INSTITUTO UNIVERSITARIO AERONÁUTICO
Departamento Mecánica Aeronáutica

PROTOCOLO DE COMUNICACIÓN PARA MICROCONTROLADOR DE 8 BIT

Informe técnico: DMA-013/17 **Revisión:** /

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

Fecha: 07/08/17

Autor:

Ing. Diego Llorens
Investigador

Revisó:

Ing. Andrés Liberatto
Director de proyecto

PROTOCOLO DE COMUNICACIÓN PARA MICROCONTROLADOR DE 8 BIT

Por:

Ing. Diego Llorens

RESUMEN

En el marco del proyecto PIDDEF 038/14 – “Paracaídas Comandado Autónomo”, se desarrolló un protocolo de comunicación simple que puede ser utilizado en un microcontrolador con arquitectura de 8bit. El mismo está basado en mensajes estructurados compuestos por un encabezado, un paquete de información y un fin de mensaje. El encabezado permite identificar mensajes específicos, el paquete de información es de longitud variable de acuerdo al mensaje y el fin de mensaje incluye un control de integridad del mismo mediante un checksum simple de toda la información que contiene el mensaje.

Se presenta una librería implementada utilizando lenguajes C/C++ para leer y escribir los mensajes en un espacio de memoria administrado mediante dos buffers circulares. A partir de la misma se pueden desarrollar los mensajes particulares que se quieran implementar para enviar y recibir información por telemetría en el proyecto “Paracaídas Comandado Autónomo”.

Se presentan, además, algunas mediciones del tiempo de ejecución de las funciones de lectura y escritura de mensajes en un microcontrolador de 8bit. Se encontró que el tiempo de ejecución promedio en 50 ciclos de medición, tanto para la lectura como para la escritura, de un mensaje con un paquete de información de 40 bytes es de 277 [µs] con el CPU operando a 16 [MHz].

Córdoba, 07 de agosto de 2017

ÍNDICE

ABREVIACIONES

Abreviación	Descripción
ASCII	American Standard Code for Information Interchange
CAN	Controller Area Network
FPU	Floating Point Unit
I^2C	Inter-Integrated Circuit
IMU	Inertial Measurement Unit
ITU	International Telecommunications Union
SAE	Society of Automotive Engineers
TCP/IP	Transmission Control Protocol/Internet Protocol
UART	Universal Asynchronous Receiver-Transmitter
UDP	User Datagram Protocol

1. INTRODUCCIÓN

En el marco del proyecto paracaídas comandado autónomo es necesario disponer de un medio de comunicación flexible para enviar datos desde el autopiloto del paracaídas hacia la estación terrena durante las pruebas de campo que se realicen.

Actualmente el código del autopiloto está usando el protocolo original de ArduPilot en conjunto con el programa de estación terrena HappyKillMore. Este protocolo funciona mediante el envío de mensajes no codificados en formato de cadenas de caracteres (ASCII) con lo cual los mismos ocupan un tamaño en bytes considerable (es posible enviar la misma información codificada en forma binaria con una cantidad mucho menor de bytes). Otro inconveniente que se presenta es que el programa de estación terrena no es abierto en cuanto a la incorporación de nuevos mensajes; esto limita la cantidad de datos de estado del autopiloto que se pueden enviar.

Un protocolo de comunicación que se utiliza en vehículos no tripulados en el campo de investigación es el MAVLink. El mismo es un conjunto de archivos de encabezado de C++ que permite codificar estructuras de datos de C y enviarlas a través de un puerto de comunicación serie (UART) [1]. Este protocolo ha sido probado en varios desarrollos abiertos de autopilotos para vehículos aéreos (PX4, PIXHAWK, APM, Parrot).

Este protocolo se encuentra ampliamente desarrollado y cuenta con más de 300 mensajes para comunicación. Ha evolucionado al punto de permitir no sólo la comunicación entre vehículos con una estación terrena, sino también de los vehículos entre sí y de sistemas internos del autopiloto entre ellos (por ejemplo, IMU con computadora de vuelo principal).

2. DESARROLLO

Se busca generar un protocolo de comunicación que cumplan con los siguientes requerimientos:

- que sea flexible para la incorporación y envío de mensajes en la medida que el proyecto lo requiera
- que los mensajes estén codificados de alguna manera particular para hacer los mismos más reducidos en tamaño (respecto a un mensaje equivalente como cadena de caracteres)
- que el protocolo en general se pueda utilizar en una arquitectura de un microcontrolador de 8 bit (con las limitaciones de memoria y espacio que tienen esta arquitectura básica).

2.1 Análisis del protocolo MAVLink

Tal como se mencionó en la Introducción, el protocolo MAVLink, cuenta con algunos de los requerimientos planteados para el protocolo a desarrollar. Es por ello que a continuación se presenta un breve análisis del mismo para tenerlo como referencia durante el desarrollo del protocolo de comunicación propio.

El formato del mensaje del protocolo MAVLink está inspirado en los protocolos de comunicación CAN y SAE AS-4 Standard y consta de una serie de bytes fijos y un paquetes de bytes que representan el contenido del mensaje que puede tener longitud variable (ver Figura 1). En la Tabla 1 está una descripción de los campos del mensaje MAVLink. El uso de dos bytes con un valor de checksum permite determinar la integridad del mensaje recibido y asegura que el contenido del mensaje está correcto.

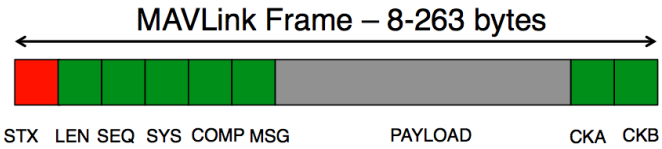

Figura 1: Formato de un mensaje del protocolo MAVLink [1]

Byte #	Ref	Contenido	Valor	Descripción
0	STX	Packet start sign	v1.0: 0xFE (v0.9: 0x55)	Indicates the start of a new packet.
1	LEN	length	0 - 255	Indicates length of the following payload.
2	SEQ	Packet sequence	0 - 255	Each component counts up his send sequence. Allows to detect packet loss
3	SYS	System ID	1 - 255	ID of the SENDING system. Allows to differentiate different MAVs on the same network.
4	COMP	Component ID	0 - 255	ID of the SENDING component. Allows to differentiate different components of the same system, e.g. the IMU and the autopilot.
5	MSG	Message ID	0 - 255	ID of the message - the id defines what the payload “means” and how it should be correctly decoded.
6 to (n+6)	PAYLOAD	Data	(0 - 255) bytes	Data of the message, depends on the message id.
(n+7) to (n+8)	CKA y CKB	Checksum (low byte, high byte)		ITU X.25/SAE AS-4 hash, excluding packet start sign, so bytes 1..(n+6) Note: The checksum also includes MAVLINK_CRC_EXTRA (Number computed from message fields. Protects the packet from decoding a different version of the same packet but with different variables).

Tabla 1: Descripción de los campos de mensajes MAVLink [1]

Los mensajes que envía el protocolo se pueden agrupar en dos grupos generales: mensajes de información y mensajes de comando. Los mensajes de información se envían de un sistema a otro y no requieren respuesta alguna desde el sistema que los está recibiendo; mientras que los mensajes de comando se utilizan para ejecutar ciertas acciones en el sistema y requieren de una comunicación

bidireccional entre ambos sistemas (envío de comando y respuesta). Por ejemplo, el manejo de información de puntos de navegación se realiza por medio de una serie de mensajes que forman el "Waypoint Protocol" y que funciona usando una máquina de estado dentro del protocolo de comunicación para administrar el envío y recepción de datos relacionados con los puntos de navegación.

En la Figura 2 se muestra el esquema de operación del sistema para recibir la lista de puntos de navegación de la misión. La comunicación se inicia enviando un mensaje especial (en este caso WAYPOINT_REQUEST_LIST); el sistema que envió el mensaje inicia una cuenta regresiva y queda esperando respuesta por parte del sistema consultado. La cuenta regresiva permite tomar una decisión en caso de no obtener respuesta del otro sistema. El sistema consultado envía una respuesta particular a la consulta efectuada (WAYPOINT_COUNT(N) en este caso) e inicia un mecanismo de espera similar al explicado antes. De esta manera ambos sistemas se comunican de manera sincronizada, esperando a que cada uno de ellos esté listo para recibir/enviar la información solicitada. El proceso de comunicación finaliza con el envío de un mensaje especial (WAYPOINT_ACK para el caso mostrado) que hace que ambos sistemas den por completado el envío / recepción de información.

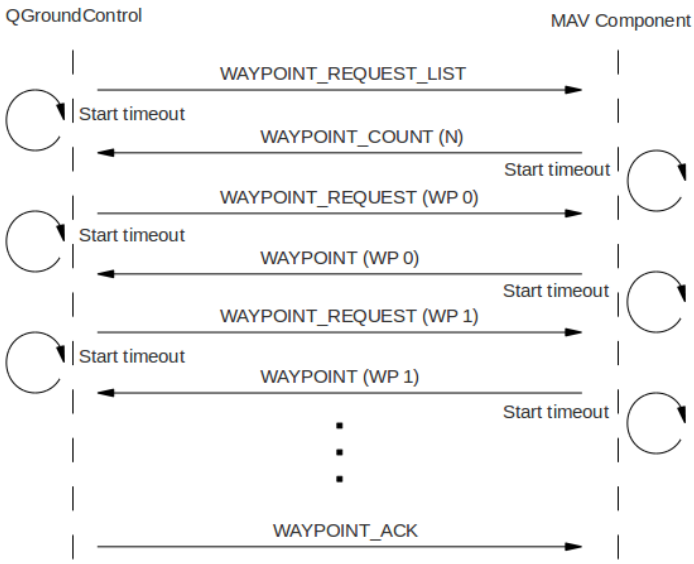

Figura 2: Esquema de comunicación para recibir la lista de puntos de navegación de la misión del protocolo MAVLink [2]

Con este mecanismo de comunicación, MAVLink, implementa todos los procesos de intercambio de datos entre sistemas y el envío de comandos para ejecutar acciones en el sistema.

Si bien este protocolo de comunicación es muy completo y tiene amplia difusión en los sistemas autónomos, la diversidad de sistemas en los que se usa ha hecho que el protocolo sea complejo en la implementación de mensajes y la variedad de mensajes que dispone no se alcanza a utilizar en su mayoría o no son aplicables al sistema en desarrollo.

Es por ello que se propone desarrollar un protocolo de comunicación basado en la filosofía de MAVLink pero simplificado en cuanto a la cantidad de mensajes y orientado en particular al sistema de paracaídas comandado autónomo.

2.2 Protocolo de comunicación propio: ADSLink

El nombre adoptado para el protocolo es ADSLink que es un acrónimo de las palabras Air Delivery System Link.

La definición del protocolo de comunicación se centra en la estructura del mensaje que se quiere enviar y recibir ya que este elemento define la forma de interpretar cada uno de los valores que forman el mensaje. Es por ello que a continuación, primero, se define la estructura del mensaje, luego se propone la representación para el “buffer” de mensajes y finalmente las operaciones de lectura y escritura para el protocolo. De esta manera se irá de los más abstracto a lo detallado en la definición e implementación del protocolo.

2.2.1 Definición del mensaje para protocolo ADSLink

La estructura de mensaje de MAVLink es compacta y completa para el envío y recepción de datos por lo que se adopta una estructura similar, pero simplificando el método de cálculo del checksum para no tener que programar un algoritmo complejo. En este caso se utilizará el mecanismo más sencillo de obtención de un checksum que es la suma del valor numérico de los bytes que forman el mensaje. Si bien este algortimo no es robusto frente a algunos errores que se pueden producir en el mensaje (ej.: intercambio de bytes entre dos posiciones dentro del mensaje: el valor de suma es igual pero la decodificación de los datos asociados a estos bytes es diferente y por lo tanto no es correcta), permite una implementación sencilla y no sobrecarga un procesador de 8bit sin FPU con operación matemáticas complejas (multiplicaciones, divisiones, módulo, etc).

La estructura de mensaje que se propone (similar a la de MAVLink) es mediante un vector de bytes (con representación de 8 bit) en donde determinadas posiciones se usan para interpretar los datos que se están enviando. En la Figura 3 se muestra un esquema con la estructura del mensaje, mientras que en la Tabla 2 se detallan el significado de cada byte del mensaje y los valores que puede adoptar.

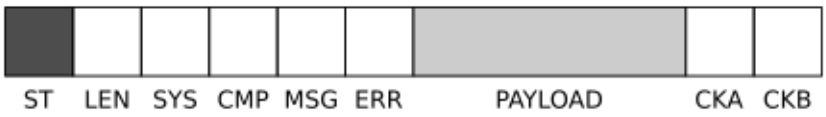

Figura 3: Estructura de mensaje del protocolo ADSLink

Si bien el tamaño del payload puede ser variable, el mismo tiene un valor máximo limitado por la representación que se está usando para enviar el largo del mensaje: 1 byte sin signo de 8 bit, el que puede adoptar un valor máximo de 255. Se propone que el tamaño del mensaje sea desde el byte #0 hasta el byte #(n+6) (último valor del payload), por lo tanto el tamaño máximo del payload puede ser de 249 bytes. Esta cantidad de bytes de payload permitiría enviar hasta 62 valores de punto flotante (asumiendo una representación de 4 bytes para los mismos) en un sólo mensaje.

El máximo valor de checksum que tendría un mensaje en caso de tener 249 bytes de payload y 6 bytes de encabezado sería de 255 x 255 = 65025, valor que se puede representar mediante un entero sin signo de 16 bit (0 a 65535) y que ocuparía los dos últimos bytes del mensaje.

Byte #	Ref	Contenido	Valor	Descripción
0	ST	Byte inicio de mensaje	0xFE	Marca el inicio de un nuevo mensaje
1	LEN	Largo	0 - 255	Indica el largo del mensaje (cantidad de bytes).
2	SYS	System ID	0 - 255	ID del sistema que envía el mensaje. Permite diferenciar diferentes sistemas en una misma red (ej. estación terrena, paracaídas, etc.).
3	CMP	Component ID	1 - 255	ID del componente que envía el mensaje. Permite diferenciar diferentes componentes en un mismo sistema (ej. IMU y la computadora principal).
4	MSG	Message ID	0 - 255	ID del mensaje. De acuerdo a este identificador se decodifican los datos del mensaje.
5	ERR	Error Status	0 - 255	Reservado – sin uso actualmente
6 to (n+6)	PAYLOAD	Datos	(0 - 255) bytes	Datos del mensaje. Depende del mensaje.
(n+7) to (n+8)	CKA y CKB	Checksum (low byte, high byte)	0 - 65535	Bytes del valor de la suma de todos los bytes del mensaje (excepto los 2 de checksum)

Tabla 2: Identificación de los bytes del mensaje para el protocolo ADSLink

2.2.2 Definición de la representación en memoria del mensaje (buffer)

En general el envío y recepción de información digitalizada opera mediante una secuencia FIFO (First Input First Output) [3] en donde los datos que se reciben se acumulan de manera secuencial en un buffer de memoria y se leen en el orden en que se recibieron.

Debido a las limitaciones de memoria que tiene un microcontrolador, resulta muy útil implementar el vector de entrada / salida de datos como un buffer circular [4]. En esta representación, los datos se almacenan en un vector de longitud finita; el manejo de la lectura / escritura en el vector tiene que tener implementada alguna lógica para que al llegar a la última posición del vector, tanto la lectura como la escritura de datos, continúe en la posición inicial del vector. De esta manera nunca se sobrepasa el tamaño máximo (overflow) del vector. Esta implementación también permite que la velocidad a la que se leen y escriben los datos sea diferente (sin llegar al punto en el que se escriben más datos que los que se leen) y que las operaciones de lectura y escritura en el buffer puedan funcionar de manera asíncrona.

Se propone implementar el protocolo usando dos buffer circulares: uno para los mensajes que se quieren enviar desde el sistema y otro para recibir los mensajes que son enviados al sistema (ver Figura 4). Si bien esta representación aumenta el tamaño en memoria del protocolo, se considera que simplifica la lógica de operación del mismo ya que los datos de se encuentran desacoplados y no se debe identificar constantemente si los mensajes pertenecen al propio sistema ó vienen de algún otro sistema.

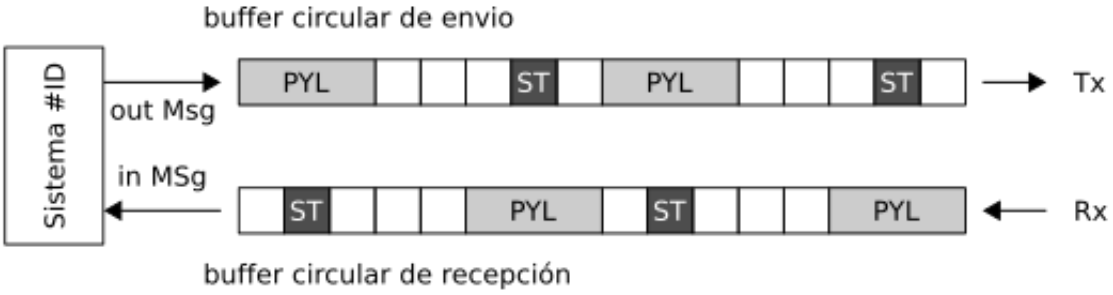

Figura 4: Esquema de buffers para envío y recepción de mensajes - protocolo ADSLink

La utilización de dos buffer circulares para el envío y recepción de mensajes hace que el protocolo pueda ser independiente del hardware que se utilice para la transmisión y recepción de los datos. Esto agrega sobrecarga en la ejecución del protocolo, pero desacopla totalmente el protocolo del hardware, siendo posible utilizarlo como una herramienta de comunicación con diferentes interfaces de comunicación (Serial, I^2C, TCP/IP, UDP, etc.).

2.2.3 Operaciones de lectura / escritura (I/O) para el protocolo

El protocolo requiere de dos pares de operaciones básicas: recepción y decodificación de mensajes (lectura) y codificación y transmisión de mensajes (escritura). Por recepción y transmisión se entiende las operaciones de lectura y escritura al buffer circular correspondiente, mientras que por decodificación y codificación se entiende las operaciones necesarias para transformar la información que se quiere recibir, o enviar, al formato del mensaje propuesto anteriormente.

Al estar la codificación de mensajes en este nivel de la implementación del protocolo, se propone que la operación de lectura analice la integridad de los mensajes que se reciben. Es por esto que se debe implementar a este nivel el cálculo del checksum del mensaje, tanto para verificar si el mensaje recibido está correcto como para agregar el valor de verificación para el mensaje que se está enviando.

Para dar flexibilidad al protocolo hay que considerar la posibilidad que la lectura y escritura de mensajes se realice a diferentes velocidades, así como también que los mensajes que se reciben lleguen en partes debido a la forma de funcionamiento del hardware de transmisión de datos.

En el estudio de las operaciones de lectura y escritura es conveniente dividir al mensaje en tres segmentos que surgen naturalmente para simplificar la programación del código fuente: un encabezado, el paquete de datos (“payload”) y la finalización de mensaje (“tail”) (ver Figura 5). El encabezado incluye los 6 bytes que preceden al payload y que son fijos mientras que la finalización de mensaje, en este caso, estaría representada por los 2 bytes del checksum.

Figura 5: Segmentos del mensaje para operaciones de lectura / escritura

En las Figura 6 y Figura 7 se muestran una propuesta de los diagramas de flujo para las operaciones de lectura y escritura de los mensajes del protocolo. La operación de lectura es más compleja ya que contempla la posibilidad de leer el mensaje de a partes (esperando que se complete el mensaje desde el dispositivo de transmisión). En cambio, la operación de escritura es muy sencilla ya que solamente hay que escribir los datos del mensaje a enviar en el orden apropiado.

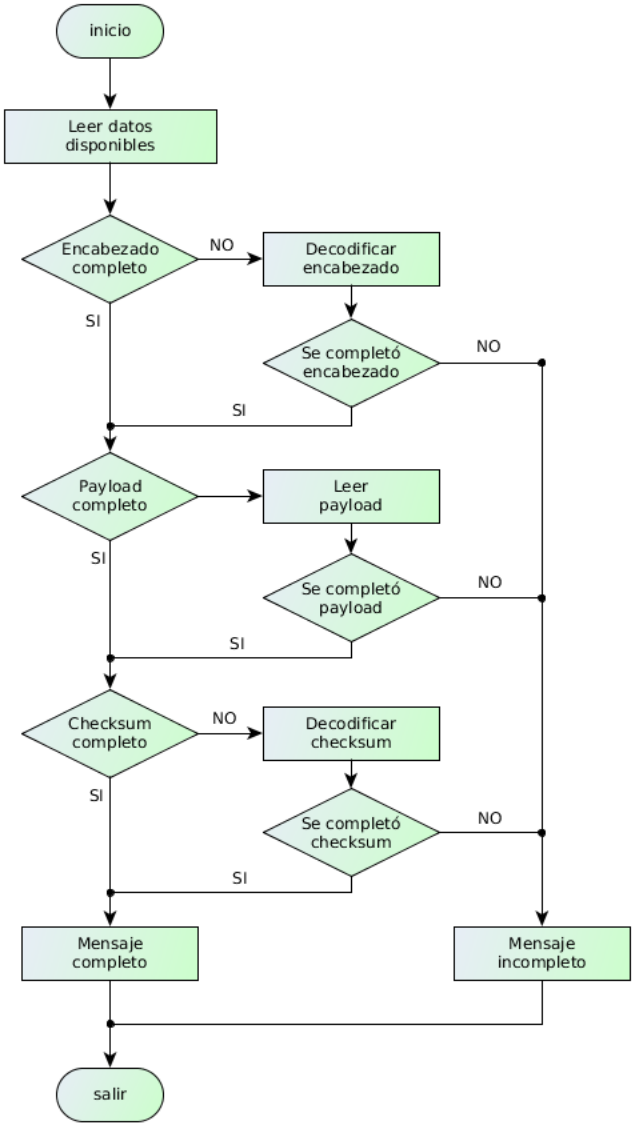

Figura 6: Diagrama de flujo para lectura de mensajes - Protocolo ADSLink

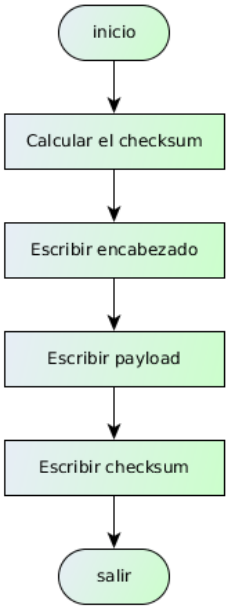

Figura 7: Diagrama de flujo para escritura de mensajes - Protocolo ADSLink

El diagrama de flujo de lectura de mensaje no incluye el control de integridad del mensaje (para no complicar el diagrama), pero el mismo se incluirá en el desarrollo del código fuente del protocolo.

2.3 Implementación del protocolo ADSLink usando C/C++

Para la implementación del protocolo se utilizarán elementos de los lenguajes de programación C y C++ con los que se está desarrollando el código fuente del autopiloto del paracaídas comandado autónomo. Se tratará de utilizar en todo momento los tipos de datos para una arquitectura de 8 bit para optimizar la ejecución del código del protocolo en estas plataformas.

El código se desarrollará al estilo de una librería del lenguaje C (estructuras de datos y funciones que operan sobre estas estructuras) y no usando programación orientada a objetos [5]. Esta aproximación permite mantener a un nivel bajo la sobrecarga por llamada a funciones y permite limitar más la abstracción de datos que usando la programación de objetos.

2.3.1 Implementación del mensaje del protocolo

La implementación del mensaje del protocolo se realiza usando la siguiente metodología: los datos que son fijos en el mensaje y que se encuentran presentes siempre en el mismo se agrupan en una estructura de datos de C mientras que el paquete de datos se representa mediante un vector de enteros sin signo de 8 bit (uint8_t).

```
#ifndef ADSLINKTYPES_MSGHDR
#define ADSLINKTYPES_MSGHDR

#define ADSLINK_MSG_HEADER_SIZE       6U      // Tamaño en bytes del encabezado del mensaje
                                              // (sin el checksum)
/*
 * Definición de los campos para los mensajes.
 * Campo        índice
 * msgHeader    0
 * msgSize (n)  1
 * systemID     2
 * componentID  3
 * msgID        4
 * msgError     5
 * data         6 to (n+5)
 * checksum     (n+6) & (n+7)
 */
typedef struct {
 uint8_t msgHeader;     // Encabezado general para todos los mensajes del protocolo
 uint8_t msgSize;       // Cantidad de bytes de los datos del mensaje (n)
 uint8_t systemID;      // Identificador de sistema
 uint8_t componentID;   // Identificador de componente dentro del sistema
 uint8_t msgID;         // Identificador de mensaje correspondiente al componente -> sistema
 uint8_t msgError;      // Identificador de errores para envió y recepción de mensajes
 uint16_t checksum;     // Checksum para controlar integridad del mensaje
} tMessageFields;

#endif
```

El control de la asignación de memoria del vector de datos se deja por fuera del protocolo para permitir ajustarlo a las necesidades de cada implementación en la que se use el protocolo, por lo tanto, el mismo, se trabaja como un puntero de memoria a la primera posición del vector.

Se define una estructura de datos adicional para encapsular los datos básicos de mensaje y que sirva como tipo de datos para identificar al mismo en las capas superiores del protocolo en las cuales no

es necesario conocer los detalles de la implementación del mensaje (byte de inicio de mensaje, checksum, etc.).

```
#ifndef ADSLINKTYPES_MSGID
#define ADSLINKTYPES_MSGID

typedef struct {
 uint8_t systemID;      // Identificador de sistema
 uint8_t componentID;   // Identificador de componente dentro del sistema
 uint8_t msgID;         // Identificador de mensaje correspondiente al componente -> sistema
 uint8_t msgError;      // Identificador de errores para envió y recepción de mensajes
 uint8_t payloadLenght;        // Cantidad de bytes del payload de este mensaje.
} tMessageID;

#endif
```

2.3.2 Implementación del buffer circular para acumular mensajes

Si bien existe la implementación en el lenguaje C de vectores de datos, los mismos no operan como un buffer circular, sino que funcionan como un espacio de memoria en donde se puede acceder directamente a las ubicaciones individuales de memoria en donde se encuentra el dato del tipo en que se declaró el vector. Esto hace que los vectores sean estructuras de datos muy rápidas, pero tienen el inconveniente que no proporcionan seguridad en el indexado de los datos ya que no es posible determinar el tamaño del vector, ó si se está escribiendo en una posición fuera del vector.

En la librería estándar de C++ esto se soluciona mediante la definición de un tipo de contenedor llamado “<vector>” [6]. Este contenedor especializado está programado como una clase y es segura para la operación de elementos (lectura, escritura, etc.). Además es un contenedor que permite variar el tamaño en forma dinámica. El inconveniente con la utilización de esta implementación en un microcontrolador de 8bit, es que la memoria y la velocidad de ejecución de este último requiere de estructuras de programación sencilla y seguras.

Es por esto que se propone una implementación propia para un buffer circular basado en un vector estándar de C para tratar de hacer lo más eficiente y rápida las operaciones de lectura y escritura en el buffer. La implementación está basada en un ejemplo de la referencia [4].

El buffer circular se representa mediante una estructura de datos de C en donde se definen los siguientes elementos:

- puntero de memoria al vector donde se acumulan los datos
- tamaño del vector
- un indice para la posición de lectura del buffer circular
- un índice para la posición de escritura del buffer circular
- un índice especial para poder volver leer elementos del vector

Como se va a utilizar en una arquitectura de 8bit se utiliza una representación de enteros sin signo de 8bit para los datos de la estructura. Esto puede ser, fácilmente extendido a otras arquitecturas definiendo tipos acordes.

```
typedef struct{
  uint8_t *pBuffer;     // Block of memory
  uint8_t size;         // Must be a power of two
  uint8_t read;         // Holds current read position: 0 to (size-1)
  uint8_t write;        // Holds current write position: 0 to (size-1)
  uint8_t free;         // Holds current free position: 0 to (size -1)
} tCircularBuffer8;
```

Se implementan 6 operaciones básicas para el buffer circular:

- Inicialización
- Lectura
- Escritura
- Copia de un elemento a una dirección de memoria
- Borrado
- Consulta de la cantidad de elementos disponibles para leer

Las operaciones se implementan utilizando aritmética del complemento de dos [7] para hacer eficientes las operaciones de lectura y escritura al llegar a los extremos del buffer sin tener que usar condicionales. Esto impone una restricción fuerte en el tamaño del vector ya que para poder usar este tipo de aritmética, el tamaño del vector debe ser un valor que se obtenga como potencia del número 2. La mayor potencia de 2 que no sobrepasa el máximo valor de un entero sin signo de 8 bit (255) es 7 (2^7 = 128). Por lo tanto los tamaños máximos de vectores con los que puede trabajar esta implementación del buffer circular en una arquitectura de 8bit es 128 elementos.

Se definen las siguientes macros para los tamaños máximos de vectores admisibles en la implementación de 8 bit:

```
#ifndef CIRCULARBUFF_DEFINES_H_
#define CIRCULARBUFF_DEFINES_H_

// Definiciones de tamaño en base 2 para los buffers circulares
#define CBSIZE_2        2U
#define CBSIZE_4        4U
#define CBSIZE_8        8U
#define CBSIZE_16       16U
#define CBSIZE_32       32U
#define CBSIZE_64       64U
#define CBSIZE_128      128U

#endif
```

La función de inicialización del buffer almacena en la estructura de datos la dirección de memoria al primer elemento del vector de datos que se quiere usar como un buffer circular así como el tamaño del mismo; además se ajustan al elemento inicial del vector (posición cero) los índices de posición (lectura, escritura y re-lectura).

```
void CBInit8(tCircularBuffer8 *cb, uint8_t* pBuffer, uint8_t size){
  cb->pBuffer = pBuffer;
  cb->size = size;
  cb->read = 0;
  cb->write = 0;
  cb->free = 0;
}
```

La función de escritura toma como argumento el elemento que se quiere agregar al buffer circular y lo copia el vector de datos. La posición a usar para escribir dicho elemento se calcula usando aritmética del complemento de dos entre la posición actual del índice de escritura y el tamaño del vector. Esto garantiza que al llegar a la última posición del vector, el próximo índice sea el del primer elemento del vector. La implementación actual no verifica si el elemento de la posición a escribir ha sido leído o no, lo cuál puede ser un inconveniente si la tasa de lectura es menor que la de escritura. Se obvió esta verificación para tratar de hacer lo más rápida posible la ejecución de la operación de escritura.

```
void CBWrite8(tCircularBuffer8 *cb, uint8_t *data){
  cb->pBuffer[cb->write] = *data;
  cb->write = (cb->write + 1) & (cb->size - 1);
}
```

La función de lectura solamente avanza el índice de posición de lectura al siguiente lugar sin devolver el elemento. Esta forma de operar está pensada desde un punto de vista de optimización para acceder al elemento del vector de datos directamente (usando el índice de lectura del buffer circular) sin tener que hacer una copia del mismo. Es por ello que el uso de la operación de lectura requiere tener ciertas precauciones a la hora de operar tanto con el vector de datos como con el índice de posición de lectura.

```
void CBRead8(tCircularBuffer8* cb){
  cb->read = (cb->read + 1) & (cb->size - 1);
}
```

Se provee una función de copia para extraer los elementos del vector de datos para aquellas operaciones donde el tiempo de ejecución no sea un problema y poder operar el vector de datos de

manera más segura. La operación hace una copia del elemento del vector de datos a la dirección de memoria que se pasa como argumento de la función seguida de una operación de lectura (avance del índice de posición de lectura).

```
void CBCopy8(tCircularBuffer8 *cb, uint8_t *data){
  *data = cb->pBuffer[cb->read];
  cb->read = (cb->read + 1) & (cb->size - 1);
}
```

La función de borrado del buffer circular reinicia todos los índices a la posición inicial del vector de datos. Esta no es una operación estricta de borrado de memoria, sino que unicamente permite reescribir el vector de datos sin tener en cuenta los datos que pueda haber en el mismo y que no se hallan leído aún. La consulta de la cantidad de elementos luego de ejecutar la función de borrado del buffer devuelve 0 elementos disponibles para la lectura.

```
void CBFlush8(tCircularBuffer8* cb){
  cb->read = 0;
  cb->write = 0;
  cb->free = 0;
}
```

La función de conteo de elementos disponibles devuelve la cantidad de elementos nuevos que hay para la lectura. La misma se basa en la aritmética del complemento de dos para calcular la diferencia entre los índices de escritura y lectura sin tener que usar un condicional para detectar cuando el índice de lectura tiene un valor mayor que el de escritura (índice de lectura por delante del de escritura).

```
uint8_t CBAvailable8(tCircularBuffer8 *cb){
  return ((cb->write - cb->read) & (cb->size - 1));
}
```

2.3.3 Implementación de las operaciones de lectura / escritura de mensajes

A continuación se detallan las operaciones del lectura y escritura para los mensajes del protocolo. Durante la implementación de estas operaciones fue necesario desarrollar una estructura para administrar estas operaciones y darle mayor flexibilidad al protocolo. Esta estructura se denominó nodo de comunicación y se explica más adelante en el informe.

2.3.3.1 Lectura de mensajes

De las operaciones sobre mensajes, la lectura es la más compleja, ya que contempla la posibilidad que los datos que forman el mensaje lleguen de a partes. Aprovechando en que era

necesario calcular el checksum del mensaje a este nivel de la implementación se incorporó el control de integridad durante la lectura del mensaje.

La lectura se lleva a cabo por medio de una función que recibe como argumentos el nodo de comunicación, un identificador de mensajes, dirección de memoria al primer elemento del vector para escribir el payload y el tamaño del vector de payload.

```
uint8_t ADSLinkReadMessage(tAdsLink* link,
         tMessageFields* msg,
         uint8_t* payload,
         const uint8_t payloadMaxSize);
```

La función devuelve un entero sin signo de 8bit que representa el resultado de la operación de lectura de mensaje, siendo el mismo, alguno de los siguientes valores definidos por medio de macros.

```
// Mensajes del lector de mensajes del protocolo
#define ADSLINK_MSG_READER_NOMSG 0U   // No se detectó ningún mensaje en el buffer
#define ADSLINK_MSG_READER_RDHDR 1U   // Se detectó el comienzo de un mensaje, pero el
                                      // encabezado está incompleto. Esperando a que se llene
                                      // el buffer.
#define ADSLINK_MSG_READER_RDDAT 2U   // Se completó la lectura del encabezado del mensaje y
                                      // se comenzó a leer el payload, pero el mismo está
                                             // incompleto. Esperando a que se llene el
buffer.
#define ADSLINK_MSG_READER_RDCHK 3U   // Se completó la lectura del payload del mensaje y se
                                      // comenzó a leer el checksum, pero el mismo está
                                      // incompleto. Esperando a que se llene el buffer.
#define ADSLINK_MSG_READER_MSGOK 4U   // Se completó la lectura de todo el mensaje y el
                                      // checksum enviado corresponde con el checksum
                                      // calculado con los datos recibidos. Mensaje OK.
#define ADSLINK_MSG_READER_BADHDR 5U  // Se detectó el comienzo de un mensaje pero hay un
                                             // problema de consistencia en el contenido del
                                             // encabezado del mensaje.
#define ADSLINK_MSG_READER_BADDAT 6U  // Se completó la lectura del encabezado del mensaje y
                                      // se comenzó a leer el payload, pero el mismo tiene un
                                      // problema de consistencia del contenido.
#define ADSLINK_MSG_READER_BADCHK 7U  // Se completó la lectura de todo el mensaje pero el
                                      // checksum recibido y el calculado con los datos no
                                      // corresponden. El mensaje está corrupto.
```

La lectura de mensajes se realiza mediante una máquina de estado que verifica qué parte del mensaje se está leyendo y además si puede o no avanzar con la lectura del siguiente segmento del mensaje. El funcionamiento de la misma no es bloqueante, es decir no detiene la ejecución de todo el programa en caso que el mensaje esté incompleto, sino que verifica el estado actual de lectura y lo almacena para la próxima llamada. Esto permite llamar a esta función a intervalos fijos de tiempo. Al salir siempre devuelve el estado en que está la lectura del mensaje actual en el buffer circular. En la Figura 8 se muestra el diagrama de bloques de la función, mientras que en el Anexo A se lista el código fuente de la misma.

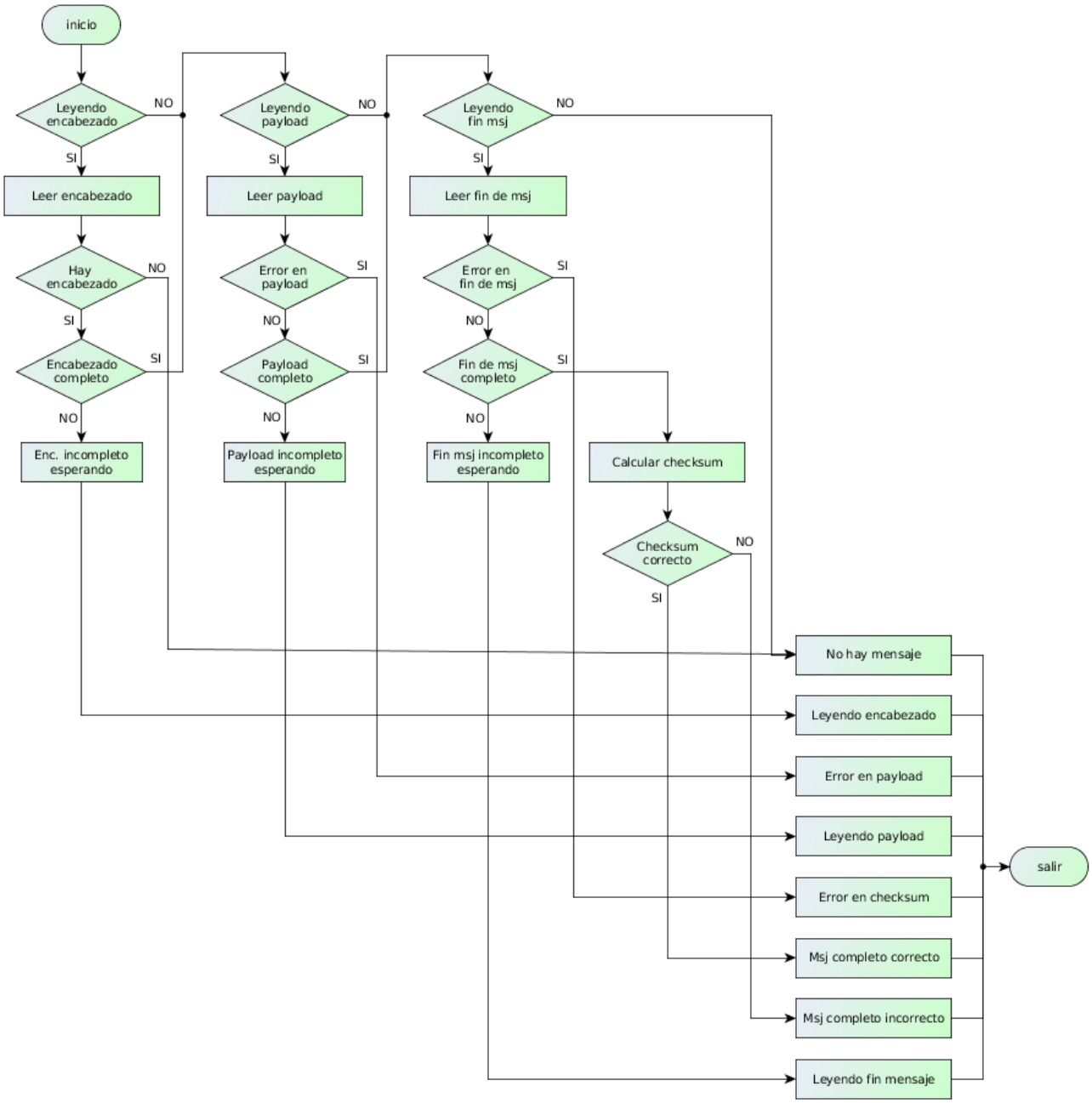

Figura 8: Máquina de estado para la lectura de mensajes del protocolo ADSLink

El procesamiento de las diferentes partes del mensaje (encabezado, payload y fin de mensaje) está encapsulado por medio de tres funciones especializadas que operan sobre los bytes almacenados en el buffer circular. A continuación se muestran los prototipos de estas funciones.

Prototipo de la función de lectura de encabezado de mensaje.

```
uint8_t ReadMsgHeader(tCircularBuffer8* buffer,
                      tMessageFields* msgFields,
                      tMessageIOManager* manager);
```

Prototipo de la función de lectura de datos de mensaje (payload).

```
uint8_t ReadMsgData(tCircularBuffer8* buffer,
                    const tMessageFields* msgFields,
                    uint8_t* dataBuffer,
                    const uint8_t dataBufferMaxSize,
                    tMessageIOManager* manager);
```

Prototipo de la función de lectura de fin de mensaje.

```
uint8_t ReadMsgTail(tCircularBuffer8* buffer,
                    tMessageFields* msgFields);
```

Estas funciones devuelven un entero sin signo de 8bit que representa el estado de la operación de lectura sobre el buffer. Los valores posibles se encuentran definidos mediante las siguientes macros. La máquina de estado de lectura de mensajes toma las decisiones según estos últimos.

```
// Estados de lectura / escritura de partes del mensaje
// Encabezado
#define ADSLINK_MSG_HDR_NOHDR    0U   // No se detecto el byte de comienzo de mensaje
#define ADSLINK_MSG_HDR_COMPLETE 1U   // Se leyo completamente el encabezamiento del mensaje
#define ADSLINK_MSG_HDR_WAITING  2U   // Se leyó el byte de comienzo de mensaje. Esperando a
                                      // que se complete el encabezamiento
// Payload
#define ADSLINK_MSG_DATA_ERROR    0U  // Error interno al intentar leer los datos del payload
#define ADSLINK_MSG_DATA_COMPLETE 1U  // Se leyeron todos los bytes del payload
#define ADSLINK_MSG_DATA_WAITING  2U  // Esperado a que se complete la cantidad de bytes
                                      // necesaria del payload
// Tail
#define ADSLINK_MSG_TAIL_ERROR    0U  // Error interno al intentar leer los bytes de la
                                      // terminación del mensaje
#define ADSLINK_MSG_TAIL_COMPLETE 1U  // Se leyeron todos los bytes de la terminación del
                                          // mensaje
#define ADSLINK_MSG_TAIL_WAITING  2U  // Esperado a que se complete la cantidad de bytes
                                      // necesaria de la terminación del mensaje
```

2.3.3.2 Escritura de mensajes

La operación de escritura de mensajes es mucho más sencilla ya que se construye el encabezado del mensaje, luego se copian los bytes almacenados en el vector de payload y finalmente se calcula el checksum del mensaje y se inserta al final del mismo.

Al igual que para la lectura, la escritura, se lleva a cabo por medio de una función que recibe los mismos argumentos que la función de lectura: el nodo de comunicación, un identificador de mensajes, dirección de memoria al primer elemento del vector para leer el payload y el tamaño del vector de payload.

```
uint8_t ADSLinkWriteMessage(tAdsLink* link,
         tMessageFields* msg,
         const uint8_t* payload,
         const uint8_t payloadSize);
```

En esta función también está previsto el retorno del estado del escritor de mensajes mediante un entero sin signo de 8bit. Para la implementación actual, y como la función es muy sencilla, hay solamente un valor del estado de escritura definido por la siguiente macro.

```
// Mensajes del escritor de mensajes del protocolo
#define ADSLINK_MSG_WRITER_MSGOK 1U          // Se escribió correctamente el mensaje
```

Al igual que para la lectura de mensajes, la escritura se realiza mediante tres funciones específicas: una para el encabezado, otra para el payload y una tercera para el fin de mensaje. A continuación se muestran los prototipos de estas funciones.

Prototipo de la función de escritura de encabezado de mensaje.

```
void WriteMsgHeader(tCircularBuffer8* buffer,
                    tMessageFields* msgFields);
```

Prototipo de la función de escritura de datos de mensaje.

```
void WriteMsgData(tCircularBuffer8* buffer,
                  const uint8_t* dataBuffer,
                  const uint8_t& lengh);
```

Prototipo de la función de escritura de fin de mensaje.

```
void WriteMsgTail(tCircularBuffer8* buffer,
                  tMessageFields* msgFields);
```

2.3.3.3 Funciones auxiliares de cálculo de checksum

A continuación se listan las dos funciones que se utilizan para calcular el checksum del mensaje (encabezado y payload). Las mismas se utilizan tanto para leer como para escribir los mensajes.

El cálculo del checksum del encabezado es la suma de los valores de los campos de la estructura de datos tMessageFields; la función recibe como argumento un puntero de memoria a la estructura de datos de encabezado de mensaje a utilizar para calcular el checksum y devuelve un entero sin signo de 16bit con la sumatoria de los valores.

```
uint16_t CalculateMsgFieldsChecksum(tMessageFields* msgFields){
 uint16_t checksum = 0U;
 checksum += msgFields->msgHeader;
 checksum += msgFields->msgSize;
 checksum += msgFields->systemID;
 checksum += msgFields->componentID;
 checksum += msgFields->msgID;
 checksum += msgFields->msgError;
 return checksum;
}
```

El cálculo del checksum del payload es la suma de los elementos del vector de datos hasta el índice que corresponda con el tamaño que tenga el mensaje particular; la función recibe como argumentos la ubicación en memoria del primer elemento del vector de datos y la cantidad de elementos a sumar y devuelve un entero sin signo de 16bit con la sumatoria de los valores.

```
uint16_t CalculateDataChecksum(const uint8_t* dataBuffer, const uint8_t& lengh){
 uint8_t i;
 uint16_t checksum = 0U;
 for( i = 0 ; i < lengh ; i++){
        checksum += *(dataBuffer + i);
 }
 return checksum;
}
```

2.3.4 Implementación de un nodo de comunicación usando protocolo ADSLink

Para poder utilizar el protocolo de comunicación se implementó una estructura de datos que representa un nodo de comunicación entre el sistema que administra el nodo y el resto de los sistemas que estén conectados al nodo a través de un dispositivo de hardware (ver Figura 9). El nodo no tiene conocimiento del tipo de hardware que se está usando para enviar los datos, solamente administra los buffers circulares para el envío y recepción de mensajes. Estos últimos se deben conectar

posteriormente a un dispositivo de hardware específico de acuerdo a la disponibilidad de cada sistema particular.

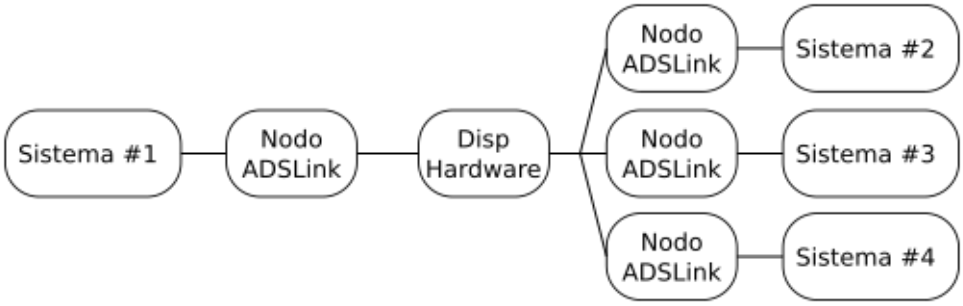

Figura 9: Esquema de una red de comunicación usando nodos ADSLink

El nodo encapsula cuatro elementos principales: el estado del lector de mensajes, los buffers circulares de envío y recepción de mensajes para el nodo y un administrador para control del envío y recepción de mensajes en el nodo.

```
typedef struct{
 uint8_t            msgReaderState;   // Estado del lector de mensajes del link
 tCircularBuffer8*  sendBuffer;       // Buffer circular para la escritura de mensajes
 tCircularBuffer8*  receiveBuffer;    // Buffer circular para la lectura de mensajes
 tMessageIOManager* msgIOManager;     // Administrador para la I/O de mensajes en el link
} tAdsLink;
```

El administrador para las operaciones de entrada / salida de mensajes tiene como finalidad almacenar el estado de lectura / escritura del nodo al que pertenece, permitiendo la existencia de varios nodos de comunicación en el mismo sistema (esto es similar a una creación de varias instancias de un mismo objeto usando Programación Orientada a Objetos, pero en este caso usando lenguaje C puro). En la etapa actual de desarrollo del protocolo, el administrador sólo tiene implementadas el seguimiento de las operaciones de lectura, siendo posible agregar luego seguimiento de operaciones de escritura.

```
typedef struct {
 uint8_t headerReaderState;    // Byte para manejar el estado del lector de encabezados de
                               // mensaje.
 uint8_t dataLeftToRead;       // Cantidad de bytes que quedan por leer de los datos del
                               // mensaje.
} tMessageIOManager;
```

Se implementó una función de inicialización para el administrador de mensajes en donde se inicia el estado del lector de encabezado en “Inicio de lectura de encabezado” y la cantidad de datos que faltan leer en cero. El argumento que se pasa a la función es la instancia del administrador de mensajes que se quiere inicializar.

```
void Init_MessageIOManager(tMessageIOManager* manager){
  manager->headerReaderState    = (uint8_t)HREADER_STATE_BEGIN;
  manager->dataLeftToRead       = 0U;
}
```

La inicialización del nodo de comunicación se realiza mediante una función en donde se pasa como argumentos punteros de memoria a los siguientes elementos: estructura de datos del nodo (tAdsLink), administrador de mensajes (tMessageIOManager), buffer circulares para envío y recepción de datos. Esta función se encarga de vincular los elementos que componen el nodo y dejarlos listos para poder enviar y recibir mensajes a través del mismo.

```
void ADSLinkInit(tAdsLink* link,
         tMessageIOManager* manager,
         tCircularBuffer8* sendBuffer,
         tCircularBuffer8* receiveBuffer){

  link->msgIOManager    = manager;
  link->sendBuffer      = sendBuffer;
  link->receiveBuffer   = receiveBuffer;
  link->msgReaderState  = (uint8_t)ADS_READER_STATE_HEADER;

  // Iniciar el administrador de I/O de mensajes
  Init_MessageIOManager(link->msgIOManager);

  // Limpiar los buffers de entrada / salida
  CBFlush8(link->sendBuffer);
  CBFlush8(link->receiveBuffer);
}
```

2.3.5 Organización de archivos para el protocolo

A continuación se listan los archivos de encabezado (.h) y de código funete (.cpp) que forman el protocolo de comunicación ADSLink (ver Tabla 3). Este conjunto básico de archivos permite generar un nodo de comunicación para enviar y recibir mensajes por el mismo. Sobre esta implementación básica hay que crear las librerías específicas para conectar el nodo de comunicación con los dispositivos de hardware específicos a utilizar.

Archivo	Descripción
adsLinkTypes.h	Definición de las estructuras de datos tMessageFields y tMessageID. En este archivo se pueden incorporar nuevos tipos específicos al mensaje del protocolo.
adsLinkProtocol.h	Archivo de encabezado del protocolo. Se define la estructura para el nodo de comunicación tAdsLink, la inicialización para la misma y las operaciones de lectura / escritura para los mensajes que se quieran enviar. También se definen los posibles estados de lectura y escritura del sistema.
adsLinkProtocol.cpp	Archivo de código fuente con las implementaciones para la inicialización del nodo de comunicación y las operaciones de lectura / escritura de mensajes.
messageIO.h	Archivo de encabezado con la declaración de las funciones particulares de lectura y escritura de cada parte del mensaje y cálculo de checksum. También aquí se encuentra definido el tipo del administrador de mensajes (tMessageIOManager) y la declaración para la inicialización del mismo.
messageIO.cpp	Archivo de código fuente con las implementaciones de las funciones de lectura / escritura específicas para cada parte del mensaje.

Tabla 3: Descripción de los archivos que forman la estructura básica del protocolo ADSLink

Archivo	Descripción
cbDefines.h	Archivo de encabezado para las definiciones específicas de los buffers circulares. En este archivo se encuentran definidos los tamaños en base 2 para los buffers circulares.
CircularBuffer8.h	Archivo de encabezado para el manejo de buffer circulares con datos de 8bit. Se define la estructura para el buffer circular de 8bit. Además se encuentran definidas la inicialización y las operaciones de lectura / escritura.
CircularBuffer8.c	Archivo de código fuente con las implementaciones para la inicialización y las operaciones de lectura / escritura del buffer circular de 8bit.

Tabla 4: Descripción de los archivos que forman el buffer circular de 8bit

2.4 Pruebas del protocolo en un microcontrolador de 8bit

Se realizaron algunas pruebas de una implementación en un microcontrolador de 8bit de la estructura básica del protocolo ADSLink. Se utilizó el microcontrolador ATmega 2560 [8] en el que está basado el autopiloto que se está usando para hacer las pruebas del paracaídas autónomo guiado. El algoritmo de prueba se mide el tiempo de ejecución de las funciones del protocolo usando una función interna de la librería del microcontrolador que devuelve la cantidad de microsegundos que han transcurrido desde el encendido del mismo.

En la Tabla 5 se muestra el tamaño que ocupan las estructuras de datos del protocolo en la memoria del microcontrolador. A partir de esta tabla se calcula el tamaño que ocuparía un nodo de comunicación operativo en la memoria del microcontrolador. El nodo requiere de los siguientes elementos:

- 1 estructura tAdsLink
- 2 estructuras tCircularBuffer8
- 1 estructura tMessageIOManager
- 3 vectores de enteros sin signo de 8bit

La capacidad de los vectores está sujeta a la cantidad de mensajes que se quiera acumular antes de enviarlos, la velocidad de actualización de las operaciones de lectura y escritura y la memoria disponible en el microcontrolador. Se asume un tamaño máximo de 128 bytes para los buffer circulares y un paquete de datos de 40 bytes (10 valores de 4 bytes cada uno) máximo para los mensajes. Con estos valores se obtiene que la implementación del nodo de comunicación requerirá 317 bytes de memoria del microcontrolador.

Estructura	Tamaño [bytes]
tMessageFields	8
tMessageID	5
tCircularBuffer8	6
tMessageIOManager	2
tAdsLink	7

Tabla 5: Tamaño en memoria de las estructuras de datos del protocolo ADSLink - Microcontrolador ATMega 2560

También se realizaron algunas mediciones de duración de las operaciones relacionadas con la lectura y escritura de mensajes del protocolo. En la Tabla 6 se muestran tiempos promedios calculados para las operaciones de lectura y escritura básicas de las partes del mensaje que no dependen de la cantidad de datos que tiene el payload (encabezado, fin de mensaje y checksum). Los tiempos son valores medios calculados en base a un ciclo de ejecución de las funciones de 50 veces. Para el caso particular de las funciones de lectura el resultado que se muestra es para el caso en que el encabezado está completo y correcto, ya que la función contempla diferentes posibilidades dependiendo el estado de los datos disponibles en el buffer circular.

Función	Tiempo medio [µs]
WriteMsgHeader()	28
WriteMsgTail()	14
ReadMsgHeader()	44
ReadMsgTail()	16
Calcular checksum (encabezado + payload)	27

Tabla 6: Tiempos de ejecución de funciones que no dependen del tamaño del paquete de datos - Protocolo ADSLink - Microcontrolador ATMega 2560

En la Tabla 7 se muestran los tiempos de ejecución de las funciones de lectura y escritura del paquete de datos en función de diferentes cantidad de datos. Al igual que antes, las funciones de lectura, se probaron para el caso en que el payload está completo y listo para ser leído. Se deberían realizar pruebas de tiempo para otros casos de ejecución cuando los paquetes de datos están incompletos. Los valores calculados representan el valor medio obtenido de 50 ejecuciones de las funciones.

Payload size [bytes]	**Tiempo medio [µs]**	
	ReadMsgData(...)	WriteMsgData(...)
4	22	28
8	44	46
12	61	64
16	72	89
20	94	106
24	110	124
28	132	143
32	147	167
36	165	186
40	181	204

Tabla 7: Tiempos de ejecución de funciones de lectura y escritura del paquete de datos - Protocolo ADSLink - Microcontrolador ATMega 2560

Por último, se midieron los tiempos de ejecución de las funciones para leer y escribir mensajes completos. Los resultados se muestran en la Tabla 8 para diferentes tamaños del paquete de datos. En el rango de tamaños analizado el crecimiento del tiempo de ejecución es lineal con la cantidad de datos que tiene el mensaje (ver Figura 10). La función de lectura se analizó para el caso de 1 mensaje que se encuentran completo y correcto. Se deberían estudiar los tiempos de demora para otros casos en que los datos del mensaje estén disponibles por partes.

Payload size [bytes]	Tiempo medio [µs]	
	ADSLinkReadMessage(...)	ADSLinkWriteMessage(...)
4	102	81
8	119	102
12	140	122
16	161	142
20	178	168
24	198	191
28	224	209
32	248	231
36	264	257
40	277	277

Tabla 8: Tiempos de ejecución de funciones de lectura y escritura de mensajes - Protocolo ADSLink - Microcontrolador ATMega 2560

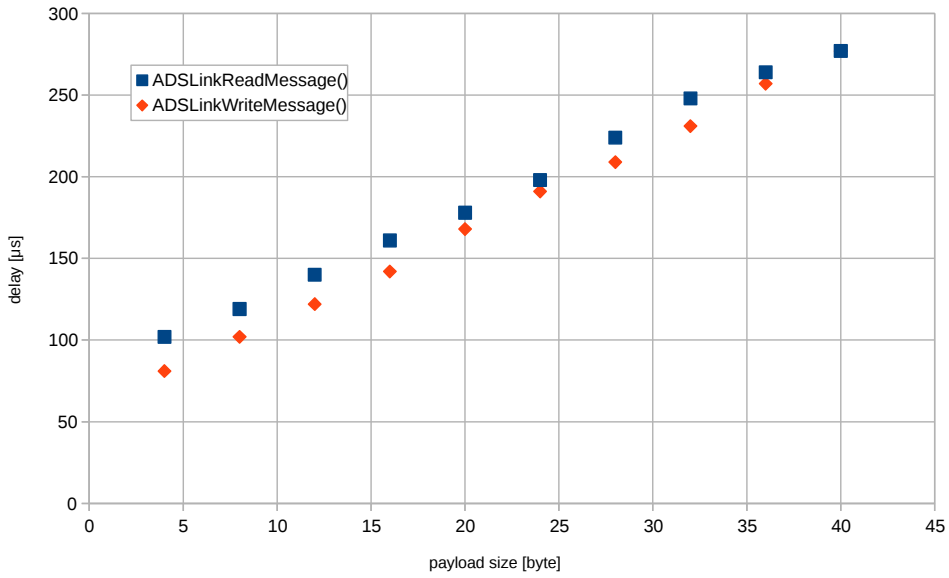

Figura 10: Tiempo de ejecución de las funciones de I/O del protocolo ADSLink – Microcontrolador ATMega 2560

Como referencia para estos tiempos de ejecución se calcularon qué porcentaje del tiempo total de un ciclo de un programa ocuparían para diferentes velocidades de ejecución de este último (ver Tabla 9). Estos valores no representan tiempos finales de ejecución, ya que para obtener el tiempo total que se emplea en enviar o recibir un mensaje hay que sumarle, a estos valores, el tiempo en copiar los datos desde y hacia los buffers de recepción y envío del hardware que se vaya a utilizar para realizar la

comunicación. En el caso de llamar tanto la función de lectura como la de escritura en el mismo ciclo de ejecución del programa, los porcentajes de ocupación de la Tabla 9 se duplican para el caso estudiado de un paquete de datos de 40 bytes.

Velocidad loop principal [Hz]	Intervalo de duración del loop Δt [ms]	Porcentaje del tiempo que ocupa [%]
10	100	0,27
20	50	0,55
50	20	1,4
100	10	2,8
200	5	5,5

Tabla 9: Porcentaje de ocupación del tiempo de un ciclo de ejecución de un programa - Función ADSLinkReadMessage(...) - Payload: 40 [bytes]

3. CONCLUSIONES

Se implementó un protocolo de comunicación con mensajes estructurados inspirado en el protocolo MAVLink. La finalidad del protocolo es poder utilizarlo en un microcontrolador de 8bit que dispone de recursos limitados (memoria y capacidad de cálculo).

El protocolo funciona mediante un nodo de comunicación que permite conectar diferentes sistemas a través de un dispositivo de hardware único; el envío y recepción de mensajes se realiza mediante dos buffer circulares en donde se leen y escriben los datos codificados de los mensajes. Se provee un control de integridad mínimo en los mensajes por medio de la transmisión de un checksum en los datos de cada mensaje. De esta manera es posible determinar si el mensaje recibido está corrupto ó no. También la lectura de mensajes es robusta para la recepción de mensajes por partes; en tales casos, se espera a que se completen los datos del mensaje sin bloquear la ejecución del programa.

La implementación actual ocupa 317 bytes de memoria el microcontrolador ATMega 2560, y demora 277 [μs] tanto para lectura como para la escritura de un mensaje completo que contiene un paquete de datos de 40 [bytes] (10 parámetros de 4 bytes cada uno). Este tiempo de ejecución representa el 2,8 [%] del tiempo total disponible en un ciclo de programa que se ejecute a 100 [Hz].

4. REFERENCIAS

[1] QGroundControl. *MAVLink Micro Air Vehicle Communication Protocol* [en línea]. [s/d: s/d], s/d., s/d. <http://qgroundcontrol.org/mavlink/start> [Consulta: 14 de Junio de 2017]

[2] QGroundControl. Waypoint *Protocol* [en línea]. [s/d: s/d], s/d., s/d. <http://qgroundcontrol.org/mavlink/waypoint_protocol> [Consulta 14 de Junio de 2017]

[3] Wikipedia. *FIFO (computing and electronics)* [en línea]. [USA: Wikimedia Foundation, Inc.], s/d., 1 de Abril de 2017. <https://en.wikipedia.org/wiki/FIFO_(computing_and_electronics)> [Consulta: 26 de Junio de 2017]

[4] White, E. *"Making Embedded Systems - Design Patterns for Great Software"*. Editado por Andy Oram y Mike Hendrickson. 1ra ed. California, USA: O'Reilly Media, 2012. 371 p. ISBN 978-1-449-30214-6. Cap. 6, Circular Buffers, p. 193–198.

[5] Wikipedia. *Object-oriented programming* [en línea]. [USA: Wikimedia Foundation, Inc.], s/d., 27 de Junio de 2017.<https://en.wikipedia.org/wiki/Object-oriented_programming> [Consulta: 27 de Junio de 2017]

[6] CplusPlus.com - The C++ Resources Network . *Vector – C++ Reference* [en línea]. [s/d.: s/d], s/d. s/d. <http://www.cplusplus.com/reference/vector/vector/> [Consulta: 20 de Julio de 2017]

[7] White, E. *"Making Embedded Systems - Design Patterns for Great Software"*. Editado por Andy Oram y Mike Hendrickson. 1ra ed. California, USA: O'Reilly Media, 2012. 371 p. ISBN 978-1-449-30214-6. Cap. 6, Circular Buffers – Representing signed numbres, p. 195-196.

[8] Microchip. *Atmega 2560 – Microcontrollers and Processors* [en línea]. [USA: Microchip Technology Inc.], s/d, s/d. <https://www.microchip.com/wwwproducts/en/ATmega2560> [Consulta: 24 de Julio de 2017]

ANEXO A

LISTADO DE CÓDIGO FUENTE DE LAS FUNCIONES DE LECTURA Y ESCRITURA DE MENSAJES DEL PROTOCOLO ADSLINK

Listado de la función de lectura de mensaje para el protocolo ADSLink.

```
uint8_t ADSLinkReadMessage(tAdsLink* link,
                        tMessageFields* msg,
                        uint8_t* payload,
                        const uint8_t payloadMaxSize)
{
 uint8_t result = (uint8_t)ADSLINK_MSG_READER_NOMSG;
 uint8_t parserResult = 0U;                          // Variable para leer el resultado del
                                                     // parseo de cada parte del mensaje

 if( link->msgReaderState == (uint8_t)ADS_READER_STATE_HEADER ){
        // Tratar de leer el encabezado completo del mensaje.
        parserResult = ReadMsgHeader(link->receiveBuffer,msg,link->msgIOManager);

        switch (parserResult) {
        case ADSLINK_MSG_HDR_NOHDR:
                // No se encontró el comienzo de mensaje en el buffer. Volver inmediatamente.
                // Seguir buscando el encabezado en la próxima ejecución.
                link->msgReaderState = (uint8_t)ADS_READER_STATE_HEADER;
                return (uint8_t)ADSLINK_MSG_READER_NOMSG;          // Salir inmediatamente.
                break;

        case ADSLINK_MSG_HDR_COMPLETE:
                // Se leyó el encabezado completo del mensaje.
                // Tratar de seguir leyendo el payload.
                link->msgReaderState = (uint8_t)ADS_READER_STATE_DATA;
                // Continuar parseando el mensaje.
                result = (uint8_t)ADSLINK_MSG_READER_RDDAT;
                break;

        case ADSLINK_MSG_HDR_WAITING:
                // Se encontró parte del encabezado del mensaje, pero está incompleto.
                // Completar el encabezado en la próxima ejecución.
                link->msgReaderState = (uint8_t)ADS_READER_STATE_HEADER;
                return (uint8_t)ADSLINK_MSG_READER_RDHDR;          // Salir inmediatamente.
                break;

        default:
                // Uppss! No deberíamos estar acá. Indicar que no se encontró un mensaje e
                // intentar leer uno en la próxima ejecución.
                // Seguir buscando el encabezado en la próxima ejecución.
                link->msgReaderState = (uint8_t)ADS_READER_STATE_HEADER;
                return (uint8_t)ADSLINK_MSG_READER_NOMSG;          // Salir inmediatamente.
                break;
        }
 }

 // Si salimos del if() anterior tratar de seguir leyendo el mensaje ya que hay bytes
 // disponibles en el buffer.
 if( link->msgReaderState == (uint8_t)ADS_READER_STATE_DATA ){

        parserResult = ReadMsgData(link->receiveBuffer,msg,payload,payloadMaxSize,link-
>msgIOManager);

        switch (parserResult) {
                case ADSLINK_MSG_DATA_ERROR:
                        // Se produjo un error interno al leer el payload.
                        // Reiniciar el parseador de mensaje en la próxima ejecución.
                        link->msgReaderState = (uint8_t)ADS_READER_STATE_HEADER;
                        return (uint8_t)ADSLINK_MSG_READER_BADDAT;  // Salir inmediatamente.
                        break;

                case ADSLINK_MSG_DATA_COMPLETE:
```

```
                        // Se leyó completamente el payload del mensaje.
                        // Tratar de seguir leyendo el checksum.
                        link->msgReaderState = (uint8_t)ADS_READER_STATE_CHKSUM;
                        // Continuar parseando el mensaje.
                        result = (uint8_t)ADSLINK_MSG_READER_RDCHK;
                        break;

                case ADSLINK_MSG_DATA_WAITING:
                        // Se encontró parte del payload, pero el mismo está incompleto.
                        // Completar el payload en la próxima ejecución.
                        link->msgReaderState = (uint8_t)ADS_READER_STATE_DATA;
                        return (uint8_t)ADSLINK_MSG_READER_RDDAT;   // Salir inmediatamente.
                        break;

                default:
                        // Uppss! No deberíamos estar acá. Indicar que no se encontró un
                        // mensaje e intentar leer uno en la próxima ejecución.
                        // Buscar un nuevo encabezado en la próxima ejecución.
                        link->msgReaderState = (uint8_t)ADS_READER_STATE_HEADER;
                        return (uint8_t)ADSLINK_MSG_READER_NOMSG;   // Salir inmediatamente.
                        break;
        }
}

uint16_t checksum = 0U;
uint8_t lenght = 0U;

// Si salimos del if() anterior tratar de seguir leyendo el mensaje ya que hay bytes
// disponibles en el buffer.
if( link->msgReaderState == (uint8_t)ADS_READER_STATE_CHKSUM){

        parserResult = ReadMsgTail(link->receiveBuffer,msg);

        switch (parserResult) {
        case ADSLINK_MSG_TAIL_ERROR:
                // Se produjo un error interno al leer el checksum
                // Reiniciar el parseador de mensaje en la próxima ejecución.
                link->msgReaderState = (uint8_t)ADS_READER_STATE_HEADER;
                return (uint8_t)ADSLINK_MSG_READER_BADCHK;          // Salir inmediatamente.

        case ADSLINK_MSG_TAIL_COMPLETE:
                // Se leyó completamente el checksum por lo que el mensaje está completo.
                // Reiniciar el parseador de mensaje para buscar un nuevo mensaje en la
                // siguiente ejecución.
                link->msgReaderState = (uint8_t)ADS_READER_STATE_HEADER;

                // Controlar si coincide el checksum enviado y el del mensaje recibido.
                lenght = msg->msgSize - ADSLINK_MSG_HEADER_SIZE;
                checksum = CalculateMsgFieldsChecksum(msg);
                checksum += CalculateDataChecksum(payload,lenght);

                if( checksum == msg->checksum ){
                        result = (uint8_t)ADSLINK_MSG_READER_MSGOK;
                        return (uint8_t)ADSLINK_MSG_READER_MSGOK;
                }
                else{
                        result = (uint8_t)ADSLINK_MSG_READER_BADCHK;
                        return (uint8_t)ADSLINK_MSG_READER_BADCHK;
                }

                break;

        case ADSLINK_MSG_TAIL_WAITING:
                // Se encontró parte del checksum, el mismo está incompleto.
```

```
                // Completar el checksum en la próxima ejecución.
                link->msgReaderState = (uint8_t)ADS_READER_STATE_CHKSUM;
                return (uint8_t)ADSLINK_MSG_READER_RDCHK;          // Salir inmediatamente.

        default:
                // Uppss! No deberíamos estar acá. Indicar que no se encontró un mensaje e
                // intentar leer uno en la próxima ejecución.
                // Buscar un nuevo encabezado en la próxima ejecución.
                link->msgReaderState = (uint8_t)ADS_READER_STATE_HEADER;
                return (uint8_t)ADSLINK_MSG_READER_NOMSG;          // Salir inmediatamente.
                break;
        }
 }

 return result;
}
```

Listado de la función de escritura de mensaje para el protocolo ADSLink.

```
uint8_t ADSLinkWriteMessage(tAdsLink* link,
        tMessageFields* msg,
        const uint8_t* payload,
        const uint8_t payloadSize)
{

 uint8_t result = (uint8_t)ADSLINK_MSG_WRITER_MSGOK;
 uint16_t checksum = 0U;

 // TODO: implementar alguna lógica para manejar el caso que se produzca algún error
 //       durante la escritura del contenido del mensaje.

 // Terminar de escribir los campos del mensaje.
 msg->msgHeader = ADS_PROTOCOL_MSG_HEADER;
 msg->msgSize = (uint8_t)ADSLINK_MSG_HEADER_SIZE + payloadSize;

 // Calcular el checksum del mensaje
 checksum = CalculateMsgFieldsChecksum(msg);
 checksum += CalculateDataChecksum(payload,payloadSize);

 msg->checksum = checksum;

 WriteMsgHeader(link->sendBuffer,msg);
 WriteMsgData(link->sendBuffer,payload,payloadSize);
 WriteMsgTail(link->sendBuffer,msg);

 return result;
}
```