INSTITUTO UNIVERSITARIO AERONÁUTICO
Departamento Mecánica Aeronáutica

DESCRIPCIÓN DEL CÓDIGO DE AUTOPILOTO PARA PARACAÍDAS COMANDADO AUTÓNOMO

Informe técnico: DMA-014/17 **Revisión:** /

Proyecto: PIDDEF 038/14 - “Paracaídas Comandado Autónomo”

Fecha: 13/09/17

Autor:

Ing. Diego Llorens
Investigador

Revisó:

Ing. Germán Weht
Investigador

DESCRIPCIÓN DEL CÓDIGO DE AUTOPILOTO PARA PARACAÍDAS COMANDADO AUTÓNOMO

Por:

Ing. Diego Llorens

RESUMEN

En el marco del proyecto PIDDEF 038/14 – “Paracaídas Comandado Autónomo”, se realiza una descripción general del código fuente que se está desarrollando para el control de un paracaídas para el lanzamiento de cargas.

El programa de autopiloto se encuentra dividido en dos bloques principales: uno asociado a la interacción con el hardware y el lazo principal de ejecución, y otro en donde se implementan las funcionalidades particulares de los sistemas de control, guiado y navegación para el comando de un paracaídas.

Los sistemas particulares que se encuentran implementados para el control de un paracaídas se componen de una librería de controladores de rumbo en donde se puede elegir la arquitectura del controlador a utilizar, un módulo de guiado sencillo a un punto, un módulo de navegación que tiene implementada una estrategia de pérdida de altura entre dos puntos y una aproximación final de cara al viento para aterrizar y un módulo de estimación de velocidad y dirección de viento en vuelo.

Córdoba, 13 de septiembre de 2017

ÍNDICE

ABREVIACIONES

Abreviación	Descripción
CPU	Central Processing Unit
EEPROM	Electrically Erasable Programmable Read-Only Memory
I^2C	Inter-Integrated Circuit
MCU	Micro Controller Unit
MIPS	Millons Instruccions Per Sencond
MSL	Mean Sea Level
PWM	Pulse Width Modulation
RAM	Random Acces Memory
RISC	Reduced Instruction Set Computer
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver-Transmitter

SISTEMA DE REFERENCIA Y CONVENCIÓN DE SÍMBOLOS

Símbolo	Unidad	Descripción
chIn	[-]	Valor en unidades de desplazamiento de un canal de entrada
chOut	[-]	Valor en unidades de desplazamiento de un canal de salida
G_1	[-/µs]	Relación de transformación de valores PWM a unidades físicas
G_2	[µs/-]	Relación de transformación de unidades físicas a valores PWM
inChInv	[-]	Valor para invertir el signo de un canal de entrada
$inPWM_{raw}$	[µs]	Valor sin procesar de un canal de entrada en valores de PWM
PWM_{ref}	[µs]	Valor de referencia de un canal de entrada en valores de PWM
outChInv	[-]	Valor para invertir el signo de un canal de salida
$outPWM_{raw}$	[µs]	Valor de un canal de salida en valores de PWM a escribir en el hardware
$outPWM_{ref}$	[µs]	Valor de referencia de un canal de salida en valores de PWM

1. INTRODUCCIÓN

El código fuente para el autopiloto del paracaídas comandado autónomo es una bifurcación del código de autopiloto ArduPilotMega. Se utilizó este desarrollo ya que se dispone hardware operativo para el mismo y se tiene experiencia previa en la implementación de algoritmos de control sobre esta plataforma [1] [2].

Para el desarrollo del código de autopiloto para el paracaídas se utilizará la librería existente de sensores y la estructura básica de funcionamiento del programa ArduPilotMega (inicialización y bucle principal). Sobre esta estructura se agregarán las funcionalidades específicas y los sistemas de control necesarios para la aplicación de paracaídas comandado autónomo para entrega de cargas.

En este informe se presentará un explicación general del código describiendo los diferentes módulos que componen el programa del autopiloto.

2. DESARROLLO

El código fuente del autopiloto está desarrollado para el hardware ArduPilotMega v1.0; este último está formado por dos placas en donde se encuentran integrados un conjunto de sensores y 2 microcontroladores que se usan para realizar el control del vehículo. Los detalles de estas placas se pueden consultar en la referencia [3].

En particular para el desarrollo del software interesa conocer las características del microcontrolador principal sobre el cual se embebe el software del autopiloto ya que las mismas, al ser limitadas en algunos aspectos, pueden restringir las estructuras de datos y los patrones de software a utilizar. La versión de hardware v1.0 de ArduPilotMega utiliza el microcontrolador ATmega 2560 [4] como computadora principal; en la Tabla 1, a continuación, se muestran las principales características técnicas del mismo.

Parámetro	**Valor**
Tipo de memoria para programa	Flash
Tamaño de memoria para programa	256 [KBytes]
Tamaño de memoria	4096 [Bytes]
Tamaño de memoria RAM	8192 [Bytes]
Arquitectura del CPU	8bit
Velocidad del CPU	16 [MIPS] @ 16 [MHz]
Set de instrucciones del CPU	Advance RISC
Cantidad de registros del CPU para uso general	32 x 8bit c/u
Unidad de punto flotante disponible	No
Periféricos para comunicación digital	4-UART, 5-SPI, 1-I^2C

Tabla 1: Caracterísitcas técnicas MCU ATmega 2560 [4]

Para el desarrollo inicial del software interesa considerar los siguientes datos de la tabla anterior:

- *Tamaño de memoria para programa:* esto limita la cantidad de datos que se pueden almacenar en el programa principal en forma de tablas, cadenas de texto y otros, además del espacio que ocupan las funciones del programa
- *Tamaño de memoria RAM:* por defecto el compilador ubica todas las variables y estructuras de datos y cadenas de texto en el espacio de memoria que se utiliza como memoria RAM a menos que se indique, mediante instrucciones especiales, que haga uso de la memoria flash para programa. En este sentido es mandatorio tratar de ubicar todas las cadenas de caracteres en la memoria flash así como las estructuras de datos grandes y que no requieran acceso constante desde el programa. Es deseable que durante la ejecución del programa esté disponible por lo menos el 50% de la memoria RAM para poder usarla con las llamadas a funciones.
- *Arquitectura del CPU:* es recomendable utilizar el tamaño de la arquitectura del CPU para la mayoría de las variables que se declaren (siempre que se a posible). Esto reduce la cantidad de instrucciones que tiene que usar el CPU para operar sobre una variable. En este caso los tipos que tienen que tener preferencia son enteros con y sin signo de 8bit (uint8_t y int8_t).
- *Tamaño de registros para uso general:* el microcontrolador utiliza estos registros para realizar las operaciones programadas en el código fuente del programa. En general, las operaciones con registros son más rápidas que una operación equivalente con acceso a memoria RAM. Si las variables que están siendo usadas tienen un tamaño mayor que el tamaño de los registros el CPU tiene que hacer mayor cantidad de operaciones para manejar dichas variables.
- *Unidad de punto flotante:* si no se dispone de una unidad dedicada a realizar operaciones con números de punto flotante, las mismas se realizan mediante implementaciones de software lo que las transforma en posibles puntos de demora en el código. Es por esto que hay que favorecer las operaciones con enteros sobre las de punto flotante.

Se tratará de aplicar estas consideraciones durante el desarrollo de los módulos siempre que sea posible.

2.1 Organización general del código de autopiloto

En las Figura 1 y Figura 2 se muestran dos diagramas de bloques generales de la organización del código de autopiloto. Cuando la placa de hardware se enciende, se ejecuta el archivo binario grabado en la memoria del micorcontrolador. El mismo realiza una inicialización de todos los sensores asociados al autopiloto y de los sistemas de control y navegación. Luego de esto, se entra en un bucle infinito que se encuentra dividido en 4 bloques de ejecución temporizados: cada 20 [ms] (50 Hz), 100 [ms] (10 Hz), 300 [ms] (3,33 Hz) y 2000 [ms] (0.5 Hz). Dentro de estos bloques se organiza la ejecución de funciones para controlar el comportamiento del vehículo de acuerdo a la prioridad que tenga cada una de ellas. Por ejemplo el control de los actuadores se ejecuta en el bucle de 50 Hz, mientras que la actualización de algoritmo de navegación se puede realizar en el bloque de 3.33 [Hz].

Las tareas mínimas que se requieren para que el autopiloto funcione y que no dependen de la aplicación específica que deba cumplir, están organizadas de la siguiente manera en los bloques temporizados:

- Tareas que se ejecutan a 50 Hz
  - Leer las señales de entrada PWM (comandos por hardware)
  - Actualizar el sistema de control de actitud del autopiloto
  - Calcular las salidas de control necesarias para controlar la actitud
  - Escribir las salidas de control en el hardware mediante señales PWM
- Tareas que se ejecutan a 10 Hz
  - Realizar una lectura de datos del sistema GPS
  - Actualizar el sistema de control de navegación
  - Realizar una lectura de la altura usando el barómetro
  - Loguear datos a la memoria Flash del autopiloto
  - Realizar una lectura del estado del sistema eléctrico
- Tareas que se ejecutan a 3,34 [Hz]
  - Actualizar el estado del sistema de modos de control

Esta organización de tareas permite que se encuentren desacopladas las operaciones básicas de funcionamiento del hardware de las específicas de control del sistema. Así se pueden generar diferentes aplicaciones para el hardware de autopiloto haciendo una implementación específica de las tareas de actualización del control de navegación y actualización del sistema de control de actitud, ej. control de una aeronave de ala fija, control de un paracaídas, etc.

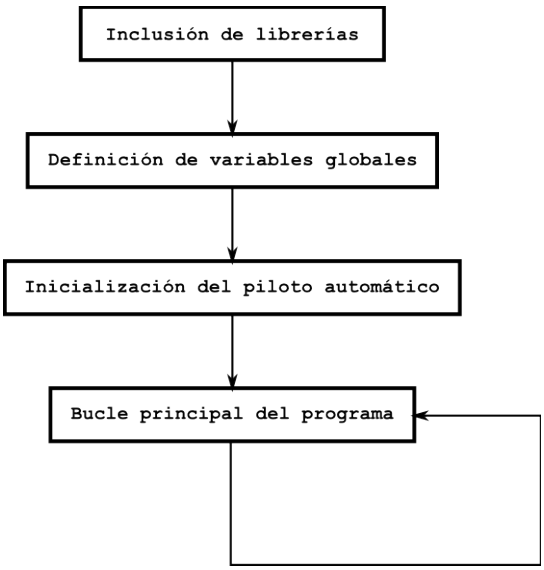

Figura 1: Diagrama de bloques general del programa de autopiloto [1]

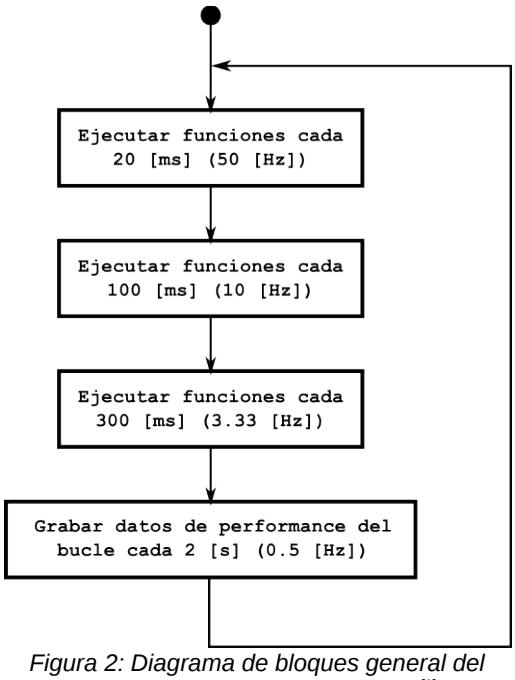

Figura 2: Diagrama de bloques general del lazo de ejecución del autopiloto [1]

A continuación se describen, primero el funcionamiento de los módulos básicos para operar el autopiloto y luego se analizarán los módulos específicos para el sistema de paracaídas comandado autónomo.

2.2 Módulos del sistema básico del autopiloto

2.2.1 Memoria global

El código del autopiloto dispone de una memoria global en donde se asigna espacio de memoria para aquellas variables que son usadas por varios módulos. Esta metodología, si bien no es la recomendada desde un punto de vista del ocultamiento de la información [5], elimina la sobrecarga que generaría disponer de funciones de lectura y escritura para cada una de las variables.

El compilador reserva espacio en la sección .bss de la memoria [6] de manera que estas variables consumen parte del espacio de la memoria RAM del microcontrolador. Hay una salvedad con respecto a las variables que almacenan cadenas literales ya que las mismas consumen mayor cantidad de bytes. Para ello se utiliza un mecanismo particular de las librerías de AVR Atmel en el cuál estas variables se asignan a la memoria flash del microcontrolador sin que ocupen espacio en la memoria RAM [7].

La memoria global se crea mediante dos archivos de código fuente:

- globalVar.h: archivo de encabezado para hacer la declaración de las variables
- globalVar.cpp: archivo de código fuente donde se definen las variables declaradas en globalVar.h

Para agregar una variable nueva a la memoria se declara la variable con la palabra clave del lenguaje C “extern” para indicarle al compilador que la variable se va a definir en otro archivo de código fuente. Esto evita la definición múltiple de variables con el mismo nombre. Luego las variables se definen en el archivo de código fuente globalVar.cpp.

Código en el archivo de encabezado.

```
extern tipo nombreVariable;
```

Código en el archivo de código fuente.

```
tipo nombreVariable = valor;
```

2.2.2 Sistema de entradas y salidas PWM

El hardware ArduPilotMega dispone de un multiplexor y un microcontrolador ATmega 328P [8] dedicado a la administración de 8 canales para enviar y recibir señales de PWM (Figura 3). La computadora principal del autopiloto está conectada al microcontrolador ATmega 328P de acuerdo al esquema de la Figura 4 [3] a través de un pin digital en ambos microcontroladores.

Todas las entradas de las señales de PWM las recibe el microcontrolador ATmega 328P y luego se retransmiten a la computadora principal ATmega 2560. En esta última, la lectura de las señales de

entrada se realiza mediante una interrupción en donde se calcula el ancho de pulso de cada canal de acuerdo al pulso de sincronización que se envía desde el pin 14 (PB2 @ ATmega 328P) hacia el pin 49 (PL0 @ ATmega 2560). Estas operaciones se encuentran implementadas en el módulo APM_RC en la carpeta de librerías del código fuente del autopiloto. Los valores de ancho de pulso se almacenan en un vector local de enteros sin signos de 16bit y luego se pueden obtener mediante un método de acceso de la librería.

Las señales de salida las administra el microcontrolador ATmega 2560 y se encuentran divididas en dos conjuntos: cuatro canales que se encuentran multiplexados con los señales que provienen directamente de las entradas de PWM (sin pasar por el microcontrolador ATmega 328P) y cuatro canales que se conectan directos a los pines de salida del microcontrolador. La computadora principal puede elegir mediante el pin 4 (PG5 @ ATmega 2560) conectado al multiplexor si se utilizan las señales provenientes del autopiloto ó de las entradas de PWM [3].

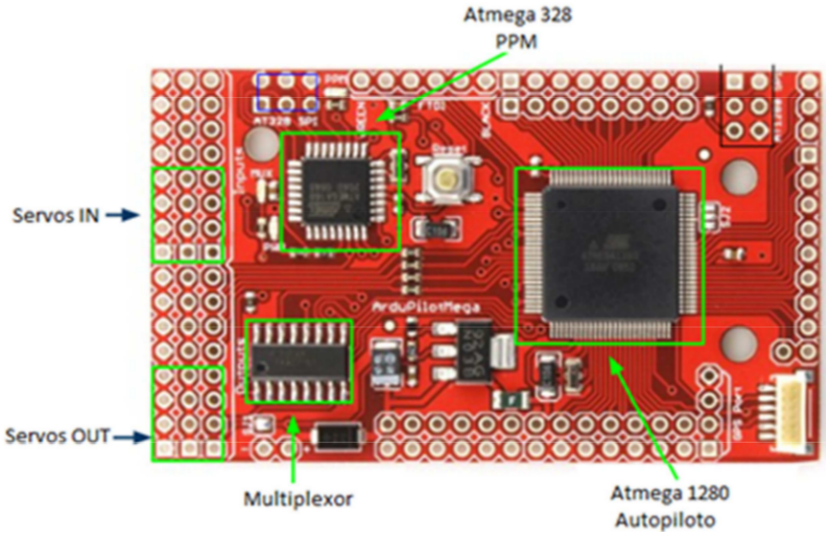

Figura 3: Placa de autopiloto ArduPilotMega

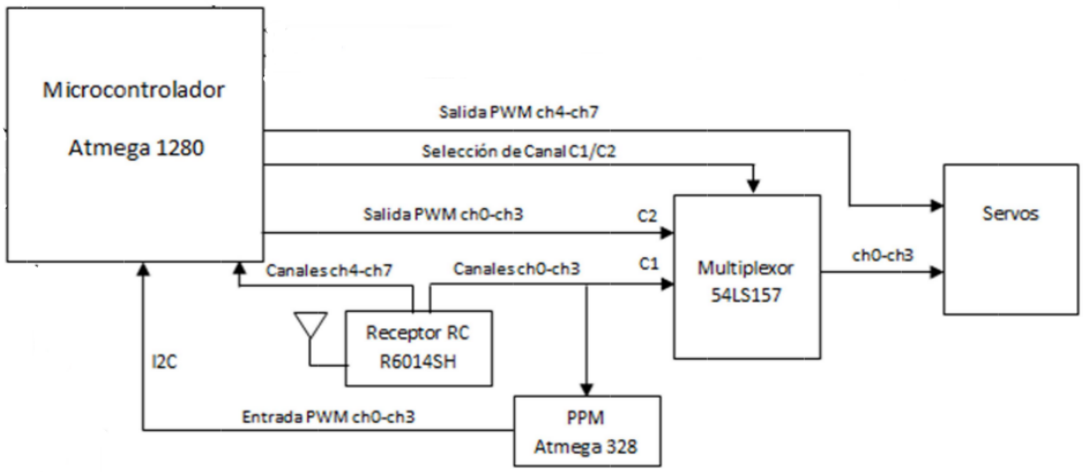

Figura 4: Conexión de las señales PWM - Hardware ArduPilotMega - Adaptado de referencia [3]

A partir de la librería básica de PWM se implementó un módulo para realizar lectura y escritura de las señales PWM, que además permitiese acumular valores de referencia y de trim para cada uno de los canales y funciones de transferencia para pasar de unidades físicas de desplazamiento, posición, etc. a valores de PWM. Al mismo se le denominó “ioControlPWM” que es un acrónimo de “Input Output Control PWM”.

El módulo está construido en torno a una serie de vectores de 8 componentes cada uno que almacenan diferentes valores relacionados con las entradas y salidas de cada canal. Algunos de estos vectores son públicos al módulo para permitir un acceso rápido para operaciones de lectura y escritura de valores. Si bien esto no es óptimo desde el punto de vista del encapsulamiento de información [5], responde a los recursos limitados del controlador y evita la sobrecarga por medio de la utilización de funciones de acceso. Como trabajo a futuro se podría evaluar utilizar funciones de acceso para los vectores y sugerir al compilador, mediante la palabra clave “inline” [9], introducir el código de la función en la sección de código en la que aparece la llamada a la misma.

Vectores para los canales de entrada de PWM.

```
extern uint16_t inputPWM[];    // Valores del bus de entrada de PWM
extern uint16_t  inRefPWM[];   // Valores de offset para las señales del bus de entrada de PWM
extern int16_t   inChValue[];  // Valores en unidades físicas del bus de entrada de PWM

static uint16_t  inputMinPWM[]; // Valores mínimos de los canales del bus de entrada PWM
static uint16_t  inputMaxPWM[]; // Valores máximos de los canales del bus de entrada PWM

static int8_t    inChReverse[];  // Valores para invertir el sentido del bus de entrada de PWM
static float     inChgPos2PWM[]; // Valores para convertir a valores físicos las entradas PWM
static float     inChgPWM2Pos[]; // Valores para convertir a valores PWM las entradas físicas
```

Vectores para los canales de salida de PWM.

```
static uint16_t  outputPWM[];   // Valores del bus de salida de PWM
static uint16_t  outRefPWM[];   // Valores de offset para las señales del bus de salida de PWM
extern int16_t   outChValue[];  // Valores en unidades físicas del bus de salida de PWM

static int8_t    outChReverse[]; // Valores para invertir el sentido del bus de entrada de PWM
static float     outChgPos2PWM[]; // Valores para convertir a valores físicos las entradas PWM
static float     outChgPWM2Pos[]; // Valores para convertir a valores PWM las entradas físicas
```

La memoria ocupada por estos vectores es la siguiente:

- 8 vectores de 8 componentes de elementos de 16bit (2 bytes): 128 bytes
- 2 vectores de 8 componentes de elementos de 8bit (1 byte): 16 bytes
- 4 vectores de 8 componentes de elementos de 32bit (4 bytes): 128 bytes

En total, la memoria global requerida es de 272 bytes.

Las funcionalidades del módulo se encuentran implementadas mediante tres funciones principales que se encargan de leer y escribir las señales PWM, un conjunto de funciones auxiliares que permiten configurar los valores de los vectores de transformación: offsets de las señales, relaciones de transformación de los valores de PWM, inversión del sentido de las señales, valores mínimos y máximos admisibles para la señales, etc. y algunas funcionalidades menores como ser almacenamiento en memoria EEPROM de los offsets, salida a consola de los valores mínimos, máximos, relaciones de transformación entre otros.

La función de inicialización del módulo debe llamarse antes de intentar usar los vectores y el resto de funcionalidades del módulo. La msima se encarga de inicializar la librería APM_RC de entrada y salida de PWM y luego asigna valores definidos a todas las componentes de todos los vectores descriptos anteriormente. Algunas de los valores que aparecen en la función a continuación se encuentran definidas mediante macros al comienzo del archivo de código fuente.

```
void IOPWMInit(void){
  APM_RC.Init();                // Inicializar la librería de comunicación PWM

  chCounter = 0;

  int8_t  chDirection = (int8_t)NORMAL_DIRECTION;      // Sentido normal para el canal.

  // EL factor 0.11111 que aparece en las ecuaciones abajo es para escalear la salida del
  // servo a valores PWM servo_out[] contiene los valores de salida para los servos en 100·DEG
  // y se estima que el máximo es 45º.
  //Entonces cuando el servo está en esta posición máxima (45º) debería dar la salida PWM
  // máxima (2000) radio_out[] = 1500 + 4500 * (500 / 4500)
  //  en donde 500/4500 = 0.111111

  float   chgPos2PWM = 1.0F / (float)DEFAULT_GPWM2POS;        // chgPos2PWM = 0.1111111111

  // Inicializar valores en los vectores de la librería
  for( chCounter = 0 ; chCounter < (uint8_t)NUM_CHANNELS ; chCounter++){

          // Inicializar los vectores de configuración de los canales PWM de entrada
          inputPWM[chCounter]    = (uint16_t)DEFAULT_PWMVAL;
          IPWM_SetChannelDirection(chCounter,chDirection);
          IPWM_SetPWMTransformation(chCounter,chgPos2PWM);
          IPWM_SetPWMRef(chCounter,(uint16_t)DEFAULT_PWMVAL);
          inputMinPWM[chCounter] = (uint16_t)MIN_PULSEWIDTH;
          inputMaxPWM[chCounter] = (uint16_t)MAX_PULSEWIDTH;
          inChValue[chCounter] = (int16_t)0;

          // Inicializar los vectores de configuración de los canales PWM de salida
          outputPWM[chCounter]   = (uint16_t)DEFAULT_PWMVAL;
          OPWM_SetChannelDirection(chCounter,chDirection);     // Sentido del canal
          OPWM_SetPWMTransformation(chCounter,chgPos2PWM);     // Relación de transformación
          OPWM_SetPWMRef(chCounter,(uint16_t)DEFAULT_PWMVAL); // Valor de referencia a cero
          outChValue[chCounter] = (int16_t)0;                  // Valor físico actual del canal
  }

  // Usar por default los valores de referencia de los canales de entrada
  IOPWMUseInputRefs();
}
```

La función de lectura de las entradas PWM se encarga de copiar dichos valores de la librería APM_RC al vector local y convertir a valores de unidades físicas del canal. El valor físico se convierte de acuerdo a la ecuación (1). Luego de hacer una llamada a esta función están disponibles los nuevos valores tanto de PWM como en unidades físicas para los 8 canales del autopiloto.

$$(chIn)_i = (inChInv \cdot (inPWM_{raw} - PWM_{ref}) \cdot G_1)_i \quad (1)$$

```
/*
 * Lee los valores de PWM de todos los canales y almacena los valores en
 * el vector global de PWM
 */
void IOPWMReadAllInputs(void){
  /*
   * Leer los valores de PWM desde las entradas y transformar los valores de PWM
   * a unidades físicas usando la relación de transformación de cada canal (gPWM2Pos).
   */
  for( chCounter = 0 ; chCounter < (uint8_t)NUM_CHANNELS ; chCounter++){
         inputPWM[chCounter] = APM_RC.InputCh(chCounter);

         inChValue[chCounter] = (int16_t)(inChReverse[chCounter] *
                        ((float)((int16_t)inputPWM[chCounter] - (int16_t)refPWM[chCounter]) *
                        inChgPWM2Pos[chCounter]));
  }
}
```

La función de escritura de señales PWM tiene un comportamiento similar a la función de lectura: convierte los valores disponibles en el vector de salida de unidades físicas a valores PWM y los envía al hardware usando la librería APM_RC de acuerdo a la ecuación (2). En este caso se utiliza como offset el valor de referencia del vector outRefPWM.

$$(outPWM_{raw})_i = (outPWM_{ref} + outChInv \cdot chOut \cdot G_2)_i \quad (2)$$

```
/*
 * Escribe los valores de PWM de todos los canales
 */
void IOPWMWriteAllOutputs(void){

  /*
   * Calcular los valores de PWM a partir de los valores físicos en el vector outChValue.
   * La salida a PWM de un canal se construye como:
   * outPWM = refPWM + direction * (value * gpos2PWM)
   */
  for( chCounter = 0 ; chCounter < (uint8_t)NUM_CHANNELS ; chCounter++){
         outputPWM[chCounter] = (uint16_t)(outRefPWM[chCounter] +
                              (int16_t)(outChReverse[chCounter]) *
                              (int16_t)(outChValue[chCounter] * outChgPos2PWM[chCounter]));
```

```
            APM_RC.OutputCh(chCounter,outputPWM[chCounter]);
    }
}
```

Estas dos funciones deben ser llamadas a intervalos regulares en el loop principal del programa del autopiloto.

2.2.3 Sensores

El módulo de sensores agrupa las funciones asociadas a los mismos. Para cada sensor se deberían implementar al menos dos funciones básicas: inicialización y lectura. Se pueden complementar las funcionalidades con otras funciones como por ejemplo salida por consola, funciones de filtrado, etc.

El desarrollo actual tiene implementadas funciones de lectura para los siguientes sensores (y en algunos casos funciones de inicialización):

- Horizonte artificial
- Barómetro
- Velocidad del aire
- Magnetómetro
- GPS
- Sensor de tensión y corriente

Como ejemplo, se muestran a continuación, las funciones asociadas al barómetro, el gps y el sensor de tensión y corriente que son los sensores con mayor relevancia para el diseño del paracaídas comandado autónomo.

El barómetro tiene dos funciones asociadas como se muestra a continuación. La función de inicialización del barómetro realiza dos ciclos de lectura de temperatura y presión. Durante el primero se controla el estado del barómetro realizando una espera mediante un ciclo de lectura hasta que el sensor devuelva un valor de temperatura y presión y luego una estabilización de las lecturas tomando 50 ciclos de lectura de temperatura y presión. El segundo ciclo se utiliza para determinar la temperatura y presión a nivel del suelo en donde se encuentra el autopiloto. Se toman 20 lecturas y se filtran mediante un promedio que tiene en cuenta el 50% del valor anterior y el 50% del valor actual.

```
void InitBarometer(void);
bool ReadPressureAltitude(void);
```

La función de lectura de presión opera de manera no bloqueante, es decir en cada ciclo de ejecución controla el estado de la conversión de temperatura y presión del barómetro y si hay valores

nuevos actualiza las variables de la memoria global del autopilot, sino sale inmediatamente liberando ciclos de procesador para otras tareas. El baromómetro Bosch BMP085 tiene una demora de 4,5 ms para realizar una conversión de temperatura y de 25,5 ms para realizar una conversión de presión en el modo de ultra alta resolución [10]. Por lo tanto estos intervalos de tiempo se pueden utilizar para otras tareas del autopiloto.

Con el valor de presión obtenido se calcula un valor de altitud actual (respecto a nivel del mar) usando el valor de altitud obtenido durante la calibración. Luego se aplica un filtrado para disponer de un segundo valor de altura más suavizado que el valor sin procesar.

```
bool ReadPressureAltitude(void){

  uint8_t bReadresult = barometer.Read();      // Leer el barometro.
  bool result = false;

  switch(bReadresult){
        case 1:        // Leida la temperatura
               baroTemperature = barometer.GetTemperature();
               break;
        case 2:        // Leida la presion
               abs_press = barometer.GetPressure();
               abs_press_filt = abs_press;          // Sin filtrado

               double x;
               double p;
               double temp;
               p = (double)abs_press_gnd / (double)abs_press_filt;
               temp = (float)ground_temperature / 10.f + 273.15f;
               x = log(p) * temp * 29271.267f;
               press_alt = (long)(x / 10) + ground_alt;     // Pressure altitude in cm.

               /*
                * Aplicar un filtro a la altura para disponer de un valor de altura
                * suavizado para realizar los cálculos de control / navegación
                */
               FiltAltitude();       // Aplicar filtro a la altura

               result = true;        // hay un valor actualizado de altura barométrica
               break;
        default:
               break;
  }

  return result;
}
```

Respecto a la determinación de la altura, el autopiloto tiene implementada una funcionalidad que permite utilizar valores de altitud para control y navegación tomando sólo la altitud de GPS ó sólo la altitud obtenida a partir del barómetro ó una mezcla de ambas. A continuación se muestra el código de esta función que se encuentra declarada e implementada en el módulo de sensores. La altitud se almacena en la estructura de datos current_loc en la memoria global del autopiloto.

```
void ReadAltitude(void){

  if( altitude_mix < 1.0e-6 ){ // Solo altura de GPS
        if( GPS.fix ){
              current_loc.alt = (int32_t)(GPS.altitude);
        }
  }
  else{
        ReadPressureAltitude();

        if( GPS.fix ){
              current_loc.alt =  (int32_t)((1.0f - altitude_mix) * (float)GPS.altitude);
              current_loc.alt += (int32_t)(altitude_mix * (float)press_alt);
        }
  }
}
```

La función de lectura de GPS es bastante simple, ya que realiza una actualización del estado del GPS para que el resto de los módulos tengan datos actualizados del mismo. Si hay datos nuevos se almacena la latitud y longitud actual del vehículo en la memoria global. Como tarea complementaria se actualiza el estado de los leds de la placa de sensores del autopiloto para tener una indicación visual del estado del mismo.

```
void ReadGPS(void)
{
  GPS.update();
  update_GPS_light();

  if (GPS.new_data && GPS.fix) {
        GPS_timer = millis();

        gps_fix_count++;                              // for performance

        current_loc.lng = GPS.longitude;              // Lon * 10**7
        current_loc.lat = GPS.latitude;               // Lat * 10**7
  }
}
```

El sensor de tensión y corriente se encuentra conectado al conversor ADC del autopiloto, por lo que la lectura del sensor se realiza consultando el valor de los canales a los cuales se encuentran conectadas las dos salidas del sensor. Estos valores se transforman de cuentas ADC a volts y amperes mediante dos macros que se definen de acuerdo a la selección del sensor que se hace. Se calcula un valor de corriente consumida acumulada.

```
void ReadBattery(void)
{
  switch (battery_monitoring) {
        case 0:               // No monitoring
               battery_voltage1 = 0.0f;
               current_amps1  = 0.0f;
               current_total1 = 0.0f;
               break;
        case 3:               // Voltage monitoring only
               battery_voltage1 = BATTERY_VOLTAGE(analogRead(BATTERY_PIN_1)) * .1 +
                                  battery_voltage1 * .9;
               current_amps1 = 0.0f;
               current_total1 = 0.0f;
               break;
        case 4:               // Voltage and current monitoring only
               battery_voltage1 = BATTERY_VOLTAGE(analogRead(BATTERY_PIN_1)) * .1 +
                                  battery_voltage1 * .9;
               current_amps1 = CURRENT_AMPS(analogRead(CURRENT_PIN_1)) * .1 + current_amps1
                                  * .9;          // reads power sensor current pin
               current_total1  += current_amps1 * (float)deltaMsMediumLoop * 0.0002778;
                      // .0002778 is 1/3600 (conversion to hours)
               break;
        default:
               battery_voltage1 = 0.0f;
               current_amps1 = 0.0f;
               current_total1 = 0.0f;
               break;
  }
}
```

2.2.4 Sistema de logueo

El autopiloto dispone de una memoria flash de 2 MB (16 Mbit) que se utiliza para guardar información durante el vuelo. La comunicación con esta memoria se realiza a través de una interfaz SPI y se encuentra implementada en la librería DataFlash disponible en la carpeta “libraries” del código fuente del autopiloto. La memoria se encuentra organizada en “páginas” de 512 bytes cada una que se administran desde esta librería.

Por otra parte la administración del formato que tienen los registros de datos (logs) que se leen y escriben a esta memoria está implementada mediante un librería que se denomina “logSystem” cuyos archivos de encabezado y código fuente son logSystem.h y logSystem.cpp. El registro de datos al que se denomina “log” se encuentra codificado de la siguiente manera (ver Figura 5):

- En la primera página de la memoria se escribe un paquete de datos que mantiene un seguimiento de la cantidad de registros escritos (logs) y las páginas de inicio y finalización de cada uno de ellos.
- A continuación a partir de la página 2 se escriben los paquetes de datos individuales codificados con un encabezado y un fin de paquete específico.

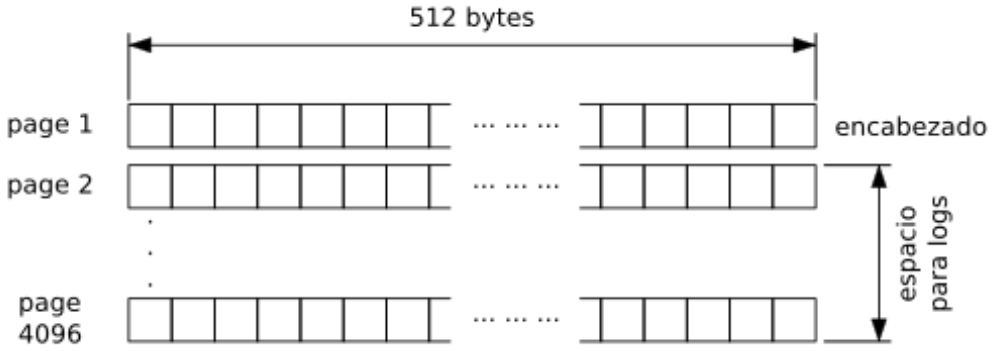

Figura 5: Estructura de la memoria flash para registro de datos

La librería provee un conjunto básico de funciones para inicializar el administrador de logs, iniciar un nuevo log y obtener la cantidad de logs ya guardados en memoria. Con estas tres funciones se puede iniciar la escritura de un nuevo log.

```
void     InitLogSystem(void);
void     start_new_log(uint8_t& num_existing_logs);
uint8_t  get_num_logs(void);
```

Para poder leer y borrar los logs de la memoria flash, la librería tiene implementado un menú de consola para poder interactuar a través del puerto de un serial. Ejecutando la función process_logs, la librería toma control del bucle de ejecución del programa y muestra un menú de consola en donde se encuentran habilitados los siguientes comandos:

- dump <n>: Lee el log “n” de la memoria y lo escribe a la salida de la consola
- erase: borra todos los logs disponibles en la memoria
- enable <name>|all: configura el sistema de logueo para que un paquete particular se grabe en la memoria durante la ejecución del programa
- disable <name>|all: configura el sistema de logueo para que un paquete particular se dejé de grabar en la memoria durante la ejecución del programa

```
int8_t   process_logs(uint8_t, const Menu::arg*);
void     Log_ConfigGroundOperation(void);
```

2.2.4.1 Mensajes del sistema de registro

Los datos que se escriben en la memoria están empaquetados en forma de mensajes con un encabezado y una finalización (Figura 6). La longitud de los datos de cada mensaje puede ser variable. En la Tabla 2 se describen el contenido de la estructura del paquete de datos.

Figura 6: Estructura de un paquete de datos del sistema de logueo

Byte #	Ref	Contenido	Valor	Descripción
0	ST1	Byte inicio de paquete	0xA3	Primer byte del encabezado de paquete de datos.
1	ST2	Byte inicio de paquete	0x95	Segundo byte del encabezado del paquete de datos.
2	ID	Packet ID	0 - 255	Identificador del paquete de datos.
3 to (n+3)	PAYLOAD	Datos del paquete	n/a	Datos del paquete de datos. Depende del mensaje.
(n+4)	END	Byte fin de paquete	0xBA	Byte para identificar el fin del paquete de datos.

Tabla 2: Identificación de los bytes del paquete de datos del sistema de logueo

La lectura y escritura de los paquetes en la memoria se debe implementar mediante dos funciones: una que realice la codificación y otra que haga la operación inversa. Como ejemplo de estas funciones se muestra, a continuación, el código fuente para leer y escribir el modo de vuelo en que se encuentra el autopiloto. La función para lectura de un paquete de datos sólo tiene que implementar la lectura del payload del paquete, ya que la identificación del paquete de datos (encabezado y ID de paquete) se realiza durante la lectura del log; al encontrar el identificador de paquete se llama a la función correspondiente para leer los datos del mismo.

```
void Log_Read_Mode(void){
  uint8_t mode;

  mode = DataFlash.ReadByte();
  Serial.print("MOD:");
  Serial.println(GetCurrentFlightModeName(mode));
}
```

La función para escribir un paquete debe, en cambio, realizar la codificación completa del mismo (escritura de encabezado y finalización) ya que estas funciones se llaman directamente en el bucle de ejecución principal (este modo de operación permite disminuir la sobrecarga, haciendo que la escritura de la memoria del log se realiza directamente sin tener que pasar a través del administrador de logs).

```
void Log_Write_Mode(uint8_t mode){
  DataFlash.WriteByte(HEAD_BYTE1);
  DataFlash.WriteByte(HEAD_BYTE2);
  DataFlash.WriteByte(LOG_MODE_MSG);
  DataFlash.WriteByte(mode);
  DataFlash.WriteByte(END_BYTE);
}
```

2.2.4.2 Configuración del sistema de logueo

La implementación del sistema de logueo tiene capacidad de administrar hasta 16 paquetes diferentes de datos. El mecanismo que se utiliza puede parecer un poco complejo pero está orientado a la performance del sistema de logueo en un microcontrolador de 8bit, tratando de disminuir a un mínimo la sobrecarga por llamadas a funciones.

La librería de logueo define 16 valores mediante macros públicas para identificar los 16 paquetes de datos y provee además otros 16 valores (también mediante macros públicas) para poder aplicar una operación de máscara de bit y determinar si un paquete se encuentra habilitado o no para logeuo (explicación a continuación).

```
// Se pueden definir hasta 16 mensajes de logueo
#define LOG_MSG_00      0x00          // Identificador de mensaje de logueo 0
#define LOG_MSG_01      0x01          // Identificador de mensaje de logueo 1
#define LOG_MSG_02      0x02          // Identificador de mensaje de logueo 2
#define LOG_MSG_03      0x03          // Identificador de mensaje de logueo 3
#define LOG_MSG_04      0x04          // Identificador de mensaje de logueo 4
#define LOG_MSG_05      0x05          // Identificador de mensaje de logueo 5
#define LOG_MSG_06      0x06          // Identificador de mensaje de logueo 6
#define LOG_MSG_07      0x07          // Identificador de mensaje de logueo 7
#define LOG_MSG_08      0x08          // Identificador de mensaje de logueo 8
#define LOG_MSG_09      0x09          // Identificador de mensaje de logueo 9
#define LOG_MSG_10      0x0A          // Identificador de mensaje de logueo 10
#define LOG_MSG_11      0x0B          // Identificador de mensaje de logueo 11
#define LOG_MSG_12      0x0C          // Identificador de mensaje de logueo 12
#define LOG_MSG_13      0x0D          // Identificador de mensaje de logueo 13
#define LOG_MSG_14      0x0E          // Identificador de mensaje de logueo 14
#define LOG_MSG_15      0x0F          // Identificador de mensaje de logueo 15
#define MASK_LOG_MSG_00         0     // Máscara para el mensaje de logueo 0
#define MASK_LOG_MSG_01         2     // Máscara para el mensaje de logueo 1
#define MASK_LOG_MSG_02         4     // Máscara para el mensaje de logueo 2
#define MASK_LOG_MSG_03         8     // Máscara para el mensaje de logueo 3
#define MASK_LOG_MSG_04         16    // Máscara para el mensaje de logueo 4
#define MASK_LOG_MSG_05         32    // Máscara para el mensaje de logueo 5
#define MASK_LOG_MSG_06         64    // Máscara para el mensaje de logueo 6
#define MASK_LOG_MSG_07         128   // Máscara para el mensaje de logueo 7
#define MASK_LOG_MSG_08         256   // Máscara para el mensaje de logueo 8
#define MASK_LOG_MSG_09         512   // Máscara para el mensaje de logueo 9
#define MASK_LOG_MSG_10         1024  // Máscara para el mensaje de logueo 10
#define MASK_LOG_MSG_11         2048  // Máscara para el mensaje de logueo 11
```

```
#define MASK_LOG_MSG_12         4096    // Máscara para el mensaje de logueo 12
#define MASK_LOG_MSG_13         8192    // Máscara para el mensaje de logueo 13
#define MASK_LOG_MSG_14         16384   // Máscara para el mensaje de logueo 14
#define MASK_LOG_MSG_15         32768   // Máscara para el mensaje de logueo 15
```

El sistema de logueo administra una variable de 16bit representada mediante un entero sin signo. En esta variable cada uno de los 16 bit que la componen corresponden a un paquete en particular por lo que si un bit tiene un valor de 1 se interpreta que el paquete asociado a ese bit está habilitado para logueo. Esta variable se almacena en la memoria EEPROM durante la configuración a través de la consola y está disponible durante la ejecución del lazo principal del programa. De esta manera, realizando una operación de máscara de bit sobre la variable con el valor correspondiente al paquete que se quiere escribir se puede determinar si el mismo está habilitado o no para logueo. Este mecanismo provee de una gran flexibilidad para habilitar y deshabilitar paquetes desde el modo de configuración de consola sin tener que recompilar y cargar nuevamente el código al hardware del auotpiloto. Como ejemplo, se muestra a continuación el código para loguear el modo de vuelo en que se encuentra el autopiloto (en la variable log_bitmask se encuentran los bits asociados a los paquetes de datos habilitados para logueo).

```
#define  MASK_LOG_MODE   MASK_LOG_MSG_06

…

if (log_bitmask & MASK_LOG_MODE){
  Log_Write_Mode(control_mode);
}
```

Para la lectura de los paquetes de datos, se provee una función para asociar un identificador de paquete con el puntero a una función para realizar la lectura de datos del mismo. Durante la lectura de un log, cuando se encuentra el identificador de paquete a continuación se ejecuta el puntero a la función asociado a dicho identificador. De esta manera, es posible desacoplar totalmente las funciones de lectura y escritura de paquetes en el log del sistema de logueo; esta mejora respecto a la versión original del sistema de logueo elimina la necesidad de compartir una variable en la memoria global para poder loguearla y permite que los paquetes de datos sean administrados por diferentes librerías. Existe una función auxiliar para acumular una cierta cantidad de funciones que se ejecutan durante la lectura de un log y antes de comenzar a leer los paquetes de datos del mismo. De esta forma es posible escribir en la salida del log otras configuraciones del autopiloto y que las mismas queden registradas en la salida a consola (ej. valores de ganancias, plan de vuelo, punto objetivo, etc.). A continuación se muestran los prototipos de ambas funciones.

```
typedef void(*ReadLogFunction)(void);                 // Puntero a una función void(void)
typedef void(*HeaderLogFunction)(uint8_t);            // Puntero a una función void(uint8_t)

…

void Log_SetReadLogFunction(uint8_t msgId, ReadLogFunction function);
void Log_SetHeaderLogFunction(HeaderLogFunction function);
```

Por ejemplo, a continuación se muestra el código fuente para asociar la función de lectura del modo de vuelo con un identificador de paquete y que la misma se ejecute al leer los datos de un log. De esta forma, queda completamente definida la lectura y escritura del modo de vuelo mediante la asociación de función de lectura y la operación de máscara de bit sobre la variable de log habilitados.

```
#define LOG_MODE_MSG    LOG_MSG_06

…

Log_SetReadLogFunction(LOG_MODE_MSG,&Log_Read_Mode);
```

Finalmente, si se quisiera mostrar los datos de valores mínimos y máximos de los canales de PWM en cada descarga de log que se realice, hay que asociar a la lista de funciones a ejecutarse antes de leer el log, la función de salida a consola de mínimos y máximos de PWM como se muestra a continuación.

```
Log_SetHeaderLogFunction(&IPWM_SendMinMaxValsToConsole);      // Valores mínimos y máximos de
                                                              // las entradas PWM.
```

2.2.5 Comunicación inalambrica

El módulo de comunicación inalambrica permite enviar y recibir mensajes usando el protocolo de comunicación ADSLink [11]. La comunicación se realiza usando una conexión por puerto serie con una antena de radio frecuencia XBee. A la antena se lo opera en el modo “transparente” [12] que significa que los datos que recibe el módulo antena a través del puerto serie, se retransmiten directamente hacia otros módulos sin realizar ninguna operación sobre los mismos (hay disponibles otros modos de transmisión).

El módulo de comunicación del autopiloto que se encuentra implementado está orientado para mantener comunicación con una estación terrena ya que, el prototipo actual del paracaídas, se utiliza para el ensayo de diferentes modelos de guiado y navegación lo que requiere la visualización del estado de los sistemas en tiempo real en una estación de monitoreo. Los mensajes que se quieren enviar se almacenen en un buffer circular de mensajes. La actualización del nodo de comunicación se realiza

mediante una única función que envía un mensaje disponible en el vector de mensajes en cada llamada.

El link de comunicación se encuentra formado por los siguientes elementos:

- Buffers circulares para enviar y recibir mensajes
- Vector para los datos de mensaje (payload)
- Buffer circular para administrar lista de mensajes a enviar
- Nodo de comunicación tADSLink
- Conector con hardware para enviar y recibir los mensajes tSerialConnectionLink

Este módulo se encarga de reservar la memoria para los buffers circulares de envio y recepción para el nodo de comunicación ADSLink. En la implementación actual se utiliza un tamaño de 128 bytes (máximo para la librería CircularBuffer @ 8 bit) para los vectores de ambos buffers.

```
#define GCS_MSG_BUFFER_SIZE     CBSIZE_128

…

static uint8_t gcsLinkSendBuffer[GCS_MSG_BUFFER_SIZE];
static uint8_t gcsLinkRecvBuffer[GCS_MSG_BUFFER_SIZE];

static tCircularBuffer8 gcsLinkSendCB;
static tCircularBuffer8 gcsLinkRecvCB;
```

También se reserva memoria para un vector de enteros sin signo de 8bit para escribir los datos del payload de los mensajes. Se está usando un tamaño máximo de 40 bytes para el mismo, lo que limita los datos a enviar y recibir a un máximo de 10 valores que ocupen 4 bytes cada uno (uint32_t, int32_t, float, etc.). La implementación actual está limitada sólo al envío o la recepción de mensajes, sin que sea posible enviar y recibir mensajes al mismo tiempo. Esto se debe a que se comparte el vector de payload para ambas operaciones e implica que luego de enviar un mensaje hay que recibir y procesar un mensaje que halla llegado por el nodo de comunicación, ya que en la siguiente actualización del nodo se sobre escribirán los datos del payload. Para mejorar esta limitación se propone a futuro separa los vectores de almacenamiento de payload para poder operar de manera asíncrona el envío y recepción de mensajes.

```
#define GCS_PAYLOAD_SIZE        40U

…

static uint8_t gcsPayloadBuffer[GCS_PAYLOAD_SIZE];
```

Finalmente se reserva memoria para un vector que se utiliza como buffer circular para acumular los mensajes que se quieren enviar al nodo de comunicación. Actualmente la lista es de 8 mensajes pendientes. Esta lista se puede ampliar a mayor cantidad de mensajes, pero teniendo en cuenta que la misma puede ocupar bastante espacio en memoria ya que cada elemento de la lista es un identificador de mensaje representado por una estructura de datos de C tMessageID que requiere 5 bytes (total: 8 identificadores x 5 bytes cada uno = 40 bytes). Este vector se administra con un buffer circular especial que se denomina tCircularBufferMsgId.

```
#define GCS_MSG_QUEUE_SIZE      CBSIZE_8

…

static tMessageID gcsLinkMsgQueueBuffer[GCS_MSG_QUEUE_SIZE];

static tCircularBufferMsgId gcsLinkMsgQueue;
```

El link de comunicación se completa con el nodo de comunicación tADSLink, el administrador de entrada / salida de mensajes tMessageIOManager y la conexión de hardware por la que se enviara y recibirá la información, en este caso un puerto de comunicación serial, tSerialConnectionLink.

```
static tAdsLink gcsLinkID;
static tMessageIOManager gcsIOMsgManager;
static tSerialConnectionLink gcsLink;
```

2.2.5.1 Utilización del sistema de comunicación inalambrica

El módulo de comunicación inalambrica tiene implementada sólo tres funciones para operar el mismo, ya que todas las tareas de codificación y decodificación de mensajes las lleva a cabo la librería ADSLink. La única funcionalidad que hay que implementar a este nivel del software es como reaccionar ante los mensajes que se reciben (escritura de valores en memoria, inicio de un proceso de envío de datos particular, etc.).

```
void ADSGCS_Init(FastSerial* port);
void ADSGCS_Update(void);
void ADSGCS_SendMessage(uint8_t componentID, uint8_t messageID);
```

La función ADSGCS_Init se encarga de llamar a las funciones apropiadas del protocolo ADSLink y SerialConnectionLink para conectar los espacios de memoria reservados con las estructuras de datos que se utilizan para administrar el link de comunicación.

La función ADSGCS_Update es el núcleo del sistema de comunicación inalámbrica. Esta función se encarga de administrar la lista de mensajes a enviar, luego realiza una actualización del conector SerialConnectionLink para enviar el mensaje que se leyó de la lista de mensajes pendientes y recibir

todos los datos disponibles de un mensaje que se halla enviado al nodo. Durante este proceso de actualización se realiza tanto la codificación como la decodificación de los mensajes. Por último, si se recibió un mensaje completo se llama a una función para realizar el procesamiento del mismo a nivel de comandos en el autopiloto.

Como esta función se encarga de leer y escribir los datos de los mensajes en el hardware, la misma debe ser llamada a una velocidad de ejecución mayor que la máxima frecuencia de mensajes que se quiera enviar y recibir.

Para enviar un mensaje se debe agregar a la lista de mensajes usando la función ADSGCS_SendMessage. La misma agrega el identificador de mensaje messageID al buffer circular; al momento de enviar el mensaje se procesa el payload asociado al mismo. Esto implica que hay una diferencia en la información que se envía entre el momento en que se agrega el mensaje a la lista y el instante en que se procesa el payload. Esto puede ser un limitante para la utilización del protocolo en la transmisión de información sincronizada a intervalos fijos de tiempo.

Código fuente de la función ADSGCS_SendMessage y la llamada a la misma para agregar un mensaje para enviar.

```
void ADSGCS_SendMessage(uint8_t componentID, uint8_t messageID){
  tMessageID msg;

  msg.componentID = componentID;
  msg.msgID       = messageID;
  msg.systemID    = 0xaf;
  msg.msgError    = 0U;

  SendMessageOnLink(&msg, &gcsLink);
}

…

ADSGCS_SendMessage(GPS_System,ADS_GPS_Status);
```

2.2.6 Controles para modos de vuelo y de propósito general

El autopiloto tiene previsto la interacción con algunas de las funcionalidades del código mediante la utilización de tres canales de PWM. Dos de estas entradas se pueden usar como selectores de 5 posiciones y la restante se puede utilizar como un botón de 2 posiciones. La implementación de estos selectores se hace mediante la librería de entrada / salida de PWM (ioControlPWM).

Uno de los selectores de 5 posiciones y el botón de 2 posiciones se utilizan para controlar el modo de vuelo en que debe ejecutar el autopiloto; de esta forma es posible probar varios modos de vuelo sin tener que recompilar y escribir el código fuente al hardware del autopiloto cada vez. El segundo selector de 5 posiciones no tiene asignada una funcionalidad específica y se puede utilizar para interactuar con el código del autopiloto según sea necesario.

Cada selector tiene implementada dos funciones básicas: una para inicializar el estado interno del switch y otra para leer el estado o posición actual del mismo. También se proporciona una definición pública mediante macros de los estados que devuelve la consulta de la posición del switch. De esta manera leyendo el estado actual del switch se puede realizar una acción determinada en el lazo principal de ejecución del autopiloto.

Declaración de prototipos para selector multipropósito de 5 posiciones.

```
#define PROPSW_POS_0    0U
#define PROPSW_POS_1    1U
#define PROPSW_POS_2    2U
#define PROPSW_POS_3    3U
#define PROPSW_POS_4    4U
#define PROPSW_POS_5    5U

…

void     PropSw_ResetSwitch(void);
uint8_t  PropSw_GetSwitchPosition(void);
```

Declaración de prototipos para selector de modo de vuelo de 5 posiciones.

```
#define SWITCH_POS_0    0U
#define SWITCH_POS_1    1U
#define SWITCH_POS_2    2U
#define SWITCH_POS_3    3U
#define SWITCH_POS_4    4U
#define SWITCH_POS_5    5U

…

void     ResetFlightModeSwitch(void);
uint8_t  GetFlightModeSwitchPos(void);
```

Declaración de prototipos para selector de 2 posiciones de modo de vuelo.

```
#define ATSW_POS_0      0U
#define ATSW_POS_1      1U

…

void     AutoSw_ResetSwitch(void);
uint8_t  AutoSw_GetSwitchPosition(void);
```

El switch y los selectores se pueden configurar para usar diferentes canales PWM mediante el cambio de la macro del canal de entrada de cada una de las implementaciones. Actualmente, se están usando los siguientes canales de PWM para cada uno de los switch:

- Switch de 2 posiciones: canal 5 (INPWM_CH4)
- Selector multipropósito de 5 posiciones: canal 6 (INPWM_CH5)
- Selector de modos de vuelo de 5 posiciones: canal 7 (INPWM_CH6)

2.2.7 Consola interactiva con el usuario

Es posible realizar la configuración y ver el estado de algunos sistemas del autopiloto mediante un modo interactivo de comandos por consola. Este sistema funciona usando uno de los puertos de comunicación serie del microcontrolador, a través del cuál se envían comandos en formato de texto y se recibe una respuesta a los mismos desde el autopiloto.

La ejecución de este modo de funcionamiento del autopiloto se realiza durante la inicialización del mismo mediante la lectura de posición de un microswitch en la placa de hardware de sensores. Una vez que se ingresó al modo no es posible pasar al modo de vuelo normal del autopiloto, teniendo que realizar una nueva inicilización del hardware para cambiar de modo. A continuación se muestra el fragmento de código en donde se lee el estado del microswitch y se ejecuta el modo de consola.

```
…

if (digitalRead(SLIDE_SWITCH_PIN) == 0) {
        digitalWrite(A_LED_PIN,HIGH);        // turn on setup-mode LED
        Serial.printf_P(PSTR("\n"
                        "Entering interactive setup mode...\n"
                        "\n"
                        "If using the Arduino Serial Monitor, ensure Line Ending is set to
Carriage Return.\n"
                        "Type 'help' to list commands, 'exit' to leave a submenu.\n"
                        "Visit the 'setup' menu for first-time configuration.\n"));

        Log_ConfigGroundOperation();

        // Configurar el sistema de log para usarlo en tierra.
        // Valores mínimos y máximos de las entradas PWM.
        Log_SetHeaderLogFunction(&IPWM_SendMinMaxValsToConsole);

        // Valores de referencia para las entradas PWM.
        Log_SetHeaderLogFunction(&IOPWM_SendRefsToConsole);

        // Valores de las ganancias de los filtros utilizandos en los sensores
        Log_SetHeaderLogFunction(&SendSensorsFiltersConfigToConsole);

        // Configuración completa del sistema de control y navegación.
        Log_SetHeaderLogFunction(&ADS_SendCtrlNavSysConfigToConsole);

        for (;;) {
                Serial.printf_P(PSTR("\n"
                                "Move the slide switch and reset to FLY.\n"
                                "\n"));
                main_menu.run();
        }
  }

…
```

La interacción con el usuario en el modo de consola se realiza por medio de tres menús dentro de los cuales se pueden realizar diferentes operaciones de configuración y testeo:

- Menú de logueo: comando “logs”
- Menú de configuración: comando “setup”
- Menú de testeo: comando “test”

- Ayuda de consola: comando “help”

Para ingresar a cualquiera de estos menús hay que enviar por el puerto serie una cadena de caracteres que contenga los caracteres del “comando” del menú seguida por el carácter de retorno de carro (“\r”) en ese orden.

Es posible agregar nuevos menús y funcionalidades para interactuar con el usuario mediante la declaración de una nueva función que ejecute las acciones deseadas y luego se agrega el puntero de esta función y el comando asociada a la misma a una estructura de datos que se almacena en memoria. El sistema de menús se encarga de administrar la llamada cuando se envía el comando por el puerto serie. A continuación se muestra un ejemplo de como se agregaría un nuevo menú a la lista principal del menús.

```
int8_t   newActionMenu(uint8_t argc,  const Menu::arg *argv);

…

const struct Menu::command main_menu_commands[] PROGMEM = {
//   command            function called
//   =======            ===============
  {"logs",              process_logs},
  {"setup",             setup_mode},
  {"test",              test_mode},
  {"help",              main_menu_help},
  {"newMenu",           newActionMenu}
}

…

int8_t   newActionMenu(uint8_t argc,  const Menu::arg *argv){
  // Implementar nuevas funcionalidades acá
}
```

2.2.8 Integración de funcionalidades de control al autopiloto

La estructura de código descripta hasta ahora permite desacoplar el funcionamiento de las capacidades básicas del autopiloto de aquellas que sean específicas de la aplicación en la que se quiera integrar el hardware. Por ejemplo, la estructura de controladores y sistema de navegación para una aeronave de ala fija pueden ser diferentes de los sistemas requeridos para un paracaídas. Una implementación general de sistemas implicaría tener funcionalidades que serían superfluas ó de poca utilidad en aplicaciones particulares. Para evitar esto, es que se estructuró el código fuente de manera de poder tener separada la interacción con el hardware de la arquitectura de control requerida para cada aplicación específica.

La integración de los módulos de control, guiado y navegación se realiza mediante la llamada a cinco funciones (ver Tabla 3) en lazo de ejecución principal del autopiloto. Estas funciones deben ser implementadas en los módulos específicos de control que se estén desarrollando.

Función	Frecuencia de actualización	Descripción – Uso sugerido
UpdateAttitudeControl(...)	50 Hz	Hacer la actualización de los sistema de control de actitud del vehículo.
Attitude2Servos(...)	50 Hz	Transferir las salidas de los sistemas de control al sistema de entrada/salida de PWM.
UpdateNavigationControl(...)	10 Hz	Hacer la actualización de los sistema de navegación y guiado del vehículo.
UpdateNavigationFlightPlan(...)	3,34 Hz	Hacer la actualización de los sistemas de navegación y plan de vuelo del vehículo.
CheckCurrentFlightMode(...)	3,34 Hz	Leer si hay que cambiar a un nuevo modo de funcionamiento automático del vehículo.

Tabla 3: Descripción de las funciones a implementar para integrar nuevos módulos de control y navegación al autopiloto

Una consideración importante a tener en cuenta es el orden en que se ejecutan estas funciones para configurar correctamente los modos de vuelo. Cuando se realiza un cambio de modo de vuelo, el mismo se detecta en la llamada a la función CheckCurrentFlightMode(…); como la misma se ejecuta a 1 Hz, la próxima función que se ejecuta es la de control de actitud del vehículo UpdateAttitudeControl(…) (lazo de 50 Hz) por lo tanto al realizar un cambio de modo de vuelo se deben re-configurar los controladores de actitud y los canales de salida de control para evitar comportamientos extraños del vehículo. De la misma manera, la actualización del control de navegación y de actualización del plan de vuelo se ejecutan luego de de la ejecución del control de actitud, por lo que se deben tener en cuenta las mismas precauciones al implementar el cambio de modo de vuelo.

2.3 Módulos del sistema de paracaídas comandado autónomo

La integración de las funcionalidades de autopiloto para el paracaídas autónomo guiado se encuentra implementada en una serie de librerías que cubren los diferentes sistemas de control, guiado y navegación necesarios. A continuación se muestra, en la Figura 7, un esquema de los módulos que se encuentran implementadas hasta ahora y la dependencia entre ellos y en la Tabla 4 hay una descripción general de cada uno. Como se puede observar de la figura, el módulo ADSControlNavigation se encarga de administrar todos los módulos individuales integrando las funcionalidades de cada uno para lograr que el autopiloto navegue con el paracaídas hasta el punto de destino elegido.

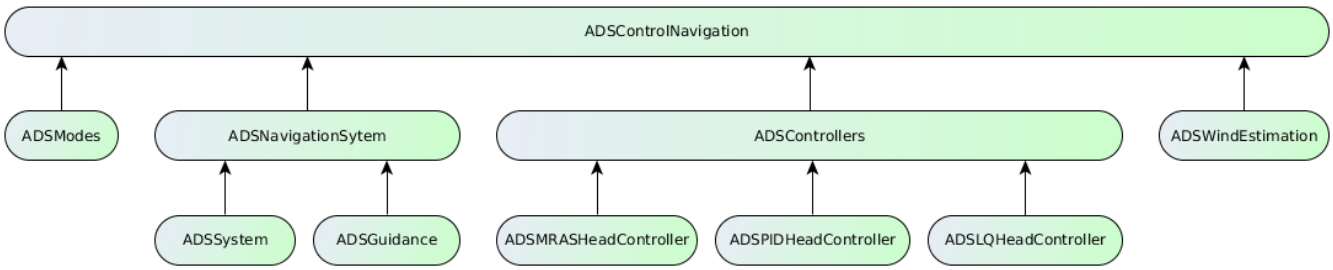

Figura 7: Esquema con la dependencia de módulos del sistema de control, guiado y navegación para paracaídas comandado autónomo

A continuación se realiza una descripción general y funcionalidad de cada uno de los módulos.

Módulo	Archivos del módulo	Descripción
ADSControlNavigation	adsControlNavigation.h adsControlNavigation.cpp	Módulo que administra los sistemas de control y navegación para el paracaídas guiado autónomo.
ADSModes	adsModes.h adsModes.cpp	Módulo en donde se declaran los modos de vuelo para paracaídas guiado autónomo.
ADSSystem	adsSystem.h adsSystem.cpp	Módulo para configurar los datos del paracaídas y la carga del mismo.
ADSGuidance	adsGuindance.h adsGuindance.cpp	Módulo que contiene las estrategias y modelos de guiado para el paracaídas comandado autónomo.
ADSWindEstimation	adsWindEstimation.h adsWindEstimation.cpp	Módulo para realizar estimación de viento durante el descenso del paracaídas.
ADSNavigationSystem	adsNavigationDefines.h adsNavigationSystem.h adsNavigationSystem.cpp	Módulo con las estrategias y modelos para la navegación del paracaídas comandado autónomo.
ADSControllers	adsControllers.h adsControllers.cpp controllersDef.h controllersGlobalData.h controllersGlobalData.cpp adsPIDHeadController.h adsPIDHeadController.cpp adsLQHeadController.h adsLQHeadController.cpp adsMRASHeadController.h adsMRASHeadController.cpp	Módulo que administra la lista de controladores de rumbo disponibles para realizar el guiado del paracaídas comandado autónomo. Se encuentran definidos tres arquitecturas de controladores: PID, LQ y MRAS.

Tabla 4: Listado de módulos del sistema de control, guiado y navegación del sistema paracaídas comandado autónomo

2.3.1 ADSControlNavigation: administrador general

Este módulo se encarga de combinar las funcionalidades de los módulos secundarios del sistema de control y navegación del paracaídas de acuerdo al modo de vuelo que se esté ejecutando. También en este módulo se encuentra la implementación de las funciones CheckCurrentFlightMode(...), UpdateAttitudeControl(…), Attitude2Servos(…) y UpdateNavigationControl(...) que se ejecutan desde el lazo principal del autopiloto y que completan la integración del autopiloto funcionando como sistema de control para un paracaídas.

El módulo provee de una función de inicialización que se ocupa de configurar todos los sub sistemas para el control, guiado y navegación. Esta función debe ser llamada durante la inicialización del autopiloto. La misma configura los canales de PWM que se usaran para controlar el paracaídas: se fijan los valores mínimos, máximos, de referencia y las relaciones de transformación de unidades físicas a valores de PWM (microsegundos) y las relaciones inversas. Luego se llama a las funciones de inicialización de los módulos particulares secundarios.

```
void InitControlAndNavigationSystem(void);
```

Fragmento de código de la función InitControlAndNavigationSystem() en donde se inicializan todos los sistemas que administra el módulo ADSControlNavigation.

```
…

ADS_InitSystemUser();           // Inicializar la configuración de usuario del paracaidas.
ADS_InitControllers();          // Inicializar controladores
ADS_InitGuidanceSys();          // Inicializar sistema de guiado
ADS_InitWindEstimation((uint8_t)UPDATEWNDEST_MS);     // Inicializar el sistema de estimación
                                                      // de viento
ADSNavSys_Init();               // Inicializar el sistema de navegación del paracaídas

…
```

2.3.1.1 Funciones de integración con el autopiloto

A continuación se describen brevemente las cuatro funciones que integran las funcionalidades particulares del paracaídas comandado al código principal del autopiloto:

- UpdateAttitudeControl(…)
- Attitude2Servos(…)
- UpdateNavigationControl(…)
- CheckCurrentFlightMode(…)

En la función UpdateAttitudeControl(…) se actualizan los controladores asociados a la actitud del vehículo (cabeceo, rolido, guiñada) y de algún otro sistema que requiera ser controlado a una frecuencia alta (50 Hz). En el caso del paracaídas comandado, por un lado, el mismo no dispone de controles de cabeceo y rolido como sería el caso de una aeronave de ala fija por lo que no se dispone de controladores en estos ejes, y por otro, la dinámica de vuelo del mismo, hace que no sea necesario realizar una actualización a alta velocidad del sistema de control de guiñada. Por lo tanto, esta función no contiene ninguna llamada para realizar control de la actitud del paracaídas.

La función Attitude2Servos(...) se encarga de componer las salidas para los canales de PWM a partir de las salidas obtenidos del sistema de control de rumbo (u otro sistema de control, por ejemplo, modos de vuelos con entradas de control en escalón para determinar características de respuesta de la planta). Se encuentran implementadas dos salidas PWM para el control independiente de dos actuadores conectadas a las mismas y una tercer salida para el control de proporcional de un variador de frecuencia para el control de un motor trifásico brushless. Las mismas se encuentran definidas en los siguientes canales usando las macros de la librería ioControlPWM:

```
#define CH_OUT_1        OUTPWM_CH1              // Canal de salida para el servo 1
#define CH_OUT_THR      OUTPWM_CH2              // Canal de salida para el mando de potencia
#define CH_OUT_2        OUTPWM_CH3              // Canal de salida para el servo 2
```

Por lo tanto, los actuadores se deben conectar a los canales 1 y 3 del hardware, mientras que el control de frecuencia del motor debe estar conectado al canal 2 del hardware.

El control del paracaídas se realiza mediante dos tipos de deflexiones diferentes del borde de fuga del mismo:

- Una deflexión simétrica de ambos extremos del borde de fuga del paracaídas
- Una deflexión antisimétrica de los extremos del borde de fuga del paracaídas

En el primer caso el efecto neto que tiene la deflexión es una disminución de la velocidad vertical con la que desciende el paracaídas mientras que en el segundo caso se genera una velocidad de guiñada que permite cambiar el rumbo del paracaídas. Estas deflexiones se logran combinando los sentidos de giro de los actuadores a los que se encuentran conectados los comandos.

A nivel de código fuente, una deflexión se encuentran representada como longitud que se recoge o que se suelta un suspente conectado a uno de los extremos del borde de fuga del paracaídas. Y además se modelan las acciones de comando como longitud a recoger/soltar en cada actuador durante una deflexión simétrica (flareControl) y longitud a recoger/soltar de manera antisimétrica en cada actuador en una deflexión antisimétrica (yawControl).

Entonces, en la función Attitude2Servos(...) se combinan las acciones de comando (flareControl y yawControl) resultado de las salidas de los controladores en los valores efectivos que tiene que recoger/soltar los actuadores conectados a los canales PWM 1 y 3 de acuerdo al modo de vuelo automático que se encuentra ejecutando el autopiloto. El control del canal asociado al regulador trifásico, en general, se copia directo desde el canal de entrada de PWM 2. Esto hace que el autopiloto no realice control sobre este canal sino que se controla de manera externa.

Al final de la función se limitan los valores de salida a valores mínimos y máximos de longitud admitidos para la configuración actual del paracaídas. Estos valores se pueden modificar cambiando los valores de las siguientes macros:

```
#define CH_1_MIN        -150                    // [mm] Valor mínimo posible para el canal 1
#define CH_1_MAX         150                    // [mm] Valor máximo posible para el canal 1
#define CH_2_MIN        -150                    // [mm] Valor mínimo posible para el canal 2
#define CH_2_MAX         150                    // [mm] Valor máximo posible para el canal 2
```

A continuación se muestra un fragmento de la función Attitude2Servos(…) en donde se encuentra implementada la transformación para los casos de modo de vuelo Manual y Modo de Navegación automática para el paracaídas comandado autónomo.

```
void Attitude2Servos(void){

  int32_t scaledInFlrValue = 0L;
  int32_t scaledInYawValue = 0L;

  switch(control_mode){

          case MANUAL:
                  scaledInFlrValue = ((int16_t)(FLARE_MIX)*inChValue[CH_IN_FLR])/100;
                  scaledInYawValue = ((int16_t)(YAW_MIX)*inChValue[CH_IN_YAW])/100;

                  outChValue[CH_OUT_1]   = (int16_t)scaledInFlrValue - (int16_t)scaledInYawValue
                                          + chTrims[CH_OUT_1];
                  outChValue[CH_OUT_2]   = (int16_t)scaledInFlrValue + (int16_t)scaledInYawValue
                                          + chTrims[CH_OUT_2];
                  outChValue[CH_OUT_THR]= inChValue[CH_IN_THR];
          break;

          // ADS Modes

          …

          case ADSNAVMODE:
                  outChValue[CH_OUT_1]   =  (int8_t)(-1) * yawControlOut  + chTrims[CH_OUT_1];
                  outChValue[CH_OUT_2]   = yawControlOut  + chTrims[CH_OUT_2];
                  outChValue[CH_OUT_THR]= inChValue[CH_IN_THR];
          break;

          …

  }

  // Limitar los recorridos mínimos y máximos de los servos.
  outChValue[CH_OUT_1] = constrain(outChValue[CH_OUT_1],
                                   (int16_t)(CH_1_MIN),
                                   (int16_t)(CH_1_MAX));
  outChValue[CH_OUT_2] = constrain(outChValue[CH_OUT_2],
                                   (int16_t)(CH_2_MIN),
                                   (int16_t)(CH_2_MAX));

}
```

La actualización del control de navegación se realiza mediante la llamada a la función UpdateNavigationControl(…). La misma actualiza las tareas relacionadas con la navegación del vehículo de acuerdo al modo de vuelo que se esté ejecutando, por ejemplo, la implementación actual para el modo de navegación primero determina el rumbo a seguir de acuerdo a la posición actual del paracaídas y el punto de navegación que tiene que alcanzar; luego este rumbo se fija como rumbo de referencia para el controlador de rumbo y se actualiza el estado del mismo para obtener el valor de la acción de control necesaria en el mando de guiñada. A continuación se muestra un extracto del código fuente de la función UpdateNavigationControl(…).

```
void UpdateNavigationControl(void){

  switch(control_mode){

  case MANUAL:
         // No hacer nada. La navegación del vehículo la controla el piloto desde el radio
         // control.
         break;

  …

  case ADSNAVMODE:

         // Hacer guiado del paracaídas usando los datos que se han actualizado en el
         // controlador de guiado.
         headingRefValue = ADS_UpdateGuidanceHeading();

         // Cambiar la referencia del controlador de rumbo.
         ADS_SetRefHeadingController(headingRefValue);

         // Actualizar el controlador de rumbo y obtener la salida de control necesaria
         yawControlOut = ADS_UpdateHeadingController();

         break;

  …

  }
}
```

La función CheckCurrentFlightMode(…) se encarga de leer el modo de vuelo en que se encuentra el paracaídas y si hay un cambio en el mismo, realizar las acciones necesarias para comenzar a ejecutar el modo de vuelo seleccionado. Todos los modos de vuelo tienen implementada una función para reiniciar los canales de salida de comandos y algunas variables particulares a cada modo de vuelo.

```
void CheckCurrentFlightMode(void){
  ReadCurrentFlightMode();      // Leer el modo de vuelo actual desde el switch

  if( lastFlightMode != control_mode ){
         switch (control_mode) {
                case MANUAL:
                       IOPWMUseInputRefs();   // Cambiar las referencias de los canales PWM
                       break;
                case ADSFLAREINPUTMODE:
                       ResetFlareStepMode(); // Resetear las variables del modo de entrada
                                             // escalón de freno
                       break;
                case ADSYAWINPUTMODE:
                       ResetYawStepMode();    // Resetear las variables del modo de entrada
                                              // escalón de guiñada
                       break;
                case ADSHEADINGMODE:
                       ResetHeadStepMode();
                       break;
                case ADSGUIDANCE2DMODE:
```

```
                        ResetGuidance2DMode();
                        break;
                case ADSWNDESTMODE:
                        ResetWindEstMode();
                        break;
                case ADSNAVMODE:
                        ResetNavigationMode();
                        break;
                default:
                        // Opps! No deberíamos estar acá. Algo salió mal con la lectura del
                        // modo de vuelo.
                        break;
        }

        // Guardar el modo actual de vuelo
        lastFlightMode = control_mode;
  }
  else{
        // No hace falta hacer algo.
        return;
  }

}
```

Por ejemplo, al iniciar el modo de navegación completo (pérdida de altura + aproximación final) se centra el mando de guiñada para que en la siguiente actualización de la función Attitude2Servos(…) el mismo quede fijo y el paracaídas vuele en línea recta hasta que comience a funcionar el control de guiado al siguiente punto de navegación del plan de vuelo. Luego se reinician todos los sistemas que se van a utilizar para realizar la navegación: sistema de guiado y controlador de rumbo. A continuación se fija el rumbo actual que tiene el paracaídas para que no halla cambios violentos en la trayectoria al iniciar el modo y finalmente se realiza un actualización manual del sistema de navegación en la que se calcula el plan de vuelo a seguir.

```
static void ResetNavigationMode(void){

  yawControlOut         = (int16_t)0;          // Poner a cero la salida del control guiñada

  ADSNavSys_Reset();
  ADS_ResetGuidanceMode();

  ADS_ResetHeadingController();
  ADS_SetRefHeadingController(currentHeading);

  ADSNavSys_Update();                          // Calcular el plan de vuelo para tener un
                                               // punto incial hacia donde ir.
}
```

El control del plan de vuelo se realiza en la función UpdateNavigationFlightPlan(…) que se encarga de actualizar y ejecutar el plan de vuelo para alcanzar el punto de destino prefijado cuando el

autopiloto se encuentra en el modo de control de navegación. Para el resto de los modos, no se realiza modificación alguna del plan de vuelo ya que no es necesaria.

```
void UpdateNavigationFlightPlan(void){
  switch(control_mode){
  case MANUAL:
          // No hay que actualizar el plan de vuelo en este modo
          break;
  case ADSFLAREINPUTMODE:
          // No hay que actualizar el plan de vuelo en este modo
          break;
  case ADSYAWINPUTMODE:
          // No hay que actualizar el plan de vuelo en este modo
          break;
  case ADSHEADINGMODE:
          // No hay que actualizar el plan de vuelo en este modo
          break;
  case ADSGUIDANCE2DMODE:
          // No hay que actualizar el plan de vuelo en este modo
          break;
  case ADSWNDESTMODE:
          // No hay que actualizar el plan de vuelo en este modo
          break;
  case ADSNAVMODE:
          ADSNavSys_Update();                  // Actualizar el algoritmo de navegación
          break;
  default:
          // Opps! No deberíamos estar acá. Algo salió mal con la lectura del modo de vuelo.
          break;
  }
}
```

Durante esta actualización, el algoritmo verifica si la altura actual de vuelo se encuentra por encima de cierto umbral, en cuyo caso ejecuta un guiado simple alternando entre dos puntos. Como resultado se obtiene una maniobra similar a un “zig-zag”. Por debajo de la altura umbral el modo entra en modo de aproximación y sigue una serie de puntos de navegación para enfrentar el viento y alcanzar el punto de destino.

2.3.1.2 Funcionalidades particulares del módulo de administración

Como se describió en la sección anterior, a continuación se detallan algunas funcionalidades particulares del módulo de administración de los sistemas del paracaídas.

El módulo tiene implementadas dos funciones para el logueo de datos a la memoria flash del autopiloto pudiéndose extender a un número mayor si fuese necesario. Uno de los paquetes de datos está relacionado con variables de estado del módulo de controladores y el otro está asociado al estado del sistema de navegación. Estas funciones se vinculan al sistema de logueo mediante la declaración de dos macros, la asignación de los punteros de las funciones a la lista de lectura de sistema de logueo y la declaración de las funciones de escritura de datos a la memoria flash (ver mecanismo de logueo en sección §2.2.4).

```
#define MASK_LOG_NTUN   MASK_LOG_MSG_05    // Usando el mensaje 05 para loguear navegación
#define MASK_LOG_CRUMB  MASK_LOG_MSG_11    // Usando el mensaje 11 para loguear controladores

#define LOG_NAV_TUNING_MSG             LOG_MSG_05
#define LOG_CONTROL_HEADING_MSG        LOG_MSG_12
…

/*
 * Declaración funciones para logueo de datos en la memoria flash
 */
void ADS_LogReadControlHeading(void);                // Estado modulo de controladores
void ADS_LogWriteControlHeading(void);

void ADS_LogReadNavigationTun(void);                 // Estado modulo de navegación
void ADS_LogWriteNavigationTun(void);
…

// Asignación de funciones de lectura al sistema de logueo
Log_SetReadLogFunction(LOG_CONTROL_HEADING_MSG,&ADS_LogReadControlHeading);
Log_SetReadLogFunction(LOG_NAV_TUNING_MSG,&ADS_LogReadNavigationTun);
…
```

Otra funcionalidad es la de interactuar con los sistemas del paracaídas en tiempo real mediante el selector de 5 posiciones (descripto en la sección §2.2.6). La función encargada de realizar los cambios se denomina ADS_ReadProportionalSwitch(…); la misma se encuentra integrada en el lazo de ejecución de 3,34 Hz del autopiloto y permite, por ejemplo, cambiar el tipo de controlador a utilizar durante el vuelo en modo automático, entre otras funcionalidades.

2.3.2 Sistema de navegación

El sistema de navegación se encarga de administrar el punto de destino del paracaídas y el plan de vuelo necesario para poder alcanzar dicho punto. Este módulo tiene tres funciones principales para configurar y actualizar el estado del sistema:

```
void ADSNavSys_Init(void);
void ADSNavSys_Reset(void);
void ADSNavSys_Update(void);
```

La función ADSNavSys_Init(...) permite realizar la configuración inicial del sistema. Durante la misma se leen los datos del punto de destino almacenado en la memoria EEPROM y se prepara el sistema para calcular el plan de vuelo.

La función ADSNavSys_Reset(…) se puede usar para borrar el plan de vuelo actual y preparar el sistema para calcular un plan de vuelo nuevo.

La función ADSNavSys_Update(…) primero calcula el plan de vuelo necesario de acuerdo a las estrategias de vuelo que estén implementadas y luego se encarga de actualizar el estado del mismo verificando de acuerdo a la posición actual del paracaídas. Como resultado de este análisis se fija el

próximo punto de navegación hacia el cuál hay que dirigirse y se pasa el mismo al sistema de guiado. En la implementación actual, el sistema utiliza dos estrategias de navegación: ZigZag entre dos puntos para perder altura y luego a partir de cierta altura circuito de aproximación de cara al viento para llegar al punto de destino prefijado.

Las funcionalidades auxiliares implementadas están relacionadas con la configuración del sistema desde el modo de consola y el logueo de datos a la memoria flash.

Hay dos modos de consola para ingresar datos al sistema de navegación:

- Menú para establecer el punto de destino para el próximo vuelo (ADS_SetupLandingTarget(…))
- Menú para fijar valores para diferentes parámetros de configuración del modo de navegación (ADS_SetupNavSystem(…))

El punto de destino al que se quiere llegar se especifica ingresando valores para la latitud, longitud y altura del mismo. En el menú de configuración de parámetros se puede cambiar el valor del radio alrededor de un punto de navegación dentro del cuál el sistema considera que ha alcanzado dicho punto. Existen dos funciones más relacionadas con el modo de consola del autopiloto que solamente sirven para mostrar datos del plan de vuelo y de la configuración del sistema de navegación (ADS_SendMemFlightPlanToConsole(…) y ADS_SendNavSysConfigToConsole(…)).

```
int8_t   ADS_SetupLandingTarget(uint8_t argc, const Menu::arg *argv);
int8_t   ADS_SendMemFlightPlanToConsole(uint8_t argc, const Menu::arg *argv);

int8_t   ADS_SetupNavSystem(uint8_t argc, const Menu::arg *argv);
void     ADS_SendNavSysConfigToConsole(uint8_t format);

void     ADS_SendFlightPlanToConsole(uint8_t format);

void     ADS_LogWriteNavSysStatus(void);
void     ADS_LogReadNavSysStatus(void);
```

Las función de logueo permite guardar los siguientes datos en la memoria externa del autopiloto:

- Estado interno del administrador de tareas del sistema de navegación
- Identificador del punto de navegación actual del sistema de navegación
- Identificador del punto de navegación que se está enviando por el sistema de comunicación
- Variable que identifica si se alcanzó o no el punto de navegación actual

El identificador en el archivo de log de este paquete de datos es la cadena de caracteres “NAVST”. Como se puede observar de esta lista, las variables que se registran están orientadas a determinar el estado interno del sistema de navegación. Esta lista se puede ajustar de acuerdo a las necesidades que vayan surgiendo mediante la modificación del código de las funciones ADS_LogWriteNavSysStatus(…) y ADS_LogReadNavSysStatus(…). A continuación se muestra el código fuente de la función de escritura del mensaje de registro en el log.

```
void ADS_LogWriteNavSysStatus(void){
  DataFlash.WriteByte(HEAD_BYTE1);
  DataFlash.WriteByte(HEAD_BYTE2);
  DataFlash.WriteByte(LOG_NAV_STAT_MSG);

  // [-] Estado del administrador de navegación
  DataFlash.WriteByte(admState);

  // [-] Identificador del WP actual al que está navegando el vehículo
  DataFlash.WriteByte(admCurrentWPId);

  // [-] Identificador del WP actual al que está enviando por la telemetria
  DataFlash.WriteByte(wpTelem.id);

  // [-] Se alcanzó el waypoint ?
  DataFlash.WriteByte((uint8_t)reachTarget);

  DataFlash.WriteByte(END_BYTE);
}
```

2.3.3 Sistema de guiado

El sistema de guiado se encarga de determinar y seguir la trayectoria necesaria para alcanzar un punto de navegación determinado desde la posición actual en la que se encuentra el paracaídas. La implementación actual tiene implementado un sistema sencillo de guiado a un punto, pero es posible ampliar la capacidad del sistema implementado estrategias de guiado más complejas.

El punto de navegación al que se quiere llegar se fija mediante la función ADS_SetTargetWPGuidance(…); el algoritmo calcula la distancia al punto de navegación que se fijo y calcula el rumbo necesario para alcanzar dicho punto. Este valor se devuelve por medio de la función ADS_UpdateGuidanceHeading(…). Cuando se detecta que se alcanzó el punto fijado, el sistema mantiene fijo el rumbo en el último valor calculado antes de entrar dentro del radio del punto de navegación. Se puede determinar si se llegó al punto de destino llamando a la función ADS_ArrivedToWP(…) que devuelve un valor verdadero o falso.

```
void     ADS_InitGuidanceSys(void);
void     ADS_ResetGuidanceMode(void);
int32_t  ADS_UpdateGuidanceHeading(void);

void     ADS_SetTargetWPGuidance(const Location* wp);
```

También hay funciones auxiliares para configurar el modo de guiado desde el modo de consola del autopiloto. En el modo de consola se puede configurar el radio en torno al punto de navegación dentro del cuál se considera que se alcanzó el mismo.

```
int8_t   ADS_SetupGuidanceMode(uint8_t argc, const Menu::arg *argv);
void     ADS_SendGuidanceConfigToCondole(uint8_t format);
```

2.3.4 Pool de controladores de rumbo

El módulo de controladores está organizado mediante un administrador general que provee una interfaz general para ejecutar el control de ciertas variables de vuelo del paracaídas, permitiendo cambiar la arquitectura del controlador que se desea utilizar. En la implementación actual hay implementados tres tipos de controladores para controlar el ángulo de rumbo; las arquitecturas disponibles son:

- Controlador Proporcional
- Controlador LQ
- Controlador MRAS

Las funciones principales del módulo son las siguientes:

```
void     ADS_InitControllers(void);
void     ADS_ResetHeadingController(void);
int16_t  ADS_UpdateHeadingController(void);

void     ADS_SetRefHeadingController(int32_t heading);
int32_t  ADS_GetRefHeadingController(void);

void     ADS_SetCurrentHeadController(const uint8_t controllerId);
uint8_t  ADS_GetCurrentHeadController(void);
```

La función de inicialización se encarga de configurar el administrador de controladores y de inicializar cada uno de los controladores disponibles para realizar el control del rumbo del paracaídas. La función de ADS_ResetHeadingController(…) permite resetear el estado del controlador que esté seleccionado. Se proporciona esta interfaz para poder realizar una reinicialización manual de los controladores por fuera del administrador de controladores. Mediante una llamada a la función ADS_UpdateHeadingController(…) se actualiza el estado del controlador que se este elegido y se obtiene el valor de la acción de comando requerida para realizar la acción de control.

Para cambiar el rumbo de referencia a seguir se utiliza la función ADS_SetRefHeadingController(…) pasando como argumento en la misma el nuevo valor de rumbo. Para cambiar el controlador a utiliza para controlar el ángulo de rumbo hay que utilizar la función ADS_SetCurrentHeadController(…). En el argumento de la función se pasa un entero sin signo de 8bit que identifica alguno de los controladores disponibles. Para simplificar la identificación de dichos controladores se proveen macros para cada uno de ellos.

```
#define ADS_HEADCONTROLLER_PID         1                    /// Controlador de rumbo PID
#define ADS_HEADCONTROLLER_LQ          2                    /// Contrlador de rumbo LQ
#define ADS_HEADCONTROLLER_MRAS        3                    /// Controlador de rumbo MRAS
```

Como función auxiliar se provee un modo iteractivo mediante la consola del autopiloto para poder cambiar los valores de las ganancias de cada uno de los controladores.

```
int8_t ADS_SetupHeadingControllerGains(uint8_t argc, const Menu::arg *argv);
```

2.3.5 Estimador de viento

El módulo de estimación de viento permite controlar el paracaídas mediante una maniobra predefinida y realizar una medición de la dirección y velocidad del viento. Al igual que para el resto de los módulos, se encuentran implementadas tres funciones básicas: inicialización, reset y actualización.

```
void ADS_InitWindEstimation(uint8_t freqCall);
void ADS_ResetWindEstimation(void);
void ADS_EstimateWind(uint8_t mode);

int16_t ADS_GetAsimetricYaw(void);
```

El estimador de viento funciona como un modo de vuelo en sí mismo, ya que requiere ir leyendo datos de posición del paracaídas durante una maniobra predefinida. Por lo tanto el módulo dispone de una función que devuelve un valor para la deflexión mando necesaria para realizar la maniobra. Al completar la medición de viento, el valor de la salida de mando se ajusta para que la misma sea nula y el paracaídas continúe en una trayectoria de vuelo recta.

En la función de inicialización se debe pasar en el argumento un valor como entero sin signo de 8bit que representa el tiempo en milisegundos que existirá entre llamadas a la función ADS_EstimateWind(…). Este valor se utiliza internamente para realizar el cálculo de velocidad del viento. Una vez que se completó la maniobra se almacena en la memoria global los datos de velocidad y dirección de viento calculados. Si se quiere realizar una nueva estimación de viento, hay que reiniciar el estado interno del sistema llamando a la función ADS_ResetWindEstimation(…).

3. CONCLUSIONES

Se realizó una descripción general de los diferentes módulos que componen el programa de autopiloto para control, guiado y navegación de un paracaídas para lanzamiento de cargas. El código fuente del autopiloto se encuentra dividido en dos bloques generales. Uno que implementa el lazo de ejecución principal y todas aquellas tareas relacionadas con el control de los periférico del hardware que no están vinculados directamente con las tareas asociadas al control del paracaídas. Mientras que en el segundo bloque se encuentran implementadas las funcionalidades que permiten utilizar el autopiloto como computadora de vuelo para un paracaídas. Esta arquitectura de software, permite tener flexibilidad para cambiar el tipo de vehículo que se quiere controlar sin tener que reescribir todo el código fuente desde cero.

En cuanto a las funcionalidades particulares asociadas con la utilización de un paracaídas para el lanzamiento de cargas con precisión, el programa de autopiloto se encuentra dividido por sistemas que ejecutan diferentes tareas:

- sistema de control de rumbo con posibilidad de cambiar la arquitectura del controlador a utilizar.
- sistema de guiado simple a un punto

- sistema de navegación con una estrategia de pérdida de altura y de aproximación final para aterrizar en el punto de destino prefijado

Hay un sistema auxiliar que no está relacionado con las funciones de control, pero que es complementario para estas tareas, que permite estimar la velocidad y la dirección del viento durante el vuelo realizando una maniobra predeterminada.

Todos estos sistemas están implementados como librerías que pueden ser ampliadas a medida que el proyecto lo requiera.

Estos sistemas están integrados en un módulo general que se encarga de la administración de cada uno de ellos dependiendo de las tareas requeridas durante cada fase del vuelo en modo automático, el cuál se puede configurar dentro del mismo módulo.

4. REFERENCIAS

[1] LLORENS, D. *"Descripción general del funcionamiento del programa ArduPilotMega"*. Dpto de Mecánica Aeronáutica, Facultad de Ingeniería, Instituto Universitario Aeronáutico. Informe técnico DMA-015/11. Octubre de 2011.

[2] REYNOSO, S. y LLORENS, D. *"Implementación del código fuente del sistema de control de rumbo y altitud"*. Dpto de Mecánica Aeronáutica, Facultad de Ingeniería, Instituto Universitario Aeronáutico. Informe técnico DMA-003/13. Abril de 2013.

[3] REYNOSO, S. *"Descripción del hardware del autopiloto ArduPilotMega"*. Dpto de Mecánica Aeronáutica, Facultad de Ingeniería, Instituto Universitario Aeronáutico. Informe técnico DMA-017/11. Noviembre de 2011.

[4] Microchip. *Atmega 2560 – Microcontrollers and Processors* [en línea]. [USA: Microchip Technology Inc.], s/d, s/d. <https://www.microchip.com/wwwproducts/en/ATmega2560> [Consulta: 18 de Mayo de 2017]

[5] Wikipedia. *Information hiding* [en línea]. [USA: Wikimedia Foundation, Inc.], s/d., 20 de Junio de 2017. <https://en.wikipedia.org/wiki/Information_hiding> [Consulta: 7 de Septiembre de 2017]

[6] Atmel. *The .bss Section – AVR Libc Reference Manual* [en línea]. [USA: Atmel Corporation], s/d., s/d. <http://www.atmel.com/webdoc/avrlibcreferencemanual/mem_sections_1sec_dot_bss.html> [Consulta: 22 de Mayo de 2017]

[7] Atmel. *Storing and Retrieving Strings in the program space – AVR Libc Reference Manual* [en línea]. [USA: Atmel Corporation], s/d., s/d. <http://www.atmel.com/webdoc/avrlibcreferencemanual/pgmspace_1pgmspace_strings.html> [Consulta: 22 de Mayo de 2017]

[8] Microchip. *ATmega 328P – Microcontrollers and Processors* [en línea]. [USA: Microchip Technology Inc.], s/d, s/d. <https://www.microchip.com/wwwproducts/en/ATmega328P> [Consulta: 31 de Mayo de 2017]

[9] GNU Project. *Using the GNU Compiler Collection (GCC): inline* [en línea]. [USA: Free Software Foundation, Inc.], s/d. , s/d. <https://gcc.gnu.org/onlinedocs/gcc/Inline.html> [Consulta: 12 de Junio de 2017]

[10] Bosch Sensortec. *"BMP 085 Digital pressure sensor – Data Sheet"*. Rev. 1.0 ed. Reutlingen, Germany: Bosch Sensortec GmbH, 2008. 25 p. BST-BMP085-DS000-03.

[11] LLORENS, D. *"Protocolo de comunicación para microcontrolador de 8bit"*. Dpto de Mecánica Aeronáutica, Facultad de Ingeniería, Centro Regional Universitario Córdoba - Instituto Universitario Aeronáutico. Informe técnico DMA-013/17. Agosto de 2017.

[12] DIGI. *Xbee transparent mode – RF Kits Common – Digi Docs* [en línea]. [USA: Digi International Inc.] s/d. 2015. <http://docs.digi.com/display/RFKitsCommon/XBee+transparent+mode> [Consulta: 14 de Junio de 2017]