Memfault

Proactive Debugging with Offensive Programming

Tyler Hoffman, Co-Founder, Memfault

Does this look Familiar?

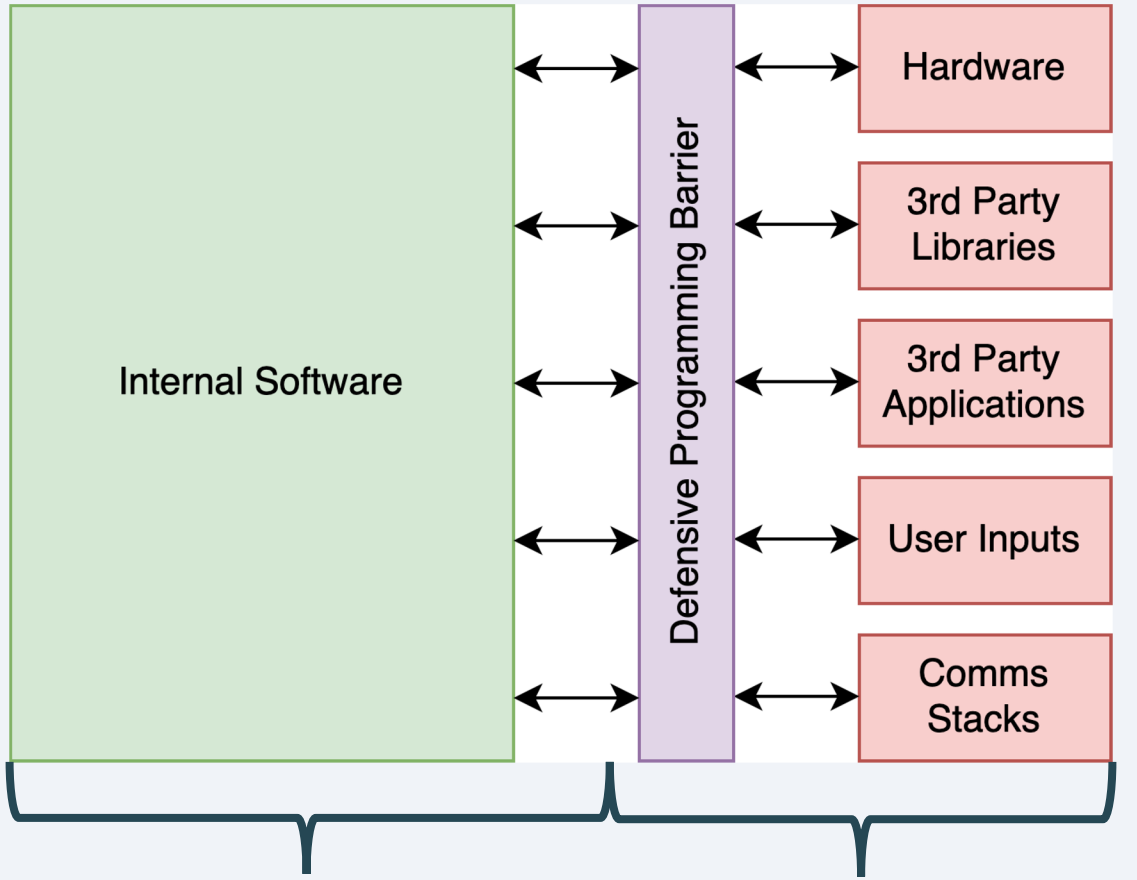
```
void process_something(sData *data) {
    uint8_t *buf = malloc(256);
    if (buf == NULL) {
        LOG("Malloc failed");
        return;
    }
}
```

It’s defensive, but poorly done

```
void process_something(sData *data) {
    uint8_t *buf = malloc(256);
    if (buf == NULL) {
        LOG("Malloc failed");
        return;
    }
}
```

- Pretends like it’s recoverable
- Requires implementation knowledge
- Often leads to silent failures and confusion
- *Someone* has to eventually deal with the error. Maybe your future self.

It can be appropriate at times



#1 takeaway from this talk

```c
int do_something(void) {
    uint8_t *buf = malloc(128);
    if (buf == NULL) {
        return -1;
    }
}
```

```c
void do_something(void) {
    uint8_t *buf = malloc(128);
    ASSERT(buf != NULL);
}
```

Agenda

1. What is Offensive Programming
2. Production Usage
3. Examples
4. Best Practices

Tyler Hoffman

Co-Founder & Lead Engineer, Memfault

- I love developer tools, primarily for embedded engineers
- Previously: Firmware Engineer @ Pebble, Fitbit
- I write on Memfault's Interrupt blog and give talks.
- https://interrupt.memfault.com

fitbit pebble Memfault

Offensive Programming

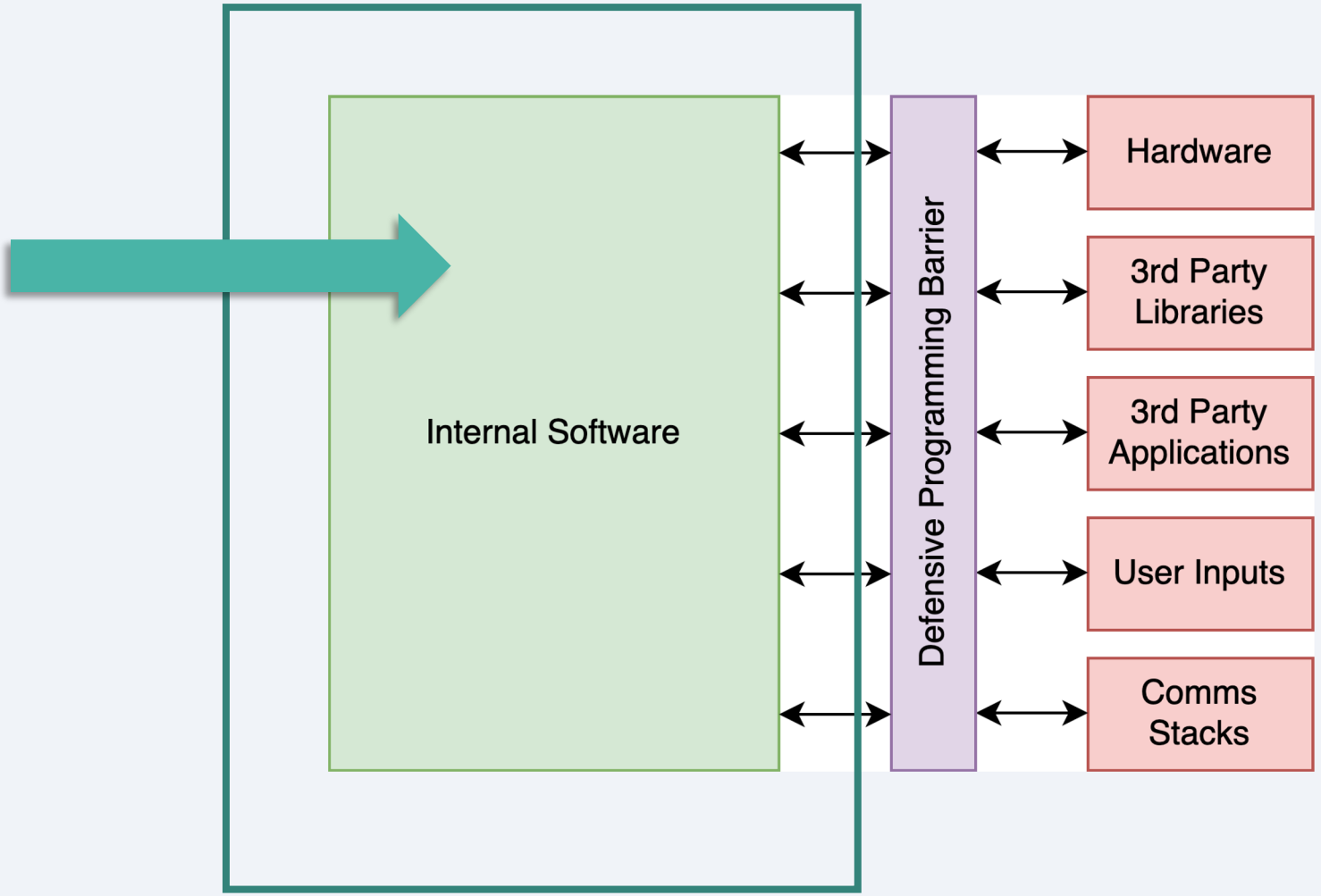
```
void do_something(void) {
    uint8_t *buf = malloc(128);
    ASSERT(buf != NULL);
}
```

Raise errors immediately – and loudly

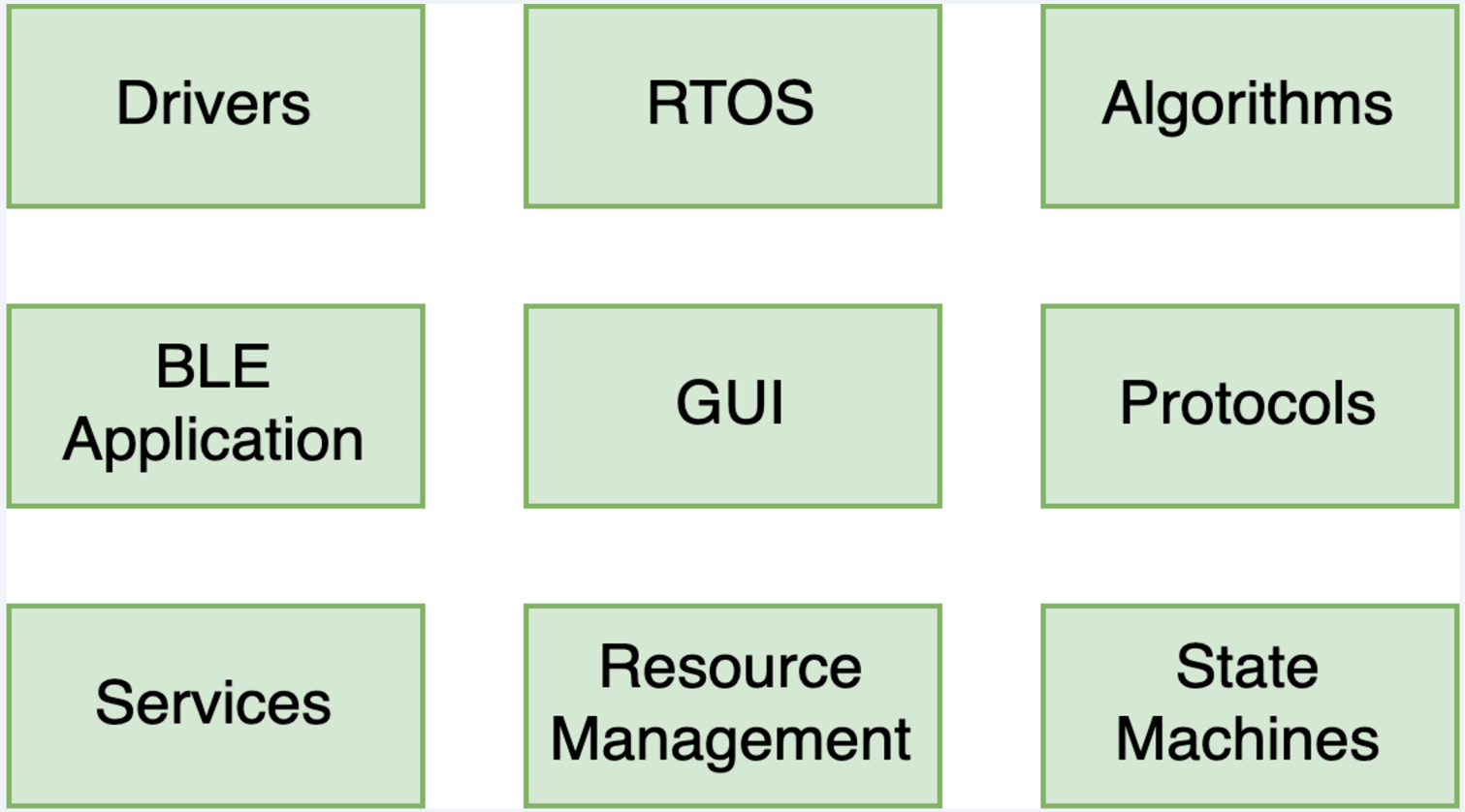
https://interrupt.memfault.com/blog/asserts-in-embedded-systems

Internal Software Modules

Assert in here



Internal Software Modules



Majority of the software stack is in our control

Reasons to use Assert

- Infinitely better than documentation
- Asserts provide breadcrumbs (file & line number)
- Raise alarms close to the root issue
- Safest thing to do in undefined state is to reset
- You control the assert handler
- Capture extra data, logs, a coredump. Anything!

```
void do_something(void) {
    uint8_t *buf = malloc(128);
    ASSERT(buf != NULL);
}
```

Fail fast! – especially during development and testing

What you should assert on

Programmer Error

- Invalid arguments
- Out-of-order API calls

Undefined Behavior

- Memory corruption
- Security issues

Resource Exhaustion

- Malloc failures
- Stack overflow

Performance

- Queues full
- Watchdog timers

Many of these are very difficult to reproduce

Wait a minute... ASSERT? Liberally!?

Yes. Even in production (most of the time)

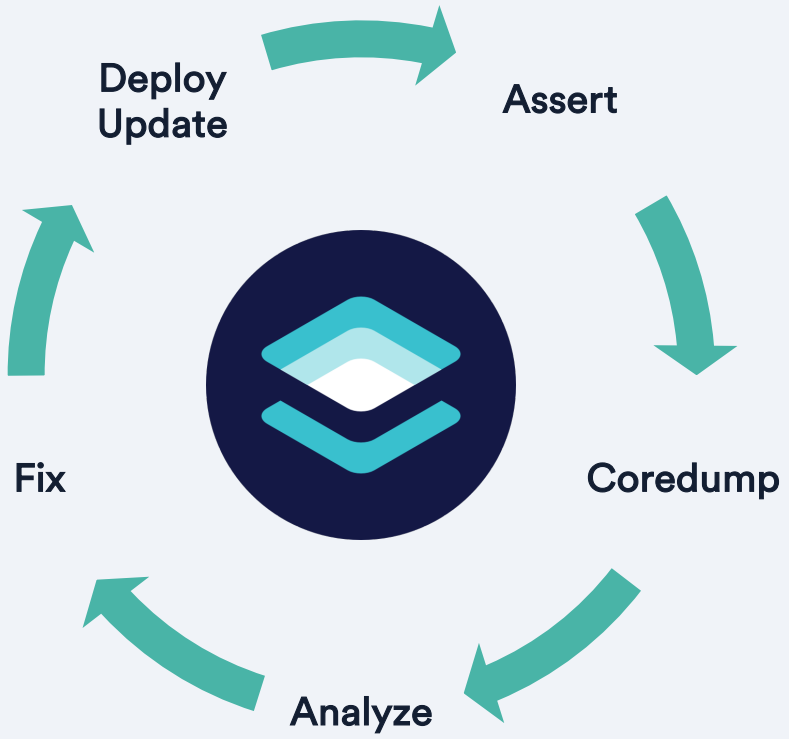
The problem is that our devices are now resetting with no debugger attached.

That's where Memfault comes in...

Production environments

Every bug will surface in production

- 1 in 10,000 bugs are real
- Production has the greatest matrix of test cases
- Log if assertions aren't possible in some cases
- **Diagnostics need to be collected**



Offensive Programming + Diagnostics = Quicker fixes

Prerequisites for Production

Diagnostics & monitoring is a requirement

- ✅ Basic telemetry & logging
- ✅ Proper fault handling
- ✅ Assert implementation
- ✅ Some path to receive error data
- ✅ Devices can firmware update

Memfault

Logs, Metrics,
Crashes

Memfault SDK

What to capture on an assert

- File and line number of the assert

  Expression value (if configured to do so)

- Backtrace of the asserted task
- Arguments and variables within the call stack
- Global and static variable values
- State of all data structures (heap, queues, etc.)

Debugger attached

or

Memfault

Threads

- accel-workq (2) STACK OVERFLOW RUNNING
  - 0 compute_fft in .../src/fft.c at line 10
  - 1 sleep_algo_compute_sleep_time in .../src/sleep_algo.c
  - 2 process_accel_data_worker_task in .../src/accel_data
  - 3 z_work_q_main in .../zephyr/lib/os/work_q.c at line 32
  - 4 z_thread_entry in .../lib/os/thread_entry.c at line 29
  - 5 0xaaaaaaaa
- Thread 3 SUSPENDED
  - 0 z_arch_irq_unlock in .../arm/asm_inline_gcc.h at line 137
  - 1 __swap in .../arch/arm/core/swap.c at line 63
  - 2 z_swap_irqlock in .../kernel/include/kswap.h at line 145
  - 3 z_swap in .../kernel/include/kswap.h at line 145
  - 4 z_tick_sleep in .../zephyr/kernel/sched.c at line 965
  - 5 z_impl_k_sleep in .../zephyr/kernel/sched.c at line 983
  - 6 k_sleep in .../syscalls/kernel.h at line 21
  - 7 eswifi_spi_poll_thread in .../eswifi/eswifi_bus_spi.c at li
  - 8 z_thread_entry in .../lib/os/thread_entry.c at line 29
  - 9 0xaaaaaaaa
- idle (4) READY
- logging (5) SUSPENDED
- net_mgmt (6) BLOCKED

Exceptions | Registers & Locals | Globa ...

Analysis

Memory management fault detected at 0×2000a3c0

Memory management fault on a data access

Fault Register	Value	Hex Value
CFSR	130	0×00000082
HFSR	0	0×00000000
SHCSR	458753	0×00070001

Memory Viewer

Find Ad... | Regions

```
0x08000000 00 e1 00 20 ...
0x08000004 95 9e 00 08 ....
0x08000008 25 9e 00 08 %...
0x0800000c 81 9d 00 08 ....
0x08000010 81 9d 00 08 ....
0x08000014 81 9d 00 08 ....
0x08000018 81 9d 00 08 ....
0x0800001c 81 9d 00 08 ....
0x08000020 81 9d 00 08 ....
0x08000024 81 9d 00 08 ....
0x08000028 81 9d 00 08 ....
0x0800002c 9d 9b 00 08 ....
0x08000030 81 9d 00 08 ....
0x08000034 81 9d 00 08 ....
0x08000038 41 9b 00 08 A...
0x0800003c ed 95 00 08 ....
0x08000040 6d 9e 00 08 m...
0x08000044 6d 9e 00 08 m...
0x08000048 6d 9e 00 08 m...
0x0800004c 6d 9e 00 08 m...
0x08000050 6d 9e 00 08 m...
0x08000054 6d 9e 00 08 m...
0x08000058 6d 9e 00 08 m...
```

Exceptions | Registers & Locals | Globals & Statics | ...

A **dft_out** = 0×2000a900 <my_stack_area+1344>
L **i** = 400
A **num_samples** = 536912536
A **raw_samples** = 0×3128115f
L **tmp** = {1, 222, 7, 84}
R **$r0** = long 536912536 (0×2000a298)
R **$r1** = long 1372324912 (0×51cc0430)
R **$r2** = long 1372324919 (0×51cc0437)
R **$r3** = long 536912832 (0×2000a3c0)
R **$r4** = long 536912508 (0×2000a27c)
R **$r5** = long 536914136 (0×2000a8d8)
R **$r6** = long 0 (0×00000000)
R **$r7** = long 536912488 (0×2000a268)
R **$r8** = long 0 (0×00000000)
R **$r9** = long 0 (0×00000000)
R **$r10** = long 0 (0×00000000)
R **$r11** = long 0 (0×00000000)
R **$r12** = long 0 (0×00000000)
R **$sp** = void * 0×2000a268 <my_stack_area2+104> (0×2000a2...
R **$lr** = long 134353149 (0×080210fd)

< 1 2 >

Search... Order by Memory Locati

▶ **lock** = k_spinlock **{...}**
overflow_cyc = volatile u32_t **0**
heap_sz = unsigned int **0**
▶ **s_mflt_packetizer_state** = sMfltTransportState **{...}**
▶ **s_active_data_source** = const sMemfaultDataSourceImpl* **{...}**
▶ **s_ds_rle_state** = sMemfaultDataSourceRleState **{...}**
▶ **s_event_storage** = sMfltCircularBuffer **{...}**
▶ **s_event_storage_read_state** = sHeartbeatStorageReadState **{...}**
▶ **s_event_storage_write_state** = sHeartbeatStorageWriteState **{...}**
▶ **s_memfault_ram_logger** = sMfltRamLogger **{...}**
▼ **s_mflt_reboot_info** = sMfltRebootInfo* **{...}**
  ▼ * = sMfltRebootInfo {...}
    magic = uint32_t 559170130
    version = uint8_t 2
    crash_count = uint8_t 1
    ▶ **rsvd1** = uint8_t[1] {...}
    coredump_saved = uint8_t 1
    last_reboot_reason = uint32_t 0
    pc = uint32_t 0
    lr = uint32_t 0
    reset_reason_reg0 = uint32_t 0
    ▶ **rsvd2** = uint32_t[10] {...}

Offensive Programming Examples

With Memfault as your “debugger”

Argument Validation

```
void device_set_name(char *name,
                     size_t name_len) {
  ASSERT(name && name_len <= 32);
  ...
}

bool device_get_name(char **buf,
                     size_t buf_len) {
  ASSERT(buf && buf_len >= 32);
  ...
}
```

Developer errors → Raise the alarm immediately

Argument Validation

State Logs

Threads

Thread 1

0 memfault_reboot_tracking_assert_handler in .../memfault_fault_handling.c at line 171
1 cli_execute in .../libraries/cli/nrf_cli.c at line 2554
2 cli_state_collect in .../libraries/cli/nrf_cli.c at line 1952
3 nrf_cli_process in .../libraries/cli/nrf_cli.c at line 2852
4 mflt_cli_try_process in src/cli.c at line 37
5 idle_state_handle in src/main.c at line 1030
6 main in src/main.c at line 1030

Exceptions Registers & Locals Globals & Statics Heap

A **extra** = <optimized out>
L **info** = { reason = kMfltRebootReason_Assert, pc = 159296, lr = 168611 }
A **lr** = <optimized out>
A **pc** = <optimized out>
R **$r0** = long 537122316 (0×2003d60c)
R **$r1** = long 559170130 (0×21544252)
R **$r2** = long -536810236 (0xe000ed04)
R **$r3** = long -2147483648 (0×80000000)
R **$r4** = long 1 (0×00000001)
R **$r5** = long 537122512 (0×2003d6d0)
R **$r6** = long 1 (0×00000001)
R **$r7** = long 240896 (0×0003ad00)
R **$r8** = long 537122448 (0×2003d690)
R **$r9** = long 240896 (0×0003ad00)
R **$r10** = long 241888 (0×0003b0e0)
R **$r11** = long 0 (0×00000000)
R **$r12** = long 1 (0×00000001)
R **$sp** = void * 0×2003d608 (0×2003d608)
R **$lr** = long 226689 (0×00037581)
R **$pc** = void (*)() 0×3750e <memfault_reboot_tracking_assert_handler+42> (

< 1 2 >

State machine transition errors

```
void on_commit(eState prev_state) {
  ASSERT(prev_state == kState_Flushing
      || prev_state == kState_Idle);
}
```

Ensure that states happen in order and as expected

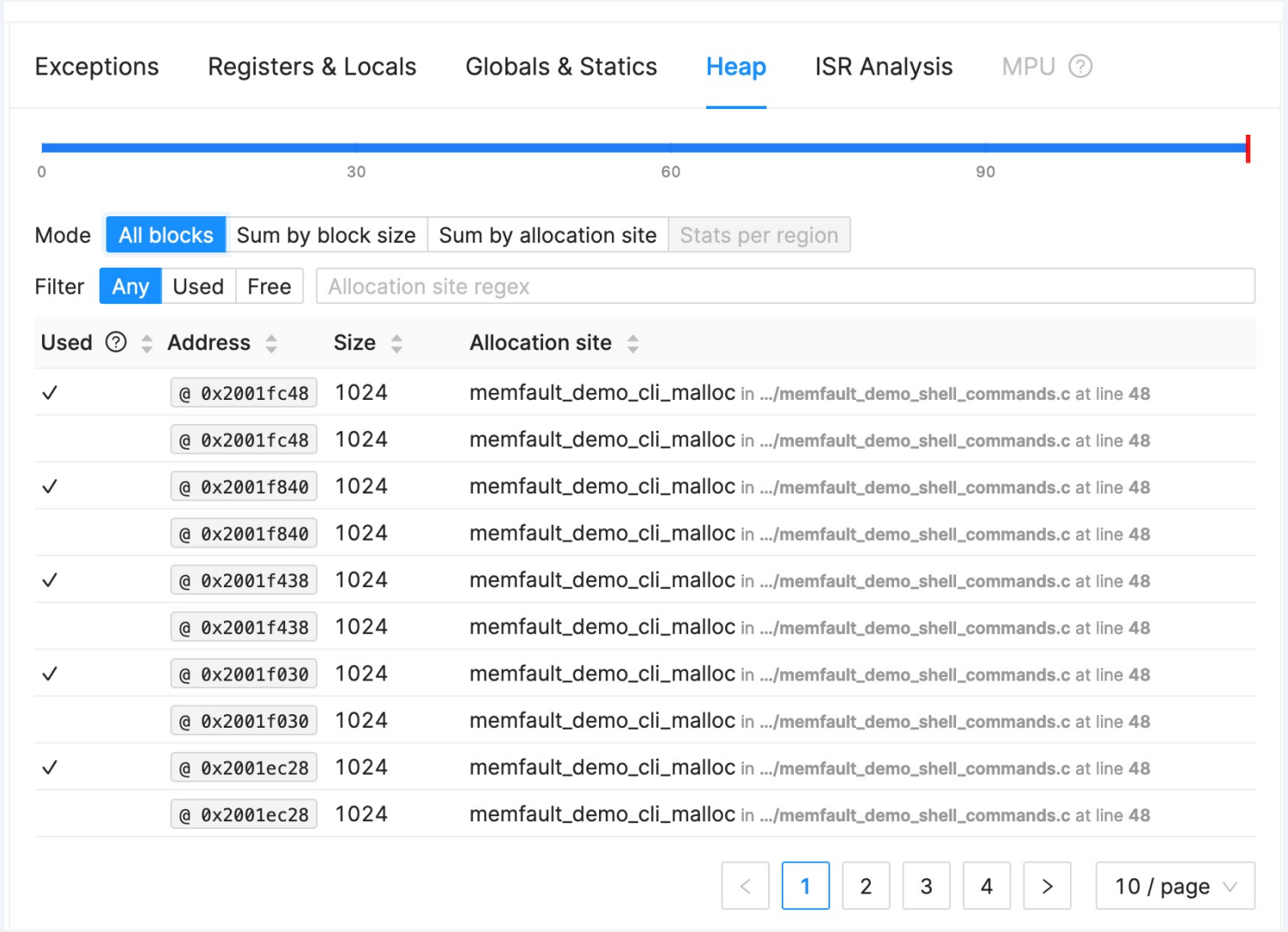
Malloc returns NULL

```
void *malloc_assert(size_t n) {
  void *p = malloc(n)
  ASSERT(p);
  return p;
  ...
}
```

For allocations that should never fail

Likely means a memory leak

Memfault Memory View



Full Queue

```
void critical_event(void) {
  ...
  const bool success =
      xQueueSend(q, &item, 1000 /* wait 1s */)
  ASSERT(success);
  ...
}
```

Track down performance issues using a timeout

Resource Depletion – full queue

```
(gdb) queue_print s_event_queue
-----
Queue Status: 10/10 events in queue (FULL!)
0: Addr: 0x200070c0, event: BLE_PACKET
1: Addr: 0x200070a8, event: TICK_EVENT
2: Addr: 0x20007088, event: BLE_PACKET
3: Addr: 0x20007070, event: BLE_PACKET
4: Addr: 0x20007050, event: BLE_PACKET
5: Addr: 0x20007038, event: BLE_PACKET
6: Addr: 0x20007018, event: BLE_PACKET
7: Addr: 0x20007000, event: BLE_PACKET
8: Addr: 0x20006fe0, event: BLE_PACKET
9: Addr: 0x20006fc8, event: BLE_PACKET
```

Weren’t processing BLE packets fast enough

Detecting software stalls

```
void timing_sensitive_task(void) {
  const bool success =
      mutex_lock(&s_mutex, 1000 /* 1 second */);
  ASSERT(success);
  {
    ...
  }
}
```

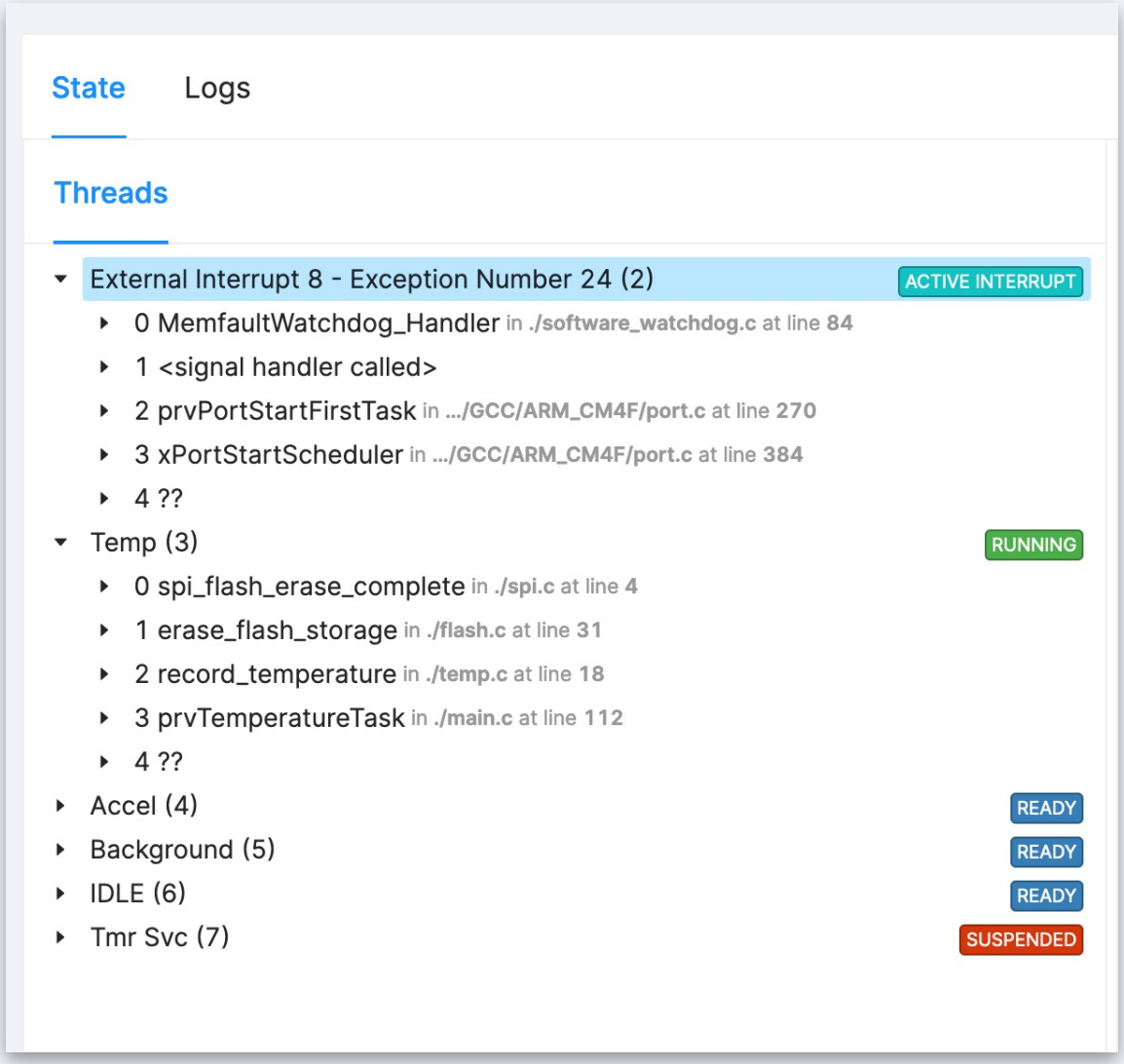
Fail if mutex not grabbed in reasonable time

Detecting software stalls

```
void timing_sensitive_task(void) {
  // Task watchdog will assert a stall
  mutex_lock(&s_mutex, INFINITY);
  {
    ...
  }
}
```

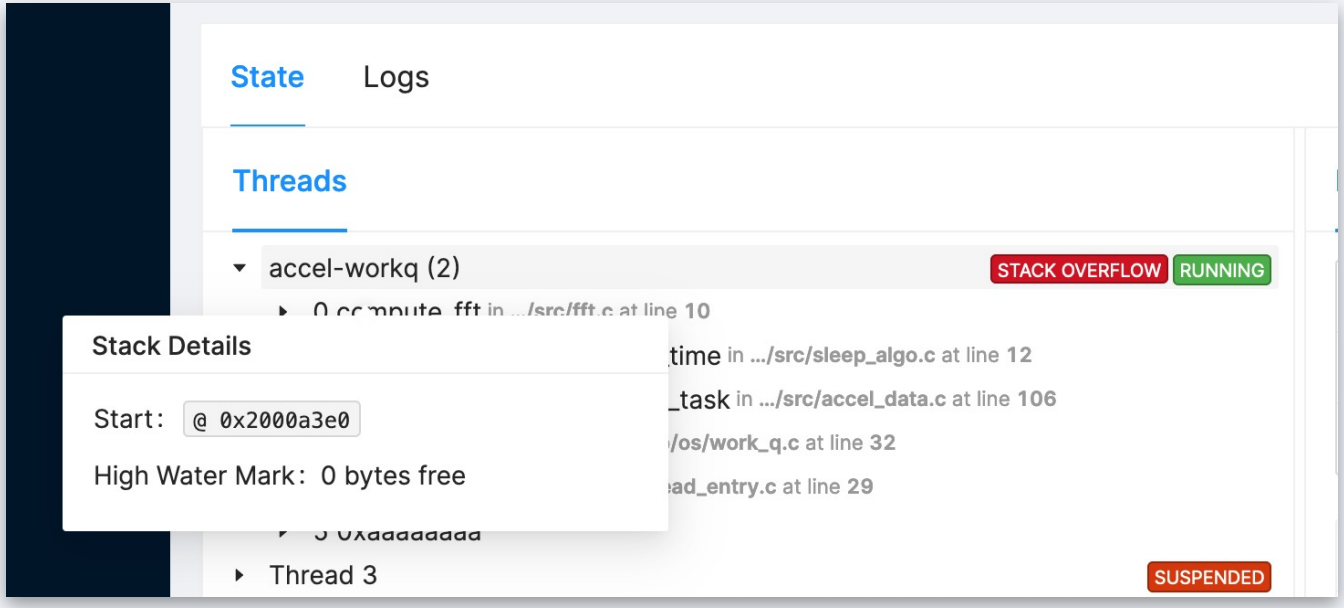
Let the software watchdog detect the stall

Detecting software stalls



Stack Overflow Detection

Many RTOS's have this built in now. Double check yours!



https://embeddedartistry.com/blog/2020/05/18/implementing-stack-smashing-protection-for-microcontrollers-and-embedded-artistrys-libc/

https://www.freertos.org/Stacks-and-stack-overflow-checking.html

https://docs.zephyrproject.org/latest/reference/usermode/memory_domain.html#hardware-stack-overflow

Failing even faster: Compile-time errors ⚡ ⚡

```c
typedef struct PACKED {
  uint32_t count;
  uint8_t buf[12];
  uint8_t new_value; // ADDED
} MyStruct;

_Static_assert(sizeof(MyStruct) <= 16, "Oops, too large!");
```

```
$ gcc test.c
test.c:14:1: error: static_assert failed due to requirement
    'sizeof(MyStruct) <= 16' "Oops, too large!"

_Static_assert(sizeof(MyStruct) <= 16, "Oops, too large!");
^              ~~~~~~~~~~~~~~~~~~~~~~
1 error generated.
```

Best Practices

Watch out for boot loops

Boot loop detection is a must.

- You control the assert handler
- Capture extra data, logs, a coredump. Anything!
- Don’t assert on boot
- Count # reboots within time interval
- Boot into safe mode
- Only FWUP, diagnostics pull, and factory reset

https://interrupt.memfault.com/blog/device-firmware-update-cookbook

Build asserts into wrappers

```
void *malloc_assert(size_t size) {
    uint8_t *buf = malloc(128);
    ASSERT(buf != NULL);

    /* Collect breadcrumbs */

    return buf;
}
```

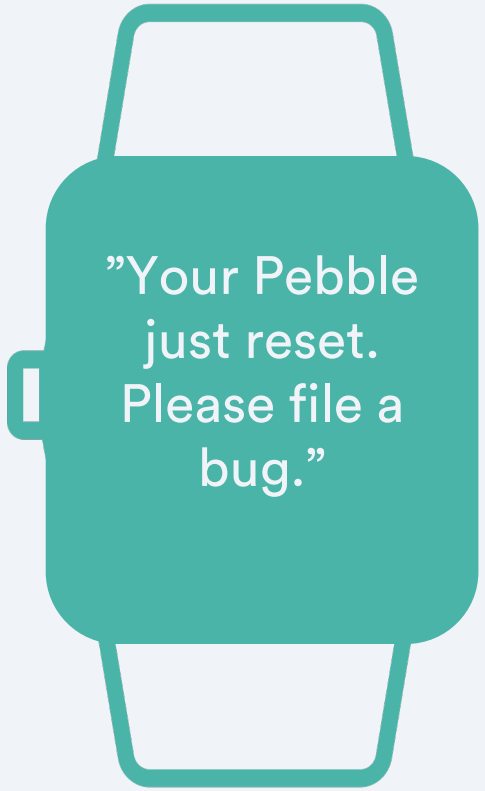
Clean and simple

Debug builds are your friend

Internal testing is the best kind of testing

On internal builds:

- Enable more aggressive asserting
- Tighten timeout durations
- Send builds to small groups externally
- Test. Experiment. Try things. Be creative.



- Every Pebble internal firmware build

Takeaways

- Don’t play defense against bugs
- Fail fast and capture data
- Test internally as much as possible
- Keep asserts in production
- We at Memfault would love to help

Thank You!

- memfault.com
- twitter.com/memfault
- linkedin.com/company/Memfault
- interrupt.memfault.com
- We're hiring!

Memfault

Co-founder, Memfault

Memfault