into account Hofstadter's Law.

Some Basic Debugging Principles

- **Look for stuff that is commonly wrong … even if you don't see bugs!**
  - Many bugs are hidden in normal use, but will bite you later
  - Especially memory and timing problems!
- **Rely on your brains more than fancy tools**
  - It's easier to play with a debugger than step back and think
  - But thinking is more likely to find the problem!
- **Concentrate on isolating and reproducing problems**
  - Localize the problem; figure out how to make it manifest repeatably
  - Divide & conquer as an approach
- **Remember where you've been**
  - Keep track of what you've tried – use a written log
  - Don't be in a hurry – you'll just have to do everything twice
  - If you modify software for debugging, save the unmodified version first
- **Try to get it right to begin with**
  - Design reviews are more cost-effective than debugging!
  - There is never enough time to do reviews, but there's always time to write more bugs

Button & LED Debug Interface

- **"printf" is often missing from embedded systems**
  - 8 LEDs and 8 Switches is a surprisingly effective debug alternative
  - (That's why you've been using it up until now!)
- **LED ideas:**
  - Monitor memory locations (periodically put memory location to LED)
  - Monitor which branches a test has exercised
  - Count number of iterations
  - If you have a hardware timer, display time taken to execute some code
- **Switch ideas:**
  - Force different paths down a piece of code for debugging
  - Run to a certain point (initialize things), then stop and wait for switch to proceed
  - Select what to show on the LEDs (e.g., which byte of a multi-byte word)

Spare Pin & O-scope

- **Oscilloscope – one or two probes monitoring analog values at high speed**
  - Generally used to look at waveforms
  - Can be helpful looking at noise
  - Newer o-scopes are often A/D converters on a PC platform for display/analysis
- **Also helpful for debugging**
  - Toggle a digital output bit every time a loop is run to look at timing variation
  - (for example – helps see how close to 100% capacity the CPU is running)

[Wikipedia]

Top 10 Worst Ways To Find Out About A Bug

10. Your module fails unit test
9. The subsystem with your module fails independent testing
8. The system fails system integration test
7. The system fails customer acceptance test
6. You get a field problem report
5. Customers wake you up at 2 AM screaming at you
4. You get an airplane ticket to a war zone to reprogram flash memories
3. You hear about the bug from CNN.COM
2. Your corporate lawyers tell you about the lawsuit filed by the widows

And, the Number One Worst Way To Find A Bug:

1. The reporters camped outside your house ask you to comment on it

Types Of Testing – Outline For Following Slides

- **Ad hoc tests – frequently used, but not very thorough**
  - Smoke testing
  - Exploratory testing
- **How can we be more methodical about testing?**
  - Black box testing
  - White box testing

Smoke Testing

- **Quick test to see if software is operational**
  - Idea comes from hardware realm – turn power on and see if smoke pours out
  - Generally simple and easy to administer
  - Makes no attempt or claim of completeness

  - Smoke test for car: turn on ignition and check:
    - Engine idles without stalling
    - Can put into forward gear and move 5 feet, then brake to a stop
    - Wheels turn left and right while stopped
- **Good for catching catastrophic errors**
  - Especially after a new build or major change
  - Exercises any built-in internal diagnosis mechanisms
- **But, not usually a thorough test**
  - More a check that many software components are "alive"

Exploratory Testing

- **A person exercises the system, looking for unexpected results**
  - Might nor might not be using documented system behavior as a guide
  - Is especially looking for "strange" behaviors that are not specifically required nor prohibited by the requirements
- **Advantages**
  - An experienced, thoughtful tester can find many defects this way
  - Often, the defects found are ones that would have been missed by more rigid testing methods
- **Disadvantages**
  - Usually no documented measurement of coverage
  - Can leave big holes in coverage due to tester bias/blind spots
  - An inexperienced, non-thoughtful tester probably won't find the important bugs

Black Box Testing

- **Tests designed with knowledge of behavior**
  - But without knowledge of implementation
  - Often called “functional” testing
- **Idea is to test what software does, but not how function is implemented**
  - Example: cruise control black box test
    - Test operation at various speeds
    - Test operation at various underspeed/overspeed amounts
    - BUT, no knowledge of whether lookup table or control equation is used
- **Advantages:**
  - Tests the final behavior of the software
  - Can be written independent of software design
    - Less likely to overlook same problems as design
  - Can be used to test different implementations with minimal changes
- **Disadvantages:**
  - Doesn’t necessarily know the boundary cases
    - For example, won’t know to exercise every lookup table entry
  - Can be difficult to cover all portions of software implementation

Examples of Black Box Testing

Assume you want to test a floating point square root function: sqrt(x)

- sqrt(0) = 0 (boundary condition)
- sqrt(1) = 1 (behavior changes between <1 and >1
- sqrt(9) = 3 (test some number greater than 1)
- sqrt(.25) = .5 (test some number less than 1)
- sqrt(-1) => error (test an out of range input)
- sqrt(FLT_MAX) => … (test maximum numeric range)
- sqrt(FLT_EPSILON) => … (test smallest positive number)
- sqrt(NaN) => NaN (test for Not a Number input values)
- Pick random positive numbers and confirm that: sqrt(x) * sqrt(x) = x
- Other types of possible results to monitor:
  - Monitor numerical accuracy/stability for floating point math
  - Check to see if software crashes on some inputs

White Box Testing

- **Tests designed with knowledge of software design**
  - Often called "structural" testing
- **Idea is to exercise software, knowing how it is designed**
  - Example: cruise control white box test
    - Test operation at every point in control loop lookup table
    - Tests that exercise both paths of every conditional branch statement
- **Advantages:**
  - Usually helps getting good coverage (tests are specifically designed for coverage)
  - Good for ensuring boundary cases and special cases get tested
- **Disadvantages:**
  - 100% coverage tests might not be good at assessing functionality for "surprise" behaviors and other testing goals
  - Tests based on design might miss bigger picture system problems
  - Tests need to be changed if implementation/algorithm changes

Examples of White Box Testing

Assume you want to test a floating point square root function: sqrt(x)

- Uses lookup table spaced at every 0.1 between zero and 10
- Uses iterative algorithm at and above value of 10

- Test sqrt(x) for negative numbers (expect error)
- Test sqrt(x) for every value of x in middle of lookup table: 0.05, 0.15, … 9.95
- Test sqrt(x) exactly at every lookup table entry: 0, 0.1, 0.2, … 10.0
- Test sqrt(x) at 10.0 + FLT_EPSILON
- Test sqrt(x) for some numbers that exercise interpolation algorithm

- **Main differences from Black Box Testing:**
  - Tests exploit knowledge of software design & coding details
  - Usually strives for 100% coverage of known properties
    - e.g., lookup table entries
  - Digs deepest at algorithmic discontinuities & branches
    - e.g., lookup table boundaries and center values

Testing Coverage

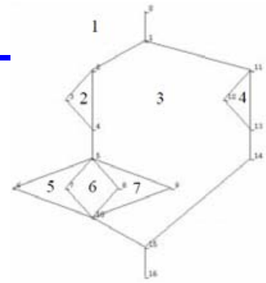
- **"Coverage" is a notion how completely testing has been done**
  - Usually a percentage (e.g., "97% branch coverage")
- **White box testing coverage (this is the usual use of the word "coverage")**
  - Percent of conditional branches where both sides of branch have been tested
  - Percent of lookup table entries used in computations
- **Black box testing can have a related coverage notion**
  - Percent of requirements tested
  - Percent of documented exceptions exercised by tests
  - But, must relate to externally visible behavior or environment, not code structure
- **Important note: 100% coverage is not "100% tested"**
  - Each coverage aspect is narrow; good coverage is necessary, but not sufficient to achieve good testing

MCDC Coverage as White Box Example

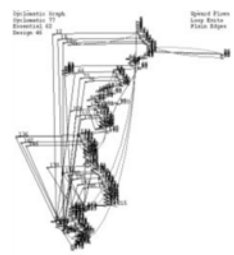
- **Used by DO-178b & DO-178c for critical aviation software testing**
- **Modified Condition/Decision Coverage (MC/DC)**
  - Each entry and exit point is invoked
  - Each decision tries every possible outcome
  - Each condition in a decision takes on every possible outcome
  - Each condition in a decision is shown to independently affect the outcome of the decision
  - For example: if (A == 3 || B == 4) need to test at least
    - A == 3 ; B != 4 (A causes branch, not masked by B)
    - A !=3 ; B == 4 (B causes branch, not masked by A)
    - A !=3 ; B != 4 (Fall-through case AND verifies A==3 and B==4 are in fact responsible for taking the branch)

Cyclomatic Complexity

- **Not all code is created equal:**
  - Some code is complex; some is simple
- **McCabe Cyclomatic Complexity metric**
  - Distinguishes straight-line from heavily branching code
  - Compute number of closed regions in a flow graph + 1
  - High complexity means difficult to test, and suggests difficult to understand
  - Complexity affects number of test paths
- **Code with structural problems is often called <u>"spaghetti code"</u>**
  - <u>Incomprehensible code due to unnecessary coupling, jumps, gotos, or high complexity</u>



Complexity=7



Complexity=77

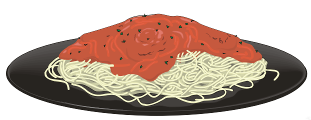
(NIST 500-235, 1996)

As the number of branches in the module or program rises, the cyclomatic complexity metric rises too. Empirically, numbers less than ten imply reasonable structure, numbers higher than 30 are of questionable structure. Very high cyclomatic numbers of more than 50 imply the application cannot be tested, while even higher numbers of more than 75 imply that every change may trigger a "bad fix". This metric is widely used for Quality Assurance and test planning purposes.

(RAC 1996, p.124)

What About Testing Data Values?

- **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
  - How do you test all these interactions? (Generally you ***<u>can't if there are many globals</u>***)



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)

When Do You Test?

- **Unit test**
  - Programmer tests own code
  - Coverage might be: execute each line of code at least once
- **Subsystem test**
  - Someone else tests the modules you wrote
  - Coverage: exercise subsystem functions and module-to-module interfaces (API)
- **System integration test**
  - Testing all the modules from all the programmers as a whole
  - Coverage: test everything in the SW reqts & HW reqts documents
- **Regression test**
  - Run previous tests again to see if a bug fix broke anything else
- **Acceptance test**
  - Customer decides if the system does what it is supposed to do
  - Coverage: test everything in the product requirements
- **Beta test**
  - End users find strange ways to use system you never thought of
  - Coverage: pick representative but demanding users

- **Most testing styles have some role to play in each testing situation**

Discussion – examples of coverage

- **100% branch coverage**
  - White box or black box?
  - How can you know you got it?

- **100% requirements coverage**
  - White box or black box?
  - How can you know you got it?

Exercise: What Test Cases Would You Use?

```
inline Average( int16 A, int16 B)
{  return ( (A+B) / 2);
}
```


Exercise: How Do You Test A Statechart?

- How do you decide what test cases to generate?
- What are good measures of coverage?

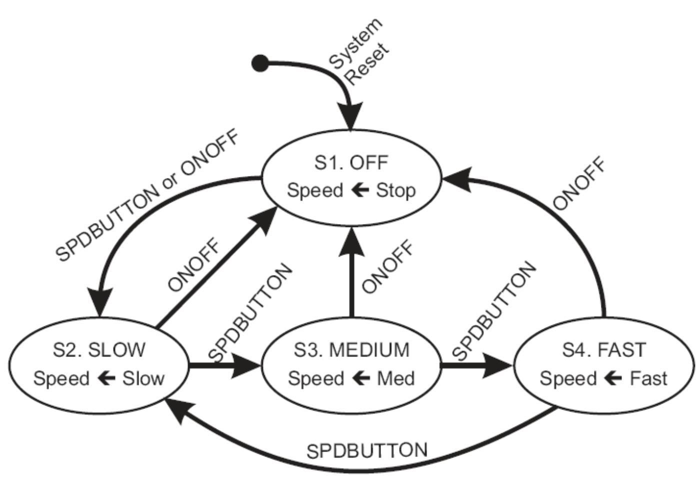


Figure 13.1. An example statechart.

How Many Defects Do We Expect?

- **Typical "good" software has from 5 to 30 defects per KSLOC**
  - The number you get varies, but these are good general ranges
  - Best we know how to do is about 0.1 defects/KSLOC for Space Shuttle
    - At extremely high cost
- **It is difficult to know what to do with this number**
  - Some defects cause bigger problems than others
    - Some will be safety issues; some won't
  - Some defects will occur more frequently than others
    - The "operational profile" of software matters – how the software is used
  - But, if you say you did nothing special and got 2 defects per KSLOC…
    - The most likely explanation is you haven't tested enough to find the other bugs
- **Software "reliability" is not an easy problem**
  - Don't believe anyone who tells you otherwise
- **Thought question: how do you know you got rid of all the bugs?**

Questions To Ask When Testing & Debugging

- What is my coverage goal for this activity?
- Is the power turned on?
- Did you check every wire in your board for correct connectivity?
- Are the configuration register values all correct?
- Did you follow the coding standard?
- Did you use lint or turn compiler warnings on?
- Are your pointers initialized and valid?
- Did you use parentheses defensively?
- Did you avoid global variables? (Global Variables Are Evil)
- Did you skimp on unit testing? (Or are you doing big bang testing?)
- Did you skimp on design reviews?
- Do you have complex code? (Nested conditionals more than 3 deep)
- Did you handle concurrency problems with shared variables?
- Have you looked in places where a bug can't possibly happen?
- Are you being impatient, or methodical?
- **There are two required reading articles for this lecture.**
  READ THEM!
  - "The quickest way to debug a program is to write a program that has no bugs"

Review

- **Difference between testing and debugging**
- **How software debugging tools work**
  - SWI-based debugging
  - What a "watchpoint" does
- **Memory bug types**
  - What's a memory leak?
  - What's a wild pointer?
  - What's a stack overflow?
- **Four types of testing and what they mean:**
  - Smoke test
  - Exploratory Testing
  - White Box Testing
  - Black Box Testing
  - What does testing coverage mean?

Lab Skills

- **Create tests with defined level of coverage**
  - White Box tests
  - Black Box tests