HC-SR04 on Teensy 4.1 with ROS 2 (Jazzy): A Low-Jitter, Multi-Sensor Handler

This tutorial walks through a compact HC-SR04 driver, explaining each function from `modules/sensors/sonar.cpp` and `sonar.h`. The goal isn't to be the ultimate solution—it's a small, easy-to-understand demo that shows how to get accurate readings at fast frame rates with low jitter for ROS 2/Nav2.

Audience: mid-level C++ devs comfortable with Arduino/Teensy and ROS 2 Jazzy.

Why HC-SR04 can be tricky (and what we'll fix)

HC-SR04 sensors are simple but fussy:

- You must shape a precise TRIG pulse: minimum 10 µs HIGH.
- The module emits 8 ultrasonic bursts (≈40 kHz), then after a short internal delay (typically a few hundred microseconds, ~0.2–0.5 ms), it raises ECHO.
- If an echo returns, ECHO goes LOW; the HIGH duration encodes round-trip time.
- Spec says max valid echo window is ≈38 ms (~4 m). In practice, many boards hold ECHO HIGH far longer when no echo occurs—I've measured ≈134 ms. You must wait this out before re-triggering.

What makes this hard on a microcontroller running other tasks (serial I/O, motor control, ROS 2 bridging):

- Busy-waiting or using `pulseIn()` blocks and adds jitter.
- Loop delays (logging, math, other peripherals) can skew timing.
- Multiple sensors can cross-talk if triggered too close together.

This demo solves it by:

- Using a hardware timer to run a tiny state machine that triggers one sensor at a time in round-robin and enforces timing.
- Using external interrupts on ECHO edges to timestamp durations precisely.
- Keeping the main `loop()` lightweight so jitter stays low—important because Nav2 performs best with steady, low-jitter sensor frames.

Sensor characteristics to keep in mind

- TRIG minimum high time: 10 µs (no explicit upper bound; we use exactly 10 µs).
- Internal send delay: ECHO rises a few hundred microseconds after TRIG goes LOW (empirically ~0.2–0.5 ms; I often see ~0.475 ms).
- Valid echo window: ≈38 ms for ~4 m round-trip time.
- Observed ECHO high with no echo: ≈134 ms on some boards. Plan for this before re-triggering.
- Distance conversion: `meters = duration_us * 0.000343 / 2.0` (speed of sound ~343 m/s at 20 °C; divide by 2 because the time is round-trip).

A list of devices (multi-sensor support)

Each sensor is described by name and GPIO pins for TRIG and ECHO. From `modules/sensors/sonar.h`:

```
struct Device {
  String name;
  uint8_t trigger_pin;
  uint8_t echo_pin;
  bool enabled = false;
  int32_t countdown_ticks = 0;  // For timing the echo.
  bool echo_found = false;
  int32_t waitout_countdown_ticks = 0;
  float distance = 0.0f;

  Device(String name, uint8_t trigger_pin, uint8_t echo_pin)
      : name(name), trigger_pin(trigger_pin), echo_pin(echo_pin),
enabled(true) {}

  Device() = default;
};
```

Usage sketch:

```
using WimbleRobotics_Teensy::SonarMonitor;
using WimbleRobotics_Teensy::Device;

Device front{"Front", 32, 33};
Device left{"Left", 30, 31};

auto& sonar = SonarMonitor::getInstance();
sonar.configureSensor(0, front);
sonar.configureSensor(1, left);
// Configure up to kMaxSonarSensors (default 4)
```

Pro tip: Configure all devices before `setup()` so pins/interrupts attach correctly.

Tunable timing constants and limits

From `sonar.h`:

```
static constexpr float    max_distance_meters     = 2.2f;     // HC-SR04
usable max
static constexpr uint32_t timer_interval_us       = 10;       // 10 µs tick
⇒ exact TRIG width
static constexpr uint32_t echo_sample_interval_us = 40'000;   // 40 ms
valid echo window
```

```
static constexpr uint32_t max_sample_interval_us  = 150'000; // 150 ms
(covers ~134 ms timeout)
```

- 10 µs timer period lets the state machine raise TRIG HIGH for exactly one tick.
- `echo_sample_interval_us` is the max time we consider a “valid” echo.
- `max_sample_interval_us` is the hard wait-out before re-triggering if no echo.

Function-by-function walkthrough

All functions live in `modules/sensors/sonar.cpp` unless noted.

1) Singleton and constructor

```
SonarMonitor* SonarMonitor::instance_ = nullptr;

SonarMonitor& SonarMonitor::getInstance() {
  if (!instance_) instance_ = new SonarMonitor();
  return *instance_;
}

SonarMonitor::SonarMonitor() : Module() {
  for (uint8_t i = 0; i < kMaxSonarSensors; ++i) {
    devices_[i].enabled = false;
  }
}
```

- A lightweight singleton; constructor disables all sensors by default. You enable/configure only the ones you use.

2) Adding sensors

```
void SonarMonitor::configureSensor(uint8_t index, const Device& device) {
  if (index < kMaxSonarSensors) {
    devices_[index] = device;
  }
}
```

- Copy the `Device` into the internal array. Typical pattern: build all `Device`s (names + pins), then call this per index.

3) Hardware setup: pins, interrupts, timer

```
void SonarMonitor::setup() {
  for (uint8_t i = 0; i < kMaxSonarSensors; ++i) {
    if (devices_[i].enabled) {
      pinMode(devices_[i].trigger_pin, OUTPUT);
      pinMode(devices_[i].echo_pin, INPUT);
      attachInterrupt(digitalPinToInterrupt(devices_[i].echo_pin),
                      echo_interrupt_handler, CHANGE);
    }
  }
  Timer1.initialize(timer_interval_us);
  Timer1.attachInterrupt(timerInterruptHandler);
}
```

- TRIG pins are outputs, ECHO pins are inputs.
- A single ISR is attached to every ECHO pin (see “Alternatives” for a per-pin version). We watch both rising and falling edges.
- A hardware timer ticks every 10 µs and drives the state machine.

Why timer-driven? So `loop()` delays can never stretch TRIG pulses or echo windows. This is the key to low jitter.

4) The timer ISR state machine

```
enum class State { PULSE_HIGH, PULSE_LOW, COUNTDOWN, WAIT_OUT_ECHO };

void SonarMonitor::timerInterruptHandler() {
  SonarMonitor& instance = SonarMonitor::getInstance();

  switch (instance.state_) {
    case State::PULSE_HIGH: {
      // pick next enabled sensor round-robin
      for (int i = 0; i < kMaxSonarSensors; ++i) {
        instance.current_sensor_index_ =
            (instance.current_sensor_index_ + 1) % kMaxSonarSensors;
        if (instance.devices_[instance.current_sensor_index_].enabled)
break;
      }
      if (!instance.devices_[instance.current_sensor_index_].enabled)
return;


digitalWrite(instance.devices_[instance.current_sensor_index_].trigger_pin
, HIGH);
      instance.state_ = State::PULSE_LOW; // hold HIGH exactly one 10 µs
tick
    } break;

    case State::PULSE_LOW: {
      auto& d = instance.devices_[instance.current_sensor_index_];
      digitalWrite(d.trigger_pin, LOW);
      d.countdown_ticks = echo_sample_interval_us / timer_interval_us; //
40 ms window
      d.echo_found = false;
      d.waitout_countdown_ticks =
          (max_sample_interval_us - echo_sample_interval_us) /
timer_interval_us; // to 150 ms
      instance.state_ = State::COUNTDOWN;
    } break;

    case State::COUNTDOWN: {
      auto& d = instance.devices_[instance.current_sensor_index_];
      if (d.echo_found) { instance.state_ = State::PULSE_HIGH; break; }
      if (--d.countdown_ticks > 0) break;      // still waiting in valid
window
      instance.state_ = State::WAIT_OUT_ECHO;   // no echo yet, wait out
remainder
    } break;

    case State::WAIT_OUT_ECHO: {
      auto& d = instance.devices_[instance.current_sensor_index_];
      if (d.echo_found) { instance.state_ = State::PULSE_HIGH; break; }
      if (--d.waitout_countdown_ticks > 0) break; // finish to 150 ms

      // Mark no-echo case (clamped distance)
      d.echo_found = false;
      d.distance = max_distance_meters;
      instance.state_ = State::PULSE_HIGH;      // next sensor
    } break;
  }
}
```

- PULSE_HIGH: assert TRIG for exactly 10 µs.
- PULSE_LOW: drop TRIG, open a 40 ms “valid echo” window, and start the longer wait-out.
- COUNTDOWN: if echo arrives, move on; else continue counting down.
- WAIT_OUT_ECHO: finish waiting to ~150 ms so we never re-trigger while ECHO might still be HIGH on certain boards.

This tightly bounds when we can fire the next sensor—preventing cross-talk and keeping frame cadence regular.

5) Echo edge handler (shared ISR)

```
void SonarMonitor::handleEcho() {
  Device& device = devices_[current_sensor_index_];
  if (!device.enabled) return;

  uint8_t level = digitalRead(device.echo_pin);
  if (level == HIGH) {
    echo_start_time_ = micros();
    is_echo_active_ = true;
  } else {
    if (is_echo_active_) {
      device.echo_found = true;
      unsigned long echo_duration = micros() - echo_start_time_;
      device.distance = echo_duration * 0.000343 / 2; // meters
      is_echo_active_ = false;
    } else {
      // Spurious LOW edge; ignored
    }
  }
}

void SonarMonitor::echo_interrupt_handler() {
  if (instance_) instance_->handleEcho();
}
```

- Rising edge: remember start time; falling edge: compute distance.
- We attach the same ISR to all ECHO pins, but only the "current" sensor should be active. This keeps code small, at the cost of relying on scheduling to ensure edges belong to the current sensor.

Improvement idea: Use one ISR per pin, or pass pin identity to the handler and map pin→Device. That removes any ambiguity if noise or wiring produces stray edges on a different ECHO pin.

6) Reporting in the main loop

```
void SonarMonitor::loop() {
  static unsigned long last_print_time = 0;
  if (millis() - last_print_time > 200) {
    for (int i = 0; i < kMaxSonarSensors; ++i) {
      auto& d = devices_[i];
      if (d.enabled) {
        String msg = "Sonar " + String(d.name) + " distance: " +
String(d.distance) + " meters";
        SerialManager::getInstance().sendMessage("SONAR", msg.c_str());
      }
    }
    last_print_time = millis();
  }
}
```

- The loop does non time-critical work—just periodic reporting. The timer ISR/state machine and echo ISRs own all the tight timing.
- If you use ROS 2 on a companion computer, parse these lines and publish `sensor_msgs/Range` (one topic per sensor). Nav2 appreciates steady, low-jitter updates.

7) Accessor for consumers

```
float SonarMonitor::getDistance(uint8_t idx) const {
  if (idx < kMaxSonarSensors) return devices_[idx].distance;
  return max_distance_meters;
}
```

- Lets other modules read the latest computed distance.

Why a timer-driven state machine helps

- Consistent TRIG width: exactly 10 µs regardless of what `loop()` is doing.
- Regular cadence: each sensor is serviced in a predictable round-robin, minimizing jitter.
- Single-sensor activity: by waiting out long ECHO highs, we avoid re-triggering while the module is still "listening," which prevents cross-talk and phantom hits.
- Loop isolation: serial prints or other tasks won't skew measurements.

This lines up with Nav2's needs: the local planners and cost-map updates behave better with low-jitter sensor frames.

Alternative approaches and trade-offs

- Simple `pulseIn()` in `loop()`
  - Easiest to write but blocks and adds jitter. Hard to scale beyond one sensor.
- Micros-based scheduler in `loop()` (no timer ISR)
  - Use `micros()` deadlines to shape TRIG and check ECHO via pin polling or pin-change interrupts. Less ISR load, but you must be disciplined to avoid blocking work.
- One ISR per ECHO pin

  - More robust mapping (pin → sensor). Slightly larger code footprint. Recommended for production.

  - Pattern example (conceptual):

    ```
    // In header:
    void handleEchoFor(uint8_t idx);

    // In setup(): attach a dedicated ISR for each enabled sensor
    for (uint8_t i = 0; i < kMaxSonarSensors; ++i) {
      if (!devices_[i].enabled) continue;
      auto pin = devices_[i].echo_pin;
      // Create a small trampoline that remembers the index
      attachInterrupt(digitalPinToInterrupt(pin), [] {
        SonarMonitor::getInstance().handleEchoFor(/* idx known at
    bind time */);
      }, CHANGE);
    }

    // Implementation:
    void SonarMonitor::handleEchoFor(uint8_t idx) {
      Device& d = devices_[idx];
      uint8_t level = digitalRead(d.echo_pin);
      if (level == HIGH) { echo_start_time_ = micros();
    is_echo_active_ = true; }
      else if (is_echo_active_) {
        d.echo_found = true; unsigned long dt = micros() -
    ```

```
echo_start_time_;
    d.distance = dt * 0.000343f / 2.0f; is_echo_active_ = false;
  }
}
```

Note: Arduino attachInterrupt requires a function pointer (not a capturing lambda). In production, implement per-index static trampolines or a small table of ISR thunks to call `handleEchoFor(idx)`.

- Hardware input-capture (Teensy FlexPWM/QuadTimers)
  - Highest precision timing with minimal CPU overhead; most complex to set up.
- Longer `timer_interval_us`
  - E.g., 50–100 µs or 1 ms with careful deadlines. Reduces ISR rate at the cost of pulse-width granularity.

Practical notes and small improvements

- Use level shifting on ECHO for 3.3 V MCUs (Teensy 4.1). Many HC-SR04 ECHO pins are 5 V.
- Consider replacing Arduino `String` with fixed buffers to avoid heap fragmentation on embedded targets.
- Timestamp readings and add a simple validity flag; consumers can ignore stale data.
- If you see occasional spurious ISR triggers, debounce edges in software (e.g., ignore pulses shorter than a few tens of microseconds) or move to per-pin ISRs.
- Beam shape is wide (~15°); surfaces at angles can yield weak or multi-path echoes—expect occasional misses. The wait-out logic prevents those misses from disturbing cadence.

ROS 2 (Jazzy) integration sketch

Bridge example on the host:

- Read serial lines like `SONAR:name=Front,meters=0.73`.
- Publish `sensor_msgs/Range` with:
  - radiation_type = ULTRASOUND
  - field_of_view appropriate to your transducer (~0.26 rad for ~15°)
  - min_range ≈ 0.02 m, max_range ≈ 2.2 m
  - header.frame_id per sensor: `sonar_front`, `sonar_left`, …
- Use one topic per sensor: `/sonar/front`, `/sonar/left`, …
- Publish on new readings; avoid fixed-rate timers to keep jitter low.

Nav2 tip: Keep the frame rate steady (e.g., 10–20 Hz per sensor is plenty) and avoid bursts. The timer-driven approach here makes that straightforward.

Companion script (added in this repo):

- `code/ultrasonic/sonar_serial_bridge.py`
  - Reads serial lines beginning with `SONAR:` or human-readable messages like `Sonar <name> distance: <meters> meters`.

- Publishes `sensor_msgs/Range` per sensor to `/sonar/<name-lowercase>`.
- Parameters: `--port /dev/ttyACM0` (macOS example: `/dev/tty.usbmodem*`), `--baud 115200`.

What this demo is (and isn't)

- Is: small, readable example showing how to achieve accurate readings with fast frame rates and low jitter, handling one or more sensors safely.
- Isn't: a production-grade, fully fault-tolerant system. You can improve it with per-pin ISRs, hardware capture, temperature compensation, filtering, and robust error handling.

Quick checklist to adapt this code

- Define your sensors (names + TRIG/ECHO pins) and call `configureSensor()` for each.
- Ensure ECHO level shifting to 3.3 V.
- Verify timer tick (10 µs) and echo windows (40 ms valid, 150 ms max) suit your boards.
- Keep `loop()` light; do heavy work off the ISR path.
- On the ROS side, parse serial and publish `sensor_msgs/Range` with consistent timestamps.

Happy pinging!