Debugging Embedded Applications

Klaus-Peter Zauner

COMP2215: Computer Systems II
Debugging

- Compile-time Errors
  - syntax and semantic errors
    → same strategies as for host programming

- Run-time Errors

- Fix the Problem

- Test → may involve hardware

- Expand Test Suit → may involve hardware (test
Debugging

- **Compile-time Errors**
  - syntax and semantic errors
    - same strategies as for host programming

- **Run-time Errors**

- **Fix the Problem**

- **Test**
  - may involve hardware

- **Expand Test Suit**
  - may involve hardware (test
Typical Run-time Errors

- Errors of Intent

- Boundary Violations
  - Counter overflow
  - Buffer overflow
  - Wrong pointer (e.g., uninitialised)

- Unanticipated Program Flow
  - Assumed atomicity, races . . .

- Incomplete State Preservation
  - e.g., non-reentrant calls from ISR

```plaintext
if (i=1); {}
```
Yes, you’ll find those things in products...
Yes, you’ll find those things in products...
Yes, you’ll find those things in products. . .

The 787 has six electrical generators. Two 250-kilovolt-ampere units are mounted on each of the two engines, and two 225-kilovolt-ampere units are used as backup generators. The generators provide power for a variety of functions on the aircraft, including running the planes avionics, pressurizing the cabin and de-icing wing parts.
Each generator is linked to a control unit. Boeing found that if the four engine generators were left on continuously for about eight months, a software internal counter would overflow and cause the control units to enter a fail-safe mode. The F.A.A. warned that this could result in a loss of all electrical power, regardless of whether the plane was in flight.

⇒ F.A.A.: reboot plane every 120 days
When is debugging hard? I

- New Hardware
- Cause and effect distant in space and/or time
  - Impact several millions of instructions later
  - Symptoms give no hint of root cause
When is debugging hard? II

- Programmer has wrong model
  - misinterpreted macro expansion
  - code optimization may lead to very different code than what the programmer uses for reasoning (removal/reordering of instructions)
  - wrong assumption about architecture (e.g. memory alignment)
When is debugging hard? III

- Environmental Effects
- Instrumentation alters the scene
  - bug does not appear in debugger
  - tools may be impractical
    (e.g., not enough memory)
When is debugging hard? III

- **Stealth Bugs**
  - evidence removed by the bug

- **Concurrent, event driven, real-time**
  - timing related, rare bugs
  - start/stop debugging (breakpoints) not suitable
  - often multiple microcontrollers interacting in a system
General Approach to Debugging

Debugging is not an Art, it’s (experimental) Science!
Strategies

- **Gather Data**
  - find how to make it reproducible
  - record symptoms (trace)

- **Isolate**
  - Binary and heuristic search in code space-time and run-time state space

- **Confirm your assumption**
  - incl. hardware and signals

- **Controlled experiments** → find root cause
Device Under Test

User Interface

Device Under Test (DUT)

Seriell Port

Instrumentation

GPIO Pins
Gather Data: Access I

- **Software Instrumentation**
  - Difficult because of resource constraints (memory, speed)

- **Hardware Instrumentation**
  - Easier in embedded systems, but free pins are rare
Gather Data: Access II

- **Re-Purpose pins**
  - Bus lines while peripheral not enabled/selected
  - EEPROM write line without applying write-voltage
  - Input switches, SD card detection switch, etc...

- **Use exiting LED(s)**
  - Slow patterns for humans
  - Fast flicker for communication
    (requires (simple) reading hardware, or mobile phone camera)
Gather Data: Timing

- Switch pins on function entry/exit and show on oscilloscope
- Very low cost: Listen on pins through a headphone
Make Bug Reproducible

- What is the fault condition?
- What discriminates it from normal operation?
  - Use as trigger for data gathering
  - Use as stop for trace ring-buffer
Debugging is a Science

- Design debugging in
- Make targeted minimal changes
- Keep a good record of the change and the effect
  - versioning system (branches/tags)
    - possibly include log output / timing profiles
  - log book
- Construct Hypotheses
Tools

- Simulators
- In-circuit emulators
- JTAG/BDM debuggers
- Digital (storage) oscilloscopes
  - possibly with DAC
- Logic analyzers
Where can things go wrong?

- Hardware
- Source Code
- Fuse Settings
- Compiler Options
- Persistent parameters (EEPROM or Flash)
Where can things go wrong?

- Hardware
- Source Code
- Fuse Settings
- Compiler Options
- Persistent parameters (EEPROM or Flash)
Effects?

- Fuse Settings
- Compiler Options
  - Clock Speed setting
  - Optimisation options
Effects?

- **Fuse Settings**

- **Compiler Options**
  - Clock Speed setting
  - Optimisation options
Effects?

- Fuse Settings
- Compiler Options
  - Clock Speed setting
  - Optimisation options
Effects?

- Fuse Settings
- Compiler Options
  - Clock Speed setting
  - Optimisation options
Effects?

- Fuse Settings
- Compiler Options
  - Clock Speed setting
  - Optimisation options

⇒ Always document the requirements in the source code.
⇒ Automate and version the build process.
Typical Scenarios

- **Device Dead**
  - Watchdog?
  - Undefined Interrupt Vectors?
  - Error in Initialisation?

- **Periodic Failure**
  - Boundary Violation?
  - Memory Problem?

- **Random Failure**
  - Memory Problem?
  - Timing Problem?
  - Atomicity violation?
Typical Scenarios

- **Device Dead**
  - Watchdog?
  - Undefined Interrupt Vectors?
  - Error in Initialisation?

- **Periodic Failure**
  - Boundary Violation?
  - Memory Problem?

- **Random Failure**
  - Memory Problem?
  - Timing Problem?
  - Atomicity violation?
Memory Problems

- Corruption
- Leaks
- Fragmentation
Memory Leak

- Program allocates small amounts of memory and never frees it up
- Over time system runs out of memory and (if well behaved) resets
- Cycle repeats

Know when a library call allocates memory! Beware that `malloc()` and `free()` are not reentrant!
Memory Leak

- Program allocates small amounts of memory and never frees it up
- Over time system runs out of memory and (if well behaved) resets
- Cycle repeats

Know when a library call allocates memory!
Beware that malloc() and free() are not reentrant!
Memory Fragmentation

- `malloc()` serves consecutive memory
- `free()` calls merge accent blocks

Pay attention to the memory map produced my the linker and watch the available stack space.

Note that you might use `malloc()` indirectly, eg. `printf()` may use it.
Available Stack Space

Determine memory available for dynamic allocation/stack:

```
avr-nm -n myprog.elf
```

Look for the symbol `_end`. It is the first address in RAM that is not allocated by a variable.

The addresses from `_end` to the end of SRAM (see datasheet) is what is available for stack and `malloc()`.

See avr-gcc FAQ.
Memory Corruption

- Dangling Pointer
- Stack touches Heap
- Array index exceeds boundary
- Outdated preserved state
  - E.g. A struct still holds pointer to memory that was freed

No memory protection on microcontrollers: delays detection of memory corruption.
How to Debug Memory Problems?

On Host Computer or large Embedded System:

- **valgrind**’s memory tools

On a Microcontroller, there is not enough RAM for fancy tools... especially not if there is already a memory issue.

- **malloc()** is nice code, you can interface with it

See Code to Observe Fragmentation for an example of looking into the malloc() data structures to track fragmentation.
How to Debug Memory Problems?

On Host Computer or large Embedded System:

- valgrind's memory tools

On a Microcontroller, there is not enough RAM for fancy tools... especially not if there is already a memory issue.

- malloc() is nice code, you can interface with it

See Code to Observe Fragmentation for an example of looking into the malloc() data structures to track fragmentation.
Compiler Support

- **Assert:**
  
  ```c
  #define assert(p) if (p) else ( ACTION )
  ```

- **Predefined labels:**
  
  ```c
  __FILE__, __LINE__
  ```

- **Pre-processor:**
  Conditional compilation
Compiler Support

- **Assert:**
  
  ```c
  #define assert(p) if (p) else ( ACTION )
  ```

- **Predefined labels:**
  
  ```c
  __FILE__, __LINE__
  ```

- **Pre-processor:**

  Conditional compilation

  Carefully consider whether the debug code should stay in the product or not.
Compiler Support

- **Assert:**
  
  ```
  #define assert(p) if (p) else ( ACTION )
  ```

- **Predefined labels:**
  
  `__FILE__`, `__LINE__`

- **Pre-processor:**

  **Conditional compilation**

  Carefully consider whether the debug code should stay in the product or not.

  → this is often a trade-off between reliability and security