NC STATE UNIVERSITY

Strategies and Methods for Debugging

These lecture notes created by Mr. James B. (Jim) Carlson, NCSU

Debugging

- The best way of debugging is to *avoid creating the bugs*
  - Stop writing big programs, since complexity scales exponentially
  - Follow good coding practices: modular code, OOP, top-down decomposition, code walk-throughs, etc
- Embedded systems allow limited visibility of system state, so develop as much of the system on a more friendly platform
- *"It should be working, but it isn't – it doesn't make sense!"* really means *"One of my assumptions is wrong. I need to find out which it is and why."*

Debugging Concepts – Think and Test

- Determine what works and what does not
  - As system grows more complex, faults become visible later in program, obscuring the sources of error
  - Your brain is an essential part of the debug process, regardless of tools
  - ***Assume everything you have created is broken until you prove it works***
    - Rely on as little of system as possible when debugging
  - ***Blame yourself first and the hardware last***
    - Assume that the chip is doing what you tell it to do

Example Embedded System

- Reflective infrared object detector
  - IR emitter
    - Software: initialization, operation
    - Hardware: IR LED + current-limiting resistor
  - IR sensor
    - Software: initialization, operation
    - Hardware: IR phototransistor + pull-up resistor, LED output

Debugging with Divide and Conquer

- Doesn’t work? Divide and conquer. What *does* work?
- Externally observable signals
  - Does the IR LED go on and off?
    - Oscilloscope/DVM, cellphone camera
  - Does the IR phototransistor voltage fall as IR increases?
    - Oscilloscope/DVM
  - Does the indicator LED go on and off?
    - Oscilloscope/DVM
- Software

Debugging Concepts – Observation and Repeatability

- A processor can execute many millions of instructions in one second, and humans can’t handle time on that scale
  - Set output bits to indicate when specific events occur, allowing scope to trigger on useful info
- Need to make inputs repeatable
  - Repeatability is fundamental for debugging
    - Code must have same control flow and data values
    - Need to be sure changes in system behavior are a result of changes to source code rather than system inputs
  - Embedded systems read I/O devices
    - Must configure devices to behave the same each time
      - e.g. stimulus file in simulator
    - May need to write test functions that emulate input devices

Debugging Concepts - Tools

- Need two tools for embedded real-time systems
  - Debugger: examine functional behavior
    - How many times does the loop execute?
    - Does the program recognize the error condition and execute this branch of the conditional?
    - Examples: gdb, KD30
  - Monitor: examine temporal behavior
    - When does the pulse occur?
    - How long does it take to respond to the interrupt?
    - Examples: oscilloscope, logic analyzer

Functional Debugging 1

- Supply inputs, run system, compare outputs against expected results
- Can add debugging instruments (modify your source code) to supplement/enhance debugger
- Single stepping or trace
  - Can step at source level or assembly level
  - Can step down into called functions or over them, or finish existing ones
- Breakpoints
  - Can halt program execution at a given point in source code
  - *Conditional breakpoints* are aware of program state, reduce false triggers

Trade-Offs

- Single-stepping (SS) vs. Breakpoints (BP)
  - SS gives complete control and visibility into the program's control flow, but may require many, many steps
    - Scales very badly as program increases
  - Fast execution up to BP, but you don't know what code executed before it
- Forward vs. Backward Search
  - Forward: Find point in program where state is good, then work forward until bad data/behavior is found
    - Need to be methodical and keep track of location
  - Backward: Find point in program where state is bad, then rerun to earlier points
    - The original bug's nature may be masked by code which follows it, complicating debugging
    - Garbage In, Garbage Out: just because this function's output is bad doesn't mean the function has a bug!
    - One bug may trigger other bugs, so you may end up tracking multiple bugs to fix one
- Forward search is much more efficient

How Do We Know If The Program State Is Good?

- Motivation
  - The sooner we find a bug, the sooner we can fix it
  - The sooner we know a bug has executed, the sooner we can find it.
- Helps to have functions which check the program state to see if it is good or bad
  - Simple for basic data types
  - More sophisticated data structures should have a check function
  - Can conditionally compile the check code, leaving it out of production (release) code for speed
  - Might still want to leave it in to get more detailed bug reports later

Common Symptoms of Bugs

- ISR
  - never runs
  - never returns
- Subroutine
  - never runs
  - never returns
  - returns wrong value
- Variable has wrong value
- Uncontrolled execution
  - processor resets
  - processor hangs

Common Bugs

- Misuse of C – Read the C manual, or Patt & Patel
- Missing header file, so function name is undefined
- ISR
  - vector not initialized
  - interrupt controller not enabled properly
  - not declared as interrupt
- Peripherals
  - misconfiguration
  - misunderstanding of operation
- Variable corruption
  - out-of-bounds array access
  - stack under/overflow
  - casting needed
  - signed/unsigned problem
  - invalid pointer
- Infinite loop

Debugging instrument

- Code added to program for debugging
  - Print statement, output bit twiddling
  - Can also enhance power of existing debugger
- How are instruments enabled or disabled?
  - Dynamically: instruments check global flag before executing

    ```
    if (debug) p3_5 = 1;
    ```

    - Run-time overhead always incurred
  - Statically: use conditional compilation/assembly

    ```
    #define DEBUG_ENABLE 1
    #if DEBUG_ENABLE
    #define DEBUG_OUT(a,b) {a=b;}
    #else
    #define DEBUG_OUT(a,b)
    #endif
    DEBUG_OUT(p3,val)
    ```

    - Run-time overhead incurred only when compiled in
- Monitoring with software affects system behavior

Functional Debugging 2

- Conditional breakpoints
  - Can halt program execution as above, when certain logical conditions are true
    - debugger: cond 1 (buffer.length > 33)
    - instrument:

      if (buffer.length > 33)

      filler statement with breakpoint
  - Filters out many breaks we aren't interested in
  - Can also ignore first N instances of breakpoint
    - debugger: ignore 1 33
    - instrument:

      if (++times_bkpt_hit > 33)

      filler statement with breakpoint

Functional Debugging 3

- Print statements
  - Need to get information out of embedded systems, which typically have limited visibility
  - use printf or similar function
    - may not have a video display or serial port
    - time delay of printf
      - slows down rest of system
      - can’t practically be coordinated with observing an event on a scope
    - printf requires large amounts of code memory
    - Usually it is easy to add your own output handler

Functional Debugging 4

- Dump into a local buffer
  - Store data into a buffer for later examination
    - Can store data values (e.g. ADC, stack pointer, UART Rx buffer size)
    - Can also use event codes (e.g. over-temperature condition, UART Rx buffer overflow)
  - Later examine or dump data values with instrumentation code or debugger
    - Can use an array (simple) or circular buffer (more flexible, allows last N events to be tracked)

- Use a fast monitoring device (e.g. alphanumeric LCD, LEDs)
  - Limited amount of information can be displayed (e.g. eight LEDs)
  - LCD controller interface may be relatively slow, raising CPU load