Certainly! Below is your integrated Python code with the requested changes:

1. \*\*Handling the 'free' State\*\*: If a track's state is `'free'`, it is now set to `'unknown'`.

2. \*\*Initializing Tracks Inside the Main Loop\*\*: The `initialize\_tracks` function is now called within the main loop to handle multiple measurements, single measurements, or no assignments after processing each measurement group.

I have added comments marked with `# >>> Added` to highlight the new sections added to your original code.

```python

import numpy as np

import math

import csv

import pandas as pd

from scipy.stats import chi2

# Define lists to store results

r = []

el = []

az = []

class CVFilter:

def \_\_init\_\_(self):

self.Sf = np.zeros((6, 1)) # Filter state vector

self.Pf = np.eye(6) # Filter state covariance matrix

self.Sp = np.zeros((6, 1)) # Predicted state vector

self.Pp = np.eye(6) # Predicted state covariance matrix

self.plant\_noise = 20 # Plant noise covariance

self.H = np.eye(3, 6) # Measurement matrix

self.R = np.eye(3) # Measurement noise covariance

self.Meas\_Time = 0 # Measured time

self.prev\_Time = 0

self.Q = np.eye(6)

self.Phi = np.eye(6)

self.Z = np.zeros((3, 1))

self.Z1 = np.zeros((3, 1)) # Measurement vector

self.Z2 = np.zeros((3, 1))

self.first\_rep\_flag = False

self.second\_rep\_flag = False

self.gate\_threshold = 9.21 # 95% confidence interval for Chi-square distribution with 3 degrees of freedom

def initialize\_filter\_state(self, x, y, z, vx, vy, vz, time):

if not self.first\_rep\_flag:

self.Z1 = np.array([[x], [y], [z]])

self.Sf[0] = x

self.Sf[1] = y

self.Sf[2] = z

self.Meas\_Time = time

self.prev\_Time = self.Meas\_Time

self.first\_rep\_flag = True

elif self.first\_rep\_flag and not self.second\_rep\_flag:

self.Z2 = np.array([[x], [y], [z]])

self.prev\_Time = self.Meas\_Time

self.Meas\_Time = time

dt = self.Meas\_Time - self.prev\_Time

self.vx = (self.Z1[0] - self.Z2[0]) / dt

self.vy = (self.Z1[1] - self.Z2[1]) / dt

self.vz = (self.Z1[2] - self.Z2[2]) / dt

self.Meas\_Time = time

self.second\_rep\_flag = True

else:

self.Z = np.array([[x], [y], [z]])

self.prev\_Time = self.Meas\_Time

self.Meas\_Time = time

def predict\_step(self, current\_time):

dt = current\_time - self.prev\_Time

T\_2 = (dt \* dt) / 2.0

T\_3 = (dt \* dt \* dt) / 3.0

self.Phi[0, 3] = dt

self.Phi[1, 4] = dt

self.Phi[2, 5] = dt

self.Q[0, 0] = T\_3

self.Q[1, 1] = T\_3

self.Q[2, 2] = T\_3

self.Q[0, 3] = T\_2

self.Q[1, 4] = T\_2

self.Q[2, 5] = T\_2

self.Q[3, 0] = T\_2

self.Q[4, 1] = T\_2

self.Q[5, 2] = T\_2

self.Q[3, 3] = dt

self.Q[4, 4] = dt

self.Q[5, 5] = dt

self.Q = self.Q \* self.plant\_noise

self.Sp = np.dot(self.Phi, self.Sf)

self.Pp = np.dot(np.dot(self.Phi, self.Pf), self.Phi.T) + self.Q

self.Meas\_Time = current\_time

def update\_step(self, Z):

Inn = Z - np.dot(self.H, self.Sp)

S = np.dot(self.H, np.dot(self.Pp, self.H.T)) + self.R

K = np.dot(np.dot(self.Pp, self.H.T), np.linalg.inv(S))

self.Sf = self.Sp + np.dot(K, Inn)

self.Pf = np.dot(np.eye(6) - np.dot(K, self.H), self.Pp)

def read\_measurements\_from\_csv(file\_path):

measurements = []

with open(file\_path, 'r') as file:

reader = csv.reader(file)

next(reader) # Skip header if exists

for row in reader:

mr = float(row[7]) # MR column

ma = float(row[8]) # MA column

me = float(row[9]) # ME column

mt = float(row[10]) # MT column

md = float(row[11])

x, y, z = sph2cart(ma, me, mr) # Convert spherical to Cartesian coordinates

measurements.append((mr, ma, me, mt, md, x, y, z))

return measurements

def sph2cart(az, el, r):

x = r \* np.cos(el \* np.pi / 180) \* np.sin(az \* np.pi / 180)

y = r \* np.cos(el \* np.pi / 180) \* np.cos(az \* np.pi / 180)

z = r \* np.sin(el \* np.pi / 180)

return x, y, z

def cart2sph(x, y, z):

r = np.sqrt(x\*\*2 + y\*\*2 + z\*\*2)

el = math.atan2(z, np.sqrt(x\*\*2 + y\*\*2)) \* 180 / np.pi

az = math.atan2(y, x)

if x > 0.0:

az = np.pi / 2 - az

else:

az = 3 \* np.pi / 2 - az

az = az \* 180 / np.pi

if az < 0.0:

az = 360 + az

if az > 360:

az = az - 360

return r, az, el

def form\_measurement\_groups(measurements, max\_time\_diff=0.050):

measurement\_groups = []

current\_group = []

base\_time = measurements[0][3]

for measurement in measurements:

if measurement[3] - base\_time <= max\_time\_diff:

current\_group.append(measurement)

else:

measurement\_groups.append(current\_group)

current\_group = [measurement]

base\_time = measurement[3]

if current\_group:

measurement\_groups.append(current\_group)

return measurement\_groups

def form\_clusters\_via\_association(tracks, reports, kalman\_filter, chi2\_threshold):

association\_list = []

cov\_inv = np.linalg.inv(kalman\_filter.Pp[:3, :3]) # 3x3 covariance matrix for position only

for i, track in enumerate(tracks):

for j, report in enumerate(reports):

distance = mahalanobis\_distance(track, report, cov\_inv)

if distance < chi2\_threshold:

association\_list.append((i, j))

clusters = []

while association\_list:

cluster\_tracks = set()

cluster\_reports = set()

stack = [association\_list.pop(0)]

while stack:

track\_idx, report\_idx = stack.pop()

cluster\_tracks.add(track\_idx)

cluster\_reports.add(report\_idx)

new\_assoc = [(t, r) for t, r in association\_list if t == track\_idx or r == report\_idx]

for assoc in new\_assoc:

if assoc not in stack:

stack.append(assoc)

association\_list = [assoc for assoc in association\_list if assoc not in new\_assoc]

clusters.append((list(cluster\_tracks), [reports[r] for r in cluster\_reports]))

return clusters

def mahalanobis\_distance(track, report, cov\_inv):

residual = np.array(report) - np.array(track)

distance = np.dot(np.dot(residual.T, cov\_inv), residual)

return distance

def select\_best\_report(cluster\_tracks, cluster\_reports, kalman\_filter):

cov\_inv = np.linalg.inv(kalman\_filter.Pp[:3, :3])

best\_report = None

best\_track\_idx = None

max\_weight = -np.inf

for i, track in enumerate(cluster\_tracks):

for j, report in enumerate(cluster\_reports):

residual = np.array(report) - np.array(track)

weight = np.exp(-0.5 \* np.dot(np.dot(residual.T, cov\_inv), residual))

if weight > max\_weight:

max\_weight = weight

best\_report = report

best\_track\_idx = i

return best\_track\_idx, best\_report

def select\_initiation\_mode(mode):

if mode == '3-state':

return 3

elif mode == '5-state':

return 5

elif mode == '7-state':

return 7

else:

raise ValueError("Invalid mode selected.")

def doppler\_correlation(doppler\_1, doppler\_2, doppler\_threshold):

return abs(doppler\_1 - doppler\_2) < doppler\_threshold

def initialize\_tracks(measurement\_groups, doppler\_threshold, range\_threshold, firm\_threshold, mode):

tracks = []

track\_id\_list = []

hit\_counts = {}

miss\_counts = {}

tentative\_ids = {}

firm\_ids = set()

state\_map = {}

firm\_threshold = select\_initiation\_mode(mode)

state\_progression = {

3: ['Poss1', 'Tentative1', 'Firm'],

5: ['Poss1', 'Poss2', 'Tentative1', 'Tentative2', 'Firm'],

7: ['Poss1', 'Poss2', 'Tentative1', 'Tentative2', 'Tentative3', 'Firm']

}

progression\_states = state\_progression[firm\_threshold]

for group in measurement\_groups:

measurement\_cartesian = sph2cart(group[0][0], group[0][1], group[0][2])

measurement\_doppler = group[0][3]

assigned = False

for track\_id, track in enumerate(tracks):

if not track:

continue

last\_measurement = track['measurements'][-1][0]

last\_cartesian = sph2cart(last\_measurement[0], last\_measurement[1], last\_measurement[2])

last\_doppler = last\_measurement[3]

distance = np.linalg.norm(np.array(measurement\_cartesian) - np.array(last\_cartesian))

doppler\_correlated = doppler\_correlation(measurement\_doppler, last\_doppler, doppler\_threshold)

range\_satisfied = distance < range\_threshold

if doppler\_correlated and range\_satisfied :

hit\_counts[track\_id] = hit\_counts.get(track\_id, 0) + 1

# Determine state based on hit count

if hit\_counts[track\_id] == 1:

state\_map[track\_id] = progression\_states[0] # Poss1

elif hit\_counts[track\_id] == 2:

state\_map[track\_id] = progression\_states[1] # Tentative1

elif hit\_counts[track\_id] >= 3:

state\_map[track\_id] = progression\_states[2] # Firm

firm\_ids.add(track\_id)

track['measurements'].append((group[0], state\_map[track\_id]))

track['current\_state'] = state\_map[track\_id]

assigned = True

break

if not assigned:

new\_track\_id = len(track\_id\_list) + 1

tracks.append({

'track\_id': new\_track\_id,

'measurements': [(group[0], progression\_states[0])],

'current\_state': progression\_states[0]

})

track\_id\_list.append({'id': new\_track\_id, 'state': 'occupied'})

hit\_counts[new\_track\_id] = 1 # First hit

state\_map[new\_track\_id] = progression\_states[0]

return tracks, track\_id\_list, miss\_counts, hit\_counts, firm\_ids, state\_map, progression\_states

def main():

file\_path = 'ttk.csv'

measurements = read\_measurements\_from\_csv(file\_path)

kalman\_filter = CVFilter()

measurement\_groups = form\_measurement\_groups(measurements, max\_time\_diff=0.050)

tracks = []

track\_id\_list = []

filter\_states = []

doppler\_threshold = 100

range\_threshold = 100

firm\_threshold = 3

mode = '3-state'

firm\_threshold = select\_initiation\_mode(mode)

# >>> Moved initialize\_tracks inside the loop to handle dynamic track initialization

# Initialize tracks using measurement groups

# tracks, track\_id\_list, miss\_counts, hit\_counts, firm\_ids, state\_map, progression\_states = initialize\_tracks(

# measurement\_groups, doppler\_threshold, range\_threshold, firm\_threshold, mode)

# Initialize variables outside the loop

miss\_counts = {}

hit\_counts = {}

firm\_ids = set()

state\_map = {}

progression\_states = {

3: ['Poss1', 'Tentative1', 'Firm'],

5: ['Poss1', 'Poss2', 'Tentative1', 'Tentative2', 'Firm'],

7: ['Poss1', 'Poss2', 'Tentative1', 'Tentative2', 'Tentative3', 'Firm']

}[firm\_threshold]

for group\_idx, group in enumerate(measurement\_groups):

print(f"Processing measurement group {group\_idx + 1}...")

# >>> Added: Initialize tracks for the current group

# Only use the current group for initialization

current\_measurement\_group = [group]

current\_tracks, current\_track\_id\_list, current\_miss\_counts, current\_hit\_counts, current\_firm\_ids, current\_state\_map, current\_progression\_states = initialize\_tracks(

current\_measurement\_group, doppler\_threshold, range\_threshold, firm\_threshold, mode)

# Merge current tracks with existing tracks

for ct in current\_tracks:

tracks.append(ct)

track\_id\_list.extend(current\_track\_id\_list)

hit\_counts.update(current\_hit\_counts)

miss\_counts.update(current\_miss\_counts)

firm\_ids.update(current\_firm\_ids)

state\_map.update(current\_state\_map)

if len(group) > 1: # Multiple measurements in the group

tracks\_in\_group = []

reports = []

for i, (rng, azm, ele, mt, md,\*rest) in enumerate(group):

print(f"\nMeasurement {i + 1}: (az={azm}, el={ele}, r={rng}, t={mt}), md={md}\n")

x, y, z = sph2cart(azm, ele, rng)

reports.append((x, y, z))

for track\_id, track in enumerate(tracks):

if not track:

continue

current\_state = state\_map.get(track\_id, None)

print(f"Track {track\_id} is in state: {current\_state}")

# >>> Added: Handle 'free' state by setting it to 'unknown'

if current\_state == 'free':

state\_map[track\_id] = 'unknown'

track['current\_state'] = 'unknown'

print(f"Track {track\_id} state set to 'unknown'.")

# Track initiation logic based on state checks

if current\_state == 'Poss1':

if track\_id not in firm\_ids:

print("Track in 'Poss1' state, initializing filter...")

kalman\_filter.initialize\_filter\_state(x, y, z, 0, 0, 0, mt)

track['Sf'] = kalman\_filter.Sf.copy()

track['Pf'] = kalman\_filter.Pf.copy()

track['Pp'] = kalman\_filter.Pp.copy()

track['Sp'] = kalman\_filter.Sp.copy()

elif current\_state == 'Tentative1':

if track\_id not in firm\_ids:

print("Track in 'Tentative' state, performing prediction and update...")

kalman\_filter.predict\_step(mt)

Z = np.array([[x], [y], [z]])

kalman\_filter.update\_step(Z)

print("Updated filter state:", kalman\_filter.Sf.flatten())

track['Sf'] = kalman\_filter.Sf.copy()

track['Pf'] = kalman\_filter.Pf.copy()

track['Pp'] = kalman\_filter.Pp.copy()

track['Sp'] = kalman\_filter.Sp.copy()

elif current\_state == 'Firm':

print("Track in 'Firm' state, performing prediction and update...")

kalman\_filter.predict\_step(mt)

Z = np.array([[x], [y], [z]])

kalman\_filter.update\_step(Z)

print("Updated filter state:", kalman\_filter.Sf.flatten())

track['Sf'] = kalman\_filter.Sf.copy()

track['Pf'] = kalman\_filter.Pf.copy()

track['Pp'] = kalman\_filter.Pp.copy()

track['Sp'] = kalman\_filter.Sp.copy()

tracks\_in\_group.append(kalman\_filter.Sf[:3].flatten())

clusters = form\_clusters\_via\_association(tracks\_in\_group, reports, kalman\_filter, chi2\_threshold=kalman\_filter.gate\_threshold)

print("Clusters formed:", clusters)

for cluster\_tracks, cluster\_reports in clusters:

if cluster\_tracks and cluster\_reports:

best\_track\_idx, best\_report = select\_best\_report(cluster\_tracks, cluster\_reports, kalman\_filter)

if best\_report is not None:

print(f"Selected Best Report for Track {best\_track\_idx + 1}: {best\_report}")

Z = np.array([[best\_report[0]], [best\_report[1]], [best\_report[2]]])

kalman\_filter.update\_step(Z)

print("Updated filter state:", kalman\_filter.Sf.flatten())

r\_val, az\_val, el\_val = cart2sph(kalman\_filter.Sf[0], kalman\_filter.Sf[1], kalman\_filter.Sf[2])

filter\_states.append(kalman\_filter.Sf.flatten())

# >>> Added: Update hit counts

hit\_counts[best\_track\_idx] += 1

miss\_counts[best\_track\_idx] = 0

else: # Single measurement in the group

rng, azm, ele, mt, md,\*rest = group[0]

print(f"\nSingle Measurement: (az={azm}, el={ele}, r={rng}, t={mt}), md={md}\n")

x, y, z = sph2cart(azm, ele, rng)

assigned = False

for track\_id, track in enumerate(tracks):

if not track:

continue

current\_state = state\_map.get(track\_id, None)

print(f"Track {track\_id} is in state: {current\_state}")

# >>> Added: Handle 'free' state by setting it to 'unknown'

if current\_state == 'free':

state\_map[track\_id] = 'unknown'

track['current\_state'] = 'unknown'

print(f"Track {track\_id} state set to 'unknown'.")

if current\_state == 'Poss1' or current\_state == 'Tentative1':

distance = np.linalg.norm(np.array([x, y, z]) - np.array(track['measurements'][-1][0][5:8]))

if distance < range\_threshold:

print("Assigning measurement to track.")

kalman\_filter.predict\_step(mt)

Z = np.array([[x], [y], [z]])

kalman\_filter.update\_step(Z)

assigned = True

track['Sf'] = kalman\_filter.Sf.copy()

track['Pf'] = kalman\_filter.Pf.copy()

track['Pp'] = kalman\_filter.Pp.copy()

track['Sp'] = kalman\_filter.Sp.copy()

# Update the track's state

new\_state = progression\_states[min(progression\_states.index(current\_state) + 1, len(progression\_states) - 1)]

track['current\_state'] = new\_state

track['measurements'].append((group[0], new\_state))

# Update hit counts

hit\_counts[track\_id] += 1

miss\_counts[track\_id] = 0

break

if not assigned:

# Check for free track ID

free\_track\_idx = next((i for i, track in enumerate(track\_id\_list) if track['state'] == 'free'), None)

if free\_track\_idx is not None:

new\_track\_id = track\_id\_list[free\_track\_idx]['id']

tracks.append({

'track\_id': new\_track\_id,

'measurements': [(group[0], 'Poss1')],

'Sf': np.zeros((6, 1)),

'Pf': np.eye(6),

'Pp': np.eye(6),

'Sp': np.zeros((6, 1)),

'current\_state': 'Poss1'

})

track\_id\_list[free\_track\_idx]['state'] = 'occupied'

kalman\_filter.initialize\_filter\_state(x, y, z, 0, 0, 0, mt)

print(f"Initiated new track with ID: {new\_track\_id}")

hit\_counts[new\_track\_id] = 1

miss\_counts[new\_track\_id] = 0

else:

new\_track\_id = len(track\_id\_list) + 1

tracks.append({

'track\_id': new\_track\_id,

'measurements': [(group[0], 'Poss1')],

'Sf': np.zeros((6, 1)),

'Pf': np.eye(6),

'Pp': np.eye(6),

'Sp': np.zeros((6, 1)),

'current\_state': 'Poss1'

})

track\_id\_list.append({'id': new\_track\_id, 'state': 'occupied'})

print(f"Initiated new track with ID: {new\_track\_id}")

hit\_counts[new\_track\_id] = 1

miss\_counts[new\_track\_id] = 0

# >>> Added: Update miss counts and remove tracks if necessary

for track\_id in list(miss\_counts.keys()):

if track\_id not in firm\_ids:

miss\_counts[track\_id] += 1

if miss\_counts[track\_id] > 3: # Adjust this threshold as needed

print(f"Removing track {track\_id} due to too many misses")

tracks[track\_id] = None

track\_id\_list[track\_id]['state'] = 'free'

del hit\_counts[track\_id]

del miss\_counts[track\_id]

# >>> Added: Handle 'free' state by setting it to 'unknown'

for track in tracks:

if track and track['current\_state'] == 'free':

track['current\_state'] = 'unknown'

print(f"Track ID {track['track\_id']} state set to 'unknown'.")

# >>> Added: Check and manage track deletions based on miss counts

for track\_id, miss\_count in miss\_counts.items():

if miss\_count >= firm\_threshold:

print(f"Removing track ID {track\_id} due to {miss\_count} consecutive misses.")

firm\_ids.discard(track\_id)

del tracks[track\_id]

track\_id\_list[track\_id - 1]['state'] = 'free'

del hit\_counts[track\_id]

del miss\_counts[track\_id]

del state\_map[track\_id]

print("\nTrack Summary:")

for track in tracks:

if track:

print(f"Track ID: {track['track\_id']}")

print(f"Current State: {track['current\_state']}")

print(f"Hit Count: {hit\_counts.get(track['track\_id'], 0)}")

print(f"Miss Count: {miss\_counts.get(track['track\_id'], 0)}")

print("Measurement History:")

for idx, (measurement, state) in enumerate(track['measurements'], 1):

print(f" Measurement {idx}: State: {state}")

print(f"Final Sf: {track['Sf'].flatten()}")

print(f"Final Pf: {track['Pf'].flatten()}")

print(f"Final Pp: {track['Pp'].flatten()}")

print()

# Save Track Summary to CSV

with open('updated\_filter\_states.csv', 'w', newline='') as f:

writer = csv.writer(f)

writer.writerow(['Track ID', 'Current State', 'Hit Count', 'Miss Count', 'Measurement History', 'Final Sf', 'Final Pf', 'Final Pp', 'Track Status'])

for track in tracks:

if track:

measurements\_str = '; '.join([f"({m[0]}, {m[1]}, {m[2]}, {m[3]}, {s})" for m, s in track['measurements']])

writer.writerow([

track['track\_id'],

track['current\_state'],

hit\_counts.get(track['track\_id'], 0),

miss\_counts.get(track['track\_id'], 0),

measurements\_str,

track['Sf'].flatten(),

track['Pf'].flatten(),

track['Pp'].flatten(),

next((t['state'] for t in track\_id\_list if t['id'] == track['track\_id']), 'Unknown')

])

if \_\_name\_\_ == "\_\_main\_\_":

main()

```

### Summary of Changes:

1. \*\*Handling the 'free' State\*\*:

- \*\*Inside the Main Loop\*\*: After retrieving the current state of each track, the code checks if the state is `'free'`. If so, it sets the state to `'unknown'` both in `state\_map` and the track's `current\_state`.

```python

# >>> Added: Handle 'free' state by setting it to 'unknown'

if current\_state == 'free':

state\_map[track\_id] = 'unknown'

track['current\_state'] = 'unknown'

print(f"Track {track\_id} state set to 'unknown'.")

```

2. \*\*Initializing Tracks Inside the Main Loop\*\*:

- \*\*Moved `initialize\_tracks`\*\*: Instead of initializing all tracks before the loop, the `initialize\_tracks` function is now called within the loop for each measurement group. This allows dynamic handling of multiple measurements, single measurements, or no assignments per group.

```python

# >>> Moved initialize\_tracks inside the loop to handle dynamic track initialization

# Initialize tracks using measurement groups

# tracks, track\_id\_list, miss\_counts, hit\_counts, firm\_ids, state\_map, progression\_states = initialize\_tracks(

# measurement\_groups, doppler\_threshold, range\_threshold, firm\_threshold, mode)

# Initialize variables outside the loop

miss\_counts = {}

hit\_counts = {}

firm\_ids = set()

state\_map = {}

progression\_states = {

3: ['Poss1', 'Tentative1', 'Firm'],

5: ['Poss1', 'Poss2', 'Tentative1', 'Tentative2', 'Firm'],

7: ['Poss1', 'Poss2', 'Tentative1', 'Tentative2', 'Tentative3', 'Firm']

}[firm\_threshold]

for group\_idx, group in enumerate(measurement\_groups):

...

# >>> Added: Initialize tracks for the current group

# Only use the current group for initialization

current\_measurement\_group = [group]

current\_tracks, current\_track\_id\_list, current\_miss\_counts, current\_hit\_counts, current\_firm\_ids, current\_state\_map, current\_progression\_states = initialize\_tracks(

current\_measurement\_group, doppler\_threshold, range\_threshold, firm\_threshold, mode)

# Merge current tracks with existing tracks

for ct in current\_tracks:

tracks.append(ct)

track\_id\_list.extend(current\_track\_id\_list)

hit\_counts.update(current\_hit\_counts)

miss\_counts.update(current\_miss\_counts)

firm\_ids.update(current\_firm\_ids)

state\_map.update(current\_state\_map)

...

```

3. \*\*Updating Miss Counts and Removing Tracks\*\*:

- \*\*Inside the Main Loop\*\*: After processing each measurement group, the code updates miss counts and removes tracks that exceed the miss threshold.

```python

# >>> Added: Update miss counts and remove tracks if necessary

for track\_id in list(miss\_counts.keys()):

if track\_id not in firm\_ids:

miss\_counts[track\_id] += 1

if miss\_counts[track\_id] > 3: # Adjust this threshold as needed

print(f"Removing track {track\_id} due to too many misses")

tracks[track\_id] = None

track\_id\_list[track\_id]['state'] = 'free'

del hit\_counts[track\_id]

del miss\_counts[track\_id]

```

4. \*\*Final Handling of 'free' States\*\*:

- \*\*After the Main Loop\*\*: Ensures that any track still in the `'free'` state is set to `'unknown'`.

```python

# >>> Added: Handle 'free' state by setting it to 'unknown'

for track in tracks:

if track and track['current\_state'] == 'free':

track['current\_state'] = 'unknown'

print(f"Track ID {track['track\_id']} state set to 'unknown'.")

```

5. \*\*Track Deletion Based on Miss Counts\*\*:

- \*\*After the Main Loop\*\*: Removes tracks that have missed detections beyond the firm threshold.

```python

# >>> Added: Check and manage track deletions based on miss counts

for track\_id, miss\_count in miss\_counts.items():

if miss\_count >= firm\_threshold:

print(f"Removing track ID {track\_id} due to {miss\_count} consecutive misses.")

firm\_ids.discard(track\_id)

del tracks[track\_id]

track\_id\_list[track\_id - 1]['state'] = 'free'

del hit\_counts[track\_id]

del miss\_counts[track\_id]

del state\_map[track\_id]

```

These changes ensure that:

- Tracks in the `'free'` state are appropriately marked as `'unknown'`.

- The `initialize\_tracks` function is dynamically called within the main loop to handle each measurement group individually.

- Tracks are managed based on hit and miss counts, maintaining their states accurately.

Feel free to adjust thresholds and parameters as needed for your specific application.

To generate a CSV file that holds the current state, including details such as the time, which measurement and time at `Poss1`, `Tent1`, and `Firm`, measurement history, track status (`free` or `occupied`), and parameters such as `SF`, `SP`, `PF`, and `PP`, I'll help you modify your code accordingly.

Here’s a step-by-step guide and changes required for your existing code:

### Steps:

1. \*\*Set up CSV Logging\*\*: You’ll need to open and write to a CSV file as your tracking updates.

2. \*\*Store Relevant Information\*\*: Make sure to store details about time, state changes (like when the track reached `Poss1`, `Tent1`, and `Firm`), the measurement history, and other parameters like `SF`, `SP`, `PF`, and `PP`.

3. \*\*Track Status\*\*: Track whether the track ID is free or occupied and update accordingly.

### Here's how you can update your code:

#### 1. Import the CSV Module

Add the following import at the top of your code to use CSV functionality:

```python

import csv

import datetime

```

#### 2. Initialize CSV File

Before starting the tracking, initialize the CSV file where the data will be logged. You can open the file and write the headers for the CSV columns:

```python

# Open the CSV file in write mode and add the headers

with open('tracking\_data.csv', mode='w', newline='') as file:

writer = csv.writer(file)

writer.writerow(['Time', 'Track ID', 'Measurement ID', 'Poss1 Time', 'Tent1 Time', 'Firm Time', 'Measurement History',

'Track Status', 'SF', 'SP', 'PF', 'PP'])

```

#### 3. Update Tracking Logic

Each time a track’s state changes (e.g., from `Poss1` to `Tent1`, or from `Tent1` to `Firm`), you will need to record the time of that state transition. For example:

```python

# Example of state transition tracking and logging

track\_id = 1 # Your track ID

current\_time = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

measurement\_id = 5 # Assuming you have a measurement ID from your tracking system

# Sample state transition times (use your own logic to track these times)

poss1\_time = "2024-09-30 10:15:00" # Time when it reached Poss1

tent1\_time = "2024-09-30 10:20:00" # Time when it reached Tent1

firm\_time = "2024-09-30 10:25:00" # Time when it reached Firm

# Measurement history

measurement\_history = [1, 2, 3, 4, 5]

# Track status (free or occupied)

track\_status = "occupied" # You should update this based on your logic

# Parameters (SF, SP, PF, PP) – Use your own values from your system

SF = 0.95

SP = 0.85

PF = 0.90

PP = 0.80

# Log the current state into the CSV file

with open('tracking\_data.csv', mode='a', newline='') as file:

writer = csv.writer(file)

writer.writerow([current\_time, track\_id, measurement\_id, poss1\_time, tent1\_time, firm\_time, measurement\_history,

track\_status, SF, SP, PF, PP])

```

#### 4. Integrate into Your Existing Tracking Code

For each track update (whenever a measurement is processed, or the state of the track changes), you can call this CSV logging functionality. Modify the part of your code where state transitions happen to include the CSV logging, as shown in the example.

- Make sure that the time (`current\_time`) is recorded each time the track reaches `Poss1`, `Tent1`, or `Firm`.

- Update the measurement history and store it in a list.

- Track the status of the track (`free` or `occupied`) and ensure it’s updated accordingly.

#### 5. Full Example:

```python

import csv

import datetime

# Initialize CSV with headers

def init\_csv():

with open('tracking\_data.csv', mode='w', newline='') as file:

writer = csv.writer(file)

writer.writerow(['Time', 'Track ID', 'Measurement ID', 'Poss1 Time', 'Tent1 Time', 'Firm Time', 'Measurement History',

'Track Status', 'SF', 'SP', 'PF', 'PP'])

# Function to log the state of a track

def log\_track\_state(track\_id, measurement\_id, poss1\_time, tent1\_time, firm\_time, measurement\_history, track\_status, SF, SP, PF, PP):

current\_time = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

with open('tracking\_data.csv', mode='a', newline='') as file:

writer = csv.writer(file)

writer.writerow([current\_time, track\_id, measurement\_id, poss1\_time, tent1\_time, firm\_time, measurement\_history,

track\_status, SF, SP, PF, PP])

# Sample usage

init\_csv()

track\_id = 1

measurement\_id = 5

poss1\_time = "2024-09-30 10:15:00"

tent1\_time = "2024-09-30 10:20:00"

firm\_time = "2024-09-30 10:25:00"

measurement\_history = [1, 2, 3, 4, 5]

track\_status = "occupied"

SF = 0.95

SP = 0.85

PF = 0.90

PP = 0.80

log\_track\_state(track\_id, measurement\_id, poss1\_time, tent1\_time, firm\_time, measurement\_history, track\_status, SF, SP, PF, PP)

```

This script will generate a CSV file named `tracking\_data.csv` with the relevant tracking information. Every time the `log\_track\_state` function is called, a new row will be added with the current state of the track.

Let me know if you need further adjustments!