Elcano C4 Planner Documentation

Getting Started With C4 Planner

Before you can use the Elcano C4 Planner, you need to follow these steps. If you've already downloaded all of the Elcano files and imported all of the libraries, you can skip these steps.

1. **Clone the Elcano repository**
   Go to https://github.com/elcano/elcano and use your preferred method to clone the repository. For example, if you want to clone it using https, you would select "HTTPS" on the right-hand side, click "Copy to clipboard," and paste it into your git interface.

   If you're using Git Bash, you would type in:

   ```
   $ git clone https://github.com/elcano/elcano.git
   ```

2. **Open the C4 Planner file**
   From the Arduino IDE, select File > Open and navigate to \elcano\Elcano_C4_Planner\Elcano_C4_Planner.ino

3. **Add the Elcano libraries**
   In the Arduino IDE, select Sketch > Import Library > Add Library. Navigate to \elcano\libraries\Common. Repeat for \elcano\libraries\IO. These are the minimum required for C4 Planner, though you may want to import the rest if you will be working with other Elcano files.

Once the repository is cloned and the libraries are added you should be able to compile the C4 planner .ino file.

Elcano C4 Planner File System

MAP_DEFS.TXT

This file provides the latitude and longitude coordinates for each of the maps, followed by the file name for that map. Commas also separate each map.

The format should be as follows:

```
latitude_0,longitude_0,filename_0.txt,
...,
latitude_n,longitude_n,filename_n.txt
```

A practical example would look like this:

```
47.758949,-122.190746,MAP001.txt,
47.6213,-122.3509,MAP002.txt
```

Map Files

These files provide the junction data. The junction struct has the following variables:

long east_mm
long north_mm
int destination[4]
long Distance[4]

- east_mm is the position East of the origin in millimeters
- north_mm is the position North of the origin in millimeters
- destination is an array of indeces into the Nodes[] array that connect to this node
- Distance is an array of longs holding the distances from this node to each of the destinations in millimeters.

The file format is a comma delimited list of the values in the struct, with a comma after each junction. It should be formatted as follows:

```
east_mm_0,north_mm_0,destination_0[0],destination_0[1],destination_0[2],des
tination_0[3],Distance_0[0],Distance_0[1],Distance_0[2],Distance_0[3],
...,
east_mm_n,north_mm_n,destination_n[0],destination_n[1],destination_n[2],des
tination_n[3],Distance_n[0],Distance_n[1],Distance_n[2],Distance_n[3],
```

A practical example would look like this:

```
-183969,380865,1,2,END,END,1,1,1,1,
-73039,380865,0,3,7,END,1,1,1,1,
-182101,338388,0,3,4,5,1,1,1,1
```

File Names

The Arduino SD Card library uses a FAT file system. FAT file systems have a limitation when it comes to naming conventions. You must use the 8.3 format, so that file names look like “NAME001.EXT”, where “NAME001” is an 8 character or fewer string, and “EXT” is a 3 character extension. People commonly use the extensions .TXT and .LOG. It is possible to have a shorter file name (for example, mydata.txt, or time.log), but you cannot use longer file names.

Elcano C4 Planner Code Descriptions

Initial position. Specifies the starting location and orientation. Velocity is zero.
If this is a file, it is read by C4 (Path Planner) and passed to C6 (Navigator).
If it is user input, it is read by C6 (Navigator).

The present C4 module is the only micro-controller that has a file system.

<u>USB:</u> All Arduinos contain a USB port that lets them download code from a PC.
Since a USB connection is not a network, the Arduinos cannot talk to each other over USB.
To enable USB, one of the Arduinos must include a USB server. If there is a USB server, it might be
C4, which may have an OS, or
C6, which needs to talk to lots of instruments, or
C7, which may have a USB link to a camera, or
we could have another processor whose sole function is communication.

Nodes and Links define the road network.
The path for the robot is an **array of pointers to Nodes**.
LATITUDE_ORIGIN and LONGITUDE_ORIGIN define (0,0) on the mm coordinate system.
Start is the position of the robot as it begins a new leg of the journey.
Destination is the next goal position in mission.
Path joins Start and Destination. Finding it is the major task of this Path Planner module.
The **Distance** numbers are *initially multipliers* giving path roughness.
They are *replaced by their product* with the actual distance.
Route is a finer scale list of waypoints from present or recent position.
Exit is the route from Path[last]-> location to Destination.

Functions and data

Defines

PI ((float) 3.1415925)
NULL 0
START -1
EMPTY -2
CONES 5
DESIRED_SPEED_mmPs 2235 // == 5mph, conversion chart in file
MAX_WAYPOINTS 40

```
MAX_MAPS 10

void DataReady();            //an interrupt is attached to this in the setup, must be
implemented
extern bool DataAvailable;   //TODO: implement this bool and the above function
```

Goals

These are hardcoded POIs that make up the planned route

```
long goal_lat[CONES] = {  47621881,   47621825,   47623144,   47620616,   47621881};
long goal_lon[CONES] = {-122349894, -122352120, -122351987, -122351087, -122349894};


struct curve Links[20];                    // the links between junctions
struct junction Nodes[MAP_POINTS]          // the hardcoded lat&long, nodes pointed to, and
                                           // cost multipliers for each junction node
struct AStar
{
   int ParentID;
   long CostFromStart;
   long CostToGoal;
   long TotalCost;
} Open[MAP_POINTS];       // list used in A* to meant to track unvisited nodes

class waypoint Origin, Start;
waypoint Path[MAX_WAYPOINTS];              // course route to goal
waypoint FinePath[MAX_WAYPOINTS];          // a low level part of path that smoothes the
corners.

waypoint mission[CONES];                   // aka MDF
int waypoints = CONES;
```

void ConstructNetwork(junction *Map, int MapPoints)

Takes an array of junctions (intersections) and an integer representing the number of junctions in the array as input. Calculates the distance for all of the edges in the junction and updates the distance value from the multiplier value the array is created with to the distance in mm.

void GetGoals(waypoint *Waypoint, int Goals)

Takes the Mission waypoint array and constructs the Goals array for building the path to take. Currently does not use the *Waypoint or Goals arguments, and instead uses the global Mission and Goals arrays. Can be updated to use the arguments passed in or to remove the arguments if needed.

long distance(int i, int *k, long east_mm, long north_mm, int* perCent)

Takes the Nodes index (i), a pointer to an index (*k), the east and north coordinates in millimeters, and the percentage of the segment completed and calculates the distance. This is used by the FindClosestRoad function.

void FindClosestRoad(waypoint *start, waypoint *road)

Takes two waypoints, finds the closest junction to the first waypoint and points the second waypoint to that junction. For example, if your map has junctions at (1,1), (-1,1), (1, -1), and (-1,-1) and you pass in your starting waypoint as (2,2), the road waypoint will be updated to (1,1), which is the closest junction to (2,2).

int BuildPath (int j, waypoint* start, waypoint* destination)
int FindPath(waypoint *start, waypoint *destination)
int PlanPath (waypoint *start, waypoint *destination)

These three functions work together to implement the A* pathing. They take in start and destination waypoints which are on the road network given in Nodes and set the waypoints on the path starting with Start and ending with Destination. A* is traditionally done with pushing and popping node from an Open and Closed list. However, since we have a small number of nodes, we instead reserve a slot on global variables Open and Closed for each node.

void SendPath(waypoint *course, int count)

Sends the path to the C3 pilot module via Serial bus.

boolean LoadMap(char* fileName)

The LoadMap function loads the map nodes from a file. It takes in the name of the file to load and loads the appropriate map. LoadMap returns true if the map was loaded. It returns false if the load failed.

void SelectMap(waypoint currentLocation, char* fileName, char* nearestMap)

The SelectMap function determines which map to load. It does this by taking in the current location as a waypoint and a string with the name of the file that contains the origins and file names of the maps. The function then determines which origin is closest to the waypoint and returns the filename as a string. It assumes that the file is formatted according to the specifications.

void initialize()

Initializes the Origin, reads the map definitions from SD, loads the appropriate map file, and builds the desired path based on the mission files.