Code Walkthrough (MCU + Linux)

---

1. MCU Side (Arduino / Cortex-M)

1.1 Includes and data structure

```
#include <Arduino_RouterBridge.h>

struct CanFrameMsgV0 {
  uint8_t  version;     // = 0
  uint32_t timestamp;   // millis()
  uint32_t can_id;
  uint8_t  dlc;
  uint8_t  data[8];
  uint8_t  flags;       // bit0 = extended
};
```

What this does

- `#include <Arduino_RouterBridge.h>`
  Brings in the UNO Q **RouterBridge** API, which gives us the `Bridge` object used to talk to the Linux side (MPU).

- `struct CanFrameMsgV0`
  This is our **local C representation** of the CAN frame message we want to send:

  - `version`: message format version (we use `0` for now).
  - `timestamp`: `millis()` at time of sending (ms since MCU boot).
  - `can_id`: 11-bit or 29-bit arbitration ID.
  - `dlc`: data length code (0–8).
  - `data[8]`: raw CAN bytes.
  - `flags`: bitfield; we currently just use bit 0 to say **extended vs standard** ID.

Important: this struct is **for convenience on the MCU**. Over the Bridge we send the fields as separate parameters (because MsgPack can't serialize a raw `uint8_t[8]` array directly).

1.2 Test frame provider (temporary stand-in for real CAN)

```
// TEMP: fixed test frame – later this will read from the CAN buffer
bool get_one_can_frame(CanFrameMsgV0 &msg) {
  msg.version   = 0;
  msg.timestamp = millis();
  msg.can_id    = 0x18FF50E5;   // example extended ID
  msg.dlc       = 8;
  msg.data[0]   = 0x03;
  msg.data[1]   = 0x11;
  msg.data[2]   = 0xFF;
  msg.data[3]   = 0xFF;
  msg.data[4]   = 0xFF;
  msg.data[5]   = 0xFF;
  msg.data[6]   = 0x33;
  msg.data[7]   = 0x00;
  msg.flags     = 0x01;         // extended frame (bit0 = 1)

  return true;
}
```

What this does

- Right now, we **don't pull from the real CAN buffer** yet.
  Instead, we build a **known, fixed test frame**:
  - Extended ID `0x18FF50E5`
  - DLC = 8
  - Payload bytes: `03 11 FF FF FF FF 33 00`
  - `flags = 0x01` → mark as extended frame.
- `millis()` is called here so the timestamp reflects roughly when we decided to send it.
- `return true;`
  We designed it as "try to get a frame from somewhere":
  - future: it will return `false` if the ring buffer is empty
  - now: it always returns `true` (always "has a frame").

This separation means later we can swap in **real CAN buffering** without changing the bridge logic.

1.3 Setup: initialize Bridge and Monitor

```
void setup() {
  // IMPORTANT: init Bridge first, then Monitor (this worked for us in
4.2)
  Bridge.begin();
  Monitor.begin();
  delay(300);

  Monitor.println("MCU: Phase 4.4 starting");
  Monitor.println("MCU: Bridge + Monitor ready");

  // Give Python time to fully start
  delay(5000);
}
```

What this does

- `Bridge.begin();`
  Initializes the **RPC client** on the MCU core so it can talk to the Linux core.

- `Monitor.begin();`
  Instead of `Serial`, UNO Q uses `Monitor` to push text to the App Lab Serial Monitor.
  This is the only way we see MCU logs in the web IDE.

- `delay(300);`
  Short pause to let things settle.

- Two `Monitor.println(...)` calls:
  Human-friendly markers so we know the sketch actually started and Bridge/Monitor are ready.

- `delay(5000);`
  This is a **critical detail** we discovered experimentally:

    - The Linux-side Python environment takes some time to start.
    - If we send RPC calls too early, they may be lost or cause confusing behaviour.
    - 5 seconds is a safe, simple **startup buffer**.

1.4 Loop: 1 Hz CAN frame over the bridge

```
void loop() {
  static uint32_t lastSend = 0;
  uint32_t now = millis();

  // Send at ~1 Hz
  if (now - lastSend < 1000) {
    return;
  }
  lastSend = now;

  CanFrameMsgV0 msg;
  if (get_one_can_frame(msg)) {
    Bridge.notify("can_frame_v0",
                  msg.version,
                  msg.timestamp,
                  msg.can_id,
                  msg.dlc,
                  msg.data[0],
                  msg.data[1],
                  msg.data[2],
                  msg.data[3],
                  msg.data[4],
                  msg.data[5],
                  msg.data[6],
                  msg.data[7],
                  msg.flags);

    Monitor.print("MCU: sent CAN frame ID=0x");
    Monitor.println(msg.can_id, HEX);
  }
}
```

What this does

Rate limiting

```
static uint32_t lastSend = 0;
uint32_t now = millis();

if (now - lastSend < 1000) {
  return;
}
lastSend = now;
```

1.
    - Stores the last send time in `lastSend`.
    - Only proceeds if **at least 1000 ms** have passed.
    - This gives us a clean **1 Hz send rate**:
        - Easy to read in logs
        - Puts almost no load on the bridge
        - Keeps behaviour deterministic while we debug.

Build / fetch a frame

```
CanFrameMsgV0 msg;
if (get_one_can_frame(msg)) {
    ...
}
```

2.
    - Allocates a `CanFrameMsgV0` on the stack.
    - Calls `get_one_can_frame(msg)` to fill it.
    - Check the return value (`true` means “we have a frame to send”).

Bridge.notify: send to Linux

```
 Bridge.notify("can_frame_v0",
                  msg.version,
                  msg.timestamp,
                  msg.can_id,
                  msg.dlc,
                  msg.data[0],
                  msg.data[1],
                  msg.data[2],
                  msg.data[3],
                  msg.data[4],
                  msg.data[5],
                  msg.data[6],
                  msg.data[7],
                  msg.flags);
```

3.
    - Calls the **remote function** named `"can_frame_v0"` on the Linux side.
    - Arguments are serialized by MsgPack under the hood.
    - Note: we send the 8 data bytes as **eight separate arguments**:
        - This is necessary because the Bridge + MsgPack stack can't pack a `uint8_t[8]` C-array directly.
        - The Linux handler will receive them as `b0, b1, ..., b7`.

MCU logging

```
 Monitor.print("MCU: sent CAN frame ID=0x");
Monitor.println(msg.can_id, HEX);
```

4.
    - Confirms from the MCU side what ID was just sent.
    - This lets us **match MCU logs with Linux logs** line-by-line.

1.5 MCU libraries involved

- **Arduino_RouterBridge**
    - Provides `Bridge.begin()` and `Bridge.notify(...)`.
    - Underneath, uses Arduino_RPClite + MsgPack to serialize arguments.
- **Monitor (from Arduino_RouterBridge environment)**
    - Provides `Monitor.begin()` and `Monitor.print/println()` to send text back over the bridge to the Serial Monitor in App Lab.
- **Core Arduino APIs**
    - `millis()`, `delay()`, and `HEX` for printing.

2. Linux / MPU Side (Python)

2.1 Imports and startup message

```
import time
from arduino.app_utils import App, Bridge
print("Python: Phase 4.4 CAN frame receiver starting", flush=True)
```

What this des

- `from arduino.app_utils import App, Bridge`
  This is the **official Python API** for UNO Q App Lab:
    - `App` controls the lifecycle (start/stop, main loop).
    - `Bridge` lets Python expose and call named functions that map to MCU calls.
- `print(..., flush=True)`
  Makes sure the startup message appears immediately in the App Lab log (no buffering delays).

2.2 Bridge handler for CAN frames

```
def can_frame_v0(version,
                 timestamp,
                 can_id,
                 dlc,
                 b0, b1, b2, b3, b4, b5, b6, b7,
                 flags):
    ext = bool(flags & 0x01)
    data = [b0, b1, b2, b3, b4, b5, b6, b7]

    print(
        f"Linux: CAN frame v{version} "
        f"ts={timestamp}ms "
        f"id=0x{can_id:X} "
        f"{'EXT' if ext else 'STD'} "
        f"dlc={dlc} "
        f"data={[hex(b) for b in data[:dlc]]}",
        flush=True
    )
```

What this does

Signature matches the MCU `Bridge.notify` call exactly:

```
Bridge.notify("can_frame_v0",
              version, timestamp, can_id, dlc,
              data[0]..data[7],
              flags);
```

➪

```
def can_frame_v0(version, timestamp, can_id, dlc,
                 b0, b1, b2, b3, b4, b5, b6, b7,
                 flags):
```

- `ext = bool(flags & 0x01)`
  - Extracts the **extended frame flag**.
  - If bit 0 is set, we treat it as an extended ID.
  - In our test, `flags = 0x01`, so we print `EXT`.
- `data = [b0, b1, ... b7]`
  - Rebuilds the array of data bytes into a Python list for easier handling.
  - Later, this list can be passed into decoding / MF4 writers.
- `data[:dlc]`
  - We only display the bytes up to `dlc`.
  - Even though we always send 8 bytes, this respects the DLC field and matches real CAN semantics.
- The big `print(...)`
  - Builds a human-readable line with:
    - version
    - timestamp in ms
    - CAN ID in hex
    - EXT/STD tag
    - dlc
    - data bytes as hex strings
  - `flush=True` ensures each line appears immediately in the log.

This printout is what you saw:

```
Linux: CAN frame v0 ts=5551ms id=0x18FF50E5 EXT dlc=8 data=['0x3',
'0x11', '0xff', '0xff', '0xff', '0xff', '0x33', '0x0']
```

2.3 Registering the handler and keeping the app alive

```
Bridge.provide("can_frame_v0", can_frame_v0)

def loop():
    time.sleep(0.1)

App.run(user_loop=loop)
```

What this does

- `Bridge.provide("can_frame_v0", can_frame_v0)`
    - Registers the function name `"can_frame_v0"` with the Bridge runtime.
    - When the MCU calls `Bridge.notify("can_frame_v0", ...)`, those arguments are **delivered to this Python function**.
- `def loop(): time.sleep(0.1)`
    - A tiny loop that just sleeps.
    - Its job is **not** to do work, just to keep the Python process alive and responsive.
    - Without this, the script could exit and the App Lab container would shut down.
- `App.run(user_loop=loop)`
    - Starts the application framework.
    - Runs your `loop()` repeatedly in the main thread.
    - Handles incoming Bridge messages in the background.

2.4 Python-side libraries involved

- **arduino.app_utils.App**
  - Manages the Python application lifecycle in UNO Q App Lab.
- **arduino.app_utils.Bridge**
  - Manages RPC message routing:
    - `Bridge.provide("name", func)` registers functions callable from MCU.
    - (We used `Bridge.notify` on the MCU side to call into Python).
- **time**
  - Only used for `sleep` to keep the loop alive at a gentle pace.

---

3. What this validation proves

With this pair of programs running, we have:

- A **known CAN frame** created on the MCU side.
- A **stable 1 Hz** send rate.
- Message fields serialized over the RouterBridge using MsgPack.
- Python receiving and reconstructing:
  - version
    timestamp
  - CAN ID
  - DLC
  - data bytes
  - extended/standard flag
- Matching logs on both sides (MCU & Linux) showing the same CAN ID and data.

This is the **first full end-to-end validation** of the path:

CAN frame (synthetic for now) → MCU → Bridge → Linux → log