Mailboxes on FreeRTOS

Roy Ratcliffe

Abstract

FreeRTOS equips a message buffer and a task abstraction for building embedded systems. Erlang's Open Telephony Platform demonstrates a simple but very powerful abstraction for decomposing concurrent software. This article presents a purely functional implementation of a concurrent system mailbox for lightweight, low-footprint embedded usage via FreeRTOS.

Keywords: C, FreeRTOS, Erlang

1. Erlang'ian Mailboxes

Erlang's Open Telephony Platform (OTP) is legendary for its robustness and resilience[1]. It lives by the motto, "Let it crash." Erlang itself as a language recapitulates Prolog. Think of Erlang as Prolog without trail-stack backtracking but *with* term unification. The OTP layer built using Erlang adds a lightweight, low-memory footprint with low-overhead scheduling[2].

The entire OTP system sits on top of the mailbox processor concept. Is it possible, is it practical, to mimic the mailbox processor for embedded systems using FreeRTOS?

2. FreeRTOS Mailbox

Is it possible? The short answer is: yes indeed! The result is quite straightforward and requires only around 150 lines of C. See Gist. The implementation reuses the FreeRTOS message buffer as a messaging mechanism combined with thin Message Pack wrappers for binding and unifying messages.

Figure 1 below depicts an Erlang-style mailbox for FreeRTOS. The mailbox is just a message buffer with a task—not much more than that. The figure draws the mailbox using Unified Modelling Language as a conceptual class diagram although the implementation for FreeRTOS is purely functional C. The diagram simplifies the function names for brevity and clarity, `send_msg` becomes `xMailboxSendMsg` in reality.

2.1. Spawn and Link

Fundamentally, the mailbox-oriented system builds a linked tree of mailbox processors acting as concurrent messaging handlers. The structure allows for actor decomposition.

Construct and start a mailbox processor using:

*Corresponding author
Email address: `roy@ratcliffe.me` (Roy Ratcliffe)

[1] Robustness and resilience sound like the same thing. However, one is proactive the other reactive: how easily can a system fall down versus how well it recovers when something untoward happens.

[2] No locking via semaphores and critical sections is required.

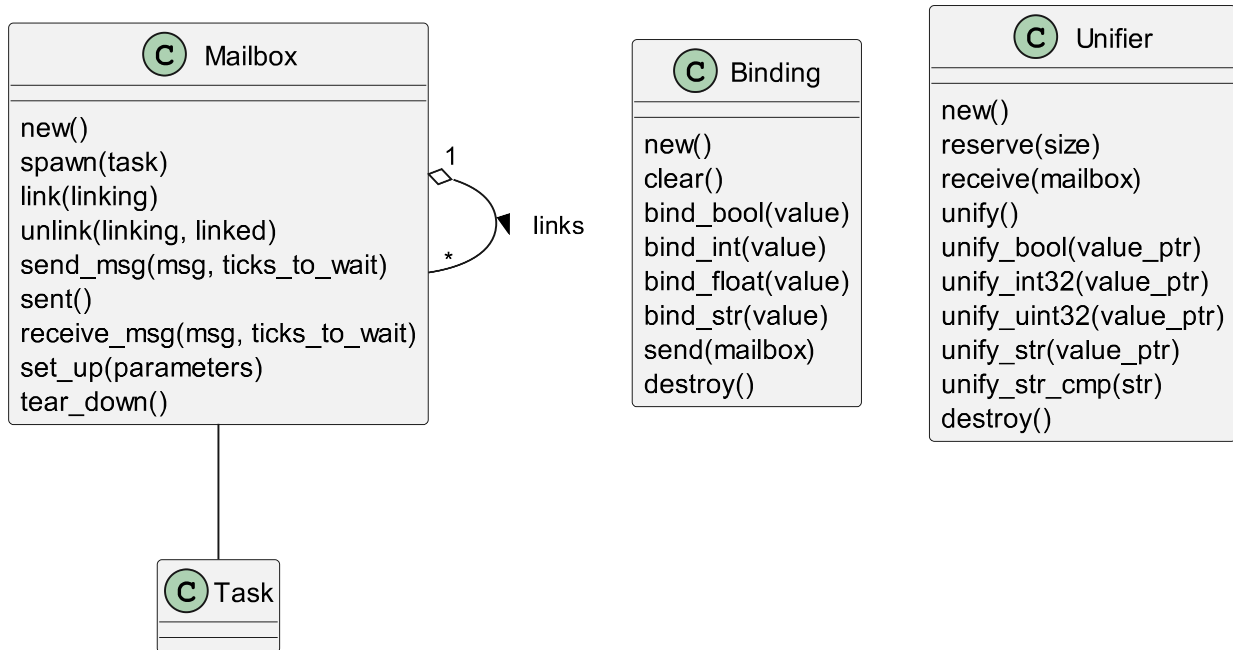

Figure 1: Mailbox, Binding, Unifier and Task abstractions

```
  MailboxHandle_t xMailbox = xMailboxNew();
  xMailboxSpawn(xMailbox, prvEchoMailboxTask, "echo");
```

Spawning starts its associated processor task, a normal FreeRTOS task albeit with a built-in mailbox.

The mailbox processing task implementation eats messages.

```
static portTASK_FUNCTION(prvEchoMailboxTask, pvParameters) {
  vMailboxSetUp(pvParameters);
  MsgBindingHandle_t xMsgBinding = xMsgBindingNew();
  MsgUnifierHandle_t xMsgUnifier = xMsgUnifierNew();
  for (;;) {
    uint32_t ulNotified;
    xTaskNotifyWait(0UL, ULONG_MAX, &ulNotified, portMAX_DELAY);
    if (ulNotified & mailboxSENT_NOTIFIED)
      // Pattern-match the mailbox messages one by one. Do not wait for
      // additional messages, hence no ticks to wait.
      while (xMailboxReceiveMsg(NULL, xMsgUnifier, 0UL)) {
        if (xMsgUnify(xMsgUnifier) != eMsgUnifySuccess)
          continue;

        // Match an optional mailbox sender handle but not by sending to itself.
        // Perform a basic sanity check. The sender must not be the receiver.
        MailboxHandle_t xMailbox;
        if (xMsgUnifyMailbox(xMsgUnifier, &xMailbox)) {
          if (xMailbox == xMailboxSelf())
            continue;
```

```
          if (xMsgUnify(xMsgUnifier) != eMsgUnifySuccess)
            continue;
        } else
          xMailbox = NULL;

        // Handle ping by sending back pong.
        if (xMsgUnifyStrCmp(xMsgUnifier, "ping")) {
          if (xMailbox) {
            vMsgBindingClear(xMsgBinding);
            xMsgBindStr(xMsgBinding, "pong");
            xMailboxSendMsg(xMailbox, xMsgBinding, portMAX_DELAY);
            xMailboxSent(xMailbox);
          }
          continue;
        }
      }
    }
    vMsgUnifierDestroy(xMsgUnifier);
    vMsgBindingDestroy(xMsgBinding);
    vMailboxTearDown();
    vTaskDelete(NULL);
  }
```

The processor handles messages by unifying the message stream sequentially and maintains structured concurrency. It interfaces with other, non-streaming events, by waiting for general notifications. The message "sent" notification is one of many other possibilities. Other notifications might include asynchronous hardware events triggered by interrupt-service routines.

2.2. Bind and Unify Using MsgPack

Messages have an arbitrary type, including compound types: strings, integers, floats, arrays, maps and any combinations thereof. Build messages by binding specific values. Unpack messages by unifying expected types. "Actual" message contents may *fail* to match the expected. That allows for unification and future-proof message development, embuing an embedded architecture with event streaming properties and commensurate benefits. Kafka, Akka, ActiveMQ, RabbitMQ and others adopt such an architecture for its scalable concurrency benefits. The event becomes the locus of functionality and the development building block for backwardly-compatible unification.

Sending a message requires a 'binding' and looks like this.

```
    // Ping the echo mailbox. Bind a task handle followed by a string.
    // Send the message then notify.
    MsgBindingHandle_t xMsgBinding = xMsgBindingNew();
    xMsgBindMailbox(xMsgBinding, NULL);
    xMsgBindStr(xMsgBinding, "ping");
    xMailboxSendMsg(xMailbox, xMsgBinding, portMAX_DELAY);
    vMsgBindingDestroy(xMsgBinding);
    xMailboxSent(xMailbox);
```

This snippet destroys the binding after sending the message because the sender no longer requires it; the binding has already coded the packed message so not needed. The sender could also retain the binding for reuse by clearing it. The binding process progressively builds up an internal message buffer of packed messages.

Unification reverses the binding process: bytes in, message components out. The concept originates with first-order logic, [1]. Fundamentally, 'unifying' expectations with actual message contents involves a matching case block. The snippet from the previous extract illustrates:

```
while (xMailboxReceiveMsg(NULL, xMsgUnifier, 0UL)) {
  if (xMsgUnify(xMsgUnifier) != eMsgUnifySuccess)
    continue;

  if (xMsgUnifyStrCmp(xMsgUnifier, "ping")) {
    // handle ping
    continue;
  }
}
```

Read pending messages from the mailbox. Attempt a unification. Throw away failed messages; they are unification mismatches, future as-yet-unknown types. Handle when the message stream matches an expectation—pong when pinged, in this case.

3. Conclusions

Dependencies include a message binding and unifier mechanism. Message Pack is used in the supplied implementation. The C library is small, fast and flexible even for complex messages such as arrays and maps. In pure logic, the Message Pack protocol reduces to simple predicates.

It is not the most trivial encoding technique. Sending pure strings would be an alternative, for example; the sender could `printf` a message and the receiver could `scanf` the messages. Message Packing does nevertheless allow for fast unification over complex types because the 'type' encodes first and matches first which amounts to a simple 8-bit comparison. If types match, length comes next and that amounts to a fast integer comparison. Only thereafter the contents compare by early-out matching over the encoded size.

While not the only transcoding technique, Message Pack supplies a very useful embed-friendly compromise of competing requirements, proving fast, compact and flexible. Its approach implicitly encodes type information making full or partial unification very fast. Handling quickly collapses to success or failure; success quickly focuses on the type and its contents—ideal for partial matching.

The default C implementation of the Message Pack library uses large buffer-size defaults, see Table 1. In practice, these sizes far exceed the capacity and needs of an embedded system. Hence 8K becomes more like 256 bytes for this style of usage (right shift by five). The unpacker sizes need even less: 128 and 64 bytes respectively for the initial and reserve sizes.

Table 1: Sizes of Message Pack buffers and chunks

Manifest constant	Default
MSGPACK_SBUFFER_INIT_SIZE	8K
MSGPACK_UNPACKER_INIT_BUFFER_SIZE	64K
MSGPACK_UNPACKER_RESERVE_SIZE	32K
MSGPACK_VREFBUFFER_CHUNK_SIZE	8K
MSGPACK_ZBUFFER_INIT_SIZE	8K
MSGPACK_ZBUFFER_RESERVE_SIZE	512
MSGPACK_ZONE_CHUNK_SIZE	8K

The proposed implementation also relies on `offsetof` C compiler extension in order to navigate by offset from a list item to its container. Most if not all modern C compilers supply a built-in version.

FreeRTOS makes mailbox decomposition easy: one message buffer, one task. The implementation relies on FreeRTOS thread-local storage for linking a task to its mailbox. Apart from concurrency, the paradigm acts as a structural decomposition technique and there useful for wrapping system components in general. If you need interacting concurrency within your embedded system, this approach is not the worst by far.

4. Future Work

"Atoms" would be a handy abstraction for future development (reference for atoms). Atoms are string constants passed by reference. Matching references implies matching a string and obviates the need for string comparisons. They could pack and unpack using the Message Pack extension @ type[3].

References

[1] F. Baader, W. Snyder, Unification Theory, handbook of automated reasoning (2001).
URL https://www.cs.bu.edu/fac/snyder/publications/UnifChapter.pdf

[3] ASCII code 64_{10}