

Jawne FromISR: UART RX i GPIO edge

jeden IsrContext, jedna decyzja yield, jawny backpressure

Karta	K15 / 16	Czas	30 minut
Platforma	Hazard3 / RV32I	IRQ	machine external
RX queue	capacity 1	Overflow	drop-newest
Wersja	v00.01	UUID	02f6cd71-...
		karty	

Jawny kontrakt handlera

```
IsrContext isr;
auto byte = uart.isr_view().read_rx_from_isr();
rx.send_from_isr(byte, isr);
notify.give_from_isr(isr);
isr.yield_if_needed();           // dokładnie raz
```

Kontrakt K15. ISR view nie ma timeoutu ani blocking API. Każde ...FromISR dokłada żądanie wake do jednego contextu. Handler czyści źródła, publikuje bounded dane i dopiero na końcu raz podejmuje decyzję yield.

Plan 30 minut

Czas	Tryb	Dowód
0–5	views	task view kontra ISR view
5–10	IRQ	mcause=0x8000000b, ISR stack
10–15	UART	0x41 queued, 0x42 dropped
15–20	GPIO	edge 1–0, notification=1
20–25	wake	flag true, yield calls=1
25–30	order	receiver przed resume stimulus

Predykcja

Czy drugi bajt zmieści się w pełnej kolejce capacity 1? _____. Ile razy handler powinien wywołać yield dla dwóch publikacji? _____.

1 Realny external IRQ i bounded handler

stimulus: UART[41,42] + GPIO edge + SET_IRQ

↓ machine external trap / ISR stack

↓ queue FromISR + notify FromISR + clear sources

↓ one yield - rx-worker - resume stimulus

Pomiar	Predykcja	Odczyt
mcause	0x8000000b	_____
ISR SP różny od task SP	true	_____
external IRQ state po clear	0	_____
UART ready przed / po ISR	1 / 0	_____
GPIO edge przed / po ISR	1 / 0	_____

2 UART RX i backpressure

Dwa bajty czekają w modelu rejestrów, lecz queue ma jedno miejsce. Handler ma limit dwóch odczytów; nie wykonuje parsowania ani oczekiwania.

Pomiar	Predykcja	Odczyt
queued count / byte	1 / 0x41	_____
dropped count / byte	1 / 0x42	_____
overflow policy	drop-newest	_____
receiver byte	0x41	_____

Porównaj drop-newest, drop-oldest i overwrite dla UART:

3 Task view i ISR view

Wrapper	Task view	ISR view
Uart	enable/configure, test inject	ready/read, bez wait
GpioPin	direction/write output	edge pending/clear
Queue	receive(timeout)	send_from_isr
Notification	task odbiera	give_from_isr

4 Akumulacja wake i jeden yield

```
BaseType_t local = pdFALSE;
xQueueSendFromISR(queue, &byte, &local); isr.merge(local);
local = pdFALSE;
vTaskNotifyGiveFromISR(worker, &local); isr.merge(local);
isr.yield_if_needed();
```

Pomiar	Predykacja	Odczyt
notification sent/taken	1 / 1	_____
higher priority task woken	true	_____
yield requested	true	_____
yield_if_needed calls	1	_____
receiver tick	1	_____

Dowód natychmiastowego przełączenia

- receiver widzi `stimulus_after=0`;
- po powrocie z IRQ stimulus widzi `receiver_done=1`;
- receiver ustawia GPIO output z bitu 0 bajtu 0x41: data=1.
Co zmieniłby brak yield przy wake=true?

Granica ISR

Parsing, debounce, logowanie i retry są pracą taska. ISR tylko odczytuje bounded FIFO, czyści źródło i publikuje. Static queue/task storage daje heap delta 0.

5 Hazard3/GDB i zaliczenie

```
make check
riscv64-unknown-elf-gdb build/task01_isr_drivers/prog.elf
b isr_drivers_debug_checkpoint
```

```
p/x g_mcause
p g_uart_status_before_isr
p g_uart_status_after_isr
p g_gpio_edge_before_isr
p g_gpio_edge_after_isr
p g_rx_queued
p/x g_queued_byte
p g_rx_dropped
p/x g_dropped_byte
p g_notification_taken
p g_higher_priority_task_woken
p g_isr_yield_calls
p g_stimulus_after_seen_by_receiver
p g_receiver_done_at_stimulus_resume
p g_isr_drivers_pass
```

Zaliczenie

- ☐ pokazuję realny external IRQ i osobny ISR stack;
- ☐ używam wyłącznie jawnych metod `from_isr`;
- ☐ kolejkuje 0x41 i jawnie liczę drop-newest 0x42;
- ☐ clear UART/GPIO zmienia status 1 na 0;
- ☐ akumuluję wake=true i wywołuję yield dokładnie raz;
- ☐ receiver działa przed resume stimulus i ustawia GPIO=1.

Wyjście

Dlaczego automatyczne zgadywanie task/ISR contextu osłabia API?

Gdzie zaimplementujesz parsing i debounce, i dlaczego?

Następna karta K16: integracja usług RTOS, end-to-end trace, memory/stack report, pressure injection i uzasadnienie każdej granicy.