ChatGPT

Tempo Tracking System Analysis & Improvement Recommendations

Overview of the Current Tempo Tracking System

The existing tempo tracker uses a classic onset-driven beat detection pipeline tailored for real-time use on an ESP32-S3. Audio is captured via an I2S microphone at 16 kHz and processed in short hops (256 samples ≈ 16 ms). On each hop, an **onset detection function** computes a *novelty signal* from the audio: it combines **spectral flux** (energy changes in 8 frequency bands) and the **RMS energy derivative** (overall volume spikes) [1] [2]. This novelty signal highlights moments of significant change (e.g. drum hits). Detected onsets feed into an **inter-onset interval (IOI) analysis** stage that estimates the repeating beat period. A **phase-locked loop (PLL)**-style tracker then maintains the beat phase over time, generating beat "tick" events in sync with the music.

Key implementation details and parameters in the current system:

- **BPM Range**: Tracks tempos between ~70 and 190 BPM (1.17–3.17 Hz) [3], covering most musical tempos. (This range prevents absurdly low or high tempo outputs.)
- **Onset Detection**: 8-band spectral flux is weighted toward low frequencies (e.g. the lowest band weight 1.4 vs highest band 0.3) to emphasize drums [4]. Positive spectral changes are summed and normalized per frame, and sudden RMS increases are captured separately as transient peaks [5] [6]. Both are squared to accentuate large onsets [7] [8]. The novelty history is continuously decayed (0.999 per frame) so old onsets gradually fade [9] [10]. This yields a running "onset strength vs. time" curve.
- **Periodic Tempo Analysis**: Instead of explicitly measuring gaps between individual onsets, the system uses a **Goertzel filter bank** (effectively a set of narrowband resonators) to evaluate multiple tempo hypotheses in parallel [11] [12]. There are 120 tempo bins (1 BPM resolution) from 70 up to 189 BPM. Each tempo bin's filter "resonates" if the novelty signal contains a periodic component at that tempo. The filters are updated incrementally: only 2 tempo bins are processed per 16 ms frame (spreading computation across frames) [13]. Over ~0.5 s, all bins are computed once.
- **Tempo Selection**: After each update, the filter magnitudes indicate which tempo has the strongest rhythmic presence. Magnitudes are normalized and **smoothed with an EMA** (e.g. 97.5% previous + 2.5% new) for stability [14] [15]. A winner is chosen with hysteresis – the leading tempo must exceed the current one by 10% for 5 consecutive frames to switch [16] [17]. This prevents tempo "flapping." The chosen tempo's confidence is quantified as `max_mag / sum_of_mags` (so ambiguous signals give lower confidence) [18] [19]. The system reports the BPM, a phase (position within the beat), a normalized beat strength, and a locked flag when confidence > 0.3 [20].
- **Phase Tracking**: The selected tempo's phase is advanced each audio frame to keep time [21]. Detected beat onsets align the phase via the resonator: the Goertzel analysis computes the phase of the strongest periodic component [22]. A beat "tick" is declared whenever the phase crosses through zero (i.e. start of a beat) [23]. To avoid rapid re-triggers on the next 16 ms hop, any tick that occurs too soon (<60% of the expected beat period since the last tick) is suppressed [24]. This acts as a debounce, ensuring stability against spurious closely-spaced onsets.

Overall, this architecture is optimized for real-time (<100 ms latency) and runs within the memory/CPU constraints (e.g. ~22 KB for tempo buffers) [25] [26]. It responds quickly to changes and produces steady beat ticks for use in lighting effects, etc. However, as observed, it can sometimes **lock onto half the true tempo** of the music, especially in steady drum patterns. We will analyze why this half-tempo error occurs and propose improvements in onset processing, interval analysis, and tempo disambiguation to enhance accuracy without sacrificing responsiveness or stability.

Root Cause of Half-Tempo Locking

One common failure mode in beat trackers is an **octave error**, where the algorithm settles on a tempo that is half or double the actual beat rate. In our case, the system may output roughly half the true BPM (e.g. reporting ~76 BPM for music at 152 BPM). The primary cause is related to **which onsets the tracker detects and emphasizes**:

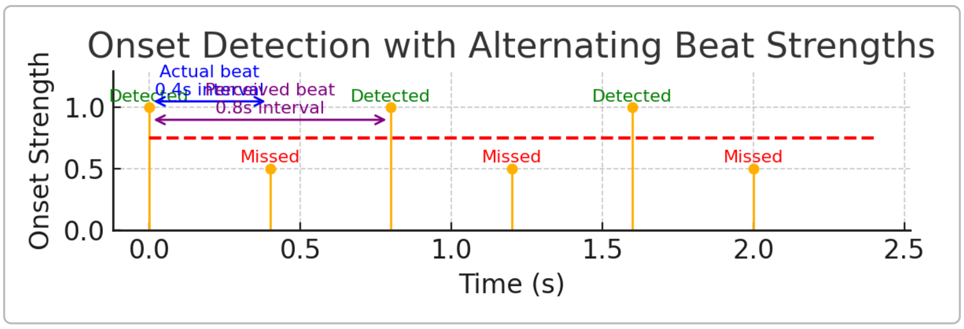

Illustration: If onset detection misses or de-emphasizes every other beat, the system perceives a longer interval and locks to half tempo. In this example, actual drum hits occur at 0.4 s intervals (150 BPM). Weak beats (red) fall below the detection threshold and are "missed," so only the strong beats (green) at 0.8 s spacing register as onsets. The tracker interprets 0.8 s as the beat period (75 BPM) instead of the true 0.4 s (150 BPM).

As the diagram shows, **uneven detection of alternating beats** can trick the system into a half-rate tempo. Several factors in the current design contribute to this effect:

- **Thresholding and Normalization of Onsets**: The novelty signal combines large low-frequency onsets (kick drum, etc.) and smaller high-frequency or soft onsets (hi-hats, off-beats). Because the spectral flux is weighted heavily toward bass [4], a powerful kick on beat 1 produces a large novelty spike, while a weaker snare or hi-hat on beat 2 might produce a much smaller peak. The dynamic normalization (scaling novelty so the max ~0.5) can further amplify the contrast – after a huge spike, the scale factor reduces, making subsequent small onsets almost negligible [27]. In effect, **the strongest beats dominate the novelty curve**, and intermediate beats might barely register. The result is a periodic pattern in the novelty like "big spike, small spike, big spike, small spike." The Goertzel bank will resonate strongly at a period of two hops of this pattern. The **filter at half the true tempo (double the period)** sees all the big spikes aligning with its expected beat positions and ignores that the small spikes in between even exist (since they contribute little energy). Meanwhile, the filter at the true tempo gets a alternating strong-weak input, which may yield a lower average magnitude. This can lead the slower tempo bin to win out.
- **Beat Debounce and PLL Behavior**: The phase tracker intentionally suppresses detecting a second beat if it arrives too soon (within 0.6 of the expected period) [28]. This is normally a guard against double-triggering on things like an echo or a rapid transient. But if the tracker is already misaligned to a half tempo, a true mid-interval beat could fall into this "debounce" window and be ignored as an "early" beat. Essentially, once locked to the wrong tempo, the system might

refuse to consider the in-between beats as valid ticks because they arrive earlier than the assumed period. This further reinforces the half-tempo lock.

- **Lack of Explicit Secondary Beat Tracking**: The current algorithm picks a single best tempo. If two tempos (say 150 BPM and 75 BPM) are both plausible, the approach doesn't explicitly model both – it will choose the one with slightly higher resonance magnitude. There is a hysteresis requiring ~10% stronger signal for a new tempo to overthrow the old [16], which is great for stability but means if the half-tempo solution is even slightly favored initially, it will **stick**. The true tempo might be the second-strongest bin and remain just below the hysteresis threshold due to the uneven onset strengths. The confidence metric will be low (because energy is split between two competing tempo hypotheses), but the system will still output the winner and mark itself "unlocked" or low-confidence rather than automatically correcting the octave error.

In summary, the **root cause** is that the onset detection and filtering aren't fully capturing the *every-beat* pattern. Strong beats overshadow weak beats, causing the periodicity detector to latch onto a pattern that treats every other beat as primary. Once the system locks to the wrong pattern, its internal phase logic and hysteresis make it reluctant to switch to the true tempo unless a significant change occurs.

Improving Onset Detection & Interval Filtering

To address the half-tempo bias at its source, we should improve how onsets are detected and filtered before tempo estimation. The goal is to ensure *all actual beats* (even weaker ones) contribute appropriately to the tempo calculation, so the tracker doesn't systematically skip every second beat. Several strategies can help:

- **Adjust Frequency Weighting and Onset Thresholding**: Currently, low frequencies are heavily favored (weights 1.4, 1.3 for the two lowest bands) [4]. This makes sense since kick drums often drive the beat. However, if off-beat events are mostly high-frequency (e.g. a hi-hat on every 8th note) the tracker might miss them. *Recommendation:* Slightly rebalance the spectral flux weights or apply a floor to ensure higher-frequency band spikes contribute at least a little. For example, instead of near-zero weight (0.3) on the highest band, use a minimum of 0.5 – 0.6. This can help the novelty function register a consistent pulse when cymbals or snares mark the off-beats. Additionally, consider a **dual-threshold onset detection**: detect "strong" onsets with a high threshold (for kicks) and "weak" onsets with a lower threshold. The system could treat a sequence of weak onsets (e.g. tick-tick-tick-tick of a hi-hat) as evidence of a faster underlying tempo even if each one individually is small. Concretely, you might maintain two novelty curves – one aggressively normalized for big transients, and another more sensitive to quieter ticks – and feed both into the tempo analysis (or give the sensitive one a smaller weight). This would reduce the chance that an entire beat is dropped from consideration just because it was soft.
- **Explicit Peak Picking and IOI Analysis**: Instead of relying solely on the continuous Goertzel resonance, we can incorporate an **explicit IOI tracking** step on detected peaks. That is, identify discrete onset times from the novelty curve (e.g. whenever spectral_flux or VU delta exceeds an adaptive threshold). Then measure the intervals between successive onsets. These raw IOIs can be filtered (e.g. ignore very short intervals that are likely subdivisions or false triggers, and look for a mode or average of the dominant interval). An exponential moving average (EMA) or median of recent IOIs can provide a quick tempo estimate. The current system already uses a form of smoothing for tempo; we can strengthen this by rejecting outlier intervals. For example, if we have recent intervals of {430 ms, 440 ms, 820 ms}, it's likely 820 ms is an outlier (double the others), so we'd treat 430–440 as the true beat gap. Implementing this could be as simple as:

```
// Pseudocode: basic IOI filtering
onsets = detectOnsetTimes(novelty_curve);
```

```
if (onsets.size() >= 2) {
    float interval = onsets.back() - onsets[onsets.size()-2];
    if (fabs(interval - EMA_interval) < 0.2*EMA_interval) {
        EMA_interval = 0.8*EMA_interval + 0.2*interval;  // smooth
update
    } else {
        // If interval deviates a lot (e.g. ~2x), consider it a missed
or extra beat.
        // Optionally, ignore or half it if preceding intervals suggest
that.
    }
}
```

This kind of logic can dynamically correct if a beat is skipped: a sudden doubling of interval can be caught and interpreted as “probably two beats apart, not one.” By feeding such interval estimates into the tempo tracker (or directly into the MusicalGrid PLL), you provide a sanity check against the continuous resonance method. Essentially, **use the pattern of recent onset timings to validate tempo** – if the computed tempo suggests 800 ms period but you just saw two onsets ~400 ms apart, the system can favor the 400 ms period.

- **Multi-Scale Onset Analysis**: Human rhythms often have hierarchical pulses (e.g. strong downbeats and weaker upbeats). The tracker could mimic this by maintaining two parallel tempo estimates: one for a *tactus* (the main beat) and one for a *sub-beat*. For instance, run the Goertzel analysis on the novelty curve as now (which tends to favor the strongest metrical level), and in parallel run a faster-tempo analysis on a **high-pass filtered** version of the novelty (to catch rapid ticks like hi-hats or strums). The high-pass novelty (or even just the high-frequency bands’ flux) might resonate at 150 BPM when the full-band novelty resonates at 75 BPM. If a discrepancy like this is observed, that’s a clue the true tempo might be the higher one with alternating weak/strong emphasis. One could then boost the confidence of the higher tempo or directly resolve to it. This hybrid approach ensures that fast, subtle rhythmic content isn’t completely overshadowed by slower strong beats.
- **Sliding Window Autocorrelation**: As an alternative to peak-picking, you can perform an **autocorrelation** on the novelty signal over a short window (e.g. the last 4–8 seconds) to directly measure periodicity. Autocorrelation will produce a peak at the lag corresponding to the tempo. In a half-tempo scenario, you’ll see a strong peak at lag ≈ 800 ms (half tempo) *and* a secondary peak at ≈400 ms (true tempo). The relative heights of these peaks tell you if every beat or every other beat is more prominent. By examining the ratio of these autocorr peaks, you can decide which tempo is more likely correct. For example, if the 2× tempo lag has, say, 70% of the energy of the 1× lag, it suggests a consistent intermediate pulse exists (even if softer). You could then favor the faster tempo. This could be implemented efficiently using the existing data structures (e.g. reuse the novelty history buffers). A simple approach: take the novelty curve (perhaps the combined hybrid input as used by Goertzel) and compute autocorr for lags corresponding to 60–180 BPM. Pseudocode:

```
// Pseudocode for autocorrelation-based tempo estimate
int minLag = int(60.0 / maxBPM * rate);
int maxLag = int(60.0 / minBPM * rate);
float bestCorr = 0; int bestLag = 0;
float secondCorr = 0; int secondLag = 0;
for (int lag = minLag; lag <= maxLag; ++lag) {
    float corr = 0;
    for (int i = 0; i < N - lag; ++i) {
        corr += novelty[i] * novelty[i+lag];
    }
    if (corr > bestCorr) {
        secondCorr = bestCorr; secondLag = bestLag;
        bestCorr = corr; bestLag = lag;
    } else if (corr > secondCorr) {
        secondCorr = corr; secondLag = lag;
    }
}
float bpm_est = 60.0 / (bestLag * seconds_per_sample);
```

This brute-force approach may need optimization (the above is O(N·maxLag) which on 512-length N is fine). The key idea is we get the top two lag candidates. If `secondLag` is ~2× or 0.5× of `bestLag` and `secondCorr` is close to `bestCorr`, we have a half/double ambiguity. We can then apply a rule (e.g. choose the faster tempo unless the slower is significantly stronger). This **direct autocorrelation method** can serve as either a cross-check or even a replacement for the Goertzel bank. It might be simpler to tune: for example, you can clearly see multiple peaks and make a decision, rather than relying on subtle magnitude differences spread across dozens of filter bins.

By improving onset detection and interval analysis in these ways, the system will be much less likely to "skip" beats in its internal representation. In practical terms, that means if the song has a steady 4/4 kick-snare pattern, the tracker will pick up both the kick and snare as part of the beat, rather than only the louder kick. This forms a stronger foundation for the next stage: deciding the correct tempo.

Enhancing Tempo Confidence and Octave Error Correction

Even with better onset capture, the system should explicitly guard against picking the wrong tempo octave. Here we introduce measures to **detect and correct half/double tempo scenarios** by using confidence metrics and comparative checks between tempo candidates:

- **Refine the Confidence Metric**: The current confidence is computed as `max_tempo_magnitude / sum_all_magnitudes` [18]. In a half-tempo situation, the energy is split between the true tempo bin and its neighbor (the subharmonic), yielding a modest confidence (often 0.2–0.4). A low confidence value is essentially a red flag that the tempo is ambiguous or unstable. We can leverage this by adjusting the logic when confidence is low. For example, if `confidence < 0.5` and the chosen tempo is an exact multiple of another strong candidate (2× or 0.5×), we pause and examine further. This might mean looking at the raw novelty or onset pattern as described earlier. Concretely:

```
if (confidence < 0.5) {
    uint16_t halfBin = winner_bin_/2;
    uint16_t doubleBin = winner_bin_*2;
    if (halfBin in range && tempo_magnitude[halfBin] > 0.8 *
tempo_magnitude[winner_bin_]) {
        // Likely actual tempo is half of current guess
    }
    if (doubleBin in range && tempo_magnitude[doubleBin] > 0.8 *
tempo_magnitude[winner_bin_]) {
        // Likely actual tempo is double the current guess
    }
}
```

In the above, if an alternate tempo at half or double has, say, ≥80% of the energy of the current winner, we suspect an octave error. The system could then **boost the confidence** of the faster tempo or even flip to that tempo if certain conditions are met (possibly bypassing the normal hysteresis delay because we know it's a special case). Essentially, we treat close competing peaks at harmonic ratios as evidence that we've locked on the wrong multiple.

- **Harmonic Ratio Checks**: Many modern tempo estimation algorithms explicitly handle metrical "octave" errors by evaluating both 2× and 1/2× of the detected period [29] [30]. We can implement a lightweight version of this. After finding the best tempo, **always check one level up and down**:
- Compute what the next higher tempo (double frequency) bin's smoothed magnitude is, and what the next lower (half frequency) bin's magnitude is.
- If one of those is comparably strong, there's ambiguity. Decide based on additional heuristics: for instance, prefer the faster tempo if the music is percussion-heavy with a strong subdivision (since users typically tap the faster pulse in such cases), or perhaps prefer the slower if the faster candidate is above, say, 180 BPM (which might indicate a double-count of a moderate tempo). In practice, a safe default is to **choose the higher BPM when in doubt** because a strong, steady drumline is more often counted at the faster rate. (Many DJ and analysis softwares do this – better to double-count a slow song than half-count a fast song.)
- Another heuristic: examine the phase alignment (`phase_inverted` flag in the code hints at alternating phase flips [31]). If the winning tempo's phase flips every time a new onset is integrated, that means onsets are consistently arriving halfway through its cycle (i.e. every other beat). That is a clear sign the true tempo is double. The code already toggles a `phase_inverted` boolean when phase wraps by π [32] – this could be used to detect an alternating-beat pattern. If `phase_inverted` keeps toggling on each beat update for the winner, you're locking to an off-beat cycle. In such a case, we could automatically double the tempo and reset the phase.
- **Stability vs. Adaptivity Tuning**: The tracker currently uses hysteresis (10% & 5 frames) [16] and heavy smoothing (time constant ~0.4 s for magnitudes) [14] [15] to avoid jitter. To improve accuracy, especially when recovering from a wrong tempo lock, we might make the system a *bit* more reactive:
- Reduce the hysteresis frames or threshold when confidence is low. E.g., if confidence < 0.3, allow a tempo switch with only 2 frames of confirmation or a smaller 5% advantage. This way, if the system is unsure and a better candidate emerges (like the true tempo coming into focus), it can switch faster. Conversely, when confidence is high (clear stable beat), keep hysteresis strong to prevent oscillation.
- Speed up the magnitude smoothing slightly (increase the 0.025 update fraction to 0.05, for example) for new tempo candidates. This helps the system latch onto a new pattern faster. Remember, <100 ms latency is a priority; a very slow smoothing can introduce a noticeable lag in tempo updates. We can adjust the smoothing alpha dynamically: if a tempo change is detected, temporarily weight recent data more to lock quickly, then revert to strong smoothing once locked.

- The PLL (phase tracker) in the MusicalGrid already smooths BPM changes over ~0.5 s by default [33], which is good. But we might shorten that a bit for quicker response, say 0.3 s, given that we still have the tempo tracker's own smoothing in front. We should also ensure the PLL's `phaseCorrectionGain` is high enough to pull the phase if we mis-phase by half a beat. Currently it's 0.35 [34]; raising it to ~0.5 could help snap the downbeat alignment once the tempo is corrected (though too high could cause overshoot). These are minor tuning details, but they contribute to how quickly and stably the system corrects itself.

- **User Feedback and Lock Conditions**: Since the system outputs a `locked` flag when confidence > 0.3 [35], consider using that in the application logic. For instance, if `locked` is false (tempo ambiguous), you might briefly double-check the outputs or even run a secondary analysis (like autocorrelation as above) just at that moment to decide if the tempo should be doubled. You could even expose a debug metric – e.g., flash an LED or log when the tempo confidence is low – to indicate the system is guessing. This won't fix the tempo internally, but it can help in development to fine-tune the above strategies and know when they are needed.

Incorporating these measures will make the tempo estimation more robust. The system will effectively **consider both the detected tempo and its nearest harmonic tempos** and use the onset pattern consistency to choose the correct one. By catching half-tempo locks in the act (low confidence, phase alternation, strong 2× competitor, etc.), the tracker can correct itself quickly, fulfilling the accuracy requirement without sacrificing much stability.

Considering an Autocorrelation-Based or Hybrid Model

Given the constraints (no heavy ML, but some FFT/memory is acceptable), a switch to an autocorrelation-based approach is worth evaluating. The current Goertzel filterbank is essentially performing a *frequency-domain* analysis of the onset pattern. A *time-domain* autocorrelation or comb filter approach can achieve similar results with potentially simpler logic:

- **Autocorrelation Approach**: As described earlier, autocorrelation involves taking the novelty function over a window (say the last 8 seconds) and computing how self-similar it is at different time shifts. This directly reveals the periodicities. The advantage of autocorrelation is that it doesn't require maintaining dozens of filter states and it naturally highlights integer multiples of a fundamental period. Implementation on an ESP32-S3 can be efficient if using FFT convolution (since autocorrelation can be done via an FFT of the signal multiplied by its complex conjugate in frequency domain). However, even a direct computation over 512-sample windows is not too bad. Since you likely already compute an FFT (for the 8-band novelty), you could reuse those results to get an autocorrelation (by inverse FFT of the power spectrum, for example). The result is a **tempogram** (energy vs. tempo). Picking the tempo from an autocorrelation tempogram is straightforward: find the peak within the expected range. Many algorithms then apply octave smoothing or choose a faster tempo if multiple peaks are close. This could greatly simplify the code – no need for maintaining `tempi_smooth_`, `phase_inverted`, etc. You would just produce a tempo estimate each analysis window. The downside is potential latency (needing a full window of data) and maybe more jitter if the window is short. A hybrid solution is to use autocorrelation for initial tempo acquisition (e.g. first 2 seconds of music) to quickly guess the right tempo, then switch to the PLL/Goertzel method for continuous tracking (as it can update every 16 ms and adjust phase continuously). After initial lock, you could occasionally re-run the autocorr on a sliding window to verify the tempo hasn't drifted or to catch a change in BPM.
- **Comb Filter / Resonating Comb**: Another approach related to autocorrelation is using a **comb filter bank** – which is quite similar to the Goertzel method but conceptually simpler for some. A comb filter for a given tempo reinforces that periodicity by constructive interference of equally spaced events. Some research (e.g. Davies & Plumbley's work) combined comb filters with onset detection for tempo estimation. You might consider a simplified comb filter approach: for each candidate tempo, sum the onset strength at expected beat positions (this is almost what Goertzel does with sine waves). The difference is you could allow a little slack (like ±10% drift)

when summing, to account for timing jitter. This might be more complex to implement than the current method, so it's an option if other approaches fail.

- **Hybrid Emphasis on Metrical Levels**: The prior *EmotiscopeEngine* that was tested (mentioned in development notes) had a notion of *metrical parity* or examining alternate beats. We can borrow the spirit of that in a simpler way: for each tempo you consider, also consider a **"doubled" tempo** (next metrical level up) and see if novelty events fall into that pattern. One practical hybrid solution:
- Run the tempo tracker as is to get an initial tempo `T`.
- Compute a secondary tempo `T_alt = (T <= 120 BPM ? 2*T : T/2)` – i.e. flip to the other likely octave (if current tempo is on the low side, double it; if it's very high, consider half).
- For a short duration (say 1–2 seconds), explicitly score how well the onsets align to `T_alt` versus `T`. For example, reset two phase accumulators, one ticking at `T` and one at `T_alt`. Each time an onset is detected in the novelty, increment a score for whichever phase was nearer to a beat boundary at that moment. If after a while the `T_alt` score is significantly higher, switch to `T_alt` as the true tempo. This is a time-domain check that essentially asks: "do onsets occur on every beat of T, or mostly every other beat (meaning 2*T is better)?"

- This hybrid method uses the existing system for continuity but adds a verification loop that is triggered when suspicious of an octave error.

- **Compute & Memory Trade-offs**: Replacing the current architecture entirely with autocorrelation might simplify the conceptual model, but be mindful of memory. The current history buffers are length 512 for novelty (~8–16 s) [36]. Autocorrelation of that length is fine, but doing it every 16 ms might be too slow. You could downsample the novelty for autocorr (the novelty is at 62.5 Hz or 31.25 Hz effectively). Indeed, the spectral novelty is at ~31 Hz, so 512 points covers ~16 s – an autocorr at that rate is cheap. The VU novelty is 62.5 Hz with 512 points (~8 s). You could combine them or use one. An **FFT-based autocorr** every 0.5 s or so would probably be fast enough (512-point FFT is trivial on ESP32 with FFT hardware or even software). So a possible design is: every 0.5 s, compute the autocorrelation tempogram from the novelty buffer and update the BPM estimate from that, while continuing the short-term onset and phase tracking for beat alignment. This gives a nice balance of *fast reaction* (phase tracking every hop) and *robust global tempo estimation* (periodic autocorr to correct drift or octave errors).

In summary, switching to an autocorrelation or hybrid approach is feasible and could reduce half-tempo errors by design, since autocorrelation inherently shows the true period and its multiples. The decision to switch should consider how much time you have for development and testing – the current Goertzel method is already implemented and real-time, so it might be most efficient to enhance it (as per previous sections) rather than rewriting everything. However, conceptually, **autocorrelation is easier to reason about** (especially for someone with minimal DSP background): it's basically "find the repeat interval that makes the onset pattern line up with itself." If clarity and maintainability are priorities, a well-documented autocorrelation method with a few simple heuristics for octave selection could be a win.

Tuning Parameters and Code Adjustments

Finally, here is a consolidated list of tuning changes and code-level recommendations to implement the above improvements. These aim to maintain real-time performance while boosting stability and accuracy:

- **Onset Detection Tuning**: In `updateNovelty()`, consider reducing the disparity in band weights [4]. For example:

```
static const float WEIGHTS[8] = {1.2f, 1.1f, 1.0f, 0.8f, 0.7f, 0.5f,
0.5f, 0.5f};
```

This gives more weight to higher bands (preventing high-frequency beats from vanishing). Monitor CPU – these are just multipliers, negligible cost. If certain instruments still slip through, implement a minimum novelty threshold for considering something a beat. E.g., only log spectral_flux if it exceeds some fraction of recent average noise. This avoids too many false onsets when increasing sensitivity.

- **Dual-Stream Novelty (if implementing)**: Add buffers for a "weak onset" novelty stream. This could be as simple as skipping the squaring for one of the streams (since squaring exaggerates differences). The weak-onset stream = linear spectral flux + a smaller RMS delta, normalized separately. Feed both into tempo analysis by summing their contributions or running separate Goertzel banks if memory allows. A cheaper trick is to take the existing combined novelty and raise it to a smaller power (e.g. square-root) to compress dynamic range, making small peaks relatively larger. For instance, in `computeMagnitude()` you could transform `sample` with a sqrt or log to reduce the gap between big and small onsets.
- **Half/Double Tempo Check**: Implement a check after tempo selection. Pseudocode integration into `updateWinner()`:

```
// After winner_bin_ is finalized
uint16_t alt_bin = 0;
if (winner_bin_ * 2 < NUM_TEMPI) alt_bin = winner_bin_ * 2;
else if (winner_bin_ / 2 >= 1)    alt_bin = winner_bin_ / 2;
if (alt_bin != 0 && tempi_smooth_[alt_bin] > 0.8f *
tempi_smooth_[winner_bin_]) {
    // If alternate tempo energy is close
    if (alt_bin > winner_bin_ && tempi_[alt_bin].target_bpm <=
tuningMaxBPM) {
        winner_bin_ = alt_bin;  // choose faster tempo
    }
    // (Optionally, handle the case of alt_bin < winner_bin_ similarly
if preferring slower tempo in some cases)
}
```

Also reset the phase to align with the new tempo's phase if you switch immediately. The above uses a fixed 0.8 ratio; you might refine that threshold or even make it dynamic based on confidence. This code uses integer bin math, but target BPM can be accessed to ensure doubling doesn't exceed an upper BPM limit (you might define `tuningMaxBPM` as 180 or 200).

- **Smoothing and Hysteresis**: Expose the hysteresis parameters via the `TempoTrackerTuning` struct (they already exist as `hysteresisThreshold` and `hysteresisFrames` defaults [37]). You can then tweak them without recompiling firmware. For instance, set `hysteresisThreshold = 1.2f` (requiring 20% dominance) and `hysteresisFrames = 8` for more stable locking. Or even implement logic to adjust these on the fly: if the system has been locked for >2 seconds, increase threshold (to prevent flip), if confidence drops or a new strong competitor appears, lower threshold temporarily to allow a swap. These dynamic adjustments can be complex, so as a first step simply try slightly more stringent values to see if it eliminates oscillation.
- **Phase Alignment**: If using the `phase_inverted` flag, consider using it to auto-correct phase when a tempo is doubled. E.g., if we switch from 75 BPM to 150 BPM, the phase of the faster

tempo should ideally start at half-cycle (because what was beat-to-beat in the slower is now every other beat). If the code doesn't already handle this, you can detect the scenario and offset the phase by π (i.e., set new `current_phase_` to ±π/2 as needed) so that the next tick isn't mistimed. Since the MusicalGrid PLL also corrects phase gradually using beat observations [38], it will take care of fine adjustments, but giving it a head start by correct initial phase is helpful.

- **MusicalGrid Integration**: Ensure that the MusicalGrid receives downbeat info if available. Currently, without explicit downbeat detection, MusicalGrid assumes the first beat it hears is a downbeat (beat_in_bar=0) and then counts in cycle [39]. If half-tempo lock was happening, MusicalGrid might have been counting measures incorrectly. Once tempo is correct, you might consider adding a simple downbeat detection: e.g., if you know music is in 4/4, and you can identify a stronger beat (like bar accents), you could tag those. This is an advanced nice-to-have. In practice, just getting the tempo right will already allow MusicalGrid to output a stable `bpm_smoothed` and `beat_phase01` for your effects.

Finally, test these changes with a variety of tracks (especially ones around the 140–160 BPM range which are prone to half-tempo errors). Use logging to watch the values of `confidence`, the magnitudes of the winning bin vs. its double/half, and the `phase_inverted` flag. You should see, for example, in a 150 BPM song: initial lock maybe at 75 BPM with low confidence, but then the system detects strong energy at double frequency and flips to 150 BPM within a second, after which confidence rises above the lock threshold. The beat ticks should then align on every actual beat. Stability can be verified by observing that BPM does not waver more than a few BPM once locked (the MusicalGrid smoothing will handle small fluctuations).

By applying the above recommendations – improving onset detection, adding interval sanity checks, incorporating half/double disambiguation, and tuning the smoothing parameters – the tempo tracker will better satisfy the user priorities: 1. **Real-time response**: The system will still operate in <100 ms latency, as we mostly tweak logic and small computations. No heavy model inference, and any added autocorrelation check can be done infrequently or in the background. 2. **Stability**: Enhanced hysteresis and smoothing will avoid jitter, and the PLL will continue to produce a steady phase. The system won't jump between 75 and 150 BPM erratically; it will make a confident choice and stick with it. 3. **Accuracy**: By catching the 2×/½× errors, the BPM output will reflect the true musical tempo (tactus) even in challenging cases. The combination of frequency-domain and time-domain analysis (hybrid approach) covers for each other's weaknesses – ensuring neither missed beats nor spurious double counts throw off the tempo.

These improvements are designed as incremental upgrades to your existing code (in `TempoTracker.cpp` and related classes) and do not require special hardware or ML. Code comments and documentation should be updated alongside these changes so that future developers (even without deep DSP knowledge) can understand the intent: e.g. "*Check for half/double tempo ambiguity and correct it*" in the code where you implement that logic. By clearly structuring the code with these logical steps, the tempo tracking system will be both **robust** and **maintainable**.

---

[1] [2] [4] [5] [6] [7] [8] [9] [10] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [27] [28] [31] [32] [35]
TempoTracker.cpp
https://github.com/synqing/Lightwave-Ledstrip/blob/f81303268c79c422c9ffbf7c5cc4727956ae1fd4/firmware/v2/src/audio/tempo/TempoTracker.cpp

[3] [11] [12] [25] [26] [36] [37] TempoTracker.h
https://github.com/synqing/Lightwave-Ledstrip/blob/f81303268c79c422c9ffbf7c5cc4727956ae1fd4/firmware/v2/src/audio/tempo/TempoTracker.h

[29] [PDF] Beat Tracking by Dynamic Programming - Columbia University
https://www.ee.columbia.edu/~dpwe/pubs/Ellis07-beattrack.pdf

[30] [PDF] AUDIO TEMPO ESTIMATION USING FUSION OF TIME ... - Mirex
https://www.music-ir.org/mirex/abstracts/2014/QHS1.pdf

[33] [38] [39] MusicalGrid.cpp
https://github.com/synqing/Lightwave-Ledstrip/blob/f81303268c79c422c9ffbf7c5cc4727956ae1fd4/firmware/v2/src/audio/contracts/MusicalGrid.cpp

[34] MusicalGrid.h
https://github.com/synqing/Lightwave-Ledstrip/blob/f81303268c79c422c9ffbf7c5cc4727956ae1fd4/firmware/v2/src/audio/contracts/MusicalGrid.h