INSTITUTO UNIVERSITARIO AERONÁUTICO

Actualización y mejoras sobre el software IUA-GCS

Informe Técnico: DMA 18/17 **Revisión:** 1.01

Proyecto: PIDDEF 038/14 - Paracaídas Comandado Autónomo

Fecha: 10/11/2017

Autor/es: Germán Weht

Revisó:

Ing. Diego Llorens
Investigador

Vo.Bo:

Ing. Andrés Liberatto
Investigador

Índice

1. Introducción **3**

2. Desarrollo **3**

2.1. Modificaciones gráficas 3

2.2. Modificación de las librería OpmapWidget 6

2.2.1. Estructura de datos de los waypoint 6

2.2.2. Impresión de velocidad y dirección de viento sobre mapa 8

2.3. Implementación protocolo de comunicación 10

2.4. Actualización de las rutinas 11

3. Conclusiones **12**

4. Bibliografía **13**

"Actualización y mejoras sobre el software IUA-GCS"

Por

Germán Weht

Resumen

En el marco del PIDDEF 38-14 "Paracaídas Comandado Autónomo", se realiza una descripción general de las modificaciones (Ground Control Station) utilizando Qt Creator 5. Dentro de las más relevantes se encuentran la modificación sobre el panel de instrumentos, la librería *opmapwidget* y la implementación del protocolo de comunicación *adsLink*.

La finalidad del programa es hacer el seguimiento y navegación del vehículo cuando este vuele en condiciones de vuelo manual y automático.

El código esta basado en rutinas propias y librerías OpenSource.

Córdoba, 10 de noviembre de 2017

1. Introducción

2. Desarrollo

Continuando con desarrollo del software IUA-GCS [1], se implementaron nuevas capacidades al programa de base terrena. Estas modificaciones fueron realizadas a fin de mejorar la comunicación entre el vehículo y el usuario, mostrando y emitiendo mensajes gráficos en las diferentes etapas de la misión. Cabe mencionar que la mayoría de las modificaciones fueron implementadas para el desarrollo del vehículo y no para usuario final, mostrando datos relevantes para el análisis de los algoritmos de control e ingeniería.
A continuación se listan las modificaciones más relevantes respecto de la versión anterior del software [1].

2.1. Modificaciones gráficas

Dentro de las modificaciones gráficas más relevante se lista la eliminación de la pestaña *Serial Port*. Solo se conservó la *dockwidget* que contiene las opciones de configuración del puerto serie. En principio no sería necesario conservar la ventana móvil *Serial Port*, pero por razones de desarrollo se la conserva.
La ventana *Flight Instruments* fue modificada a fin de agregar datos útiles para la navegación. A continuación se muestra una imagen con la nueva configuración:

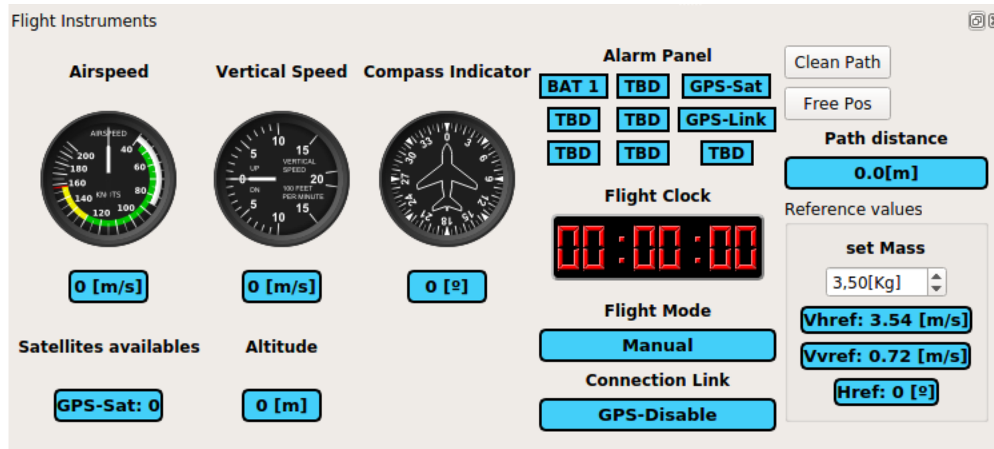


Figura 1: Ventana móvil *Flight Instruments* con la nueva distribución de instrumentos e indicadores.

De la figura (1) se observa que se modificaron los instrumentos y la disposición respecto a la versión anterior, se quitó el *Primary Flight Display* y se acomodaron de manera horizontal los indicadores *Airspeed, Vertical Speed y Compass Indicator* respectivamente. Debajo de cada instrumento de vuelo se agregó un indicador de texto de cada variable. Además de estos tres indicadores de texto se agregaron dos más que aportan datos sobre la altitud y la cantidad de satélites disponibles por la antena GPS.
Sobre la parte central del panel *Flight Instruments* se dispuso el módulo de alarmas, actualmente se hace uso 3 de las 9 implementadas; las utilizadas son: límite de carga de la batería del autopiloto $BAT1$, cantidad de satélites vistos por la antena GPS $GPS-Sat$ y conexión del GPS $GPS-Link$. Cada una de estas se inicializa en la siguiente rutina ubicada en el archivo *mainwindows.cpp*.

```
void MainWindow::FlightInstrumentsDockWidget(){
  /*
    Some other code
  */
  // Alarm panel
  alarm1 = new QGermanAlarm("BAT 1");
  alarm2 = new QGermanAlarm("TBD");
  alarm3 = new QGermanAlarm("GPS-Sat");

  alarm4 = new QGermanAlarm("TBD");
  alarm5 = new QGermanAlarm("TBD");
  alarm6 = new QGermanAlarm("GPS-Link");

  alarm7 = new QGermanAlarm("TBD");
  alarm8 = new QGermanAlarm("TBD");
  alarm9 = new QGermanAlarm("TBD");

}
```

Código 1: Fragmento de código de inicialización del panel de alarmas.

La clase *QGermanAlarm* chequea el valor de la variable, si está por debajo del umbral fijado se dispara, modificando el color del cuadro de texto en el panel de alarmas. En el siguiente fragmento de código se muestra donde se ejecuta esta acción y actualiza el entorno gráfico.

```
void MainWindow::UpdateAlarms(){
   alarm1->changeBackgroundCritic(10, Qvoltaje);
   alarm1->update();
   alarm3->changeBackgroundCritic(4,Qsatellites);
   alarm3->update();
   alarm6->changeBackgroundCritic(1, Qfix_type);
   alarm6->update();
}
```

Código 2: Código de actualización del panel de alarmas.

Del código expuesto en (2) se puede observar que el método *changeBackgroundCritic(4, Qsatellites)* necesita de dos valores, el primero es el umbral de corte y el segundo la variable a la cual se le quiere aplicar el corte, por lo tanto, cuando la variable *Qsatellites* tome el valor menor o igual a 4, la alarma se disparará. Finalmente el método *update* actualiza el entorno gráfico.

Debajo del panel de alarmas se ubica el reloj de vuelo, este se activa cuando se tiene conexión con el vehículo, esto se da cuando abre el puerto serie. Se detiene cuando se cierra el puerto serie.

Por último, sobre la parte central, se tiene dos indicadores de texto *Flight Mode* y *Connection Link* que muestran el modo de vuelo y si la antena GPS tiene conexión, respectivamente. Actualmente se tienen cargados 7 modos de vuelo: *Manual, ADSFlareStep, ADSYawStep, ADSHeading, ADSGuidance, ADSWndEst, yADSnavigation*. Para poder agregar modos de vuelo hay que entender el funcionamiento de la rutina. A continuación damos una breve explicación de su

implementación: el sistema, a través de del protocolo de comunicación *adslink*, recibe un mensaje con el modo de vuelo encriptado bajo un entero sin signo de 8 bit. Además, se tiene un vector de caracteres *flight_mode_strings*[] con los modos de vuelo listados anteriormente. El modo de vuelo queda almacenado en la base terrena en la variable *Qmode* definida en el archivo Qglobalsvar.cpp. Haciendo uso de la rutina de parseo *FlightModeParser*(*uint8_t mode*), se ingresa con la variable *Qmode* y el método devuelve el modo de vuelo como string de caracteres. A continuación se muestran fragmentos de código donde se define el vector de caracteres y cómo hacer uso de la rutina de parseo:

```
static const char* flight_mode_strings[] = {
    "Manual",
    "ADSFlareStep",
    "ADSYawStep",
    "ADSHeading",
    "ADSGuidance",
    "ADSWndEst",
    "ADSnavigation",
    "",
    "",
    "",
    "Unknown mode"
};
```

Código 3: Definición del vector de caracteres con la lista de modos de vuelo ubicado en en archivo *QGlobalvar.h*

```
void MainWindow::UpdateFlightInstruments(){
  /*
    Some code
  */
  // update flight mode text
  Fmode->setText(FlightModeParser(Qmode));
}
```

Código 4: Uso de la función *FlightModeParser*(*uint8_t mode*).

En la derecha se encuentran dos botones *Clean Path* y *Free Pos*. Cuando el vehículo se mueve sobre el mapa, este deja una línea roja que indica el recorrido; *Clean Path*, elimina del mapa el recorrido hecho por el vehículo y los *waypoints* cargados o recibidos por la comunicación. En la figura (2) se observa el recorrido que realizó el vehículo sobre la superficie junto a los *waypoints*.

Figura 2: Recorrido hecho por el vehículo a través de los waypoint.

La librería que muestra el mapa y el vehículo tiene implementado, por default, que por cada actualización de punto GPS el mapa se centre sobre el vehículo y lo enfoque a una altura prefijada. El botón *Free Pos* elimina esa opción permitiendo moverse libremente sobre el mapa tanto en plano como en altura.

Siguiendo con la descripción de los indicadores de vuelo y ayudas de navegación (ver figura (1)), sobre la derecha se tiene un indicador de texto *Path distance* que muestra la distancia recorrida por el vehículo. Finalmente, debajo de éste último, se encuentran 3 valores de referencia: velocidad horizontal Vh_{ref}, velocidad vertical Vv_{ref} y rumbo de referencia H_{ref} (ver referencia [2] para mayor detalle). Estos valores son obtenidos a partir de la masa del paracaídas y tienen la finalidad de conocer las velocidades máximas teóricas, en ambas direcciones, en condiciones normales de operación. Como trabajo futuro, se propone implementar estos valores de referencia como alarmas dentro del panel *Alarm Panel* a fin de indicar, si algún valor es superado, un desperfecto o funcionamiento anormal del paracaídas.

2.2. Modificación de las librería OpmapWidget

Se realizaron varias modificaciones sobre la librería *OpmapWidget*, entre las más destacadas se listan:

- Modificación estructura datos de los waypoint.
- Impresión de velocidad y dirección de viento sobre mapa.

A continuación se explican de forma detallada cada una de ellas.

2.2.1. Estructura de datos de los waypoint

Actualmente la base terrena recibe el plan de vuelo a través de la telemetría. Esto implica que cada waypoint debe ser procesado (ver referencia [3]) e impreso en el mapa por la librería *OpmapWidget*.
El paquete de información que contiene un mensaje de waypoint es la siguiente:

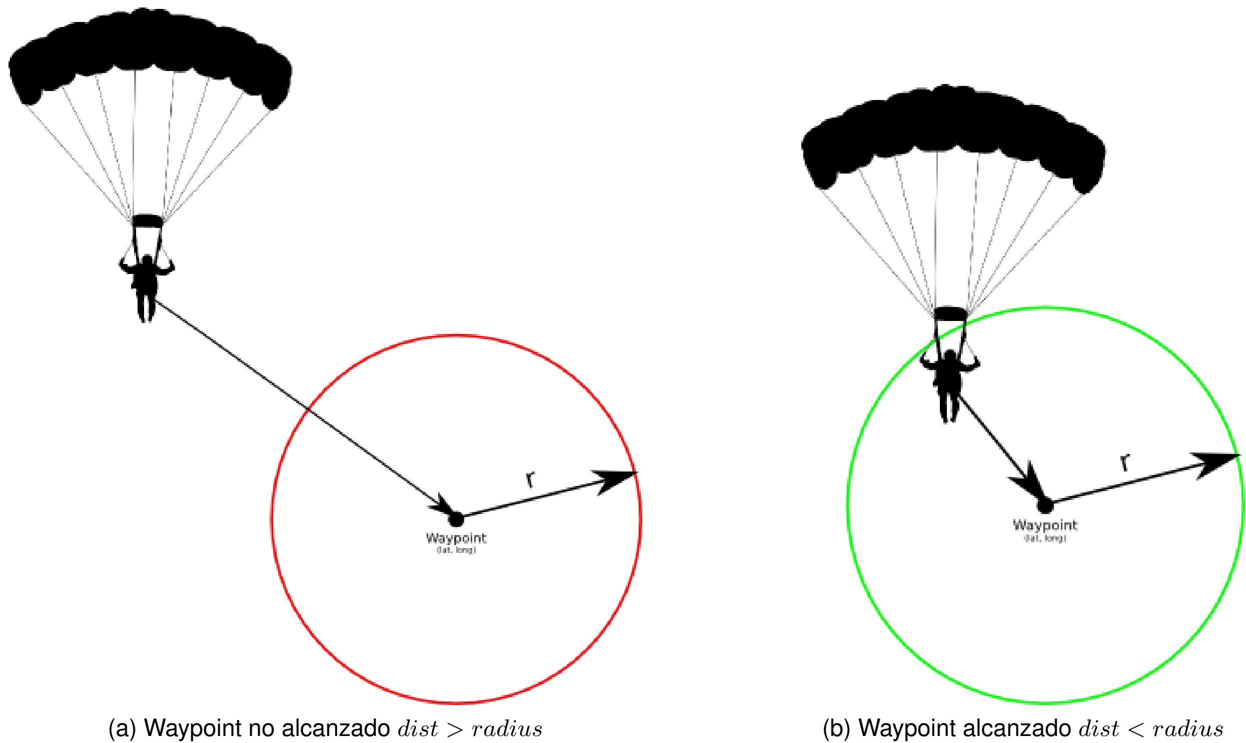
- id

- latitude
- longitude
- altitude
- radius
- heading

Mediante el id se identifica que waypoint es. Existen 5 waypoints reconocibles por la base terrena, *waypoint A, B, C, D y landing* (ver referencia [2] para conocer el significado de cada uno de ellos). Cada vez que la base terrena recibe un waypoint, desde la librería *Qadslink* se emite una señal para que la función *void MainWindow::NewWayPointAdd*(*uint8_t Identity*), ubicada en el archivo *mainwindow.cpp* reconozca a través del id que waypoint es. La variable $Identity$ es de tipo numérica (entera sin signo de 8 bit) que almacena el nombre del waypoint. Mediante una rutina de parseo, similar a la utilizada en los modos de vuelo, se reconoce para luego ser impreso sobre el mapa. En la figura (2) se observan los waypoints con un valor numérico asociado.
Los datos $latitude$ y $longitude$ son utilizados para posicionar el waypoint sobre el mapa. La variable $altitude$ es utilizada con dos fines, indicar la altitud y el tipo de waypoint. Se disponen de dos tipos de waypoints, planos (2D) y tridimensionales (3D) ver figura (3) y (4). Los waypoints planos sirven para indicar la posición del waypoint en el mapa sin importar la altura a la este se ubica, en este caso el cálculo de la distancia se realiza de forma plana $(lat\ , long)$ (un disco sobre el mapa). En los tridimensional si importa la altura y la distancia es calculada utilizando la altura $(lat\ , long,\ alt)$, (una esfera). La forma de distinguir entre un waypoint 2D y 3D es mediante el valor de la variable $altitude$, el pseudocódigo (1) explica como funciona la detección. En la figura (2) los waypoint 1, 2 y 4 son del tipo 2D y el 3 y 7 del tipo 3D.
La variable $raduis$ representa el radio de un círculo, para waypoints 2D, o una esfera, para los 3D, centrado en el origen del waypoint $(lat, long)$. Se dice que un waypoint es alcanzado por el vehículo, cuando la distancia del vehículo al waypoint es menor que el valor dado por $radius$. Para indicar esta condición gráficamente en el mapa, el waypoint cambia de color rojo a verde como se puede observar en la figura (3) y figura (4). La implementación de estos dos waypoints corresponde a la necesidad de seguir la navegación gruesa con waypoints 2D (aproximación al objetivo) y fina con waypoints 3D (descenso y aterrizaje).
Finalmente la variable $heading$ muestra el valor de rumbo por el cual el vehículo ingresa al waypoint.

Algorithm 1 Detección de waypoints 2D-3D

1: **if** *altitude* < 0 **then**
2: $waypoint\ 2D$
3: $dist = ||\,(lat,\ long)_{Waypoint} - (lat,\ long)_{Vehicle}\,||$
4: **else**
5: $waypoint\ 3D$
6: $dist = ||\,(lat,\ long,\ alt)_{Waypoint} - (lat,\ long,\ alt)_{Vehicle}\,||$
7: **end if** ▷ La distancia se calcula mediante haversine [4].



(a) Waypoint no alcanzado $dist > radius$

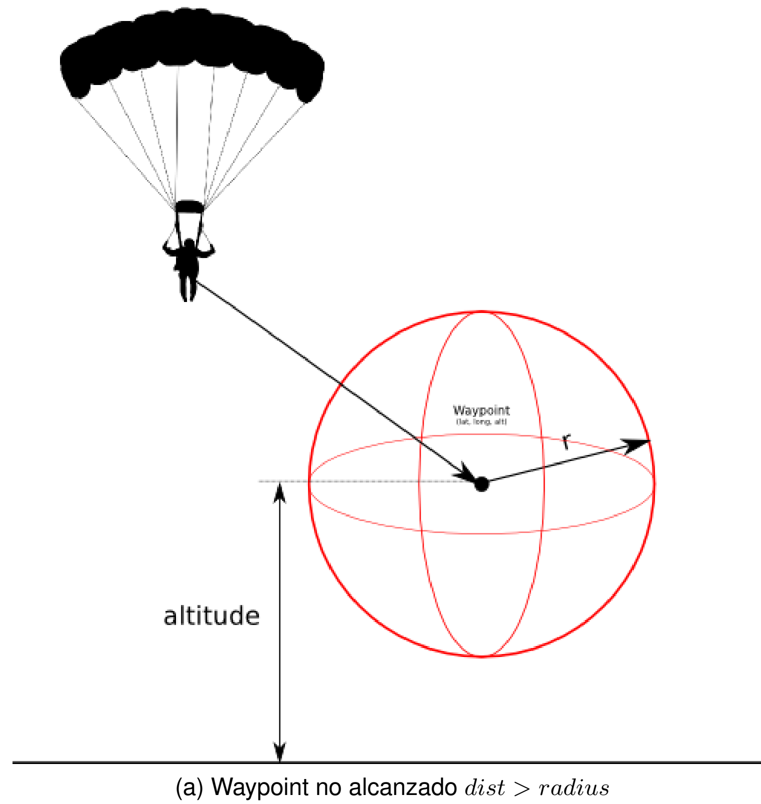
(b) Waypoint alcanzado $dist < radius$

Figura 3: Waypoints plano (2D), representado como un disco.

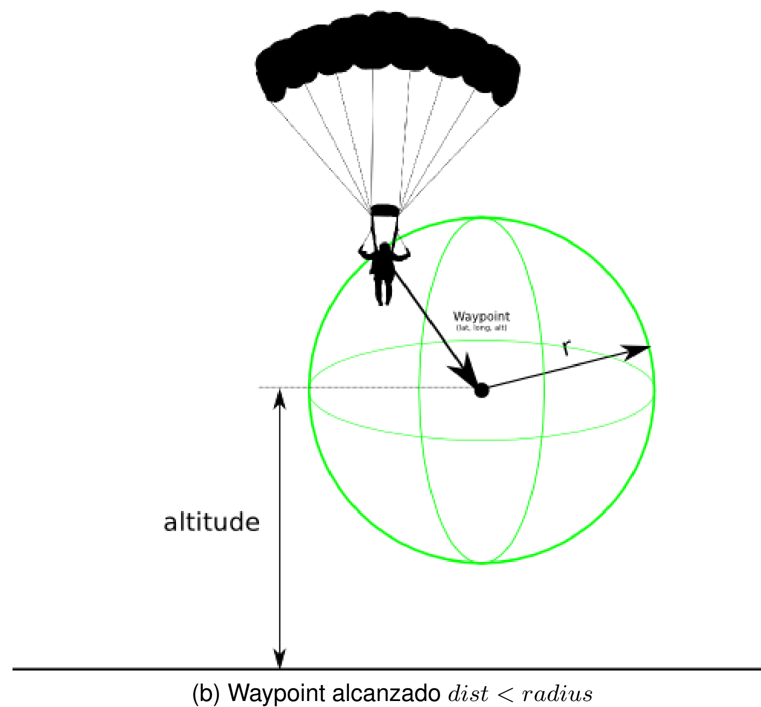
Para representa gráficamente las variables descriptas anteriormente, se utilizo una ventana de aviso tipo (*tooltip*) que aparece cuando hacemos hover sobre el waypoint. Los archivos que fueron modificados de la librería *OpmapWidget* son *waypointitem.cpp*, *uavitem.cpp* y *opmapwidget.cpp*.

2.2.2. Impresión de velocidad y dirección de viento sobre mapa

El paracaídas autónomo comandado dispone de un controlador (ver referencia [2]) que permite estimar la velocidad y dirección del viento. De la misma manera que se reciben los waypoint, se implemento una rutina que cuando se reciba el mensaje de estimación de viento, este se procese para imprimir los valores de forma gráfica sobre el mapa. Para ello se modificaron los archivos *opmapwidget.h* y *opmapwidget.cpp*. La lógica implementada es la siguiente: Se recibe el mensaje con los datos de velocidad y dirección (*adsWndEst*), este es procesado por el protocolo de comunicación. Dentro del método *QAdsLink::Update()* se emite una señal; esta es recibida por el método *MainWindow::PrintWndEst()* ubicada en el archivo *mainwindow.cpp*. *MainWindow::PrintWndEst()* es la encargada de imprimir en pantalla los datos de velocidad de viento y dirección. En la figura (5) se observa la dirección y velocidad de viento estimada, impresa en la esquina superior izquierda del mapa.



(a) Waypoint no alcanzado $dist > radius$



(b) Waypoint alcanzado $dist < radius$

Figura 4: Waypoints tridimensionales (3D), representado como una esfera.

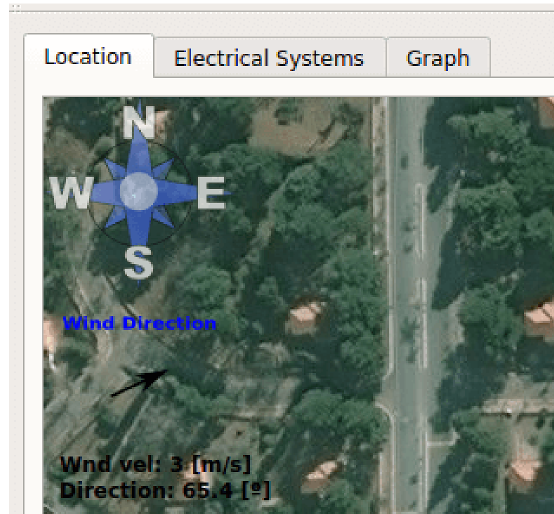


Figura 5: Impresión de velocidad y dirección de viento sobre mapa mediante el método *MainWindow::PrintWndEst()*.

2.3. Implementación protocolo de comunicación

Para anexar el protocolo de comunicación *adsLink* (referencia [3]) a la base terrena, fue necesario generar una nueva clase llamada *QAdsLink*. Esta clase es similar a la *adslink* utilizada en el código del autopiloto [5], adaptada a las librerías estándar de Qt. De igual manera que en el autopiloto, se tiene 2 archivos donde se declaran las variables globales *QGlobalvar.h* y *QGlobalvar.cpp*. Las variables declaradas dentro de estos dos archivos cumplen la función de interfaz entre el protocolo de comunicación *QAdsLink* y la clase principal de la base terrena *mainwindow*.
Los mensajes utilizados dentro del protocolo *adsLink* están implementados, por razones de eficiencia, con variables de tipo enteras. Esto genera la necesidad de tener que moldear algunas variables de manera de compatibilizarlas con el tipo que se le asigno en la base terrena. Se implementó un método (*MainWindow::UpdateQglobalVars()*) que hace una copia del valor de las variables *QGlobalvar* a un valor local moldeado al tipo que requieren las distintas librerías. Para dar mayor claridad a lo expresado en palabras, se muestra el método que realiza las operaciones de moldeado o *casteo*:

```
void MainWindow::UpdateQglobalVars(){
    Qlatitude=latitude/1.e7;
    Qlongitude=longitude/1.e7;
    Qaltitude=altitude/100.;
    Qmode =mode;
    Qsatellites=satellites;
    Qfix_type=fix_type;

    Qheading=heading/100.;
    Qgspeed=ground_speed/100.;
    Qwind_vel = wind_Vel/100.;
    Qwnd_dir = direction/100.f;

    Qvoltaje = voltaje/100.;
    Qcurrent =current/100.;
}
```

Código 5: Método encargado de moldear las variables *Qglobalvar* al tipo que requieren las librerías utilizadas por la base terrena.

De (5) vemos que la variable *Qlatitude* toma el valor de *latitude* dividida por en factor de $1.e7$. Esto es porque la variable *latitude* es de tipo $int32_t;$ y la variable *Qlatitude* es del tipo $double$.

2.4. Actualización de las rutinas

Dentro del archivo *mainwindow.cpp* se encuentran 3 métodos *void UpdateQuickValues(), void UpdateMidValues(), void UpdateLowValues()*, estos son los encargados de actualizar los distintos grupos de variables y opciones gráficas. Cada uno de ellos se ejecuta a una velocidad fija de 250, 500 y 2000 milisegundos respectivamente. La implementación de estos métodos se realizó mediante objetos *QTimer*. A continuación se muestra el fragmento de código donde se exponen los 3 métodos y las variables que actualizan.

```
void MainWindow::UpdateQuickValues(){
    if(serialPort->isOpen()){
        updateBoxVariables();
        updatePlot();
        if(recordBtnFlag) setVectorsVal();
        if(fix_type>1) updateGPSpoint();
        // refresh at 250
    }
}

void MainWindow::UpdateMidValues(){
    if(serialPort->isOpen()){
        UpdateFlightInstruments();
        UpdateAlarms();
        updateElectricalSystemsTab();
        updateLauchHome();
        UpdateTitles();
        vehiculeLink->getgcsLinkStats();
        // refresh at 500
    }
}
void MainWindow::UpdateLowValues(){
    if(serialPort->isOpen()){
        if(Qfix_type<1) PlaySound("/sounds/GPSLost.mp3");
    }
    // refresh at 2000msec
}
```

Código 6: Métodos encargados de realizar la actualización de variables numéricas y entorno gráfico

Además de los *QTimer* anteriores se dispone de uno más (*updateVehiculeLinkTimer*) que es el encargado de chequear la telemetría. La conexión se actualiza cada 10 milisegundos. Respecto de los otros timers, la velocidad es muy alta, pero de esta manera aseguramos que los datos recibidos no se perderán ni que el buffer de entrada se sature y sobrescriba los mensajes.

3. Conclusiones

Se realizaron las modificaciones y mejoras sobre el software IUA-GCS. Se agregaron nuevos indicadores en la ventana *Flight Instruments*, se acomodaron de manera horizontal los indicadores *Airspeed, Vertical Speed y Compass Indicator*. Se adaptaron funciones de la librería *OpMapWidget*, la estructura datos de los waypoint y la impresión de la velocidad y dirección de viento estimada. Se incluyo el protocolo de comunicación *adslink*, permitiendo aumentar el volumen de información a procesar.
La actualización del entorno gráfico y las variables mediante objetos *QTimer* mejoro sustancialmente el desempeño del software en computadoras con recursos limitados.

4. Bibliografía

[1] G. Weht, “Descripción general software base terrena iua-gcs,” IUA, Departamento de Aeronáutica, Tech. Rep. 002/16, 09 2016.

[2] A. H. Liberatto, “Modelo matemático & algoritmos de control primario,” IUA, Departamento de Aeronáutica, Tech. Rep. Nan, 09 2017.

[3] D. Llorens, “Protocolo de comunicación para microcontroladores de 8 bit,” IUA, Departamento de Aeronáutica, Tech. Rep. 013/17, 08 2017.

[4] Wikipedia, “Haversine formula,” https://en.wikipedia.org/wiki/Haversine_formula, 2017, [Online; accessed August-2017].

[5] D.Llorens, “Descripción del código de autopiloto para paracaídas comandado autónomo,” IUA, Departamento de Aeronáutica, Tech. Rep. 014/17, 09 2017.