Debugging Techniques

Simon Chapter 10

David E. Simon, An Embedded Software Primer
“The only way to ship a product with fewer bugs is to write software with fewer bugs in the first place.”
Testing on Host Machine

- Find bugs early
- Exercise all of the code
- Develop reasonable, repeatable tests
- Leave an “audit trail” of test results
- .....all of this is hard to do on the target system
- Therefore...use host as much as possible
Testing on the Host

Target System

Hardware-independent code

Hardware-dependent code

Hardware

Test System (Host)

Hardware-independent code

Test scaffold code

Keyboard

Display

Disk

Same code
Hardware-Dependent Code

- Create a clean division between the code that directly interfaces with the hardware, and the rest of your code.
- Disconnects hardware to allow SIMULATION of hardware.
- Hides the ugliness of bit manipulations, assertion logic, etc.
- `turn_LED_ON()`
- `void turn_LED_ON(void)`

- One Method – two C files. One contains real hardware functions, the other calls simulated hardware functions.

```c
void turn_LED_ON (void)
{
    PORT1A |= LED_PIN;
}
```

```c
void turn_LED_ON (void)
{
    enable_NUMLOCK_LED();
}
```
Another Method

Always compiling the same C files, but use internal directives to control which code is compiled

```c
#define SIMULATED_HARDWARE TRUE

void turn_LED_ON (void)
{

    #ifdef SIMULATED_HARDWARE
        enable_NUMLOCK_LED();
    #endif

    #ifdef REAL_HARDWARE
        PORT1A |= LED_PIN;
    #endif

}
Development Sequence

- Usually target hardware not available first
- Develop upper level code with driver stubs
  - `turn_LED_ON()` could be keyboard LED, a buzzer, a graphical display…or nothing.
  - Functional hardware replaced by fake code
- Write actual target driver code (sometimes written by different people than higher level code)
- Integrate
Interrupts

- Structure interrupts so that hardware dependent part is encapsulated and calls hardware independent part.
- Then, simulate just the independent part in main().
Scripts and Output Files

- Pre-write a script that the program uses to simulate input to the program.

- Test scaffold “simulates” hardware interrupts - calls hardware-independent code to deal with the input data/condition.

- Test scaffold intercepts outgoing hardware commands and handles handshaking

- Useful for repeated testing

- Useful to create strange timing conditions
Loggers

- Glorified printf
- Log useful information to a file, an output stream (e.g. serial port), etc.
- Wise to avoid ASCII and encode for faster speed and reduced memory requirements
- Can be very valuable as code coverage tool
- Can radically reduce the efficiency and speed of a system
- Can cause latency and priority inversion problems
The assert Macro

- Useful for application programmers – embedded programmers generally have to implement by hand
- assert() takes 1 argument, and on fail, causes program to crash right away.
- Use it to check things you normally assume to be true (assignments, logicals, etc.)
- Allows bugs to be found very early in development cycle

```
assert (address <= ADDR_MAX);
assert (address >= ADDR_MIN);
```
Our own assert

Define what you want to happen on assert failure – get developer’s attention!

- disable INTs and spin in infinite loop
- Flash LEDs or sound buzzers in known pattern
- write values of offending parameters to memory/log

```c
#ifdef NODEBUG
    #define assert(bool_expression)
#else
    #define assert(bool_expression)
    if (bool_expression)
    ;
    else
        bad_assertion(!!(!! do something drastic));
#endif
```
Laboratory Tools

- Volt meters and ohm meters
  - Power?
  - Short circuits?
- Oscilloscopes
  - Monitor signals over time
  - Monitor more than 1 signal
  - Trace buffer allows storing data
  - Triggers
  - Checking clock signals
  - Checking output hardware control lines
  - Glitches and spikes
  - Remember to connect the ground lead!