

---

# Strategies and Methods for Debugging

# 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 written 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

# Debug Process (repeat until bug-free)

---

Observe behavior to describe the bug

Gather other related information to flesh out description of bug

Check common problems, based on experience

Generate hypothesis of possible cause of bug

- May need to gather even more info with test tools

Create experiment to test that hypothesis

- What will this experiment prove?

Run the experiment

# 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

# Serial I/O Queues

```
#define Q_SIZE (32)
```

```
typedef struct {  
    unsigned char Data[Q_SIZE];  
    unsigned int Head; // points to oldest data element  
    unsigned int Tail; // points to next free space  
    unsigned int Size; // quantity of elements in queue  
} Q_T;
```

```
Q_T tx_q, rx_q;
```

What should always be true for this type of data structure?

- $\text{Size} \leq \text{Q\_SIZE}$
- $(\text{Head} - \text{Tail} + \text{Q\_SIZE}) \% \text{Q\_SIZE} == \text{Size}$

# Round-Robin Scheduler Task Table

```
typedef struct {  
    int initialTimerValue;    // “frequency” of task  
    int timer;                // time to next “run”  
    int run;                  // binary - 1 = “run now”  
    void (* task)(void); // address of function  
} task_t;
```

```
task_t GBL_task_list[MAX_TASKS];  
int    GBL_run_scheduler=0;
```

What should always be true for this type of data structure?

- timer, initialTimerValue > 0
- run == 0 or 1
- task is a valid function address – MCU and code-dependent!
  - Look in datasheet or MapViewer to see where is valid: E.g. 0x0f0000 to 0x0ff800

# 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

# Debugging Example

---

Program has a local array of 300 data values (table)

- Initialized so that data value = its index (0,1,2, ..., 299)

Calls function **sum\_elements**

- Adds up **count** elements starting with element # **offset**
- Also looks for out-of-range elements (beyond **MIN\_VAL** or **MAX\_VAL**)
- Puts result in location in memory indicated by pointer argument
- Function returns 1 if successful, 0 otherwise

# sum\_elements function

```
int sum_elements(int a[], ARG_TYPE offset, int count, int * retval)
{
    int local_table[LOCAL_TABLE_SIZE];
    int i, i_eff, sum=0;
    int i_search;
    /* check for out-of-range values */
    for (i_search=offset; i_search < offset+count; i_search++) {
        if ((a[i_search] > MAX_VAL) || (a[i_search] < MIN_VAL))
            return 0; /* error code */
    }
    for (i=0; i<count; i++) {
        sum += a[i+offset];
    }
    MYSTERY_CODE1
    *retval = sum;
    return 1;
}
```

# Debugging Example – Bug 1

MCU never returns from function sum\_elements

- Where are we?
  - No source code highlighted. Can't even bring up Disasm view of code –
  - What is the current value of the PC? Open BasicWindows->Register Window to examine it
    - 0EEEEFh
- Where is this? Is it valid?
  - Examine Map file (via ToolManager)
  - Not valid code address (Program ranges from 0f0000h to 0f00edh)
- Where does the code go off course?
  - Usual Suspect: Corrupted stack frame
    - Enter the subroutine with the Step command (may require multiple steps)
    - Come to end of subroutine
    - Check the return address (FB+2 through FB+4) before the EXITD is executed
      - » It should match the address of the instruction following the subroutine call (JSR)
  - Find out when the stack frame is corrupted

# Debugging Example - Bug 2

Sum is incorrect: Should be 760, not 26165

- Examine processor activity within function
  - Step into code to see how sum is incremented
  - BTW, the loop which searches for out-of-range values never executes
    - Save for later
  - sum starts at 0, becomes 5237 after adding first element (which should be 150)
- First element ( $a[i+\text{offset}]$ ) should be 150, not 5237
  - Two likely possible problems
    - A. Data element is corrupted
    - B. Reading from wrong location



# Debug Methods for Bug 2

## A. Check to see if data element is correct

- Basic Windows -> C Watch Window -> Global Window.
  - Is slow for large structures
- Data element is correct.

## B. Check to see from where MCU is actually reading

- Toggle Program Window to Mix mode
- Examine instructions to see which register holds the pointer
  - `ADD.W [A0], -4H[FB]`
- Examine Register Window to see that A0 holds 032Ch, but array **a** starts at 0400h and takes 600 bytes. Out of bounds array access!

## B'. Check array index components in source code

- `i = 0`, `offset = -106`
- Offset should be 150, is wrong

# Bug 2 – Tracking down bad offset

Want to find out when offset was last good, and where it went bad

- Put breakpoint at start of function, since offset is an argument passed in
- Reset MCU and rerun code
- offset is wrong beginning at the procedure entry!

Problem: out-of-range data

- Using a signed character, has range of -128 to 127
  - Trying to pass value of 150
- Replace parameter type with something large enough
  - unsigned char: 0 to 255
  - integer: -32768 to 32767
  - unsigned integer: 0 to 65535
  - etc.

# 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

# 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