

Queue<T> i StaticQueue<T,N>

typowana granica nad kopiowaniem bajtów FreeRTOS

Karta	K10 / 16	Czas	30 minut
Platforma	Hazard3 / RV32I	Wiadomość	Sample, 12 B
Capacity	2 elementy	Timeout	jawny, 0 lub 2 ticki
Wersja	v00.01	UUID karty	66967c2d-...

Kontrakt typu przed uruchomieniem

```
static_assert(std::is_trivially_copyable<T>::value,
    "Queue<T> requires T to be trivially copyable");
```

Kernel zapamiętuje `sizeof(T)` bajtów. Nie wywołuje konstruktora, destruktor ani operacji kopiowania C++. Dlatego owning `VectorV1` nie może być wiadomością by value.

Kontrakt K10. `send` kopiuje bajty w chwili wywołania. Późniejsza zmiana źródła nie może zmienić elementu w kolejce. Timeout jest jawny; wrapper nie ukrywa `portMAX_DELAY`.

Plan 30 minut

Czas	Tryb	Dowód
0–5	gate	trivial Sample przechodzi; VectorV1 nie
5–10	create	dynamic heap cost kontra static delta 0
10–15	copy	send, mutate, receive oryginalnej wartości
15–20	bytes	snapshot 12 B caller storage
20–25	full	trzeci send odmawia po 2 tickach
25–30	FIFO	drain 0x2222, 0x3333; empty receive=false

Predykcja

`sizeof(Sample)` = _____ B. Po wysłaniu wartości `0x1111` i zmianie źródła na `0xDEAD` odbiorę: _____.

1 Dynamiczna i statyczna własność

Właściwość	Queue<Sample>	StaticQueue<Sample,2>
create API	xQueueCreate	xQueueCreateStatic
control	heap	caller StaticQueue_t
item bytes	heap	caller uint8_t[24]
item size	sizeof(Sample)	sizeof(Sample)
destructor	delete + heap free	delete, bez free caller bytes

Pomiar heap i adresów

Pomiar	Predykcja	Odczyt
dynamic cost	> 0	_____ B
static create delta	0	_____ B
heap po obu destruktorach	baseline	_____
static handle	control address	_____
control/storage domain	.bss, poza heap	_____

2 Kopia jest snapshotem

```
Sample source{1, 0x1111, 3, 0x55};
dynamic.send(source, 0);
source.value = 0xDEAD;
dynamic.receive(received, 0);
```

Wartość	Źródło po mutate	Odebrana kopia
value	0xDEAD	_____
flags	0xAA	_____

Co byliby błędem przy xQueueCreate(2,sizeof(Sample*)), gdy send dostaje &sample zamiast &samplePointer?

3 Full, timeout, FIFO i empty

```
send(first,0) → count 1
send(second,0) → count 2 / FULL
send(third,2) → task blokuje maks. 2 ticki / false
receive → first, potem second
receive(empty,0) → false
```

Dowód	Predykcja	Odczyt
count przed trzecim send	2	_____
third send result	false	_____
tick delta	≥ 2	_____
FIFO first value	0x2222	_____
FIFO second value	0x3333	_____
empty receive, timeout 0	false	_____

4 Zajrzyj do caller storage

Po pierwszym static send porównaj pierwsze 12 bajtów `g_static_queue_storage.bytes` z reprezentacją `first`.

Pole	Wartość	Little-endian bytes
sequence	2	_____
value	0x2222	_____
channel/flags	4 / 0x10	_____

Ćwiczenie typu

Zaprojektuj trivial UART message: kierunek, długość, maks. 8 bajtów payload i sequence. Zapisz definicję oraz policz `sizeof`:

Dlaczego sam fakt, że wskaźnik jest trivially copyable, nie rozwiązuje lifetime wskazywanego obiektu?

5 Hazard3/GDB i zaliczenie

```
make check
riscv64-unknown-elf-gdb build/task01_queue/prog.elf
b queue_debug_checkpoint
```

```
p g_dynamic_queue_cost
p g_static_control_address
p g_static_bytes_address
p g_static_first_item_bytes
p g_copy_source_after_mutation
p g_copy_received_value
p g_full_count
p g_timeout_ticks
p g_fifo_values
p g_queue_pass
```

Negatywny test jest częścią kontraktu

```
struct VectorV1 {
    unsigned* data;
    ~VectorV1() { data = nullptr; }
};
freertos::Queue<VectorV1> forbidden{2}; // must not compile
```

Oczekiwana przyczyna, nie przypadkowy błąd linkera: `Queue<T>` requires `T` to be trivially copyable.

Zaliczenie

- ☐ wyjaśniam byte copy i gate trivially-copyable;
- ☐ pokazuję 128 B dynamic cost i 0 B static delta;
- ☐ dowodzę snapshot 0x1111 po mutacie do 0xDEAD;
- ☐ wiążę 12 B item size z typed API i storage;
- ☐ pokazuję full, timeout 2, FIFO i empty=false;
- ☐ negatywny test VectorV1 kończy się właściwym static assert.

Wyjście

Kiedy przesyłać wartość, a kiedy wskaźnik? Jak udokumentować ownership?

Następna karta K11: Mutex, własny LockGuard i obserwowalne dziedziczenie priorytetu.