Priority Queue

Summary

`bucketed_pqueue` is a simple, efficient strict-priority queue for FreeRTOS systems where:

- items fall into a small fixed set of priority levels
- producers may be tasks or ISRs
- a single consumer drains work in priority order

Its design is intentionally lightweight:

- one FreeRTOS queue per priority
- one bitmap to track which priorities are active
- optional task notification for efficient wakeup

> Used correctly, it is a clean fit for embedded event dispatch and deferred work handling.

> Used incorrectly, mainly with multiple consumers or inconsistent bucket item types, it becomes a fine little trap with excellent timing and poor manners.

Purpose

> The `bucketed_pqueue` module implements a **strict-priority queue** on top of standard FreeRTOS queues.

Instead of storing all items in one queue, it uses **multiple FIFO queues**, called **buckets**, where each bucket represents one priority level.

- **Bucket 0** = lowest priority
- **Bucket** `num_buckets - 1` = highest priority

When consuming items, the module always returns an item from the **highest non-empty priority bucket**.

Within the same priority bucket, ordering remains **FIFO**, because each bucket is an ordinary FreeRTOS queue.

This gives the system:

- **strict priority across buckets**
- **FIFO ordering within each bucket**
- support for **task producers**
- support for **ISR producers**
- a lightweight way to wake a consumer task when new work arrives

When to use this module

Use this module when:

- work items have a **small fixed number of priority levels**
- **higher-priority items must always be processed first**
- you want to keep FreeRTOS queue semantics
- producers may run in both **task** and **interrupt** context
- only **one consumer** is responsible for draining the queue

Typical use cases:

- event dispatching where alarms/errors must preempt normal work
- deferred interrupt processing with different urgency levels
- message handling where command classes have discrete priority bands

When to **not** use this module

> This module is **not** a general-purpose concurrent priority queue.

Do **not** use it if:

- you need **multiple consumer tasks** calling `pop()` or `peek()` concurrently
- you need **arbitrary numeric priorities** rather than a small fixed bucket range
- you need **blocking pop** behavior built into the module
- you need stable ordering across different priorities based on timestamp or insertion order

The implementation is designed around **multiple producers, single consumer**.

High-Level Design

The queue is made from:

- an array of FreeRTOS queue handles
- a count of how many buckets exist
- a 32-bit bitmap tracking which buckets are currently believed to be non-empty
- an optional task handle to notify when something is pushed

Core idea

Each priority level has its own FreeRTOS queue.

The module maintains a bitmap:

- bit `n` set = bucket `n` is believed to contain at least one item
- bit `n` clear = bucket `n` is believed to be empty

This bitmap lets the consumer avoid blindly probing every queue all the time.

Example

If there are 4 buckets:

- bucket 0 = low
- bucket 1 = medium
- bucket 2 = high
- bucket 3 = critical

And the bitmap is:

```
non_empty_mask = 0b1010
```

then bucket 1 and bucket 3 are non-empty.

A call to `bucketed_pqueue_pop()` will check from highest to lowest, so it will try:

1. bucket 3
2. bucket 2
3. bucket 1
4. bucket 0

and return the first available item it finds.

Data Structure

```
typedef struct {
  QueueHandle_t *buckets;
  uint8_t num_buckets;
  uint32_t non_empty_mask;
  TaskHandle_t notifier;
} bucketed_pqueue_t;
```

Fields

`buckets`

Pointer to an array of `QueueHandle_t`.

Each entry is a FreeRTOS queue representing one priority bucket.

> This array is **not owned** by the module. The caller must create the queues and ensure the array remains valid for the entire lifetime of the priority queue.

`num_buckets`

Number of buckets in the `buckets` array.

Valid range: **1 to 32**.

The upper limit exists because `non_empty_mask` is a 32-bit bitmap.

`non_empty_mask`

Bitmap used as a fast summary of which buckets contain items.

- bit `0` corresponds to bucket `0`
- bit `1` corresponds to bucket `1`
- etc.

This bitmap is updated on push, and cleared when the consumer determines a bucket is empty.

`notifier`

Optional task to notify when an item is successfully pushed.

If not `NULL`, a push sets the notification bit:

```
1UL << prio
```

in the target task’s notification value.

This is useful when the consumer task waits on task notifications instead of polling.

Important behavioral guarantees

This module provides the following behavior:

Strict priority

Higher-priority buckets are always preferred over lower-priority buckets.

If bucket 3 and bucket 1 both contain items, `pop()` will always return from bucket 3 first.

FIFO within one priority

Because each bucket is a FreeRTOS queue, items in the same bucket are processed in insertion order.

Non-blocking pop/peek

`pop()` and `peek()` do not wait. If no item is available, they return `RESULT_ERR_NOT_FOUND`.

Multi-context producers

There are separate APIs for:

- task context: `bucketed_pqueue_push()`
- ISR context: `bucketed_pqueue_push_from_isr()`

Single-consumer design

The implementation assumes only one consumer performs `pop()` and `peek()`.

That is not just a suggestion. It is a design constraint.

Required setup

This module does **not** create FreeRTOS queues itself.

The caller must:

1. create one FreeRTOS queue for each priority level
2. store those queue handles in an array
3. initialize a `bucketed_pqueue_t` using that array

Example setup

```
#define NUM_BUCKETS 4

static QueueHandle_t bucket_handles[NUM_BUCKETS];

bucketed_pqueue_t pq;

void app_init(void) {
  bucket_handles[0] = xQueueCreate(8, sizeof(my_msg_t));
  bucket_handles[1] = xQueueCreate(8, sizeof(my_msg_t));
  bucket_handles[2] = xQueueCreate(8, sizeof(my_msg_t));
  bucket_handles[3] = xQueueCreate(8, sizeof(my_msg_t));

  bucketed_pqueue_init(&pq, bucket_handles, NUM_BUCKETS, consumer_task_handle);
}
```

Strong recommendation

> All buckets should use the **same item type or at least size**.

Technically the module does not enforce that. Practically, mixing queue item sizes makes the API awkward and error-prone, because `pop()` and `peek()` write into a single `out` buffer and the caller has no type information at that point.

Usage model

Producer side

A producer decides the priority and pushes into the matching bucket.

Example:

```
my_msg_t msg = { ... };
bucketed_pqueue_push(&pq, PRIORITY_HIGH, &msg, pdMS_TO_TICKS(10));
```

From an ISR:

```
BaseType_t higher_woken = pdFALSE;
my_msg_t msg = { ... };

bucketed_pqueue_push_from_isr(&pq, PRIORITY_HIGH, &msg, &higher_woken);
portYIELD_FROM_ISR(higher_woken);
```

Consumer side

The consumer repeatedly pops the highest-priority item available.

Example:

```
my_msg_t msg;

while (bucketed_pqueue_pop(&pq, &msg) == RESULT_OK) {
  process_msg(&msg);
}
```

Because `pop()` is non-blocking, a typical design is:

1. consumer task blocks on a task notification
2. on wakeup, it calls `pop()` in a loop until `RESULT_ERR_NOT_FOUND`

Example pattern:

```
for (;;) {
  uint32_t notified_bits = 0;
  xTaskNotifyWait(0, UINT32_MAX, &notified_bits, portMAX_DELAY);

  my_msg_t msg;
  while (bucketed_pqueue_pop(&pq, &msg) == RESULT_OK) {
    process_msg(&msg);
  }
}
```

This pattern works well with the module's `notifier` mechanism.

Notification behavior

If `notifier` is provided during initialization, every successful push sends a task notification with:

```
1UL << prio
```

using `eSetBits`.

This means:

- pushes do **not overwrite** previous notification bits
- multiple buckets can be represented in the task notification value
- repeated pushes to the same priority keep the same bit set

What the notification means

The notification indicates that at least one push occurred into that bucket.

It does **not** guarantee:

- how many items are in that bucket
- that the bucket is still non-empty by the time the task wakes
- that the bitmap and queues are perfectly synchronized at all times

It is a wakeup hint, not a count.

That is fine. The consumer should drain the queue with repeated `pop()` calls rather than assuming one notification equals one item.

Concurrency model and assumptions

This section matters more than the function list.

Supported access pattern

Supported

- multiple producer tasks
- ISR producers

- one consumer task calling `pop()` and/or `peek()`

Not supported

- multiple concurrent consumers calling `pop()`
- one task peeking while another task pops in a way that assumes strong synchronization guarantees
- external code modifying the underlying bucket queues directly behind this module's back

> The module uses a bitmap plus queue operations, but it does **not** implement a full multi-consumer synchronization scheme around dequeue behavior.

Why single-consumer matters

The bitmap is read, scanned, and repaired in steps.

That is acceptable with one consumer, because any race is limited to producer updates and queue state changes, and the consumer can repair stale bits safely.

With multiple consumers, two tasks could:

- observe the same bitmap snapshot
- both decide the same bucket has data
- one drains the queue
- the other sees an empty queue and clears the bit

That alone is survivable, but once multiple consumers are simultaneously probing and repairing, reasoning about ordering and fairness gets messy fast. This module avoids that entire circus by assuming a single consumer.

Critical sections

The bitmap is protected with FreeRTOS critical sections:

- `taskENTER_CRITICAL()` / `taskEXIT_CRITICAL()`
- `taskENTER_CRITICAL_FROM_ISR()` / `taskEXIT_CRITICAL_FROM_ISR()`

These critical sections protect **bitmap access**, not the whole queue operation sequence.

That means queue operations and bitmap updates are not one indivisible transaction.

This is intentional and mostly fine for the intended model, but maintainers should understand that the bitmap is a **best-effort summary**, not a perfect mirror of queue state.

Known implementation characteristics

Priority scan is linear in number of buckets

`pop()` and `peek()` scan from highest to lowest priority:

```
for (int prio = num_buckets - 1; prio >= 0; prio--)
```

So the dequeue cost is `O(num_buckets)` in the worst case.

Since `num_buckets <= 32`, this is usually acceptable in embedded systems.

> If someone later decides to turn this into 128 priorities with a 32-bit bitmap, you might want to consider a better scanning system for all buckets.

That said, the loop is incredibly efficient doing only a bit-wise check to know if the bucket is populated or not.

Bitmap may be temporarily stale

The module can have these transient states:

- queue contains data but bitmap not yet updated
- bitmap says non-empty but queue is empty

The code handles the second case explicitly by clearing stale bits when `xQueueReceive()` or `xQueuePeek()` fails.

The first case is shorter-lived and happens between a successful queue send and the bitmap update, or if initialization starts with pre-filled queues.

This is why the bitmap should be viewed as a hint structure.

No ownership of queue storage

> The module does not allocate or destroy the bucket queues.

It only stores the queue handles provided by the caller.

The caller is responsible for:

- creating queues before initialization
- keeping them alive while the priority queue is in use
- ensuring item types and queue lengths are appropriate

No reset/deinit API

There is no deinitialization function.

If needed, the caller must manage queue lifecycle itself.

If a reset feature is ever added, it must consider:

- draining or recreating each bucket
- resetting `non_empty_mask`
- possible interaction with producers still running

Practical example

Scenario

A system has three priorities:

- `0`: telemetry
- `1`: commands
- `2`: emergency actions

All use the same message type:

```
typedef struct {
  uint8_t type;
  uint32_t value;
} app_msg_t;
```

Setup

```
#define APP_NUM_PRIORITIES 3

static QueueHandle_t app_buckets[APP_NUM_PRIORITIES];
static bucketed_pqueue_t app_pq;
```

```
void app_queue_init(TaskHandle_t consumer_task) {
  app_buckets[0] = xQueueCreate(16, sizeof(app_msg_t));
  app_buckets[1] = xQueueCreate(16, sizeof(app_msg_t));
  app_buckets[2] = xQueueCreate(8, sizeof(app_msg_t));

  bucketed_pqueue_init(&app_pq, app_buckets, APP_NUM_PRIORITIES, consumer_task);
}
```

Producer task

```
void send_command(uint32_t cmd) {
  app_msg_t msg = {
    .type = 1,
    .value = cmd,
  };

  (void)bucketed_pqueue_push(&app_pq, 1, &msg, 0);
}
```

ISR producer

```
void emergency_isr(void) {
  BaseType_t higher_woken = pdFALSE;

  app_msg_t msg = {
    .type = 2,
    .value = 0xDEADU,
  };

  (void)bucketed_pqueue_push_from_isr(&app_pq, 2, &msg, &higher_woken);
  portYIELD_FROM_ISR(higher_woken);
}
```

Consumer task

```
void consumer_task(void *arg) {
  (void)arg;
```

```
  for (;;) {
    uint32_t notify_bits;
    xTaskNotifyWait(0, UINT32_MAX, &notify_bits, portMAX_DELAY);

    app_msg_t msg;
    while (bucketed_pqueue_pop(&app_pq, &msg) == RESULT_OK) {
      handle_message(&msg);
    }
  }
}
```

Result

If telemetry, commands, and emergency messages all arrive, the consumer will process:

1. emergency
2. commands
3. telemetry

Within each class, messages remain FIFO.

Error handling

The module uses `result_t`, which is defined elsewhere.

Based on the implementation, these results are used:

`RESULT_OK`

Operation succeeded.

`RESULT_ERR_INVALID_ARG`

Returned when the caller passes invalid arguments, such as null pointers or out-of-range priority.

`RESULT_ERR_OVERFLOW`

Returned by push functions when the selected bucket queue cannot accept the item.

This usually means the bucket queue is full, or the send timed out.

`RESULT_ERR_NOT_FOUND`

Returned by `pop()` or `peek()` when no item is available.

This is not necessarily an error in the usual sense. It is the normal result for an empty priority queue.

Maintenance notes

If you change the number of buckets beyond 32

You must also change:

- the bitmap type
- all bit shift logic
- input validation
- notification assumptions based on `1UL << prio`

Right now, 32 buckets is a **hard architectural limit**.

If you add blocking pop behavior

Be careful.

A naive implementation that blocks on each bucket in order would break strict priority or become awkward and expensive.

A better approach is usually to keep the existing design:

- producers notify a task
- consumer waits on notification
- consumer drains with non-blocking `pop()`

If you still add a blocking API, document its wakeup semantics very clearly.

If you want multiple consumers

This requires redesign.

You would need to revisit:

- bitmap synchronization
- dequeue race handling
- semantics of `peek()`
- fairness between consumers
- whether the notifier model still makes sense

Do not label it “thread-safe” just because critical sections exist.

If different buckets need different item types

The current API is not a good fit for that.

`pop()` and `peek()` return into one generic `out` buffer with no explicit type metadata.

If you need heterogeneous payloads, safer patterns are:

- use a tagged union message type
- store pointers to separately managed objects
- wrap payloads in a common envelope struct

If queues are pre-filled before init

The bitmap starts at zero during `bucketed_pqueue_init()`.

So pre-filled queues will not be visible until something later sets the relevant bits, or until code is changed to rebuild the bitmap.

If supporting pre-filled queues matters, one possible improvement is for `init()` to inspect each bucket using `uxQueueMessagesWaiting()` and initialize `non_empty_mask` accordingly.

That behavior does **not** exist today.

Revision #4
Created 2026-04-14 10:09:54 UTC by Nikolaos Diamantopoulos
Updated 2026-04-15 08:15:34 UTC by Nikolaos Diamantopoulos