Zephyr Fatal Error Policy

Roy Ratcliffe[1,*]

Abstract

This article presents a hazard-aware alternative to fatal error handling for Zephyr-based firmware that utilises Zephyr's "iterable section" feature to define extra application-specific non-delayed emergency safety policies.

Keywords: Embedded, C, Zephyr

Zephyr [1] is a very useful Real-Time Operating System. Just like most operating systems of the embedded variety, Zephyr halts on a fatal error. This works well in most cases but not in all situations, particularly in hazardous embedded environs.

Fatal errors include:

1. memory faults or unhandled CPU exceptions
2. stack overflow
3. spurious or unhandled interrupts
4. unprotected memory-allocation failures
5. bugs

When operating potentially dangerous equipment, firmware needs its own special policy. The last reason has special significance during firmware development.

1. Fatal Error Policy

Zephyr applies its fatal-error policy using a weakly-defined `k_sys_fatal_error_handler` function. Strong symbols override weak ones. The toolchain works that way. See more about weak function attributes in the GNU compiler's online documentation.

The exact Zephyr-default implementation appears below. Find the actual code on GitHub.

```
extern void sys_arch_reboot(int type);

__weak void k_sys_fatal_error_handler(unsigned int reason,
                                      const z_arch_esf_t *esf)
{
    ARG_UNUSED(esf);

    LOG_PANIC();
    LOG_ERR("Halting system");
    arch_system_halt(reason);
    CODE_UNREACHABLE;
}
```

Zephyr does *not* declare `sys_arch_reboot` in a publicly accessible header file; the source code needs a prototype as above. It locks the CPU against additional interrupt requests and enters a forever `for`-loop spin that never returns.

There is a problem here, however. Logging when controlling hazardous equipment takes time, precious time. The default `LOG_PANIC` performs a register and stack dump. The elaborate logging mechanism in Zephyr allows for custom logging frontends, processing and backends. Backends can include multiple loggers that require processing time and possibly peripheral delays in order to output the log messages.

1.1. Nordic Policy

Nordic's nRF Connect SDK uses Zephyr RTOS but adds some customisations, including fatal error handling.

Nordic's version of the Zephyr kernel's default fault handler has two alternative execution paths: two approaches depending on `CONFIG_RESET_ON_FATAL_ERROR`. Either the system halts *or* reboots in nRFxxx kernels. Kernel "oops" behaviour triggers a hard fault, switching the kernel to panic mode. The kernel logs an error and either reboots or halts. The same limitation exists for Nordic's SDK, regardless of which halt-or-reset choice the firmware developer takes.

Nordic defines their policy *strongly*; the developer *cannot* override it. The project configuration `prj.conf` can switch off the strong-override fatal-error handler by including:

```
CONFIG_RESET_ON_FATAL_ERROR=n
```

This configuration reverts to the default 'weak' Zephyr handler.

2. Custom Fatal Error Handlers

The idea is to set up an "iterable section" [2] of decoupled system fatal error handlers—zero or more function pointers to mandatory fatal safety procedures. Whenever fatal errors occur, the primary handler first invokes the special section before logging the error and either halting the system or rebooting it.

*Corresponding author
Email address: roy@ratcliffe.me (Roy Ratcliffe)
[1] See more hackery at GitHub.

How does it work? The application defines one or more fatal error handlers using a `SYS_FATAL_ERROR_DEFINE` macro. The replacement fatal error handler will promptly invoke them all, one by one, on critical errors. Find the full implementation on GitHub. It includes a new `sys/fatal_error.h` header file. Extract below. The macro adds a handler structure to the fatal error section.

```
#include <zephyr/kernel.h>

struct sys_fatal_error {
  void (*handler)();
};

#define SYS_FATAL_ERROR_DEFINE(_name)                    \
  static const STRUCT_SECTION_ITERABLE(sys_fatal_error, \
    _CONCAT(sys_fatal_error_, _name))
```

This approach protects developers during testing when the likelihood of such problems increases due to changes in source code. Even small changes can trigger faults and exceptions. If a module or driver requires some special crash response then adding this behaviour simply requires that the developer add a new handler using the `SYS_FATAL_ERROR_DEFINE` macro. Exemplar follows.

2.1. Testing

Take a straightforward test. The firmware boots and then waits a few seconds before *explicitly* crashing. Along these lines:

```
int main(void) {
  k_sleep(K_SECONDS(2));
  k_oops();

  return 0;
}
```

Full source code appears on GitHub. It adds some LED sugar to demonstrate the feature more visibly by initially driving LED 0 using a PWM channel at its slowest period at 50% duty cycle.

The test switches off the pulse modulator when the system crashes, as follows.

```
void pwm_led0_off_fatal_error() {
  pwm_set_dt(&pwm_led0, 0, 0);
  LOG_INF("Switched off %s", pwm_led0.dev->name);
}

SYS_FATAL_ERROR_DEFINE(pwm_led0_off) = {
  .handler = pwm_led0_off_fatal_error
};
```

The log dumps the following trace.

```
*** Booting nRF Connect SDK v3.5.99-ncs1-1 ***
[00:00:02.428,344] <err> os: r0/a1:  0x00000003  r1/a2:  0x00000000  r2/a3:  0x00000000
[00:00:02.428,375] <err> os: r3/a4:  0x0000000c r12/ip:  0x00000001 r14/lr:  0x0000453f
[00:00:02.428,375] <err> os:  xpsr:  0x41000000
[00:00:02.428,405] <err> os: Faulting instruction address (r15/pc): 0x000007f2
[00:00:02.428,436] <err> os: >>> ZEPHYR FATAL ERROR 3: Kernel oops on CPU 0
[00:00:02.428,466] <err> os: Current thread: 0x200007c8 (main)
[00:00:02.428,527] <inf> main: Switched off pwm@4001c000
[00:00:03.308,197] <err> fatal_error: Halting system
```

At first blush, the log tells a misleading story. Note the timestamps, however. Logging does not immediately flush the backends. Log messages go to memory first. Adding a breakpoint to the code during a debug session proves that the iterable section of extended fatal error handlers runs first.

3. Conclusions

System faults matter.

The firmware must supply a fault handler as a bare minimum baseline. All safety operations critically depend on the fault-free execution of machine code. If code *cannot* execute for any reason, all safety features become redundant. Safety *is* code within the core or cores; no running code, no safety.

Many fault potentialities exist at different layers. Fundamental faults exist at the most basic level; they include stack overflow, memory allocation failure, internal peripheral faults such as MMU and FPU faults, plus package-level faults originating from internal and external peripherals. These lie at the lowest level, the interface between hardware and software, the CPU, its RAM and its peripheral hardware components. Bugs and critical assertion failures, e.g. null pointer exceptions, sit at the next higher layer. They originate in the purely informational domains triggered by programming assumptions that fail when the metal meets the road. Such include arguments and return values out of expected ranges or timing requirements out of expected bounds.

The fault handler has one and only one goal: to shut down the external hazards. Nothing else matters. The handler runs at the most fundamental level, even higher than the highest priority thread and all the nested interrupt handlers. The fault handler represents the *last* sequence of machine-code instructions the core will execute before it shuts down. The core freezes with interrupts disabled and waits for a reset. Thus, before reaching that point, the core must ensure that the GPIO or other signals safely power down the equipment so that hazardous external components do not continue running uncontrollably.

In summary, the system fault handler has a crucial safety role by executing pre-defined instructions to protect the system from damage in the event of a fault or malfunction. Primarily and ultimately from an overall hazard perspective, safety applies to the equipment's human operators, be they test engineers or end users. The very first thing that the software should intelligently do *and* the very last thing from within safety-critical hardware is "make it safe"—everything else in between is secondary.

The safety features represent what the author would want the software to do if using such equipment personally. You might call me overly scrupulous.

> "If you prick us, do we not bleed?"—William Shakespeare

Bibliography

[1] Zephyr Project, Zephyr RTOS (May 2024).
URL https://www.zephyrproject.org/

[2] Zephyr Project, API documentation (May 2024).
URL https://docs.zephyrproject.org/latest/index.html