GEOFENCES IN A SCRIPT

1. Introduction

Senquip devices support geofencing as a mechanism to determine whether a remote asset is located within a defined geographic area. For simple use cases, a single circular geofence—defined by a radius around a fixed latitude and longitude—can be configured directly through device settings. This approach is suitable when only one location boundary is required and when the boundary can be reasonably approximated as a circle.

Many real-world applications, however, require multiple geofences and more flexible boundary definitions. Examples include vehicles servicing multiple sites, trailers moving between depots, or assets operating within irregularly shaped areas such as farms, industrial facilities, or exclusion zones. In these cases, device settings alone are insufficient, and script-based geofencing is required.

This application note describes two script-based approaches to geofencing on Senquip devices:

- **Multiple circular geofences**, each defined by a centre point (latitude and longitude) and a radius.
- **Arbitrarily shaped geofences**, defined by a series of latitude/longitude points forming a polygon and evaluated using the ray-casting algorithm.

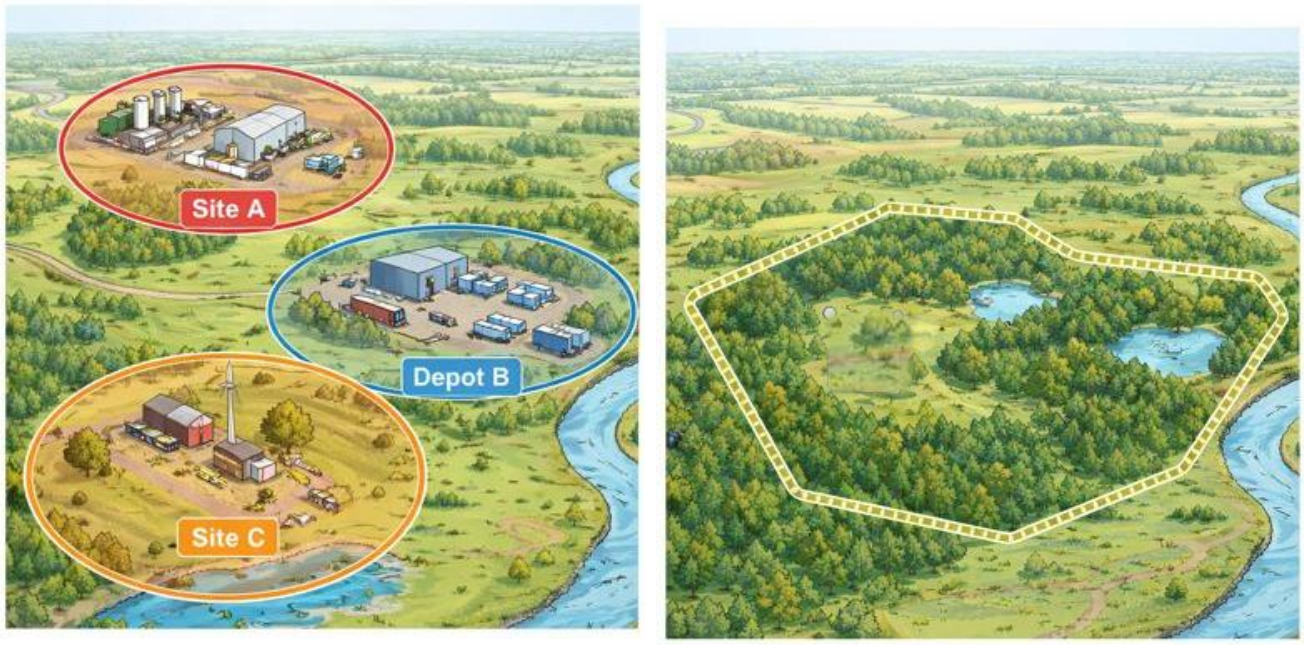

Figure 1 - Circular and Polygon Geofences

In both approaches, geofence definitions are stored externally in a CSV file loaded onto the device. This allows geofences to be modified without changing firmware and provides a clear, auditable configuration mechanism.

For circular geofences, each CSV row represents a single geofence, containing a site name and a central latitude/longitude. A single radius value is applied uniformly by the script.

For polygon geofences, each CSV row represents one geofence, beginning with a name followed by an arbitrary number of latitude/longitude point pairs that define the polygon boundary. The number of points is not fixed and determines the length of each row.

This document focuses on practical geofence algorithms suitable for execution within Senquip's scripting environment.

Disclaimer

The information provided in this application note is intended for informational purposes only. Users of the remote machine control system described herein should exercise caution and adhere to all relevant safety guidelines and regulations. By utilising the information provided in this application note, users acknowledge their understanding and acceptance of the associated risks. The authors and contributors disclaim any warranties, expressed or implied, regarding the accuracy or completeness of the information presented.

2. Circular Geofences

A circular geofence defines an "on site" area as a radius around a single latitude/longitude point. This is the simplest and most common geofence representation and works well for depots, yards, plants, farms, compressor sites, and other locations where a circular approximation is acceptable.

Senquip devices support a single circular geofence directly through device settings. When multiple sites are required, a script can apply the same concept repeatedly by comparing the device's GPS position to a list of site centre points.

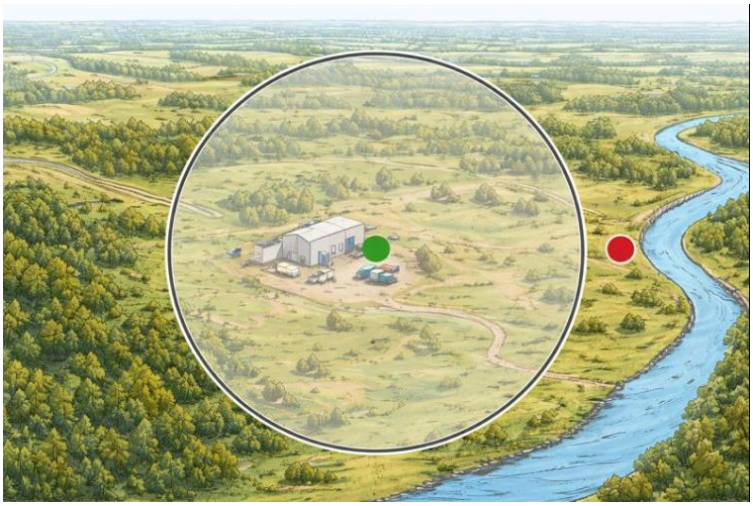

Figure 2 - Devices Inside and Outside a Circular Geofence

2.1. Geofence File Format (CSV)

For circular geofences, the user uploads a CSV file called geofence.csv. Each row defines one site using a name and a centre point. The script will load this file at boot.

Format:

Site Name	Lat	Lon
Pickup Dell Yard	34.09005	-117.581
Pickup Axion Plaza	35.512861	-119.191
Pickup Sykes	35.528028	-119.272
Dropoff Downtown LA	35.51379189	-119.191
Reiker Fuel Station	33.840697	-118.242
Hass Farm Dropoff	34.21268	-118.594
Liam Matter Fuel Depot	33.9272126	-118.381

Notes

- Fields are separated by commas.
- Line endings may be CR, LF, or CRLF; the script normalises these automatically.
- The first row is treated as a header and ignored.
- Site names may contain spaces.
- Latitude and longitude are signed decimal degrees.
- Each site should have a unique name.

2.2. Radius Selection

The script uses a configurable radius (e.g., 500 m). Choose a radius that reflects:

- GPS accuracy at the installation location
- the physical size of the site
- How strict arrival and departure detection needs to be

If the radius is too small, the device may oscillate between “on site” and “travelling” due to GPS drift. If too large, the geofence may include adjacent roads or nearby facilities.

2.3. Entry/Exit Stability and Hysteresis

GPS positions naturally vary even when an asset is stationary. To prevent false arrivals and departures, the circular geofence logic includes:

- **Entry confirmation**: multiple consecutive samples inside a geofence are required before declaring an arrival.
- **Exit confirmation**: multiple consecutive samples outside the geofence are required before declaring a departure.
- **Exit margin (hysteresis)**: departure processing only begins once the asset moves beyond *(radius + margin)*.

This combination provides robust behaviour in real deployments and prevents nuisance state changes due to GPS jitter.

2.4. Overlapping Circular Geofences

Circular geofences overlap whenever two centre points are closer than twice the configured radius. If a device is inside more than one geofence at the same time, a script may:

- Alternate ("flip-flop") between site names depending on loop order and GPS noise
- Report a site that is technically valid but operationally incorrect

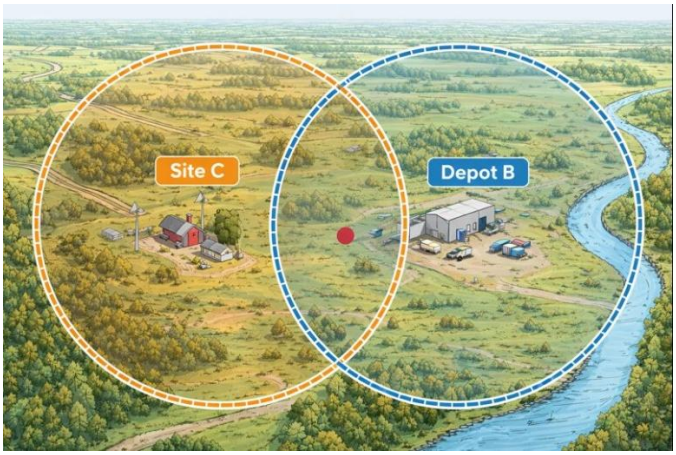

Figure 3 - Overlapping Geofences

When using circular geofences:

- Place site centres far enough apart for the chosen radius, or
- Combine very close locations into a single site, or
- Reduce the radius or use polygon geofencing where separation is required

The example script includes a runtime warning when multiple geofences match simultaneously, helping identify ambiguous configurations during commissioning.

3. Downloading Files to the Senquip Device

Files are uploaded to Senquip devices:

- On a single device basis from settings > Files > File Download
- To a group of devices from Group Settings > Bulk Configure > Send File
- Using the Senquip API

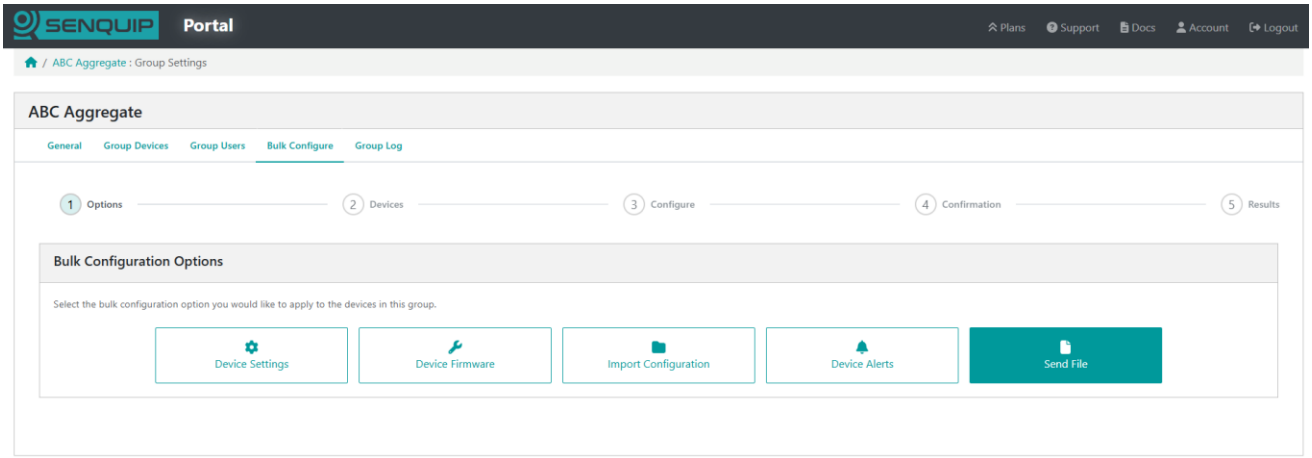

Figure 4 - Bulk File Import Using Groups

4. Circular Geofence Script

This section describes what each function does and why it exists. It focuses on the behaviour that matters when commissioning, modifying, or troubleshooting the script. For brevity, some functions are shortened here; the complete script is provided in Appendix A. For descriptions of included libraries or functions used, please see the Senquip Scripting Guide.

4.1. Parameters

Configuration parameters define entry and exit behaviour and should be adjusted based on the application:

- **range (m)**: A site is considered “inside” if the distance is less than this value.
- **exit_margin (m)**: Departure processing only begins once the distance exceeds *(range + exit_margin)*.
- **entry_needed / exit_needed**: Number of consecutive confirmations required before changing state.

```
let range = 200;         // metres - inside this is consided arrived
let exit_margin = 50;    // metres - must be this more than range to leave a site
let entry_needed = 3;    // number of positives before entry accepted
let exit needed  = 3;
```

Runtime state variables track the current location and stability:

- **current_site**:
  - 'Initialising' – boot state before GPS is available
  - 'Travelling' – not inside any site
  - or a site name from the CSV
- **entry / exit**: Counters used to confirm arrivals and departures. These increment when conditions are met and decay slowly between samples.
- **all_sites**: Array of CSV rows loaded from geofence.csv.
- **file_found**: Flag indicating whether the CSV file was successfully loaded.

```
let current_site = 'Initialising';
let entry = 0;
let exit  = 0;
let all_sites = [];
let file_found = 0;
```

4.2. load_sites()

Loads the geofence.csv file from device storage, normalises its line endings, and splits it into individual rows for processing. A status flag is set so the script can detect and report a missing or empty file at runtime. This function is called once at script startup.

```
function load_sites() {
  let file_data = File.read('geofence.csv');

  if (file_data) {
    let normalised = normalise_newlines(file_data);
    all_sites = SQ.split(normalised, "\x0a");
    file_found = 1;
  } else {
    all_sites = [];
    file_found = 0;
  }
}
```

4.3. normalise_newlines(str)

Removes carriage-return characters from the CSV file, ensuring rows can be split reliably on line-feed characters. This avoids parsing issues caused by different newline conventions used by tools such as Excel.

```
function normalise_newlines(str) {
  let parts = SQ.split(str, "\x0d");
  let result = "";
  for (let i = 0; i < parts.length; i++) {
    result = result + parts[i];
  }
  return result;
}
```

4.4. SQ.set_data_handler(...)

This function runs the geofence logic whenever new data is received from the device. It handles missing files and invalid GPS gracefully, raises events for arrivals, departures, and configuration issues, and publishes the current site state for monitoring and logging.

To improve stability when GPS reception is intermittent, the entry and exit counters are slowly reduced between updates so that isolated GPS samples do not accumulate indefinitely. This allows occasional valid fixes to reinforce the current location without causing repeated arrivals and departures as GPS coverage is gained and lost.

```
SQ.set_data_handler(function(data) {
  let obj = JSON.parse(data);

  if (file_found === 0) {
    SQ.dispatch_event(1, SQ.WARNING, "geofence.csv not found or empty");
    return;
  }
  else if (typeof obj.gps_lat === "number" && typeof obj.gps_lon === "number") {
    let matches = process_geofences();
    if (matches > 1) {SQ.dispatch_event(1, SQ.WARNING, "Multiple geofence matches found");}

    entry = entry - 0.1; if (entry < 0) {entry = 0;}
    exit = exit - 0.1; if (exit < 0) {exit = 0;}

    SQ.dispatch(1, current_site);

  }
  else {SQ.dispatch event(1, SQ.WARNING, "No valid GPS");}

}, null);
```

4.5. process_geofences()

This function compares the device's current GPS position with every configured site to determine the correct location state: travelling, arriving at a site, or leaving a site. Arrival is only declared after multiple consecutive position samples fall within a site's radius, while departure is only declared after multiple samples fall beyond the radius plus an additional hysteresis margin, providing stable behaviour in the presence of GPS drift.

The function also tracks how many geofences are matched at the same time and raises a warning when more than one site applies, helping identify overlapping or ambiguous geofence configurations.

```
function process_geofences() {
  let matches = 0;

  if (current_site === 'Initialising') {current_site = 'Travelling';}

  for (let i = 1; i < all_sites.length; i++) {
    let line = all_sites[i];
    let site = parse_site_line(line);
    let distance = GPS.distance(site.lat, site.lon);

    if (distance < range) {matches++;}

    if (current_site === site.name) {
      if (distance > (range + exit_margin)) {
        exit++;
        if (exit >= exit_needed) {
          SQ.dispatch_event(1, SQ.INFO, "Leaving " + site.name);
          current site = 'Travelling';
          exit = 0;
        }
      }
    } else {
      if (distance < range) {
        entry++;
        if (entry >= entry_needed) {
          SQ.dispatch_event(1, SQ.INFO, "Arriving at " + site.name);
          current_site = site.name;
          entry = 0;
        }
      }
    }
  }
  return matches;
}
```

4.6. parse_site_line(line)

This function parses a single CSV row into a site definition by splitting the line by commas, ignoring blank lines and the header row and validating latitude and longitude values. Invalid coordinates are skipped and flagged with a warning, preventing malformed data from affecting script operation.

```
function parse_site_line(line) {
  if (line === "") {return { ok: 0 };}

  let parts = SQ.split(line, ',');
  if (parts.length < 3) {return { ok: 0 };}

  let name = parts[0];

  // Ignore header rows (with or without UTF-8 BOM)
  if (name === "Site Name" || name === "\ufeffSite Name") {return { ok: 0 };}

  let lat = JSON.parse(parts[1]);
  let lon = JSON.parse(parts[2]);

  if (isNaN(lat) || isNaN(lon)) {return { ok: -1, name: name };}

  return { ok: 1, name: name, lat: lat, lon: lon };
}
```

5. Polygon Geofences

Polygon geofences define an “on site” area using an arbitrarily shaped boundary rather than a simple radius. The boundary is described by a series of latitude/longitude points that form a closed polygon. A device is considered “inside” the geofence when its GPS position lies within this polygon.

Compared to circular geofences, polygon geofences provide higher spatial accuracy at the cost of increased configuration complexity and processing effort.

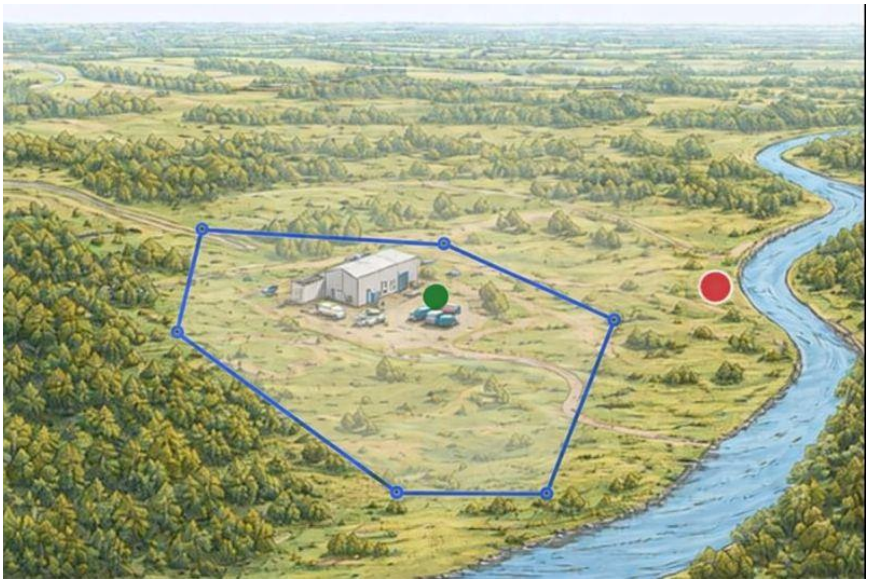

Figure 5 - - Devices Inside and Outside a Polygon Geofence

5.1. Polygon Geofence File Format (CSV)

Polygon geofences are defined in a CSV file named geofence.csv. Each row represents a single geofence and contains:

1. A unique site name
2. A sequence of latitude/longitude pairs defining the polygon vertices

The polygon is implicitly closed by connecting the final point back to the first point.

Format:

Site Name	Lat1	Lon1	Lat2	Lon2	Lat3	Lon3	...	
Farm North	34.2125	-118.595	34.2131	-118.59	34.2098	-118.587	34.2074	-118.592
Fuel Exclusion Zone	33.9271	-118.382	33.928	-118.38	33.9265	-118.379	33.9258	-118.381

Notes:

- Fields are separated by commas.
- Line endings may be CR, LF, or CRLF; the script normalises these automatically.
- The first row is a header and is ignored.
- Site names may contain spaces.
- Latitude and longitude values are signed decimal degrees.
- Each site must contain at least three latitude/longitude point pairs to form a valid polygon.
- Points should be listed in order around the boundary (clockwise or counter-clockwise).
- Polygons must not self-intersect.

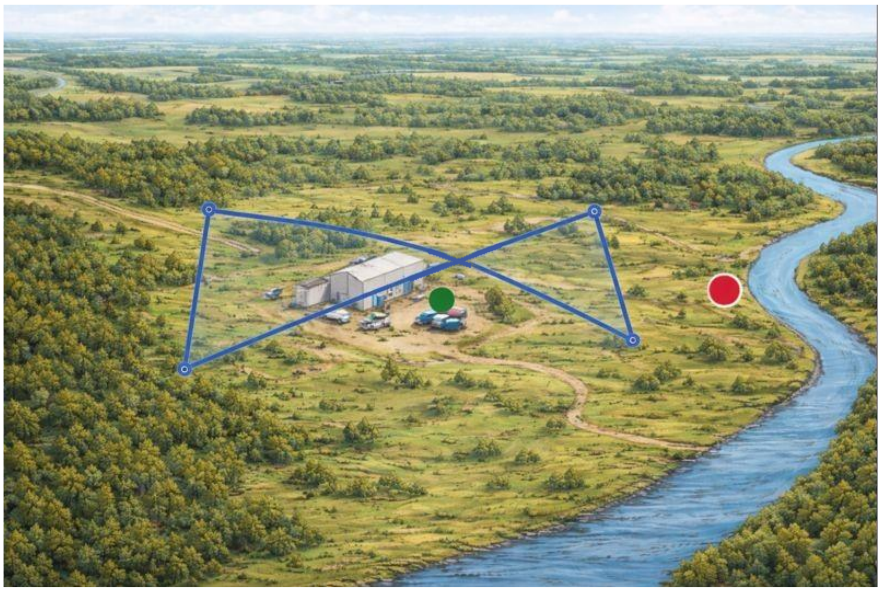

Figure 6 - Unacceptable Intersecting Polygon likely caused by incorrect order of points

5.2. Point-in-Polygon Evaluation (Ray Casting)

To determine whether the device is inside a polygon geofence, the script uses the ray casting algorithm. For a good introduction to the Ray Casting algorithm, see this video.

The algorithm works as follows:

- A horizontal ray is projected from the device's GPS position.
- The number of times this ray intersects the polygon's edges is counted.
- If the number of intersections is **odd**, the point lies inside the polygon.
- If the number of intersections is **even**, the point lies outside the polygon.

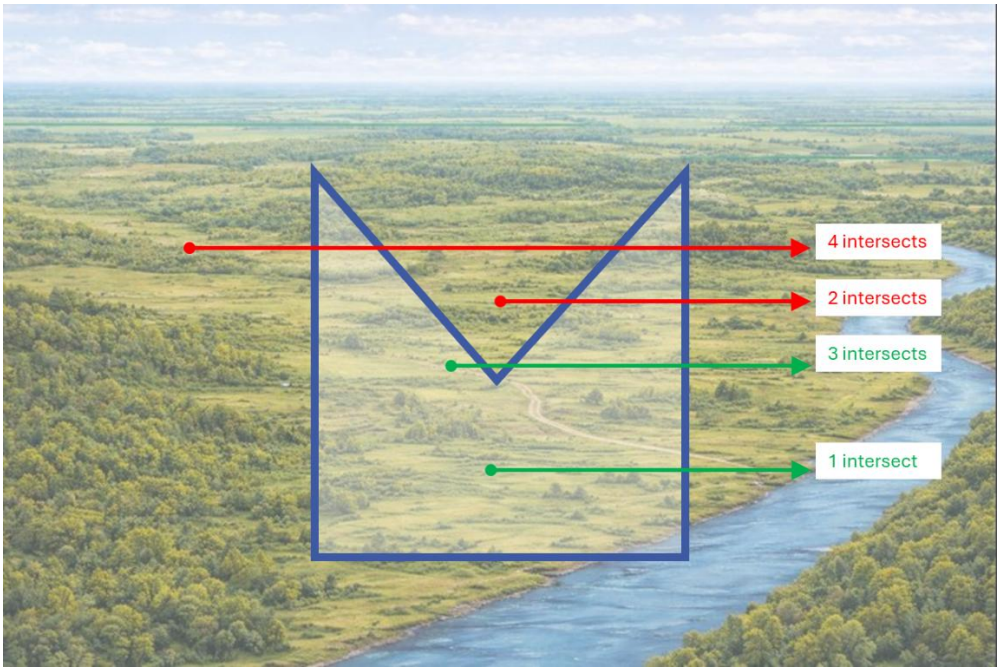

Figure 7 - Ray Casting Algorithm

This method is computationally efficient, robust, and well suited to execution within Senquip's scripting environment.

5.3. Entry/Exit Stability for Polygon Geofences

As with circular geofences, GPS drift can cause a device near a boundary to move in and out of a polygon even when stationary. To ensure stable behaviour, polygon geofencing uses the same confirmation principles:

- **Entry confirmation**: multiple consecutive "inside polygon" samples are required before declaring arrival.
- **Exit confirmation**: multiple consecutive "outside polygon" samples are required before declaring departure.

Unlike circular geofences, polygon geofences do not use a distance-based hysteresis margin. Stability is achieved purely through sample confirmation and counter decay.

5.4. Overlapping Polygon Geofences

Polygon geofences may overlap either intentionally or accidentally. When a device is located within more than one polygon at the same time, the script may:

- Alternate ("flip-flop") between site names depending on loop order and GPS noise
- Report a site that is technically valid but operationally incorrect

To minimise ambiguity:

- Avoid overlapping polygons where possible
- Merge overlapping areas into a single geofence if they represent the same logical site

As with circular geofences, the example script detects and reports multiple simultaneous matches to assist with commissioning and validation.

5.5. Performance Considerations

Polygon geofence evaluation requires iterating through every edge of every configured polygon. While this is well within the capabilities of Senquip devices for typical configurations, the following guidelines are recommended:

- Keep the number of polygon points to the minimum required to describe the boundary accurately
- Avoid defining very large numbers of polygons in a single script
- Prefer circular geofences where appropriate for simple sites

For most practical deployments, polygon geofencing performs reliably with negligible impact on device operation.

6. Polygon Geofence Script

The polygon geofence implementation is derived directly from the circular geofence script described in Section 3. To simplify commissioning, maintenance, and troubleshooting, the overall script structure, state machine, and entry/exit logic are intentionally unchanged.

Only the minimum functionality required to support arbitrarily shaped geofences has been added. In particular:

- File loading, newline normalisation, and CSV handling remain unchanged
- Entry and exit confirmation behaviour is identical to circular geofencing
- State variables and event generation are unchanged
- Function names are preserved wherever possible

The key differences are limited to how geofences are defined and how “inside” versus “outside” is determined.

6.1. Summary of Changes from Circular Geofencing

Compared to the circular geofence script, the following changes are made:

- Each geofence is defined as a polygon rather than a centre point and radius
- A new point-in-polygon test replaces the distance calculation
- The CSV parser is extended to support an arbitrary number of latitude/longitude pairs per site
- Entry and exit decisions are based on polygon inclusion rather than distance thresholds

All other behaviour remains the same.

6.2. Polygon Geofence File Format

In the polygon implementation, each row of geofence.csv defines one site as a closed polygon.

- The first column contains the site name
- Subsequent columns contain latitude/longitude pairs

- The number of points is variable and determined by the length of the row
- At least three point pairs are required to form a valid polygon

Trailing empty columns (for example, those produced by spreadsheet tools) are safely ignored by the script.

6.3. parse_site_line(line) – Polygon Version

This function is extended to parse polygon definitions instead of single centre points.

Rather than returning a single latitude and longitude, the function now:

- Splits the CSV row into fields
- Parses latitude/longitude pairs sequentially
- Stops parsing when empty fields are encountered
- Validates that at least three valid points are present
- Returns a list of polygon vertices for each site

Invalid or malformed rows are skipped, and sites with invalid coordinates are flagged with a warning. This prevents configuration errors from disrupting script execution.

```
function parse_site_line(line) {

  if (line === "") {return { ok: 0 };}

  let parts = SQ.split(line, ',');
  if (parts.length < 7) { // name + (lat,lon)*3 minimum
    return { ok: 0 };
  }

  let name = parts[0];

  // Ignore header rows (with or without UTF-8 BOM)
  if (name === "Site Name" || name === "\ufeffSite Name") {return { ok: 0 };}

  // Parse lat/lon pairs
  let pts = [];
  for (let i = 1; i + 1 < parts.length; i += 2) {

    // STOP when trailing empty columns begin (Excel-style CSVs)
    if (parts[i] === "" || parts[i + 1] === "") {break;}

    let lat = JSON.parse(parts[i]);
    let lon = JSON.parse(parts[i + 1]);

    if (isNaN(lat) || isNaN(lon)) {return { ok: -1, name: name };}

  pts.push({ lat: lat, lon: lon });
}

  if (pts.length < 3) {return { ok: -1, name: name };}

  return { ok: 1, name: name, pts: pts };
}
```

6.4. point_in_polygon(lat, lon, pts)

This function implements the ray casting algorithm to determine whether a GPS position lies inside a polygon.

The algorithm works by projecting a horizontal ray from the GPS position and counting how many times it intersects the polygon edges:

- An odd number of intersections indicates the point is inside the polygon
- An even number indicates the point is outside

The function uses only basic arithmetic and comparison operations and is efficient enough for execution within the Senquip scripting environment.

Latitude is treated as the vertical axis and longitude as the horizontal axis, which is appropriate for the small geographic areas typically used in geofencing applications.

```
function point_in_polygon(lat, lon, pts) {
  let inside = false;
  let n = pts.length;

  // Need at least 3 vertices
  if (n < 3) {
    return false;
  }

  // i = current vertex, j = previous vertex
  let j = n - 1;
  for (let i = 0; i < n; i++) {
    let yi = pts[i].lat;
    let xi = pts[i].lon;
    let yj = pts[j].lat;
    let xj = pts[j].lon;

    // Check if the horizontal ray crosses the edge (j -> i)
    // Condition: edge straddles the test latitude
    let straddles = ((yi > lat) !== (yj > lat));
    if (straddles) {
      // Compute X coordinate where the edge crosses the ray at 'lat'
      // Avoid divide-by-zero naturally because straddles implies yi != yj.
      let x_cross = (xj - xi) * (lat - yi) / (yj - yi) + xi;

      if (lon < x_cross) {
        inside = !inside;
      }
    }

    j = i;
  }

  return inside;
}
```

6.5. process_geofences() – Polygon Evaluation

The process_geofences() function retains its overall structure and state handling but replaces distance-based evaluation with polygon inclusion testing.

For each configured site:

- The current GPS position is evaluated using point_in_polygon()
- A site is considered “matched” if the position lies inside its polygon
- Arrival is declared after multiple consecutive inside samples
- Departure is declared after multiple consecutive outside samples

As with circular geofences, the function counts how many sites match simultaneously and raises a warning if more than one polygon applies, helping identify overlapping or ambiguous configurations.

```
function process_geofences() {
  let matches = 0;

  if (current_site === 'Initialising') {current_site = 'Travelling';}

  for (let i = 1; i < all_sites.length; i++) {
    let line = all_sites[i];
    if (line === "") {continue;}

    let site = parse_site_line(line);

    if (site.ok === -1) {
      SQ.dispatch_event(1, SQ.WARNING, "Invalid polygon for site: " + site.name);
      continue;
    }

    if (site.ok !== 1) {continue;}

    // Ray casting: is the current GPS point inside this polygon?
    let inside = point_in_polygon(gps_lat, gps_lon, site.pts);

    if (inside) {matches++;}

    if (current_site === site.name) {
      // Leaving condition: outside polygon
      if (!inside) {
        exit++;
        if (exit >= exit_needed) {
          SQ.dispatch_event(1, SQ.INFO, "Leaving " + site.name);
          current_site = 'Travelling';
          exit = 0;
        }
      }
    } else {
      // Arriving condition: inside polygon
      if (inside) {
        entry++;
        if (entry >= entry_needed) {
          SQ.dispatch_event(1, SQ.INFO, "Arriving at " + site.name);
          current_site = site.name;
          entry = 0;
        }
      }
    }
  }

  return matches;
}
```

Unlike circular geofences, polygon geofences do not use a distance-based hysteresis margin. Stability is achieved entirely through entry and exit confirmation counters.

This approach avoids artificially expanding or contracting polygon boundaries while still providing reliable behaviour in the presence of GPS jitter.

The decay of entry and exit counters between samples is unchanged from the circular implementation, ensuring consistent behaviour when GPS reception is intermittent.

7. Conclusion

Script-based geofencing on Senquip devices provides a flexible and robust way to determine asset location when simple, single-radius geofences are insufficient. By extending the existing circular geofence script with only minimal, well-defined changes, polygon geofencing can be implemented without altering the overall behaviour, stability, or commissioning process.

The use of external CSV configuration files allows geofence definitions to be updated independently of firmware, while entry and exit confirmation logic ensures reliable operation in real-world GPS conditions. Together, these approaches enable accurate location awareness across a wide range of applications, from simple depots to complex, irregularly shaped sites, using a consistent and maintainable scripting model.

8. Appendix A – Circular Geofence Script

```
load('senquip.js');
load('api_config.js');
load('api_math.js');
load('api_file.js');

// ------------------
// Configuration
// ------------------
let range = 200;         // metres - inside this is consided arrived
let exit_margin = 50;    // metres - must be this more than range to leave a site
let entry_needed = 3;     // number of positives before entry accepted
let exit_needed  = 3;

// ------------------
// State
// ------------------
let current_site = 'Initialising';
let entry = 0;
let exit  = 0;

let all_sites = [];
let file_found = 0;

// ------------------
// Remove all CR characters.
// ------------------
function normalise_newlines(str) {
  let parts = SQ.split(str, "\x0d");
  let result = "";
  for (let i = 0; i < parts.length; i++) {
    result = result + parts[i];
  }
  return result;
}

// ------------------
// Load geofence file
// ------------------
function load_sites() {
  let file_data = File.read('geofence.csv');

  if (file_data) {
    let normalised = normalise_newlines(file_data);
    all_sites = SQ.split(normalised, "\x0a");
    file_found = 1;
  } else {
    all_sites = [];
    file_found = 0;
  }
}

// ------------------
// Parse one CSV line
// ------------------
function parse_site_line(line) {
  // ok = 1 -> valid site
  // ok = 0 -> ignore silently
  // ok = -1 -> invalid lat/lon (warn)
```

```
  if (line === "") {
    return { ok: 0 };
  }

  let parts = SQ.split(line, ',');
  if (parts.length < 3) {
    return { ok: 0 };
  }

  let name = parts[0];

// Ignore header rows (with or without UTF-8 BOM)
if (name === "Site Name" || name === "\ufeffSite Name") {
  return { ok: 0 };
}

  let lat = JSON.parse(parts[1]);
  let lon = JSON.parse(parts[2]);

  if (isNaN(lat) || isNaN(lon)) {
    return { ok: -1, name: name };
  }

  return { ok: 1, name: name, lat: lat, lon: lon };
}

// ------------------
// Process all geofences
// ------------------
function process_geofences() {
  let matches = 0;
  let i = 0;

  if (current_site === 'Initialising') {
    current_site = 'Travelling';
  }

  for (i = 1; i < all_sites.length; i++) {
    let line = all_sites[i];

    if (line === "") {
      continue;
    }

    let site = parse_site_line(line);

    if (site.ok === -1) {
      SQ.dispatch_event(1, SQ.WARNING, "Invalid lat/lon for site: " + site.name);
      continue;
    }

    if (site.ok !== 1) {
      continue;
    }

    let distance = GPS.distance(site.lat, site.lon);
    if (isNaN(distance)) {
      continue;
    }

    if (distance < range) {
      matches++;
    }

    if (current_site === site.name) {
      if (distance > (range + exit_margin)) {
```

```
        exit++;
        if (exit >= exit_needed) {
          SQ.dispatch_event(1, SQ.INFO, "Leaving " + site.name);
          current_site = 'Travelling';
          exit = 0;
        }
      }
    } else {
      if (distance < range) {
        entry++;
        if (entry >= entry_needed) {
          SQ.dispatch_event(1, SQ.INFO, "Arriving at " + site.name);
          current_site = site.name;
          entry = 0;
        }
      }
    }
  }

  return matches;
}

// ------------------
// Initial load
// ------------------
load_sites();

// ------------------
// Data handler
// ------------------
SQ.set_data_handler(function(data) {
  let obj = JSON.parse(data);

  if (file_found === 0) {
    SQ.dispatch(1, "No geofence file");
    SQ.dispatch_event(1, SQ.WARNING, "geofence.csv not found or empty");
    return;
  }
  else if (typeof obj.gps_lat === "number" && typeof obj.gps_lon === "number") {

    let matches = process_geofences();

    if (matches > 1) {
      SQ.dispatch_event(1, SQ.WARNING, "Multiple geofence matches found");
    }

    entry = entry - 0.1;
    if (entry < 0) {
      entry = 0;
    }

    exit = exit - 0.1;
    if (exit < 0) {
      exit = 0;
    }

    SQ.dispatch(1, current_site);
    SQ.dispatch(3, entry);
    SQ.dispatch(4, exit);

  } else {
    SQ.dispatch(1, "No GPS");
    SQ.dispatch_event(1, SQ.WARNING, "No valid GPS");
  }

}, null);
```

9. Appendix B – Polygon Geofences

```
load('senquip.js');
load('api_config.js');
load('api_math.js');
load('api_file.js');

// ------------------
// Configuration
// ------------------
let entry_needed = 3;
let exit_needed  = 3;

// ------------------
// State
// ------------------
let current_site = 'Initialising';
let entry = 0;
let exit  = 0;

let all_sites = [];
let file_found = 0;

// Current GPS position (set by data handler)
let gps_lat = 0;
let gps_lon = 0;

// ------------------
// Remove all CR characters.
// ------------------
function normalise_newlines(str) {
  let parts = SQ.split(str, "\x0d");
  let result = "";
  for (let i = 0; i < parts.length; i++) {
    result = result + parts[i];
  }
  return result;
}

// ------------------
// Load geofence file
// ------------------
function load_sites() {
  let file_data = File.read('geofence.csv');

  if (file_data) {
    let normalised = normalise_newlines(file_data);
    all_sites = SQ.split(normalised, "\x0a");
    file_found = 1;
  } else {
    all_sites = [];
    file_found = 0;
  }
}

// ------------------
// Ray casting point-in-polygon
// Treat lon as X, lat as Y.
// Returns true if point is inside polygon.
// ------------------
function point_in_polygon(lat, lon, pts) {
  let inside = false;
  let n = pts.length;
```

```
  // Need at least 3 vertices
  if (n < 3) {
    return false;
  }

  // i = current vertex, j = previous vertex
  let j = n - 1;
  for (let i = 0; i < n; i++) {
    let yi = pts[i].lat;
    let xi = pts[i].lon;
    let yj = pts[j].lat;
    let xj = pts[j].lon;

    // Check if the horizontal ray crosses the edge (j -> i)
    // Condition: edge straddles the test latitude
    let straddles = ((yi > lat) !== (yj > lat));
    if (straddles) {
      // Compute X coordinate where the edge crosses the ray at 'lat'
      // Avoid divide-by-zero naturally because straddles implies yi != yj.
      let x_cross = (xj - xi) * (lat - yi) / (yj - yi) + xi;

      if (lon < x_cross) {
        inside = !inside;
      }
    }

    j = i;
  }

  return inside;
}

// ------------------
// Parse one CSV line (POLYGON VERSION)
// ------------------
function parse_site_line(line) {
  // ok = 1 -> valid site
  // ok = 0 -> ignore silently
  // ok = -1 -> invalid lat/lon (warn)

  if (line === "") {
    return { ok: 0 };
  }

  let parts = SQ.split(line, ',');
  if (parts.length < 7) { // name + (lat,lon)*3 minimum
    return { ok: 0 };
  }

  let name = parts[0];

  // Ignore header rows (with or without UTF-8 BOM)
  if (name === "Site Name" || name === "\ufeffSite Name") {
    return { ok: 0 };
  }

  // Parse lat/lon pairs
  let pts = [];
  for (let i = 1; i + 1 < parts.length; i += 2) {

    // STOP when trailing empty columns begin (Excel-style CSVs)
    if (parts[i] === "" || parts[i + 1] === "") {
      break;
    }

    let lat = JSON.parse(parts[i]);
```

```
    let lon = JSON.parse(parts[i + 1]);

    if (isNaN(lat) || isNaN(lon)) {
      return { ok: -1, name: name };
    }

  pts.push({ lat: lat, lon: lon });
}

  if (pts.length < 3) {
    return { ok: -1, name: name };
  }

  return { ok: 1, name: name, pts: pts };
}

// ------------------
// Process all geofences (POLYGON VERSION)
// ------------------
function process_geofences() {
  let matches = 0;

  if (current_site === 'Initialising') {
    current_site = 'Travelling';
  }

  for (let i = 1; i < all_sites.length; i++) {
    let line = all_sites[i];
    if (line === "") {
      continue;
    }

    let site = parse_site_line(line);

    if (site.ok === -1) {
      SQ.dispatch_event(1, SQ.WARNING, "Invalid polygon for site: " + site.name);
      continue;
    }

    if (site.ok !== 1) {
      continue;
    }

    // Ray casting: is the current GPS point inside this polygon?
    let inside = point_in_polygon(gps_lat, gps_lon, site.pts);

    if (inside) {
      matches++;
    }

    if (current_site === site.name) {
      // Leaving condition: outside polygon
      if (!inside) {
        exit++;
        if (exit >= exit_needed) {
          SQ.dispatch_event(1, SQ.INFO, "Leaving " + site.name);
          current_site = 'Travelling';
          exit = 0;
        }
      }
    } else {
      // Arriving condition: inside polygon
      if (inside) {
        entry++;
        if (entry >= entry_needed) {
          SQ.dispatch_event(1, SQ.INFO, "Arriving at " + site.name);
```

```
          current_site = site.name;
          entry = 0;
        }
      }
    }
  }

  return matches;
}

// ------------------
// Initial load
// ------------------
load_sites();

// ------------------
// Data handler
// ------------------
SQ.set_data_handler(function(data) {
  let obj = JSON.parse(data);

  if (file_found === 0) {
    SQ.dispatch(1, "No geofence file");
    SQ.dispatch_event(1, SQ.WARNING, "geofence.csv not found or empty");
    return;
  }
  else if (typeof obj.gps_lat === "number" && typeof obj.gps_lon === "number") {

    // Make GPS available to process_geofences()
    gps_lat = obj.gps_lat;
    gps_lon = obj.gps_lon;

    let matches = process_geofences();

    if (matches > 1) {
      SQ.dispatch_event(1, SQ.WARNING, "Multiple geofence matches found");
    }

    entry = entry - 0.1;
    if (entry < 0) { entry = 0; }

    exit = exit - 0.1;
    if (exit < 0) { exit = 0; }

    SQ.dispatch(1, current_site);
    SQ.dispatch(3, entry);
    SQ.dispatch(4, exit);

  } else {
    SQ.dispatch(1, "No GPS");
    SQ.dispatch_event(1, SQ.WARNING, "No valid GPS");
  }

}, null);
```