Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Diseño e Implementación de un simulador del protocolo ARINC 429 sobre UDP

Autor: Fco. Javier Vargas García-Donas
Tutor: Ángel Rodríguez Castaño

Dep. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2013

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Diseño e Implementación de un simulador del protocolo ARINC 429 sobre UDP

Autor:
Fco. Javier Vargas García-Donas

Tutor:
Ángel Rodríguez Castaño

Dep. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2015

Trabajo Fin de Grado: Diseño e Implementación de un simulador del protocolo ARINC 429 sobre UDP

Autor: Fco. Javier Vargas García-Donas

Tutor: Ángel Rodríguez Castaño

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2015

El Secretario del Tribunal

A mi familia

A mis profesores

A mis compañeros

Agradecimientos

A Rafael Bastida, Daniel Rodríguez, Germán Caramel y Alejandro Catalán, por ayudarme a recordar que durante el camino hay tiempo para todo.

Fco Javier Vargas García -Donas

Sevilla, 2015

Resumen

Hoy en día existen numerosos protocolos de comunicación en diferentes ámbitos como las redes de ordenadores, las comunicaciones interna de sistemas embebidos, las comunicaciones industriales, las comunicaciones en la aviónica, entre otros. Cada protocolo puede ubicarse en algún nivel del modelo OSI y puede ser dependiente o independiente del protocolo del nivel inmediatamente inferior que lo soporta. El protocolo de bus de aviónica ARINC 429 especifica tanto la forma de codificar la información como la forma de transmitirla a nivel físico. Dicho protocolo, por tanto, define tanto el nivel físico como el de enlace, niveles 1 y 2 del modelo OSI, y establece una clara dependencia entre ellos.

El objetivo de este Trabajo Fin de Grado es implementar un simulador del protocolo ARINC 429 que logre una independencia entre la codificación de la información y la transmisión a nivel físico en el bus. Ofreciendo de este modo una herramienta que permita profundizar en la codificación de la información de este protocolo con independencia del *hardware* necesario para soportar la capa física.

Abstract

Nowadays there are several communication protocols for different fields such as computer networks, internal communications in embedded systems, industrial communications, avionic communications, among others. Each protocol can be located at any level of the OSI model and can be dependent or independent of the immediately lower level protocol that supports it. The avionics bus protocol ARINC 429 specifies both how to encode information and how to transmit it into the physical level. This protocol therefore defines both the physical and the link layers, levels 1 and 2 of the OSI model, and establishes a clear dependence between them.

The objective of this Final Project is to implement a simulator of the ARINC 429 protocol that achieves independency between encoding information and transmitting it on a physical level on the bus. Thus providing a tool to deepen in the information encoding of this protocol with independency of the needed hardware to support the physical layer.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

Inglés	Palabras en inglés
Ficheros	Nombre de ficheros
`Código`	Texto del código fuente
33 unidad	Unidades de ingeniería
`Comando`	Comandos de la terminal
Ejemplos	Ejemplos

1 INTRODUCCIÓN

La separación de los problemas de comunicación de red en capas funcionales ha modularizado la forma en la que se resuelven. Cuando los problemas a resolver son de bajo nivel, según el modelo OSI (International Telecommunication Union, 1994), existe una dependencia con el medio físico sobre el que se establece la comunicación. Dada la independencia funcional entre las capas, el uso de simuladores es frecuente a la hora de virtualizar una o varias capas, entre ellas, el medio físico.

La especificación ARINC (Aeronautical Radio Inc) 429 define los estándares de *hardware* y protocolos para la transferencia de datos digitales entre los sistemas de aviónica (Woodward, 2002). Por tanto, este estándar vincula tanto la codificación de la información, capa dos del modelo OSI, como la transmisión sobre el medio físico, capa uno del modelo OSI. Esta vinculación condiciona a que la mayoría de simuladores comerciales ofrezcan una solución de adaptación tanto *hardware* como *software*.

Este documento describe una solución para desvincular esta independencia ofreciendo un simulador que opera únicamente a nivel de codificación de datos.

1.1 Antecendentes

Este Trabajo Fin de Grado se desarrolla como una extensión del trabajo final de la asignatura de Redes Industriales, asignatura de cuarto de Grado de Ingeniería en Tecnologías de Telecomunicación.

Este trabajo pretende ser una herramienta para profundizar en la codificación de datos según el estándar ARINC 429. La herramienta propuesta ha sido diseñada y desarrollada desde cero acorde al estándar mencionado.

1.2 Objetivos

El objetivo principal de este Trabajo Fin de Grado es diseñar y desarrollar un simulador de la comunicación en un bus de aviónica según el estándar ARINC 429.

Se persigue profundizar en el tratamiento y la codificación de la información de acuerdo al estándar.

Se pretende independizar la comunicación de la capa física descrita en el estándar simulando un entorno de comunicación entre transmisor ARINC 429 y receptor.

Se persigue que el simulador se adapte en lo posible a los requerimientos de los tiempos de transmisión e

intervalos del estándar cuando sea posible.

Con motivos didácticos se requiere que el simulador provea de una herramienta capaz de mostrar información sobre el proceso de codificación del mensaje con distintos niveles de profundidad.

Buscando un entorno de simulación común se pretende que el simulador sea capaz de establecer una comunicación entre dos ordenadores conectados en red.

Se pretende buscar un diseño modular con posibilidades de extender funcionalidades en un futuro.

Se busca ofrecer interfaces estructuradas y extensibles tanto para los datos de entrada como para la base de datos del simulador.

1.3 Metodología

El simulador se ha proyectado en diferentes etapas, una de diseño, una de desarrollo y una de pruebas. Para ello se ha utilizado el siguiente entorno de desarrollo:

- Sistema Operativo: Ubuntu 14.04.2 LTS[1].
- Kernel: 3.16.
- Orden de bits: *Little-Endian*.
- Entorno de ejecución: Terminal Unix de comandos Bash.
- Compilador: Gcc.
- Procesador de texto: Vim, configurado con las siguientes características:

```
syntax on
set tabstop = 4
set softtabstop = 4
set shiftwidth = 4
set noexpandtab
set smartindent
set number
```

Ejemplo 1-1: Configuración VIM

1.4 Estructura de la memoria

La memoria se estructura en cinco bloques que detallan la motivación del presente trabajo, las especificaciones del mismo, el diseño y la implementación de la solución, las pruebas de validación y las conclusiones finales.

1.4.1 Introducción

La introducción ofrece una perspectiva general del trabajo desarrollado. Presenta la motivación del mismo, declara el origen de la idea, ofrece una perspectiva sobre el entorno de trabajo y resume su estructura.

1.4.2 Descripción del estándar

El segundo capítulo ofrece una visión del estándar ARINC 429 centrada en los aspectos relevantes para este trabajo. Especifica luego las características generales del simulador y por último detalla los aspectos del estándar que debe cumplir.

[1] LTS (Long Term Supported)

1.4.3 Diseño y desarrollo del simulador

Este capítulo detalla el diseño del simulador justificando las decisiones tomadas en el mismo. Describe también la implementación desarrollada detallando cada aspecto para facilitar su comprensión. Por último ofrece instrucciones de puesta en marcha con diferentes características del simulador desarrollado.

1.4.4 Pruebas y validación

Se describen en este capítulo las especificaciones validadas. Se indica el mecanismo de validación y en algunos casos se detallan las pruebas y casos límites a los que se ha sometido el simulador.

1.4.5 Conclusiones y desarrollos futuros

En el último capítulo se razonan unas conclusiones sobre lo aprendido, se describen los principales problemas encontrados y se ofrecen algunas vías de ampliación y mejora.

2 DESCRIPCIÓN DEL ESTÁNDAR ARINC 429 Y ESPECIFICACIÓN DEL SIMULADOR

En este capítulo se introducen los aspectos más relevantes del estándar ARINC 429 para la especificación del simulador. Se detallan los requisitos del simulador, tanto los generales como los referidos al estándar. Se diferenciará entre los aspectos del estándar que el simulador debe cumplir, los que no debe cumplir y los aspectos que debe adaptar.

2.1 Descripción y breve reseña del estándar ARINC 429

ARINC 429 define los estándares de la industria del transporte aéreo para la transferencia de datos digitales entre los elementos de los sistemas de aviónica (ARINC SPECIFICATION 429, PART 1, 2004). La especificación define las características eléctricas, las estructuras de palabras y el protocolo necesario para establecer una comunicación sobre el bus (AIM GmbH Avionic Databus Solutions, 2010).

Aunque ARINC 429 es un vehículo para la transferencia de datos no se adecúa a la definición normal de un bus de datos ya que un bus de datos ofrece transferencia de datos multidireccional entre diferentes puntos sobre un conjunto de cables mientras que ARINC 429 especifica solo flujo de datos en un sentido (Avionics Handbook, 2014).

El estándar ARINC 429 ha sido instalado en muchos aviones comerciales de transporte incluyendo: Airbus A310/A320 y A330/A340; Bell Helicopters; Boeing 727, 737, 747, 757, y 767; y McDonnell Douglas MD-11 (CONDOR Engineering, 2010). Aunque los fabricantes no están bajo la obligación de cumplir con esta especificación, diseñar sistemas de aviónica que cumplan con ARINC 429 asegura la interoperabilidad entre unidades funcionales de diferentes fabricantes (AIM GmbH Avionic Databus Solutions, 2010).

La especificación ARINC 429 se compone de tres partes:

- ARINC Specification 429, PART 1-15: Descripción Funcional, Interfaz Eléctrica, Asignación de Etiquetas y Formatos de Palabras.
- ARINC Specification 429, PART 2-15: Estándares de Palabras de Datos Discretos.
- ARINC Specification 429, PART 3-15: Técnicas de Transferencia de Ficheros.

(AIM GmbH Avionic Databus Solutions, 2010)

En esta memoria se describe los aspectos de la primera parte del estándar que tienen mayor relevancia para la especificación del simulador.

2.1.1 Transmisor y receptor

Las palabras ARINC 429 se transmiten desde una única fuente en el bus hacia los receptores. La especificación especifica un máximo de 20 receptores recibiendo a la vez. El transmisor puede estar transmitiendo periódicamente o en el estado NULL (CONDOR Engineering, 2010).

El orden de transmisión empieza por el bits menos significativo, el bit 1 (ver Figura 2-1) y termina en el más significativo, el bit 32 (ARINC SPECIFICATION 429, PART 1, 2004).

2.1.2 Formato de palabras y asignación de etiquetas

El formato de palabra de ARINC 429 es siempre de 32 bits. Se transmiten datos desde una única fuente en el bus hacia y hasta veinte receptores como máximo. El transmisor está siempre transmitiendo ya sea diferentes palabras o el estado NULL. Un transmisor puede transmitir diferentes palabras en orden (AIM GmbH Avionic Databus Solutions, 2010).

En la Figura 2-1 se representa una palabra ARINC 429 genérica con sus respectivos campos.

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
P	**SSM**	**Datos**																				**SDI**		**Etiqueta**							

Figura 2-1: Palabra ARINC 429

2.1.2.1 Etiqueta (Label)

Los bits 1-8 contienen la Etiqueta del Identificador de Información. La Etiqueta se expresa como 3 dígitos codificados en octal por separado. Se reservan bits 1 y 2 para codificar el primer dígito de la Etiqueta y los 6 restantes para codificar los dos últimos tal y como se expresa en la Tabla 2-1. Con esta estructura solo se puede codificar hasta 255 Etiquetas (AIM GmbH Avionic Databus Solutions, 2010).

Bit	8	7	6	5	4	3	2	1
Cifra	**Cifra3**			**Cifra2**			**Cifra1**	

Tabla 2-1: Campos de la Etiqueta

La Etiqueta determina la codificación, los parámetros necesarios para la misma y los intervalos de transmisión. Se muestra un ejemplo de ello en Tabla 2-2 y en Tabla 2-3.

Etiqueta	ID de Equipo	Nombre Parámetro	Unidades	Escala Rango	Dígitos	+	Resolución	Min Tasa Tx (ms)	Max Tasa Tx (ms)
010	002	Posición Actual - Latitud	Grados – Minutos	180N – 180S	6	N	0.1	250	500
014	004	Rumbo Magnético	Grados	0 – 359.9	4		0.1	250	500

Tabla 2-2: Ejemplo de Etiquetas BCD

Etiqueta	ID de Equipo	Nombre Parámetro	Unidades	Escala Rango	Bits	Resolución	Min Tasa Tx (ms)	Max Tasa Tx (ms)
064	03C	Presión de Neumáticos	Psi	1024	10	1.0	50	250
102	002	Altitud Seleccionada	Pies	65536	16	1.0	100	200

Tabla 2-3: Ejemplo Etiquetas BNR

2.1.2.2 Identificador Origen/Destino (SDI: Source/Destination Identifier)

Su utilización es opcional y puede ser usado para la identificación de la fuente o el destino de la palabra en caso de redundancia en el equipamiento. También puede ser usado para obtener mayor resolución en vez de usarlos como SDI (AIM GmbH Avionic Databus Solutions, 2010).

2.1.2.3 Paridad (P)

Con propósito de detección de errores, no de su corrección, ARINC define el bit 32 como 1 si el número de unos lógicos transmitidos en los bits restantes de la palabra es par. Se configura el bit de paridad a 0 si el número de unos lógicos transmitidos en los bits restantes de la palabra es impar (AIM GmbH Avionic Databus Solutions, 2010).

2.1.2.4 Datos (Data)

Los bits 11 – 29 contienen la información de la palabra ARINC 429. La información se puede codificar de formas diferentes. La forma en que se codifica queda determinado por la Etiqueta tal y como se muestra en Tabla 2-2 y en Tabla 2-3 (AIM GmbH Avionic Databus Solutions, 2010).

Previamente a la codificación del dato se usa un parámetro que determina la Etiqueta llamado Resolución (ver Tabla 2-1 y Tabla 2-2). La Resolución es una constante que divide el Dato antes de que éste sea codificado en el transmisor y que multiplica al Dato antes de ser decodificado en el receptor. El uso de la Resolución permite obtener diferentes niveles de granularidad en la representación de rangos de Datos (Avionics Handbook, 2014).

Los tipos de codificación son:

- Codificación en binario (BNR): la información se codifica en binario y en complemento a 2 de modo que el bit 29 es el más significativo y el que indica si un número es positivo o negativo. Cada Etiqueta además define el número de bits a usar de los 19 disponibles (ver Tabla 2-3) empezando por el más significativo y configurando a 0 los bits de relleno o menos significativos que no se usen. En la Figura 2-2 se representa la distribución del campo de Datos para un número que solo usa 7 bits en su codificación.

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
P	**SSM**		**Dato BNR**							**Relleno**												**SDI**		**Etiqueta**							

Figura 2-2: Formato de palabra ARINC 429 codificando en BNR

- Decimal codificado en binario (BCD): utiliza 4 bits del campo de datos para representar cada dígito decimal con la excepción del dígito más significativo que usa solo 3. Hasta 5 dígitos pueden ser representados. Si son representados 5 dígitos el mayor solo podrá representar un valor máximo de 7. Si el dígito más significativo es mayor que 7, los bits 27-29 se rellenan con ceros y el segundo subcampo se convierte en el dígito más significativo permitiendo 4 valores binarios en lugar de 5 a ser representado, es decir, el número a representar solo puede ser de cuatro cifras. El campo SSM proporciona la información del signo o de dirección como se muestra en la Tabla 2-4.

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
P	**SSM**		**Dígito 1**			**Dígito 2**				**Dígito 3**				**Dígito 4**				**Dígito 5**				**SDI**		**Etiqueta**							

Figura 2-3: Formato de palabra ARINC 429 codificando en BCD

- Datos discretos: los datos discretos pueden ser una mezcla de valores binarios y decimal codificado en binario. Cada bit puede tener un significado propio como Activado/Desactivado y pueden actuar como banderas

2.1.2.5 Matriz de Signo/Estado (SSM: Sign/Status Matrix)

Dependiendo de la Etiqueta y de cómo esté codificada la información puede tomar un significado u otro. Algunos valores toman sentidos de orientación o de signo del Dato como se puede observar en la Tabla 2-4 (AIM GmbH Avionic Databus Solutions, 2010).

Codificación SSM para datos BNR			Codificación SSM para datos BCD		
Bit 30	Bit 31		Bit 30	Bit 31	
0	0	Aviso de Fallo	0	0	Mas, Norte, Sur, Este, Derecha, A, Arriba
0	1	Test de Funcionalidad	0	1	Test de Funcionalidad
1	0	Datos No Procesados	1	0	Datos No Procesados
1	1	Operación Normal	1	1	Menos, Sur, Oeste, Izquierda, De, Abajo

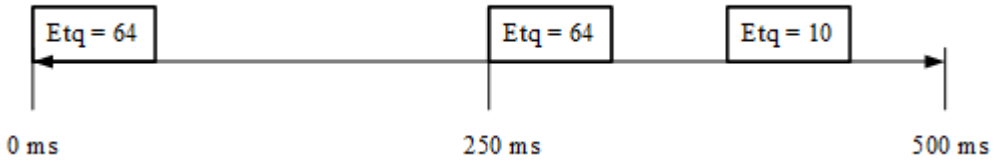
Tabla 2-4: Tabla de valores del campo SSM

2.1.3 Intervalos entre palabras

La palabra digital a transmitir deberá estar sincronizada a un intervalo de 4 veces el tiempo de bits como mínimo. El principio del primer bit transmitido tras este intervalo indica el inicio de una nueva palabra. El tiempo de bit puede ser **100 kbps**, en modo de operación de alta velocidad, o **12 -14.5 kbps**, en baja velocidad (ARINC SPECIFICATION 429, PART 1, 2004).

Normalmente las palabras son enviadas repetidamente. Cada palabra debe ser transmitida en intervalos mayores que su Tasa Mínima de Transmisión e inferiores a su Tasa Máxima de Transmisión. El orden de transmisión de cada palabra debe satisfacer esta condición para cada Etiqueta transmitida (CONDOR Engineering, 2010). La Tasa Mínima y Máxima de Transmisión queda determinada por la Etiqueta (ver Tabla 2-2 y Tabla 2-3).

El cumplimiento de estos requerimientos conlleva el establecimiento de un intervalo periódico en el que cada Etiqueta se transmite con una Tasa de Transmisión entre los intervalos determinados. En la Tabla 2-2 y la Tabla 2-3 se aprecia como la Etiqueta 10 debe transmitirse entre una Tasa de Transmisión Mínima y Máxima de **250 ms** y **500 ms** respectivamente así como la Etiqueta 64 debe transmitirse entre una Tasa de Transmisión Mínima y Máxima de **50 ms** y **250 ms**. Si para la Etiqueta 10 se seleccionara una Tasa de 500 ms y para la Etiqueta 64 una tasa de 250 ms se generaría un intervalo de transmisión periódico como en el del Ejemplo 2-1.



Ejemplo 2-1: Intervalo de transmisión de Etiquetas

2.1.4 Codificación y decodificación

La codificación de los datos en el transmisor y su decodificación en el receptor es posible debido a que ambos comparten la misma base de datos de Etiquetas. Las Etiquetas están regularizadas en el Apéndice 1-1 del estándar.

2.2 Especificación y requisitos del simulador

En este apartado se exponen las especificaciones con las que se ha desarrollado el simulador. En la primera parte se describe las especificaciones propias del simulador. En la segunda parte se detallan que aspectos del estándar ARINC 429 debe cumplir, que aspectos no debe cumplir y que aspectos se han adaptado.

2.2.1 Especificaciones generales

2.2.1.1 Sistema operativo

Las especificaciones son las siguientes:

- Sistema Operativo: El simulador debe poder ser ejecutado en alguna versión del sistema operativo Ubuntu.
- Fichero Makefile: El programa debe ofrecer un fichero Makefile para la compilación del cliente y del servidor.

2.2.1.2 Arquitectura de red y encapsulación

Las especificaciones son las siguientes:

- Arquitectura cliente-servidor: El cliente debe tomar el rol de transmisor ARINC 429 y el servidor de receptor.
- Encapsulación de palabra ARINC 429 en protocolo UDP: Los datos que el simulador debe encapsular en el paquete UDP deben ser justo lo que un transmisor ARINC 429 entregaría a la electrónica de transmisión del bus.
- Red de operación: El simulador debe usar una red IP para la comunicación entre cliente y servidor.
- Simulación: El simulador debe ofrecer independencia entre la capa de enlace y la capa física.

2.2.1.3 Interfaz de entrada y base de datos

Las especificaciones son las siguientes:

- Fichero de entrada con una estructurada que permita la generación de paquetes ARINC 429 a transmitir.
- Base de datos ampliable de Etiquetas ARINC 429 con su codificación.
- Mecanismo de protección ante Etiqueta ARINC 429 no encontrada en base de datos.

2.2.1.4 Herramienta de comprobación

Las especificaciones son las siguientes:

- Que sea capaz de mostrar el paquete enviado en binario y en hexadecimal.
- Que en la representación binaria se separen los diferentes campos de la palabra ARINC 429.
- Que muestre la información leída en la base de datos en comparación con el mensaje codificado.
- Que muestre la codificación de la Etiqueta.

- Que disponga de varias opciones para mostrar diferentes grados de información incluyendo no mostrar nada.

2.2.2 Especificaciones del estándar ARINC 429

2.2.2.1 Elementos relacionados con los mensajes

Las especificaciones son las que siguen:

- Destino de la información: El simulador debe establecer comunicación entre un transmisor y un receptor.
- Elemento de información: El simulador debe usar palabras de 32 bits con sus cinco campos respectivos tal y como se especifica en el estándar.
- Etiqueta: El simulador debe usar la etiqueta codificada en octal con 3 subcampos tal y como se especifica en el estándar.
- SDI: El campo SDI puede ser estático debido a que el estándar lo propone de uso opcional.
- SSM: El simulador debe usar el campo SSM tal y como se especifica en el estándar con la excepción de las opciones de aviso de error y de datos no procesados que podrán ser adaptadas o eliminadas debido a la falta de una capa física que genere estos avisos.
- Datos: El simulador debe implementar la codificación de datos en BCD y en BNR tal y como se especifica en el estándar. El simulador no contemplará la implementación de los datos discretos dado que éstos se codifican de forma diferente en función de la etiqueta y por tanto hay gran variedad de combinaciones.

2.2.2.2 Elementos relacionados con la capa física

El simulador no debe implementar funciones de la capa física, esto incluye: la interconexión física entre transmisor y receptor, la modulación, los niveles de voltaje sobre el bus, la tolerancia a los fallos ni el aislamiento de fallos de la señal.

Dado que el simulador opera solo a nivel de bits, todas las recomendaciones de la especificación que refieran a la capa física no deben ser implementadas.

2.2.2.3 Elementos lógicos del procesado y la transmisión

Las especificaciones son las siguientes:

- Orden de transmisión: El simulador debe implementar el orden de transmisión de los bits tal y como se especifica en el estándar.
- Detección de errores: El simulador debe implementar la detección de errores mediante el bit de paridad tal y como se especifica en el estándar.

2.2.2.4 Elementos relacionados con los intervalos de transmisión

Las especificaciones son las siguientes:

- Tasa de bits: El simulador no tiene porqué respetar la tasa de bit recomendada en el estándar dado que se pide independencia de la capa de enlace y el medio físico.
- Sincronización de reloj: El simulador no debe implementar la sincronización de reloj ya que esto es un requisito de la capa física.
- Sincronización de palabras: El simulador debe respetar como mínimo 4 veces el tiempo de bit entre transmisión de palabras. El tiempo de bit será de **33 ms** que corresponde a la operación a baja velocidad.

3 DISEÑO Y DESARROLLO DEL SIMULADOR

Este capítulo se desarrolla en tres bloques: el diseño del simulador, su implementación y las instrucciones para su ejecución. En el primer bloque se describen y se justifican las decisiones tomadas para el diseño del simulador acorde a las especificaciones dadas. En el segundo se detalla el funcionamiento del simulador, sus partes funcionales y las limitaciones de la implementación. En el tercer bloque se describen los aspectos necesarios para la ejecución del simulador.

3.1 Diseño

3.1.1 Entorno de programación

La herramienta de simulación ha sido desarrollada en el Sistema Operativo Ubuntu 14.04.2 por su propiedad de soporte extendido que asegura que esta versión de Ubuntu seguirá existiendo y tendrá soporte hasta el 2019 (xnox, 2013). Esta característica asegura tener una versión estable donde probar el simulador hasta el 2019.

Se ha previsto, acorde a la especificación, que el simulador se desarrolle con un fichero **Makefile** para independizar su funcionamiento con el procesador de la máquina.

3.1.2 Arquitectura Cliente-Servidor

La arquitectura de la comunicación se ha diseñado en función de las especificaciones dadas. Se ha previsto que tanto el cliente como el servidor operen con *sockets* UDP para establecer la comunicación.

3.1.2.1 El Servidor

El servidor UDP se ha diseñado con vistas a que pueda escuchar los paquetes del cliente en todas las direcciones IP que tenga el equipo donde se ejecute. Se ha previsto también que el puerto donde escucha sea configurable para evitar conflictos con los puertos en uso del equipo.

Se ha determinado que le servidor esté escuchando hasta que se detenga con una señal de interrupción del sistema. Esto es así porque facilita el uso del simulador cuando se está cambiando el fichero de entrada, analizando los datos enviados, realizando pruebas, entre otros.

3.1.2.2 El Cliente

El cliente UDP se ha diseñado con el objetivo de enviar un paquete UDP por cada mensaje ARINC 429. La dirección de destino del servidor y el puerto donde escucha se ha previsto que sean configurables.

Se ha previsto que el cliente imprima mensajes informativos de error cuando no se pueda establecer la comunicación con el servidor.

3.1.3 Cliente o transmisor ARINC 429

El diseño del transmisor ARINC 429 se ha divido en 5 bloques o estados de operación. Los estados se suceden en bucle, como se aprecia en la Figura 3-1, simulando la operación habitual de un transmisor ARINC 429 real (AIM GmbH Avionic Databus Solutions, 2010). El bucle no es infinito, a diferencia de un transmisor real, su condición de parada es haber procesado y enviado todos los datos de entrada. Se ha diseñado de este modo para que el cliente tenga un fin programado.

Los cinco bloques son:

1. Lectura de datos de entrada: Se leen los datos a transmitir de un fichero de entrada.
2. Búsqueda de etiqueta en base de datos: Se busca la Etiqueta de los datos de entrada para determinar la forma en la que se codifica la información en el paquete ARINC 429.
3. Codificación del mensaje: Se procesa la información y se genera el mensaje ARINC 429 acorde a la codificación determinada.
4. Encapsular mensaje en paquete UDP: Se inserta el mensaje ARINC 429 codificado en un *buffer* de transmisión. Este *buffer* determina los datos que se envían en el paquete UDP hacia el transmisor.
5. Espera a la transmisión del mensaje: Se espera como mínimo 4 veces el tiempo de bit entre mensajes tal y como indica el estándar.

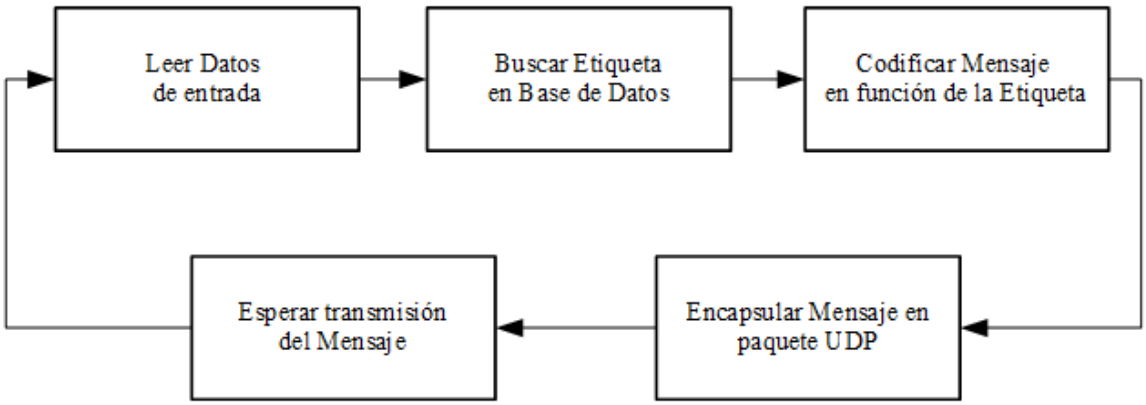


Figura 3-1: Bucle de estados del transmisor

3.1.3.1 Lectura de datos de entrada

Para la lectura de los datos de entrada se ha buscado estandarizar la interfaz con la que se obtienen. El objetivo es poder usar el *parser* por defecto del sistema operativo. Para ello se ha propuesto un fichero de texto estructurado.

Se ha fijado un nombre estático para el fichero de entrada: **input-file.txt**. Se ha determinado que el fichero esté estructurado de la siguiente forma: *Etiqueta Dato*. Entre la Etiqueta y el Dato debe haber un espacio y detrás del Dato un salto de línea. La Etiqueta debe corresponder a una etiqueta ARINC 429 válida y el Dato debe estar expresado en magnitudes de ingeniería. Cada línea determina un mensaje ARINC 429 como se muestra en el Ejemplo 3-1.

```
102 100

206 145
```

Ejemplo 3-1: Fichero input-file.txt

Se ha determinado que se lea una línea, si existe, cada vez que se entre en este primer bloque funcional.

3.1.3.2 Base de Datos de Etiquetas

Para poder codificar el mensaje el simulador debe saber cómo se codifica el mensaje para cada Etiqueta. Para ello se ha determinado que el simulador busque en una base de datos local la relación entre cada Etiqueta y su codificación.

Como base de datos se ha propuesto un fichero de texto estructurado en el que cada línea especifica cómo se codifica una Etiqueta determinada. Se ha elegido un fichero de texto como base de datos para facilitar la ampliación o modificación de las palabras ARINC 429 que contenga.

Se ha determinado que el fichero tenga un nombre estático: **db.csv**. Se ha tomado **.csv** como extensión del fichero para que sea fácilmente ampliable con *software* ofimático. Cada línea contiene 7 campos separados por el carácter *";"* entre sí, tras el último campo debe encontrarse el carácter del salto de línea. Los campos de cada línea, en orden, son:

1. Etiqueta: Indica que la información sobre la codificación de la línea pertenece a esta Etiqueta.
2. SDI: Se ha determinado que el campo SDI se configure en la base de datos y que para cada etiqueta tenga uno estático. Aunque según el estándar ARINC 429 puede haber varios SDI por etiqueta su uso es opcional por parte del programador de la aplicación ARINC 429.
3. Valor máximo: Valor máximo para el dato del fichero de entrada.
4. Valor mínimo: Valor mínimo para el dato del fichero de entrada.
5. Numero de bits a usar: Tomando sentido solo para codificación en BNR indica la cantidad de bits del campo de datos a usar para representar el número dado.
6. Resolución: Constante usada para representar rangos de datos con distinta granularidad.
7. Codificación: Determina si el dato se debe codificar como BRN o BCD.

En el Ejemplo 3-2 se muestran dos líneas de un posible fichero **db.csv**.

```
102;1;65536;0;16;0.5;0

10;0;180;-180;6;1.0;1
```

Ejemplo 3-2: Fichero db.csv

3.1.3.3 Codificación del mensaje

La codificación del mensaje se determina con los datos leídos de la base de datos. Se han propuesto dos funciones, una para cada tipo de codificación, que toman como entrada la Etiqueta, los Datos a codificar e información auxiliar de codificación y devuelven mensaje ARINC 429 completo y codificado.

Cada función es independiente y cada paquete puede ser codificado de una sola forma a la vez.

3.1.3.4 Encapsulación del mensaje

El objetivo de este estado es adecuar la información binaria a un tipo de dato soportado para la transmisión de los datos sobre el *socket* UDP. Una vez adecuado se inserta el mensaje en el *buffer* de transmisión.

Se ha previsto que el *buffer* de transmisión tenga 32 bits ajustándose a los mensajes ARINC 429. Los datos del paquete UDP por tanto solo contienen 32 bits que corresponden a la palabra ARINC 429 codificada tal y como se muestra en la Figura 3-2.

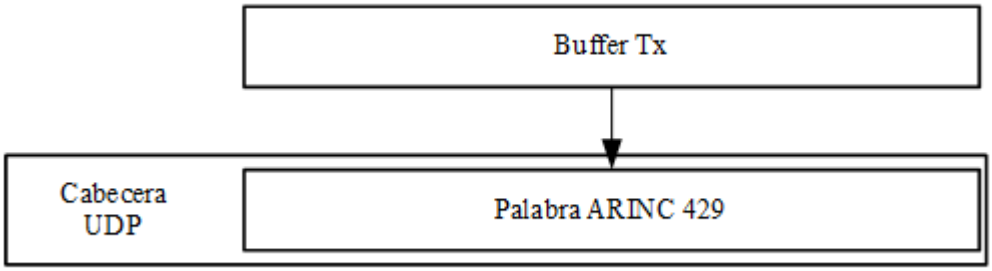


Figura 3-2: Encapsulación en paquete UDP

3.1.3.5 Espera a la transmisión del mensaje

Según el estándar debe haber un intervalo mínimo de al menos 4 veces el tiempo de bit. Aunque por especificación no se tenga acceso al medio físico se ha decidido usar las interrupciones del sistema operativo como herramienta para facilitar el control de tiempos entre mensajes.

Se ha previsto usar una interrupción de tiempo. Esta interrupción se produce cuando expira un temporizador. El temporizador se configura previamente con un tiempo mínimo de al menos 4 veces el tiempo de bit según el estándar. Se ha determinado desarrollar un método que permita configurar dinámicamente este temporizador con el objetivo de poder establecer un orden temporal para la transmisión de cada paquete.

Cuando expira el temporizador, la función manejadora de la interrupción envía el mensaje. Una vez enviado el mensaje se continúa con el bucle empezando de nuevo por el primer bloque funcional.

3.1.4 Servidor o receptor ARINC 429

El receptor de mensajes ARINC 429 ha sido diseñado para recibir los paquetes UDP del transmisor, extraer el mensaje ARINC 429, decodificar el mensaje y extraer el dato en función de la codificación del mensaje. Además, se ha previsto que no se establezca ningún mecanismo de acuse de recibo con el transmisor tal y como especifica el estándar (ARINC SPECIFICATION 429, PART 1, 2004).

Se ha dividido la operación del receptor en un bucle con los siguientes estados:

1. Recepción del paquete UDP: Se espera a la recepción de un paquete del transmisor.
2. Extracción del mensaje ARINC 429: Se extrae el mensaje del *buffer* de recepción.
3. Decodificación del mensaje: Se decodifica la Etiqueta, se busca en la base de datos la codificación para esa Etiqueta (ver Sección 3.1.3.2).
4. Extracción del Dato: Se decodifica el Dato acorde con el apartado anterior y se imprime por pantalla.

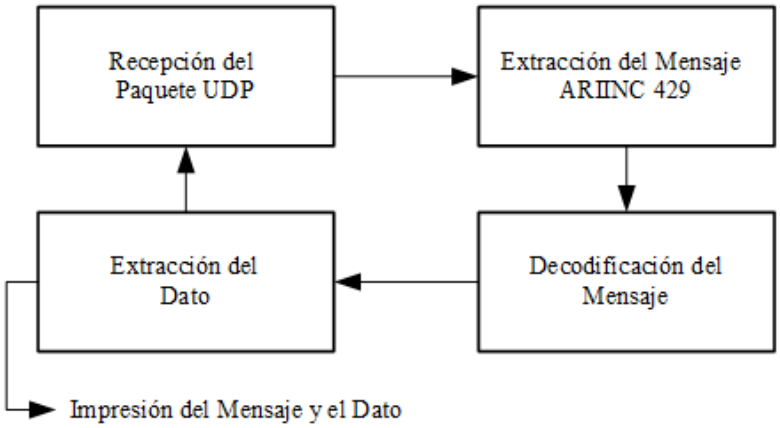


Figura 3-3: Bloques del bucle del Servidor

El primer estado del bucle es bloqueante, es decir, queda esperando la recepción indefinidamente. Además se ha previsto que el bucle sea infinito para facilitar la creación, modificación e inspección de paquetes en el transmisor y simular el proceso de escucha de un receptor ARINC 429.

3.1.5 Limitaciones del diseño

El diseño propuesto tiene algunas limitaciones:

- Los intervalos de tiempo de transmisión se pueden controlar con eficiencia hasta los milisegundos. Los valores por debajo de 1ms tienden a incorporar ciertos retrasos.
- No se tiene control de la tasa de escritura ya que no se tiene control de la capa física.
- Pueden producirse pérdidas de paquetes por la naturaleza del protocolo UDP
- Pueden producirse retrasos en recepción por la naturaleza de la red IP.

3.2 Implementación

3.2.1 Estructura de ficheros

El simulador se encuentra desarrollado bajo el directorio **arinc-429_simulator**. Este directorio está constituido por cuatro tipo de ficheros diferentes: código fuente, ejecutables y misceláneos. Los primeros son necesarios para compilar el programa. Los segundos corresponden al ejecutable del transmisor y del receptor. Los últimos están compuestos por: el fichero de datos de entrada, la base de datos de palabras ARINC 429 y el fichero de compilación. Los ejecutables y misceláneos se encuentran en el directorio principal mientras que los ficheros pertenecientes al código fuente se encuentran en el directorio **src**.

3.2.1.1 Código fuente

El código fuente se encuentra dentro del directorio **src** y se ha estructurado en los siguientes ficheros:

- **arinc-Tx.c**: Código fuente con el `main()` del programa del transmisor.
- **arinc-Rx.c**: Código fuente con el `main()` del programa del receptor.
- **arinc-Tx-func.c**: Código fuente con las funciones utilizadas por **arinc-Tx.c**.
- **arinc-Rx-func.c**: Código fuente con las funciones utilizadas por **arinc-Rx.c**.
- **arinc-Tx.h**: Código fuente con las declaraciones de funciones estructuras y variables globales utilizadas por **arinc-Tx.c**.

- **arinc-Rx.h**: Código fuente con las declaraciones de funciones y estructuras utilizadas por **arinc-Rx.c**.

3.2.1.2 Ejecutables

Los ejecutables están disponibles cuando se compila el programa usando la herramienta `make`. Se almacenan en el directorio principal de la aplicación. Los ejecutables generados son:

- **arinc-Tx**: Ejecutable del transmisor.
- **arinc-Rx**: Ejecutable del receptor.

La información sobre la puesta en ejecución del simulador se puede encontrar en la sección 3.3.

3.2.1.3 Misceláneos

Los ficheros misceláneos son los siguientes:

- **input-file.txt**: Fichero que contiene los datos de entrada.
- **db.csv**: Fichero que contiene la base de datos de palabras ARINC 429.
- **Makefile**: Fichero para la compilación.

3.2.2 Cliente o transmisor ARINC 429

En el código del transmisor ha sido implementado en tres etapas: inicialización del programa, bucle de los cinco bloques funcionales especificados en el diseño y cierre y fin del programa.

3.2.2.1 Inicialización del programa

La inicialización del programa se lleva a cabo en varias fases:

1. Se incluye la librería **arinc-Tx.h**.
2. Se procede a iniciar las variables locales dentro del `main()`. Las variables locales son:
    - `char file_name[]`: Cadena que establece el nombre del fichero de datos de entrada.
    - `int returnStatus`: Bandera para la comprobación de la conexión entre *sockets* UDP.
    - `struct arinc429 arinc429_values`: Estructura con campos del mensaje ARINC 429. La definición de la estructura se encuentra en **arinc-Tx.h** y sus campos son:
        - `unsigned int label`: Etiqueta del mensaje.
        - `unsigned int sdi`: Campo SDI del mensaje.
        - `float data`: Campo de Datos del mensaje.
        - `unsigned int ssm`: Campo SSM del mensaje.
        - `unsigned int parity`: Campo de Paridad del mensaje.
    - `struct arinc429 * sp`: Puntero a la estructura anterior usado como parámetro en las funciones.
    - `struct sockaddr_in udp_clt`: Información del *socket* del cliente.
    - `struct sigaction act`: Estructura del manejador de interrupciones.
    - `unsigned int codification`: Determina la codificación de los datos para un mensaje.
    - `int nbit_used`: Determina los bits usados en la codificación BNR de los datos de un mensaje.
3. Se comprueba que el número de argumentos pasados en la invocación del programa y si no son los adecuados se imprime un mensaje de ayuda.

4. Se comprueba si el parámetro opcional `debug` ha sido configurado y, si es así, se configura al valor establecido. Esta variable determina el funcionamiento de la Herramienta de Comprobación, cuando toma valores entre 0 y 3, o de la configuración de intervalos temporales, cuando toma el valor 4. La herramienta de comprobación está descrita en las secciones 3.2.4 y 3.3.3, la configuración de intervalos temporales se encuentra detallada en las secciones 3.2.2.4 y 3.3.4.
5. Se apunta con el puntero `sp` a la estructura `arinc429_values`.
6. Se inicializa la estructura manejadora de interrupciones y se declara que la señal de interrupción a manejar es `SIGALRM`. Esta interrupción de tiempo se genera cuando expira un temporizador configurado previamente.
7. Se crea el *socket* y se vincula a la dirección IP y puerto especificado en los parámetros pasados por línea de comandos. Usamos `returnStatus` para comprobar si hay error en la conexión e imprimir un mensaje de error informativo. Se puede observar cómo se usan dos variables globales llamadas `udp_s` y `udp_srv` que contienen información sobre el *socket* del servidor. Son globales debido a que se usan también dentro de la función de interrupción `tx_arinc()` que no acepta paso de parámetros.
8. Se abre el fichero de entrada usando la variable global `fp` como descriptor de fichero. Se ha declarado esta variable globalmente para que pueda ser usada en la función `tx_arinc()` que no admite paso de parámetros.
9. Se ejecuta una primera iteración del bucle que se explica en detalle en la sección 3.2.2.2. Se ha previsto así para que haya un mensaje preparado cuando se reciba la primera interrupción.
10. Se configura el temporizador que lanzará la primera interrupción para que ocurra en un determinado intervalo temporal en microsegundos con la función `ualarm()`.
11. Se espera a la primera interrupción con un bucle `while()` cuya condición de parada es que la variable `stop` sea igual a 1.

3.2.2.2 Bucle

Se observa como el bucle principal del programa no está contenido entre corchetes de la condición `while()`. Esto se debe a que el bucle se encuentra dentro de la función que maneja la interrupción `tx_arinc()`.

El programa queda esperando en el `while()` hasta que ocurre una interrupción, es decir, hasta que expira un temporizador. Cuando ocurre una interrupción se pasa a ejecutar la función `tx_arinc()` que es la que desarrolla los 5 bloques funcionales especificados en el diseño:

3.2.2.2.1 Lectura de datos de entrada

Se encarga de ello la función `read_pack()`. La función devuelve un 1 y lo almacena en `stop` en caso de que no haya podido leer nada. La función toma como parámetros tres punteros a los siguientes datos:

1. `fp`: El descriptor de fichero donde se leen los datos de entrada.
2. `sp->label`: Campo Etiqueta de una estructura de mensaje ARINC 429 donde se guarda la Etiqueta leída del fichero de entrada.
3. `sp->data`: Campo Datos de una estructura de mensaje ARINC 429 donde se guarda el Dato leído del fichero de entrada.

En la Figura 3-4 se muestra un diagrama de las entradas y salidas de la función.

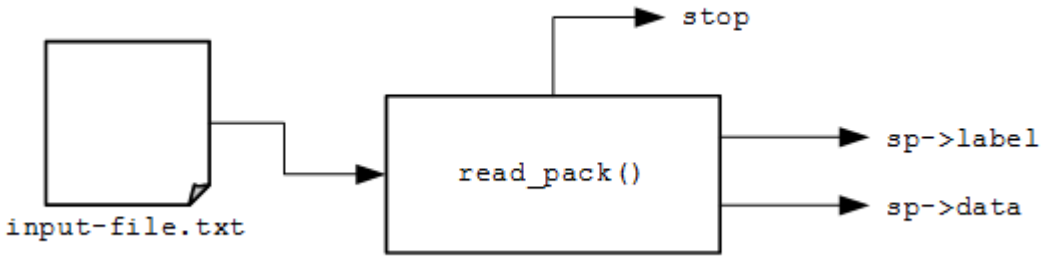


Figura 3-4: Diagrama de entradas y salidas del Bloque 1

3.2.2.2.2 Buscar Etiqueta en base de datos

Se encarga de ello la función `read_db()`. Esta función lee el fichero **db.csv** en busca de la Etiqueta pasada como parámetro para determinar la forma de codificar los Datos. La función toma como parámetros 6 punteros a los siguientes datos:

1. `sp->label`: Campo Etiqueta de una estructura de mensaje ARINC 429.
2. `sp->data`: Campo Datos de una estructura de mensaje ARINC 429.
3. `sp->sdi`: Campo SDI de una estructura de mensaje ARINC 429.
4. `sp->ssm`: Campo SSM de una estructura de mensaje ARINC 429.
5. `nbit_used`: Variable de información auxiliar para la codificación BNR.
6. `codification`: Variable que indica si la codificación del dato es BNR o BCD.

Si la función no encuentra la Etiqueta en la base de datos configura todos los valores de la estructura de mensaje ARINC 429 a 0 transmitiéndose así un paquete UDP con 32 bits vacíos, con la excepción de la paridad que se ha codificado como un bloque independiente.

Si se encuentra, la función comprueba que los datos de entrada sean acordes a la información tomada de la base de datos, esto incluye: que el Dato se encuentre en el intervalo entre Valor máximo y Valor mínimo, que el SDI solo tome valores del 0 al 3 y que la variable `nbits_used` tome valores del 0 al 19.

En caso contrario configuramos el SSM para advertir del error como sigue:

- Si la codificación es BNR: Se configura SSM a 0 que según el estándar significa Aviso de Fallo.
- Si la codificación es BCD: Se configura SSM a 2 que según el estándar significa Datos No Procesados. No se puede codificar un Aviso de Fallo porque el SSM para la codificación BCD no lo soporta en el estándar.

En la Figura 3-5 se muestra un diagrama de entradas y salidas de la función `read_db()`.

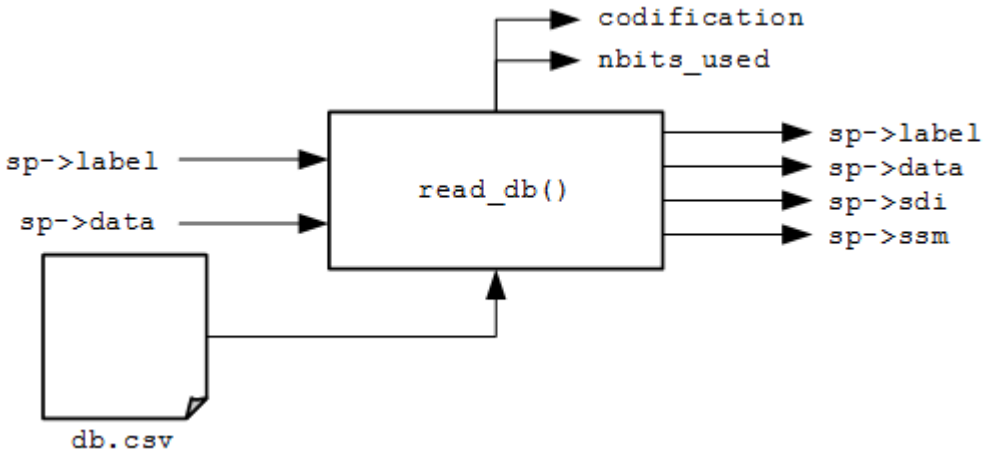


Figura 3-5: Diagrama de entradas y salidas del Bloque 2

3.2.2.2.3 Codificación del mensaje

De la codificación del mensaje se encargan dos funciones: `codify_bnr()` y `codify_bcd()`, para codificar en BRN y BCD respectivamente. Ambas devuelven el mensaje codificado en un la variable `arincMsg` de tipo `int`. Las dos funciones comparten el primer parámetro:

1. `sp`: Estructura con campos del mensaje ARINC 429.

Y `codifiy_bnr()` además usa un parámetro adicional

2. `nbits_used`: Número de bits usados en el campo de datos para codificar en BNR.

El funcionamiento de este bloque queda descrito en la Figura 3-6.

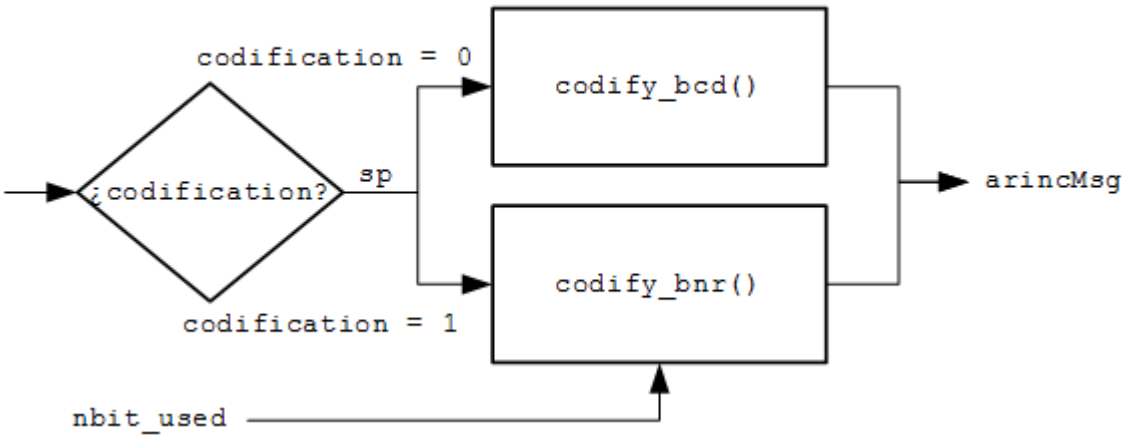


Figura 3-6: Diagrama de entradas y salidas del Bloque 3

3.2.2.2.4 Encapsulación del mensaje

La encapsulación del mensaje es llevada a cabo mediante la función `serialize_int()`. La función toma los siguientes parámetros:

1. `buf`: *Buffer* de transmisión de 32 bits.
2. `arincMsg`: Mensaje arinc 429 codificado en un int.

Esta función se encarga de estructurar el tipo de dato `int` en un *buffer* de tipo `char`. Este tipo de operación es

dependiente de como el sistema operativo almacena la información y cuanto espacio reserva para cada tipo de dato. Por ello esta función no es portable y solo funciona en sistemas operativos *Little-Endian*.

3.2.2.2.5 Espera a la transmisión

Una vez mensaje está guardado en el *buffer* se espera a la interrupción para su transmisión. Cuando se recibe una interrupción, la función manejadora de la interrupción se encarga de enviar lo que haya almacenado en el *buffer* de transmisión.

3.2.2.2.6 Condición de parada del bucle

La condición de parada del bucle se administra en la función `read_packt()` del primer bloque funcional. Cuando la función llegue al final del fichero de entrada devolverá un 1 que quedará almacenado en la variable `stop`. Esta variable se ha utilizado como condición de parada del bucle del transmisor en el `main()` del programa.

3.2.2.3 La función manejadora de la interrupción

La función manejadora de la interrupción `tx_arinc()` pasa a ejecución cuando se recibe una interrupción `SIGALRM`. Desarrolla varias funciones:

- Transmisión del paquete UDP con los datos encapsulados.
- Ejecución del bucle con los 5 bloques funcionales.
- Generación de las futuras interrupciones configurándolas para su envío de acuerdo a unos intervalos dados.

El orden en el que se ejecutan estas funciones es el siguiente:

1. Se envía el paquete UDP cuando se recibe la interrupción, es decir, cuando expira el temporizador configurado previamente. Se encarga de ello la función `sendto()`.
2. Se procede a ejecutar el primer bloque funcional descrito en la sección 3.2.2.2.1: Lectura de datos de entrada.
3. Si:
    a. No hay datos de entrada se configura `stop` a 1 y se sale de la función.
    b. Hay datos de entrada se configura el temporizador de la siguiente interrupción con la función `ualarm()`. Los tiempos con los que se configura el temporizador se encuentran almacenados en la variable `tiempo_ms` y se repiten periódicamente .La variable `tiempo_ms` puede tomar el valor estático de **45 ms** o puede tomar valores dinámicos según se describe en la sección 3.2.2.4.
4. Se ejecutan luego los cuatro bloques funcionales restantes descritos en la sección 3.2.2.2.

Se ha determinado que se configure el temporizador justo tras conocer si hay datos de entrada con el propósito de que la codificación de los datos se produzca mientras el temporizador se va decrementando. Con este enfoque se consigue un intervalo temporal entre paquetes más preciso. Esta implementación supone que la codificación de los datos se realiza en un intervalo temporal menor al de la expiración del temporizador.

En la figura Figura 3-7 se aprecia el diagrama de bloques de la función `tx_arinc()`.

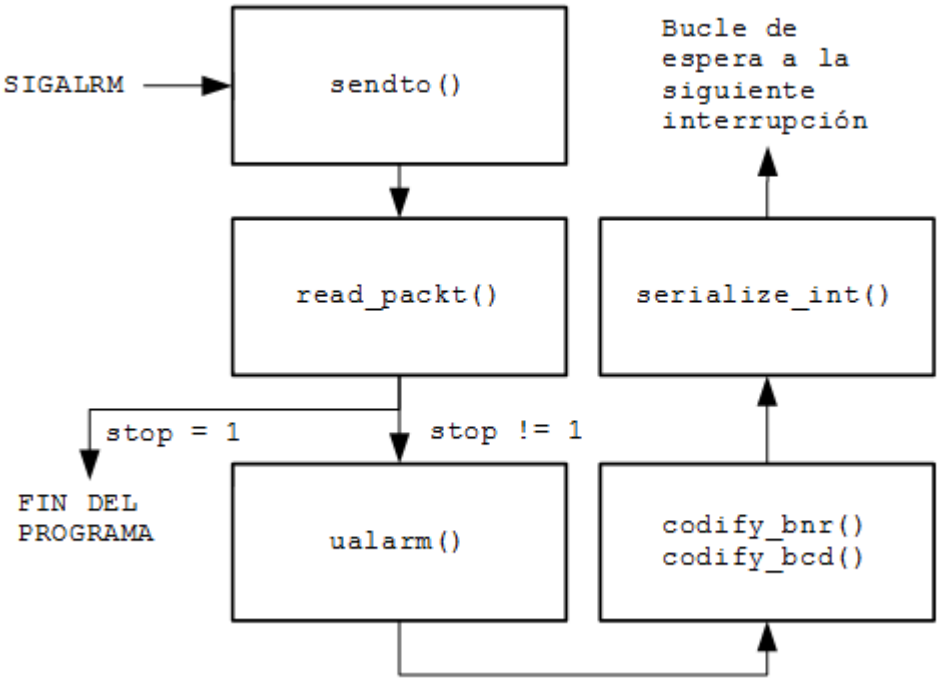


Figura 3-7: Diagrama de bloques de la función manejadora de interrupciones

3.2.2.4 Configuración de intervalos temporales

Para la configuración de los intervalos temporales la variable `debug` debe tomar el valor 4. La función `scheduling_ini()` se encarga de configurar internamente los intervalos de transmisión. Esta función toma dos parámetros:

1. `int argc`: Número de elementos pasados por línea de comandos.
2. `char ** argv`: Tabla con las cadenas de caracteres pasadas por línea de comandos en la ejecución del programa.

Esta función espera que tras la llamada al programa se inserten varios parámetros adicionales en el siguiente orden:

- Un número entero positivo que indique el número de parámetros adicionales que se pasan por línea de comando tras él.
- Una cadena de tiempos crecientes en milisegundos separados por espacios cuya longitud debe ser la misma que la expresada en el parámetro anterior.

La función reservará memoria dinámica para almacenar todos los tiempos pasados por línea de comandos usando la variable global `tiempo_ms` como puntero a esa zona de memoria. De este modo, `tiempo_ms` guarda los tiempos que la función `tx_arinc()`, descrita en la sección 3.2.2.3, usará para la configuración del temporizador. Esta configuración del temporizador es la que determina los tiempos de transmisión de los mensajes.

Si no se configuran intervalos temporales o no se cumple alguna de las condiciones especificadas en el paso de los parámetros de tiempo se configura la variable `tiempo_ms` para que los paquetes se transmitan cada **45 ms**.

3.2.2.5 Cierre y fin del programa

Cuando se da la condición de parada y se devuelve el control al `main()` se sale del bucle de espera, se cierran los *sockets* abiertos y se cierra el programa.

3.2.3 Servidor o receptor ARINC 429

El receptor ARINC 429 se ha implementado en dos etapas: inicialización del programa y bucle de operación. El receptor actúa como un servidor UDP que espera indefinidamente paquetes del cliente, los decodifica e imprime el Dato por pantalla.

3.2.3.1 Inicialización del programa

La inicialización se lleva a cabo en las siguientes etapas:

1. Se incluye la librería **arinc-Rx.h**.
2. Se inician las variables locales dentro del `main()`. Las variables son:
    - `int upd_s`: Descripto del *socket* del servidor.
    - `int returnStatus`: Variable de comprobación del enlace del *socket*.
    - `int addrlen`: Longitud de la información del cliente.
    - `struct sockaddr_in udp_srv`: Estructura con información del servidor.
    - `struct sockaddr_in udp_clt`: Estructura con información del cliente.
    - `char buf[MAXBUF]`: *Buffer* de recepción.
    - `int arincMsg`: Variable donde se almacenará el paquete tras sacarlo del buffer de recepción.
    - `strcut arinc429 arinc`: Estructura con los campos de un paquete arinc 429.
    - `strcut arinc429 * sp`: Puntero que apuntará a la estructura anterior.
    - `int nbit_used`: Número de bits a usar en la codificación en BNR.
    - `unsigned int codification`: Bandera que informa sobre la codificación del paquete recibido.
3. Se apunta con `sp` a la variable `arinc`.
4. Se comprueba que se pase los parámetros necesarios por línea de comandos.
5. Se crea el *socket* y se enlaza con la información del servidor.

A partir de este momento se entra en el bucle infinito.

3.2.3.2 Bucle

El bucle consta de las cuatro etapas descritas en la Sección 3.1.4: recepción del paquete UDP, extracción del mensaje ARINC 429, decodificación del mensaje y extracción del Dato.

3.2.3.2.1 Recepción del paquete UDP

De la recepción del paquete UDP se encarga la función de librería `recvfrom()` que toma parámetros de información del *socket* del cliente, el descriptor del *socket* del servidor y almacena los datos del paquete UDP recibido en la variable `buf`.

3.2.3.2.2 Extracción del mensaje ARINC 429

En esta etapa se extra el mensaje de la variable `buf` y se ordena en un `int`. De ello se encarga la función `r_serialize_int()`. Toma como parámetro la variable `buf` y devuelve un `int` que se almacena en la variable `arincMsg`.

3.2.3.2.3 Decodificación del mensaje

La decodificación del mensaje se lleva a cabo en varias etapas:

1. Decodificación de la Etiqueta del mensaje: con la función `dcodify_label()` que toma como parámetro la variable `arincMsg` se obtiene el número de la Etiqueta.
2. Extracción de información de codificación: Se busca la Etiqueta en la base de datos **db.csv** y se extrae información sobre su codificación. Se encarga de ello la función `read_db()` que toma los siguiente parámetros:
    a. `sp->label`: Campo Etiqueta de una estructura de mensaje ARINC 429 que contiene la Etiqueta decodificada.
    b. `nbit_used`: Variable de información auxiliar para la decodificación BNR
    c. `codification`: Variable que determina la codificación del mensaje para esa Etiqueta.

3.2.3.2.4 Extracción del dato

La decodificación del Dato se realiza acorde a la variable `codification` que determina la forma en la que está codificado el Dato.

Si el dato está codificado en BNR se usa la función `dcodify_bnr()` que toma como parámetros:

1. arincMsg: El mensaje ARINC 429 codificado.
2. nbit_used: Número de bits usados en la codificación BNR.

La función devuelve el dato y lo almacena en el campo de Dato de la estructura ARINC 429 apuntada por el puntero `sp`.

Si el dato está codificado en BCD se usa la función `dcodify_bcd()` que toma como parámetros:

1. arincMsg: El mensaje ARINC 429 codificado.

La función devuelve el dato y lo almacena en el campo de Dato de la estructura ARINC 429 apuntada por el puntero `sp`.

Tras la decodificación del Dato se procede a su impresión por pantalla.

3.2.4 Herramienta de comprobación

La herramienta de comprobación consta de una serie de funciones de impresión de información que se incluyen en los principales bloques de operación del transmisor.

El tipo de información que se muestra lo determina la variable global `debug` que puede ser configurada de forma opcional en la llamada al transmisor.

Se pueden observar a lo largo del código varias sentencias `if()` que buscan coincidencia con un valor para la variable `debug`. El valor que toma puede variar de 0 a 3. Para cada valor de `debug` la información que se imprimirá será diferente y se encuentra descrito en detalle en la sección 0.

La impresión de los distintos valores se consigue con combinaciones de la función `printf()` y las siguientes funciones:

- `showbits()`: Imprime los bits de un dato del tipo unsigned int.
- `showbits_arincBCD()`: Imprime los bits de un mensaje ARINC 429 separando los campos del mensaje y las cifras de la codificación BCD.
- `showbits_arincBNR()`: Imprime los bits de un mensaje ARINC 429 separando los campos del mensaje.

3.2.5 Limitaciones de la implementación

La implementación desarrollada presenta varias limitaciones:

- No se comprueba la corrección sintáctica de los parámetros pasados al transmisor.

- No se comprueba la corrección sintáctica de los parámetros pasados al receptor.
- La función de encapsulación de los mensajes `serialize_int()` y la función de decodificación `r_serialize_int()` son dependiente de como el sistema operativo almacena la información y por tanto no son portable a sistemas con diferente organización de la información.
- La implementación de la función `tx_arinc()` exige que la codificación de los datos y su inserción en el *buffer* de transmisión requiera un intervalo temporal menor al de expiración del temporizador de la interrupción.
- La decodificación BCD no soporta números acabados en cero o en una secuencia de ceros. Esto se debe a que mediante recursos *software* no se implementa un método para averiguar si los campos de los últimos dígitos de la codificación BCD se han usado y están a cero o no se han usado. En la capa física se hace esta distinción porque se diferencia entre tres estados: alto (voltaje positivo), nulo (voltaje a cero) y bajo (voltaje negativo). Y se puede diferenciar si se han usado y están a cero (voltaje negativo) o no se han usado (voltaje a cero).

3.3 Ejecución del simulador

Para el correcto funcionamiento del simulador se debe ejecutar primero el servidor y después el cliente tal y como se describe a continuación.

3.3.1 Ejecución del servidor

El servidor debe ser ejecutado en la terminal de comandos. Además toma como parámetro el puerto donde el *socket* UDP escuchará los mensajes del cliente. En caso de no aportar el parámetro necesario se imprimirá por pantalla un breve mensaje informativo sobre el uso del programa.

El servidor imprimirá por pantalla los paquetes que reciba y lo hará acorde a la codificación de los Datos del paquete.

Un ejemplo de llamada para la ejecución del servidor en el puerto de ejemplo 16000 que recibe un paquete codificado en BCD y otro en BNR es el Ejemplo 3-3:

```
user@ubuntu:~$ ./arinc-Rx 16000
ARINC429->   00001010 00 0000 0000 0000 1010 111 00 0    0xa0002b8
ARINC429->   01000000 00 0000000000001011011     11 1    0x400002df
```

Ejemplo 3-3: Llamada al servidor

El *socket* UDP escuchará mensajes en todas las interfaces de red del equipo.

3.3.2 Ejecución del cliente

El cliente debe ser ejecutado en la terminal de línea de comandos una vez que el servidor ya está escuchando peticiones. Para poder ser ejecutado se necesita que se aporten los parámetros necesarios y que los ficheros **input-file.txt** y **db.csv** se encuentren dentro del directorio desde el que se ejecuta el programa. Los parámetros del cliente son:

1. Dirección IP de alguna de las interfaces donde escucha el equipo que ejecuta el servidor.
2. Puerto donde se escucha el *socket* del servidor.
3. Herramienta de comprobación / Configuración de intervalos temporales: Si toma valores del 0 al 3 el valor corresponde a un nivel de información de la herramienta de comprobación descrita en la sección 3.3.3. Si toma el valor 4 se determina la configuración de intervalos temporales descrita en la sección 3.3.4.

Los dos primeros parámetros son obligatorios y el tercero es opcional. Un ejemplo de ejecución en el que el servidor se ejecuta en el mismo equipo del cliente en el puerto 16000 y con nivel 1 de la herramienta de

comprobación es el Ejemplo 3-4:

```
user@ubuntu:~$ ./arinc-Tx 127.0.0.1 16000 1
```

Ejemplo 3-4: Llamada al cliente

3.3.3 Herramienta de comprobación

La herramienta de comprobación se ejecuta en el cliente y permite mostrar información sobre el proceso de codificación en diferentes niveles. Estos niveles corresponden con el valor configurado en el cuarto parámetro en la llamada al transmisor ARINC 429.

3.3.3.1 Nivel 0

La herramienta no imprime nada por pantalla. Es el nivel por defecto. El ejemplo de una llamada al transmisor con nivel 0 y su salida es el Ejemplo 3-5:

```
user@ubuntu:~$ ./arinc-Tx 127.0.0.1 16000 0
user@ubuntu:~$
```

Ejemplo 3-5: Herramienta de comprobación con Nivel 0

3.3.3.2 Nivel 1

Se imprime el paquete enviado en binario separado por campos, el paquete en hexadecimal y el valor del Dato en decimal. Un ejemplo de ejecución para dos paquetes, uno codificado en BNR y otro en BCD es el Ejemplo 3-6:

```
user@ubuntu:~$ ./arinc-Tx 127.0.0.1 16000 1
ARINC429->   01100110 01 00011001011100000000     11 0  0x66465c06  467
ARINC429->   00001010 00 0000 0000 0000 1100 111 00 0  0xa000338  73
```

Ejemplo 3-6: Herramienta de comprobación con Nivel 1

3.3.3.3 Nivel 2

Se imprime información de cada campo del paquete y de los valores leídos de la base de datos, incluyendo: número de bits usados, valor máximo y mínimo y la resolución. Un ejemplo de ejecución es el Ejemplo 3-7:

```
user@ubuntu:~$ ./arinc-Tx 127.0.0.1 16000 2
Valores        : label  sdi  Rmax  Rmin  bitused  resol  codif  value   ssm
Valores leídos : 102    1    65536 0     16       1.000  1      467.00  3

ARINC429->   01100110 01 00011001011100000000     11 0  0x66465c06  467
```

Ejemplo 3-7: Herramienta de comprobación con Nivel 2

3.3.3.4 Nivel 3

Se imprime información de la codificación de la Etiqueta del paquete enviado. Se imprime en primer lugar la codificación octal de la Etiqueta. Se imprime luego la codificación por cifras y se muestra el resultado final. Un ejemplo de ejecución es el Ejemplo 3-8:

```
user@ubuntu:~$ ./arinc-Tx 127.0.0.1 16000 3
Label dec = 102, octal = 146
n1     =     0000000000000001
n2     =     0000000000000100
n3     =     0000000000000110
```

```
shift1 =     01000000 00 0000 0000 0000 0000 000 00 0
shift2 =     00100000 00 0000 0000 0000 0000 000 00 0
shift3 =     00000110 00 0000 0000 0000 0000 000 00 0
           +_____________________________________
ARINC429->   01100110 01 0001100101110000000     11 0  0x66465c06  467
```

Ejemplo 3-8: Herramienta de comprobación con Nivel 3

3.3.4 Configuración de intervalos temporales

Si el tercer parámetro de la llamada toma el valor 4 se determina la configuración de intervalos temporales. En este caso se deben introducir varios parámetros más:

1. Número de tiempos insertados tras este parámetro
2. Tiempo en milisegundos para la transmisión del primer paquete.
3. Tiempo en milisegundos para la transmisión del segundo paquete

   …

n. Tiempo en milisegundos para la transmisión del paquete **n**. Donde **n** es un número entero positivo.

En el Ejemplo 3-9 se muestra la ejecución del programa configurando la transmisión de dos paquetes en los tiempos **250 ms** y **500 ms**.

```
user@ubuntu:~$ ./arinc-Tx 127.0.0.1 16000 4 2 250 500
ARINC429->   01100110 01 0001100101110000000     11 0  0x66465c06  467
ARINC429->   00001010 00 0000 0000 0000 1100 111 00 0  0xa000338  73
```

Ejemplo 3-9: Configuración de intervalos temporales

4 PRUEBAS Y VALIDACIÓN

A lo largo de este capítulo se describen las pruebas desarrolladas para validar la exactitud con la que cumple el simulador las especificaciones. Se exponen los contextos en los que el simulador opera con normalidad y aquellos que escapan al límite de su control. Con este propósito se ha divido este capítulo en los mismos apartados que las especificaciones expuestas en la sección 2.2.

4.1 Validaciones generales

A continuación se exponen los resultados de las pruebas designadas para probar el cumplimiento de las especificaciones.

4.1.1 Sistema operativo

- Se ha probado satisfactoriamente la ejecución del simulador en las siguientes versiones de Ubuntu: 12.04 y 14.04.02.
- Se ha probado satisfactoriamente el fichero Makefile para compilar ambos el cliente y el servidor en las versiones de Ubuntu: 12.04 y 14.04.02.

4.1.2 Arquitectura de red y encapsulación

- El cliente UDP corresponde con el transmisor y el servidor con el receptor.
- La arquitectura cliente-servidor ha sido probada satisfactoriamente en los siguientes casos:
    - Ejecutando el cliente y el servidor en un mismo equipo.
    - Ejecutando el cliente y el servidor en dos equipos de la misma subred.
- La encapsulación de los datos ha sido comprobada con *Wireshark*. Los datos del paquete UDP corresponden exactamente con los 32 bits de la palabra ARINC 429.
- La independencia del simulador con la capa física se ha probado satisfactoriamente con la ejecución del simulador en varios equipos con:
    - Diferentes controladores para la capa de enlace: Distintas tarjetas de red.

- Diferente medio físico: Servidor con conexión cableada y cliente con conexión inalámbrica a la misma subred.

4.1.3 Interfaz de entrada y base de datos

- El fichero de entrada es estructurado y el simulador funciona siempre que haya al menos una Etiqueta y un dato que transmitir. Se ha probado satisfactoriamente el funcionamiento del simulador hasta con 100 entradas en el fichero.

- La base de datos contiene un formato estructurado y es ampliable con un editor de texto o herramienta ofimática que soporte ficheros **.csv** siempre que se respete el formato. Se ha probado satisfactoriamente el funcionamiento del simulador tras la inclusión de tres Etiquetas nuevas en la base de datos.

- Se implementa un mecanismo de protección ante Etiqueta no encontrada en la base de datos. Si se da esta condición anula todo el mensaje y se envía un paquete UDP con todos los datos a 0.

4.1.4 Herramienta de comprobación

Se ha validado en la ejecución del simulador que la herramienta de comprobación cumple con los siguientes aspectos:

- Muestra el paquete enviado en binario y hexadecimal si se configura para ello.

- Diferencia entre los campos del paquete en su representación binaria.

- Muestra la información leída en la base de datos y la codificación del paquete en base a ella si se configura para ello.

- Muestra el proceso de codificación de la Etiqueta si se configura para ello.

- Proporciona un mecanismo para seleccionar si se quiere imprimir información sobre el paquete y el tipo de información que se desea mostrar.

4.2 Validaciones relacionadas con el estándar

En este apartado se especifican las especificaciones validadas en relación con el estándar ARINC 429.

4.2.1 Mensajes

- Destino de la información: Se ha probado mediante la inspección de los estados de conexión de los *sockets* que el cliente se comunica satisfactoriamente con el servidor.

- Elemento de información: Se ha probado mediante inspección del paquete con el programa *Wireshark* y con la herramienta de comprobación que las palabras ARINC 429 transmitidas contienen 32 con sus cinco campos respectivos.

- Se ha probado satisfactoriamente con la herramienta de comprobación y mediante la inspección del paquete con el programa *Wireshark* que los campos SSM y SDI se codifican correctamente.

4.2.2 Codificación de los datos y la etiqueta

Para validar la correcta codificación del mensaje de acuerdo al estándar se han desarrollado varias pruebas que

evalúan el comportamiento del simulador de diferentes situaciones.

4.2.2.1 Codificación BCD

Se ha probado satisfactoriamente la codificación BCD del simulador con las siguientes pruebas:

- Dato positivo con primera cifra menor que 7 y dentro del rango definido para esa Etiqueta.
- Dato positivo con primera cifra mayor que 7 y dentro del rango definido para esa Etiqueta.
- Dato negativo con primera cifra menor que 7 y dentro del rango definido para esa Etiqueta.
- Dato negativo con primera cifra mayor que 7 y dentro del rango definido para esa Etiqueta.
- Dato positivo con primera cifra menor que 7, dentro del rango definido para esa Etiqueta y con una resolución de **0.5**.
- Dato positivo con primera cifra mayor que 7, dentro del rango definido para esa Etiqueta y con una resolución de **0.5**.

Se ha probado el comportamiento del simulador en las siguientes situaciones límites:

- El dato queda fuera de rango superando el límite inferior o superior definido para esa Etiqueta: El simulador configura todos los datos a 0 de la palabra ARINC 429 y configura el campo SSM para el valor de Datos No Procesados.
- El número es mayor que el mayor número codificable: El simulador configura todos los datos a 0 de la palabra ARINC 429 y configura el campo SSM para el valor de Datos No Procesados.

4.2.2.2 Codificación BNR

Se ha probado satisfactoriamente la codificación BNR del simulador con las siguientes pruebas:

- Dato positivo usando 6 bits para la codificación.
- Dato negativo usando 6 bits para la codificación.
- Dato positivo usando 12 bits para la codificación.
- Dato negativo usando 12 bits para la codificación.
- Dato positivo usando 12 bits para la codificación y con una resolución de **0.5**.
- Dato negativo usando 12 bits para la codificación y con una resolución de **0.5**.

Se ha probado el comportamiento del simulador en las siguientes situaciones límites:

- El dato queda fuera de rango superando el límite inferior o superior definido para esa Etiqueta: El simulador configura todos los datos a 0 de la palabra ARINC 429 y configura el campo SSM para el valor de Aviso de Fallo.

4.2.2.3 Codificación de la Etiqueta

Se ha probado la codificación de la Etiqueta satisfactoriamente en los siguientes casos:

- Etiqueta encontrada en la base de datos.
- Etiqueta nueva añadida a la base de datos.

Se ha probado el comportamiento del simulador en las siguientes situaciones límites:

- Etiqueta no encontrada en la base de datos: El simulador configura todo el paquete a 0, con la excepción de la paridad.
- Etiqueta fuera del rango codificable: El simulador configura todo el paquete a 0, con la excepción de la paridad.

4.2.3 Procesado y la transmisión

- Orden de transmisión: Se ha comprobado mediante inspección de paquetes con el programa *Wireshark* y con la herramienta de comprobación que el orden de transmisión de los bits respeta el estándar.

- Detección de errores: Se ha comprobado mediante inspección de paquetes con el programa *Wireshark* y con la herramienta de comprobación que la configuración del bit de paridad respeta el estándar.

4.2.4 Configuración de intervalos de transmisión

- Sincronización de palabras: se ha probado mediante inspección de paquetes con el programa *Wireshark* que los paquetes respetan los intervalos de de transmisión configurados entre paquetes con una precisión de milisegundos y con un pequeño retraso en nanosegundos que varía en función de cada equipo y que se acumula con el tiempo de ejecución.

5 CONCLUSIONES Y DESARROLLOS FUTUROS

En este capítulo se exponen las dificultades encontradas en la realización de este Trabajo Fin de Grado así como posibles desarrollos futuros, líneas de mejora del simulador actual y las conclusiones finales. Se pretende evaluar el cumplimiento de los objetivos expuestos en la sección 1.2.

5.1 Conclusiones

El objetivo principal de este Trabajo Fin de Grado es diseñar y desarrollar un simulador de la comunicación ARINC 429 que sea capaz de operar con independencia de la capa física propuesta por el estándar.

En esta memoria se ha presentado el diseño y el desarrollo de un simulador de la comunicación ARINC 429. Se ha probado cómo este simulador reproduce con fidelidad la codificación de las palabras ARINC 429 para datos codificados en BCD y en BNR. Adicionalmente se ha mostrado la implementación de una herramienta de comprobación que permite profundizar en varios aspectos de la codificación de la palabra ARINC 429.

El simulador desarrollado se ofrece como una herramienta didáctica que facilita la comprensión del funcionamiento del estándar a través de la creación, modificación e inspección de los paquetes ARINC 429 que genera.

5.2 Mejoras y desarrollos futuros

Existen dos líneas de mejoras principales para este Trabajo Fin de Grado, aquellas dirigidas a aportar robustez al funcionamiento del simulador actual y aquellas dedicadas a extender funcionalidades.

Mejoras dedicadas a la extensión de funcionalidad:

- Soporte para la codificación de tipos de datos adicionales como los datos discretos o dar soporte a la transferencia de ficheros.
- Ofrecer una interfaz gráfica para la creación, modificación e inspección de las palabras ARINC 429.
- Tomar mayor control del *hardware* de transmisión determinando el tiempo de transmisión y los intervalos con exactitud.

- Arquitectura en la que el paquete de un transmisor pueda ser escuchado por 20 receptores a la vez.

Mejoras en la robustez:

- Protección ante fallos mediante pruebas de corrección sintáctica de ficheros usados y parámetros pasados por línea de comandos.
- Establecer independencia con la organización de la información del sistema operativo: *Little Endian – Big Endian*.
- Mejorar el sistema de temporizadores con varios simultáneos.

5.3 Dificultades encontradas

Durante la realización de este Trabajo Fin de Grado se han encontrado varias dificultades en las diferentes etapas del mismo: investigación de la norma, diseño del simulador e implementación del simulador.

- Obtención del estándar AIRNC 429 perteneciente a *Airlines Electronic Engineering Committee*.
- Diseño de una máquina de estados que programe interrupciones en la misma rutina de interrupción con un fin programado.
- Diseño de la programación de interrupciones para la adaptación de intervalos temporales al estándar.
- Maduración del concepto de la serialización de un tipo de datos en un buffer.
- Depuración a nivel de bit antes del desarrollo de la herramienta de comprobación.
- Implementación de la decodificación BCD en el servidor.

REFERENCIAS

AIRLINES ELECTRONIC ENGINEERING COMMITTEE. (2004, Mayo 17). ARINC SPECIFICATION 429, PART 1. AERONAUTICAL RADIO, INC.

asdf. (14, 11 2). Retrieved 2424 31, 2344, from asdf: asdf

AIM GmbH Avionic Databus Solutions. (2010, Noviembre). ARINC 429 Specification Tutorial. Freiburg, Alemania.

Autor. (2012). Este es el ejemplo de una cita. *Tesis Doctoral, 2*(13).

Autor, O. (2001). Otra cita distinta. *revista*, p. 12.

CONDOR Engineering. (2010, Junio 7). ARINC 429 Tutorial. Santa Bárbara, Estado Unidos de América.

International Telecommunication Union. (1994). Open System Interconnection Model. Helsinki, Finlandia.

Marinec, D. A. (2014). Avionics Handbook. Carry R. Spitzer.

Solutions, A. G. (2010, Noviembre). ARINC 429 Specification Tutorial. Freiburg, Alemania.

Woodward, S. (2002, July 11). *EDN Network*. Retrieved 07 27, 2015, from http://www.edn.com/design/communications-networking/4346565/Circuit-transmits-ARINC-429-data

xnox, e. d. (2013, Octubre 23). *wiki.ubuntu*. Retrieved Julio 27, 2015, from https://wiki.ubuntu.com/LTS

CONCEPTOS

APÉNDICE

Makefile

```
#Fco Javier Vargas
all: arinc-Tx arinc-Rx

arinc-Tx : arinc-Tx.o arinc-Tx-func.o
	gcc -o arinc-Tx arinc-Tx.o arinc-Tx-func.o

arinc-Rx : arinc-Rx.o arinc-Rx-func.o
	gcc -o arinc-Rx arinc-Rx.o arinc-Rx-func.o

arinc-Tx.o : src/arinc-Tx.c src/arinc-Tx.h
	gcc -c src/arinc-Tx.c src/arinc-Tx.h

arinc-Rx.o : src/arinc-Rx.c src/arinc-Rx.h
	gcc -c src/arinc-Rx.c src/arinc-Rx.h

arinc-Tx-func.o : src/arinc-Tx-func.c src/arinc-Tx.h
	gcc -c src/arinc-Tx-func.c src/arinc-Tx.h

arinc-Rx-func.o : src/arinc-Rx-func.c src/arinc-Rx.h
	gcc -c src/arinc-Rx-func.c src/arinc-Rx.h

clean:
	rm *.o && rm src/*.h.gch
```

Código 1: Makefile

arinc-Rx.c

```
#include "arinc-Rx.h"

int main(int argc, char * argv[])
{
	/* Var declaration */
	int udp_s;
	int returnStatus = 0;			// Return var
	int addrlen;				// Length of client info
	struct sockaddr_in udp_srv;		// Server info
	struct sockaddr_in udp_clt;		// Client info
	char buf[MAXBUF];			// Buffer
	int arincMsg;
	struct arinc429 arinc;
	struct arinc429 * sp;
	int nbit_used;
	unsigned int codification;

	sp = &arinc;
	if(argc < 2)
	{
		fprintf(stderr, "Usage: %s <port>\n", argv[0]);
		exit(1);
	}


	/* 1) CREATE THE SOCKET */
	udp_s = socket(AF_INET, SOCK_DGRAM, 0);
	if(udp_s == -1)
	{
		fprintf(stderr, "Could not create socket!\n");
		exit(1);
	}
	else
	{
		printf("Socket created!\n");
	}

	/* 2) GET SERVER INFO (populate sockaddr_in upd_srv)*/
	udp_srv.sin_family = AF_INET;
	udp_srv.sin_addr.s_addr = htonl(INADDR_ANY);
	udp_srv.sin_port = htons(atoi(argv[1]));

	/* 3) BIND THE SOCKET WITH THE INFO SETTED UP*/
	returnStatus = bind(udp_s, (struct sockaddr *) &udp_srv,
					sizeof(udp_srv));
	if(returnStatus == 0)
	{
		fprintf(stderr, "Bind complete!\n");
	}
	else
	{
		fprintf(stderr, "Could not bind!\n");
		close(udp_s);
		exit(1);
	}

	/* 4) LISTEN, (RCVFROM because it limits the size of the reads bytes)*/
	while(1)
	{
		addrlen = sizeof(udp_clt);
		returnStatus = recvfrom ( udp_s,
					buf,
					MAXBUF,
					0,
					(struct sockaddr *) &udp_clt,
					(socklen_t *) &addrlen );
		if( returnStatus == -1 )
```

```
            {
                fprintf(stderr, "SERVER:Could not receive the msg!\n");
            }
            else
            {
                arincMsg = r_serialize_int(buf);
                sp->label = dcodify_label(arincMsg);
                read_db(&(sp->label),
                    &nbit_used,
                    &codification);

                printf("ARINC429->   ");
                if (codification == BNR)
                {
                    showbits_arincBNR(arincMsg);
                    sp->data = dcodify_bnr(arincMsg, nbit_used);
                }
                else
                {
                    showbits_arincBCD(arincMsg);
                    sp->data = dcodify_bcd(arincMsg);
                }
                printf ("\t%#x\t%d\n", arincMsg, sp->data);
            }
        }

        /* 5) CLEAN UP EVERYTHING */
        close(udp_s);
        return 0;

}
```

Código 2: arinc-Rx.c

arinc-Rx-func.c

```
//FUNCTIONS
#include "arinc-Rx.h"

int r_serialize_int(unsigned char *buffer)
{
	int value = 0;
	value += (int) buffer[0] << 24;
	value += (int) buffer[1] << 16;
	value += (int) buffer[2] << 8;
	value += (int) buffer[3];
	return value;
}


int read_packt(FILE ** fp, unsigned int * label, int * value)
{
	int isEOF = 0;
	isEOF = fscanf(*fp, "%u %d\n", label, value );
	if (isEOF == -1)
	{
		isEOF = 1;
	}
	return isEOF;
}


void showbits_arincBCD(unsigned int x)
{
	int i;
	for(i=(sizeof(int)*8)-1; i>=0; i--)
	{
		(x&(1<<i))?putchar('1'):putchar('0');
		if (i == 1)
			printf(" ");
		if (i == 3)
			printf(" ");
		if (i == 6)
			printf(" ");
	if (i == 10)
			printf(" ");
	if (i == 14)
			printf(" ");
	if (i == 18)
			printf(" ");
	if (i == 22)
			printf(" ");
	if (i == 24)
			printf(" ");
    }
}


void showbits_arincBNR(unsigned int x)
{
	int i;
	for(i=(sizeof(int)*8)-1; i>=0; i--)
	{
		(x&(1<<i))?putchar('1'):putchar('0');
		if (i == 1)
			printf(" ");
		if (i == 3)
			printf("     ");
		if (i == 22)
			printf(" ");
		if (i == 24)
			printf(" ");
    }
```

```
}

void showbits(unsigned int x)
{
	int i;
	for(i=(sizeof(int)*8)-1; i>=0; i--)
	{
		(x&(1<<i))?putchar('1'):putchar('0');
	}
	printf("\n");
}

void read_db(unsigned int* label,
		int* nbit_used,
		unsigned int* codification)
{
	FILE * fp_db;
	int isEOF = 0;
	int found = 0;

    //Readed values
    unsigned int label_db;
    unsigned int codification_db;
    unsigned int nbit_used_db;

    fp_db = NULL;
	fp_db = fopen("db.csv","r");
	if( fp_db == NULL )
	{
	   printf("Error while opening the db file.\n");
	}
	else
	{
		while (isEOF != -1)	//while lines in file
		{
			isEOF = fscanf(fp_db, "%u;%*u;%*u;%*u;%d;%*f;%u\n",
						&label_db,
						&nbit_used_db,
						&codification_db);
			if(isEOF != -1)
			{
				if (label_db ==	*label)
				{
					*codification = codification_db;
					*nbit_used = nbit_used_db;

					found = 1;
					//further checks to prevent errors
					if ( *nbit_used < 0 || 19 < *nbit_used )
					{
						//nbit_used is in [0, 19] interval
						*nbit_used = 19;
					}
				}
			}
		}
		if (found == 0)
		{
			//if label not found, set everything to cero
			*label = 0;
			*nbit_used = 0;
			*codification = 0;
		}
		if (isEOF == -1)
		{
			isEOF = 1;
		}
		fclose(fp_db);
	}
```

```
}

unsigned int dcodify_label(unsigned int label)
{
        unsigned int n[3] = {0,0,0};
        unsigned int dec = 0;

        //num octal = n0n1n2
        n[0] = (label >> 30) & 3;
        n[1] = (label >> 27) & 7;
        n[2] = (label >> 24) & 7;

        dec = n[0] * 64 + n[1] * 8 + n[2];
        return dec;
}


int reverse_bits(int v)
{
      int c = 0;
      c = (BitReverseTable256[v & 0xff] << 24) |
          (BitReverseTable256[(v >> 8) & 0xff] << 16) |
          (BitReverseTable256[(v >> 16) & 0xff] << 8) |
          (BitReverseTable256[(v >> 24) & 0xff]);
      return c;
}


unsigned int dcodify_bcd(unsigned int arincMsg)
{
      unsigned int data = 0;
      unsigned int ch[5] = {0,0,0,0,0};
      int i;

      ch[0] = ((unsigned int)reverse_bits( (arincMsg & 0x38) >> 3 )) >> 29 ;
      ch[1] = ((unsigned int)reverse_bits( (arincMsg & 0x3C0) >> 6 )) >> 28;
      ch[2] = ((unsigned int)reverse_bits(
                  (arincMsg & 0x3C00) >> 10 )) >> 28;
      ch[3] = ((unsigned int)reverse_bits(
                  (arincMsg & 0x3C000) >> 14 )) >> 28;
      ch[4] = ((unsigned int)reverse_bits(
                  (arincMsg & 0x3C0000) >> 18 )) >> 28;

      if(ch[4] != 0)
      {
            data = ch[0] * 10000 + ch[1] * 1000 + ch[2] * 100 + ch[3] * 10
                        + ch[4];
      }
      else
      {
            if(ch[3] != 0)
            {
                  data = ch[0] * 1000 + ch[1] * 100 + ch[2] * 10 + ch[3];
            }
            else
            {
                  if(ch[2] != 0)
                  {
                        data = ch[0] * 100 + ch[1] * 10 + ch[2];
                  }
                  else
                  {
                        if(ch[1] != 0)
                        {
                              data = ch[0] * 10 + ch[1];
                        }
                        else
                        {
                              data = ch[0];
                        }
```

```
                    }
                }
        }
        return data;
}

unsigned int dcodify_bnr(unsigned int arincMsg, int nbit_used)
{
        unsigned int data;

        data = arincMsg & 0x00CFFFF7;
        data = data >> 3;
        data = (unsigned int) reverse_bits(data);
        data = data >> (32 - nbit_used);
        return data;
}
```

Código 3: arinc-Rx-func.c

arinc-Rx.h

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <arpa/inet.h>
#include <sys/time.h>
#include <signal.h>
#include <limits.h>

#define MAXBUF 4
#define VECTOR 3
#define BCD 0
#define BNR 1
#define FREE 10
#define WAITING_TX 11

/*ARINC-429 Packet*/
struct arinc429
{
       unsigned int label;
       unsigned int sdi;
       int data;
       unsigned int ssm;
       unsigned int parity;
};

/*Functions used*/
int read_packt(FILE ** fp, unsigned int * label, int * value);
void showbits_arincBCD(unsigned int x);
void showbits_arincBNR(unsigned int x);
int r_serialize_int(unsigned char *buffer);
void showbits(unsigned int x);
unsigned int dcodify_label(unsigned int label);
void read_db(unsigned int*       label,
             int*                nbit_used,
             unsigned int*       codification);
int reverse_bits(int v);
unsigned int dcodify_bcd(unsigned int arincMsg);
unsigned int dcodify_bnr(unsigned int arincMsg, int nbit_used);
static const unsigned char BitReverseTable256[] =
{
  0x00, 0x80, 0x40, 0xC0, 0x20, 0xA0, 0x60, 0xE0, 0x10, 0x90, 0x50, 0xD0, 0x30, 0xB0,
0x70, 0xF0,
  0x08, 0x88, 0x48, 0xC8, 0x28, 0xA8, 0x68, 0xE8, 0x18, 0x98, 0x58, 0xD8, 0x38, 0xB8,
0x78, 0xF8,
  0x04, 0x84, 0x44, 0xC4, 0x24, 0xA4, 0x64, 0xE4, 0x14, 0x94, 0x54, 0xD4, 0x34, 0xB4,
0x74, 0xF4,
  0x0C, 0x8C, 0x4C, 0xCC, 0x2C, 0xAC, 0x6C, 0xEC, 0x1C, 0x9C, 0x5C, 0xDC, 0x3C, 0xBC,
0x7C, 0xFC,
  0x02, 0x82, 0x42, 0xC2, 0x22, 0xA2, 0x62, 0xE2, 0x12, 0x92, 0x52, 0xD2, 0x32, 0xB2,
0x72, 0xF2,
  0x0A, 0x8A, 0x4A, 0xCA, 0x2A, 0xAA, 0x6A, 0xEA, 0x1A, 0x9A, 0x5A, 0xDA, 0x3A, 0xBA,
0x7A, 0xFA,
  0x06, 0x86, 0x46, 0xC6, 0x26, 0xA6, 0x66, 0xE6, 0x16, 0x96, 0x56, 0xD6, 0x36, 0xB6,
0x76, 0xF6,
  0x0E, 0x8E, 0x4E, 0xCE, 0x2E, 0xAE, 0x6E, 0xEE, 0x1E, 0x9E, 0x5E, 0xDE, 0x3E, 0xBE,
0x7E, 0xFE,
  0x01, 0x81, 0x41, 0xC1, 0x21, 0xA1, 0x61, 0xE1, 0x11, 0x91, 0x51, 0xD1, 0x31, 0xB1,
0x71, 0xF1,
  0x09, 0x89, 0x49, 0xC9, 0x29, 0xA9, 0x69, 0xE9, 0x19, 0x99, 0x59, 0xD9, 0x39, 0xB9,
0x79, 0xF9,
  0x05, 0x85, 0x45, 0xC5, 0x25, 0xA5, 0x65, 0xE5, 0x15, 0x95, 0x55, 0xD5, 0x35, 0xB5,
0x75, 0xF5,
```

```
  0x0D, 0x8D, 0x4D, 0xCD, 0x2D, 0xAD, 0x6D, 0xED, 0x1D, 0x9D, 0x5D, 0xDD, 0x3D, 0xBD,
0x7D, 0xFD,
  0x03, 0x83, 0x43, 0xC3, 0x23, 0xA3, 0x63, 0xE3, 0x13, 0x93, 0x53, 0xD3, 0x33, 0xB3,
0x73, 0xF3,
  0x0B, 0x8B, 0x4B, 0xCB, 0x2B, 0xAB, 0x6B, 0xEB, 0x1B, 0x9B, 0x5B, 0xDB, 0x3B, 0xBB,
0x7B, 0xFB,
  0x07, 0x87, 0x47, 0xC7, 0x27, 0xA7, 0x67, 0xE7, 0x17, 0x97, 0x57, 0xD7, 0x37, 0xB7,
0x77, 0xF7,
  0x0F, 0x8F, 0x4F, 0xCF, 0x2F, 0xAF, 0x6F, 0xEF, 0x1F, 0x9F, 0x5F, 0xDF, 0x3F, 0xBF,
0x7F, 0xFF
};
```

Código 4: arinc-Rx.h

arinc-Tx.c

```
#include "arinc-Tx.h"

/*Main*/
int main (int argc, char * argv[])
{
	char file_name[] = "input-file.txt";
	int returnStatus;
	int status = 0;
	struct arinc429 arinc429_values;
	struct arinc429 * sp = NULL;
	struct sockaddr_in udp_clt;
	struct sigaction act;
	unsigned int codification;
	int nbit_used;

	if (argc < 3)
	{
		fprintf(stderr, "\nUsage: %s <ip address> <port>\n", argv[0]);
		fprintf(stderr, "  <ip address> : \
			IP address of the arinc-Rx machine\n");
		fprintf(stderr, "  <port>       : \
			listening port of the arinc-Rx machine\n");
		fprintf(stderr, "  <debug>      : \
			optional, takes values from 0 to 3 \n\n");
		exit(1);
	}
	if ( argc >= 4)
	{
		debug = atoi(argv[3]);
		if ( debug < 0 || debug > 4 )
		{
			debug = 1;
		}
		if ( debug == 4 )
		{
			status = scheduling_ini (argc, argv);
			sch_num = atoi(argv[4]);
			if (status == 1)
			{
				printf("Error: Scheduling wont be applied\n");
			}
		}
	}
	else
	{
		// By default debug = 0
		debug = 0;
	}
	if ( debug != 5 || status == 1)
	{
		//if there is no scheduling or
		//the scheduling went wrong
		sch_num = 3;
		tiempo_ms = (int *) calloc (sch_num, sizeof(int));
		*tiempo_ms = 45000;
		*(tiempo_ms + 1) = 45000;
		*(tiempo_ms + 2) = 45000;
	}

	sp = &arinc429_values;
	msg_id = 0;

	// Create signals and interruption handler
	act.sa_handler = tx_arinc;
	act.sa_flags = 0;
	sigemptyset(&act.sa_mask);
	sigaddset(&act.sa_mask, SIGINT);
```

```
	sigaction(SIGALRM, &act, NULL);

	// Create a socket
	udp_s = socket ( AF_INET, SOCK_DGRAM, 0);
	if (udp_s == -1)
	{
		fprintf(stderr, "Could not create a socket!\n");
		exit(1);
	}

	udp_clt.sin_family = AF_INET;
	udp_clt.sin_addr.s_addr = INADDR_ANY;
	udp_clt.sin_port = 0;

	// Bind the socket to a port
	returnStatus = bind ( udp_s, (struct sockaddr *) &udp_clt,
					sizeof(udp_clt));
	if (returnStatus != 0)
	{
		fprintf(stderr, "Could not bind to address!\n");
		close(udp_s);
		exit(1);
	}

	// Give the arinc-Rx info to the socket
	udp_srv.sin_family = AF_INET;		// AF_INET = IPv4
	udp_srv.sin_addr.s_addr = inet_addr(argv[1]); // IP_addr
	udp_srv.sin_port = htons(atoi(argv[2]));	// Port

	// Open the input file, read first packet
	// and set interruption timer
	fp = fopen(file_name,"r");
	if( fp == NULL )
	{
		printf("Error while opening the file.\n");
	}

	//The first packet is readed the next packets
	// will be readed in the interruption rutine
	stop = read_packt(&fp, &(sp->label), &(sp->data));

	read_db(&(sp->label),
			&(sp->data),
		&(sp->sdi),
		&(sp->ssm),
		&nbit_used,
		&codification);

	if (codification == 1)
	{
		arincMsg = codify_bnr(sp, nbit_used);
	}
	if (codification == 0)
	{
		arincMsg = codify_bcd(sp);
	}

	serialize_int(buf, arincMsg);	//put arincMsg in Tx buffer
	ualarm(45000, 0);			//set timer

	// Wait for the interruptions to happen
	// till stop is setted to 1 (this will
	// happen when input-file EOF is reached)
	while(stop != 1);

	// Close the socket
	close(udp_s);
	return 0;
}
```

```
/*Handler function*/
void tx_arinc()
{
	unsigned int codification;
	int nbit_used;

	struct arinc429 arinc429_values;
	struct arinc429 * sp = &arinc429_values;

	sendto (udp_s, buf, MAXBUF, 0,
		(struct sockaddr *) &udp_srv, sizeof(udp_srv));

	//Reads a new packet (if exists, otherwise it stops the program)
	stop = read_packt(&fp, &(sp->label), &(sp->data));

	if (stop != 1)
	{
		// If there is a new packet
		// set next timer
		//ualarm(tiempo_ms[msg_id], 0);
		ualarm( *(tiempo_ms + msg_id ), 0);
		if (msg_id == sch_num - 1)
		{
			msg_id = 0;
		}
		else
		{
			msg_id++;
		}

		// Check the configuration for the label
		read_db(&(sp->label),
				&(sp->data),
		&(sp->sdi),
		&(sp->ssm),
		&nbit_used,
		&codification);

		if (codification == 1)
		{
			arincMsg = codify_bnr(sp, nbit_used);
		}
		else
		{
			arincMsg = codify_bcd(sp);
		}
		serialize_int(buf, arincMsg);    //put arincMsg in Tx buffer
	}
	else
	{
		fclose(fp);
	}
}
```

Código 5: arinc-Tx.c

arinc-Tx-func.c

```
//FUNCTIONS
#include "arinc-Tx.h"
void serialize_int(unsigned char *buffer, int value)
{
  //Saves int into a char for
  //socket communications
  buffer[0] = value >> 24;
  buffer[1] = value >> 16;
  buffer[2] = value >> 8;
  buffer[3] = value;
}


int read_packt(FILE ** fp, unsigned int * label, float * value)
{
       int isEOF = 0;
       isEOF = fscanf(*fp, "%u %f\n", label, value );
       if (isEOF == -1)
       {
              isEOF = 1;
       }
       return isEOF;
}


void showbits_arincBCD(unsigned int x)
{
       int i;
       for(i=(sizeof(int)*8)-1; i>=0; i--)
       {
              (x&(1<<i))?putchar('1'):putchar('0');
              if (i == 1)
                     printf(" ");
              if (i == 3)
                     printf(" ");
              if (i == 6)
                     printf(" ");
              if (i == 10)
                     printf(" ");
              if (i == 14)
                     printf(" ");
              if (i == 18)
                     printf(" ");
              if (i == 22)
                     printf(" ");
              if (i == 24)
                     printf(" ");
    }
}


int reverse_bits(int v)
{
       int c = 0;
       c = (BitReverseTable256[v & 0xff] << 24) |
           (BitReverseTable256[(v >> 8) & 0xff] << 16) |
           (BitReverseTable256[(v >> 16) & 0xff] << 8) |
           (BitReverseTable256[(v >> 24) & 0xff]);
       return c;
}


unsigned int codify_bcd(struct arinc429 * sp)
{
       unsigned int pkt = 0;
       int data = (int) sp->data;
```

```
unsigned int ch[5] = {0,0,0,0,0};
int i;

if (sp->ssm != 2)                //ssm != not computed data
{
       if (data >= 0)            //if positive ssm = 00
       {
              sp->ssm = 0;
       }
       else
       {
              sp->ssm = 3;       //if negative ssm = 11
              data = data * (-1); //change it to positive
       }
}

// The SSM would be codified later since if the digits of the number
// are of 5 digits and the first one is higher than 7 then
// ssm=not computed data

if ( data > 9999 )               // data has 5 digits
{
       if (data /10000 <= 7)
       {
              ch[0] = data / 10000;
              ch[1] = (data - ch[0] * 10000) / 1000;
              ch[2] = (data - ch[0] * 10000 - ch[1] * 1000) / 100;
              ch[3] = (data - ch[0] * 10000 - ch[1] * 1000
                                  - ch[2] * 100) / 10;
              ch[4] = (data - ch[0] * 10000 - ch[1] * 1000
                                  - ch[2] * 100 - ch[3] * 10) / 1;
       }
       else
       {
              sp->ssm = 2;
       }
}
else
{
       if ( 9999 >= data && data > 999 )      // data has 4 digits
       {
              if ( data / 1000 <= 7 )
              {
                     ch[0] = data / 1000;
                     ch[1] = (data - ch[0] * 1000) / 100;
                     ch[2] = (data - ch[0] * 1000 - ch[1] * 100) / 10;
                     ch[3] = (data - ch[0] * 1000 - ch[1] * 100
                                         - ch[2] * 10) / 1;
              }
              else         // if first digit > 7 starts on ch[1]
              {
                     ch[1] = data / 1000;
                     ch[2] = (data - ch[1] * 1000) / 100;
                     ch[3] = (data - ch[1] * 1000 - ch[2] * 100) / 10;
                     ch[4] = (data - ch[1] * 1000 - ch[2] * 100
                                         - ch[3] * 10) / 1;
              }
       }
       else
       {
              if ( 999 >= data && data > 99 )  // data has 3 digits
              {
                     if (data / 100 <= 7)      //first digit<7 (111)
                     {
                            ch[0] = data / 100;
                            ch[1] = (data - ch[0] * 100) / 10;
                            ch[2] = (data - ch[0] * 100
                                         - ch[1] * 10) / 1;
                     }
                     else
```

```
                        {
                                ch[1] = data / 100;
                                ch[2] = (data - ch[1] * 100) / 10;
                                ch[3] = (data - ch[1] * 100
                                        - ch[2] * 10) / 1;
                        }
                }
                else
                {
                        if ( 99 >= data && data > 9 ) // data has 2 digits
                        {
                                if (data / 10 <= 7)
                                {
                                        ch[0] = data / 10;
                                        ch[1] = (data - ch[0] * 10) / 1;
                                }
                                else
                                {
                                        ch[1] = data / 10;
                                        ch[2] = (data - ch[1] * 10) / 1;
                                }
                        }
                        else
                        {
                                if ( 9 >= data  )   //data has 1 digit
                                {
                                        if (data <= 7)
                                        {
                                                ch[0] = data;
                                        }
                                        else
                                        {
                                                ch[1] = data;
                                        }
                                }
                        }
                }
        }
    }

    ch[0] = (unsigned int)reverse_bits(ch[0]);
    ch[1] = (unsigned int)reverse_bits(ch[1]);
    ch[2] = (unsigned int)reverse_bits(ch[2]);
    ch[3] = (unsigned int)reverse_bits(ch[3]);
    ch[4] = (unsigned int)reverse_bits(ch[4]);

    pkt += sp->ssm << 1;
    pkt += ch[0] >> 26;
    pkt += ch[1] >> 22;
    pkt += ch[2] >> 18;
    pkt += ch[3] >> 14;
    pkt += ch[4] >> 10;
    pkt += sp->sdi << 22;
    pkt += codify_label(sp->label);
    pkt += get_parity(pkt);

    if(debug >= 1)
    {
        if (debug == 2)
        {
                printf ("\t%u\n\n", sp->ssm);
        }
        printf ("ARINC429->   ");
        showbits_arincBCD(pkt);
        printf ("\t%#x\t%d\n", pkt, data);
    }

    return pkt;
}
```

```
unsigned int codify_bnr(struct arinc429 * sp, int nbit_used)
{
	unsigned int pkt = 0;
	unsigned int padding = 0;
	unsigned int reversed;
	int data = (int) sp->data;

	padding = 19 - nbit_used;
	reversed =  reverse_bits(data);

	//parity is calculated at the end
	if (sp->ssm != 0)
	{
		//if no error ssm = normal operation
		sp->ssm = 3;
	}
	pkt += sp->ssm << 1;
	if (data < 0)
	{
		//if negative
		pkt += ((reversed >> (10+padding)) >> 3) << 3;
	}
	else
	{
		//if positive
		pkt += reversed >> (10+padding);
	}
	pkt += sp->sdi << 22;
	pkt += codify_label(sp->label);
	pkt += get_parity(pkt);

	if(debug >= 1)
	{
		if (debug == 2)
		{
		    printf ("\t%u\n\n", sp->ssm);
		}
		printf ("ARINC429->   ");
		showbits_arincBNR(pkt);
		printf ("\t%#x\t%d\n", pkt, data);
	}
	return pkt;
}


void showbits_arincBNR(unsigned int x)
{
	int i;
	for(i=(sizeof(int)*8)-1; i>=0; i--)
	{
		(x&(1<<i))?putchar('1'):putchar('0');
		if (i == 1)
			printf(" ");
		if (i == 3)
			printf("     ");
		if (i == 22)
			printf(" ");
		if (i == 24)
			printf(" ");
	}
}

void showbits(unsigned int x)
{
	int i;
	for(i=(sizeof(int)*4)-1; i>=0; i--)
	{
		(x&(1<<i))?putchar('1'):putchar('0');
	}
```

```
	printf("\n");
}


void read_db(unsigned int* label,
		float* value,
		unsigned int* sdi,
		unsigned int* ssm,
		int* nbit_used,
		unsigned int* codification)
{
	FILE * fp_db;
	int isEOF = 0;
	int found = 0;

	//Readed values
	unsigned int label_db;
	unsigned int sdi_db;
	unsigned int codification_db;
	unsigned int nbit_used_db;
	int range_max;
	int range_min;
	double resolution;

	fp_db = NULL;
	fp_db = fopen("db.csv","r");
	if( fp_db == NULL )
	{
	   printf("Error while opening the db file.\n");
	}
	else
	{
		while (isEOF != -1)
		{
			isEOF = fscanf(fp_db, "%u;%u;%u;%u;%d;%lf;%u\n",
					&label_db,
					&sdi_db,
					&range_max,
					&range_min,
					&nbit_used_db,
					&resolution,
					&codification_db);
			if(isEOF != -1)
			{
				if (label_db ==	*label)
				{
					*sdi = sdi_db;
					*codification = codification_db;
					*nbit_used = nbit_used_db;

					if (debug == 2)
					{
						printf ("\nValores        : label
						\tsdi\tRmax\tRmin\tbitused
				\tresol\tcodif\tvalue\tssm
		\n");
						printf ("Valores leÃdos : %u\t%u\t%d
				\t%d\t%d"
							"\t%.2lf\t%u\t%.2f",
							label_db,
							*sdi,
							range_max,
							range_min,
							*nbit_used,
							resolution,
							*codification,
							*value);
					}
					found = 1;   //label is on db
```

```
                                //check if data is in range
                                if ( range_max >= *value &&
                                       *value >= range_min  )
                                {
                                       //resolution implementation
                                       *value = *value / resolution;

                                       //further checks to prevent errors
                                       if ( *nbit_used < 0
                                              || 19 < *nbit_used )
                                       {
                                              *nbit_used = 19;
                                       }
                                       if (*sdi > 3)
                                       {
                                              *sdi = 0;
                                       }
                                       if (*codification == BNR)
                                       {
                                              //ssm: normal operation
                                              *ssm = 3;
                                       }
                                       else
                                       {
                                              *ssm = 0;
                                       }
                                }
                                //if not in range
                                else
                                {
                                       //if wrong we set value to 0
                                       *value = 0;
                                       if (*codification == BCD)
                                       {
                                              //BCD: ssm = not computed data
                                              *ssm = 2;
                                       }
                                       if (*codification == BNR)
                                       {
                                              //BNR: ssm = failure warning
                                              *ssm = 0;
                                       }
                                }
                         }
                  }
            }
            if (found == 0)
            {
                  //if label not found, set everything to cero
                  *label = 0;
                  *value = 0;
                  *sdi = 0;
                  *ssm = 0;
                  *nbit_used = 0;
                  *codification = 0;
            }
            if (isEOF == -1)
            {
                  isEOF = 1;
            }
            fclose(fp_db);
      }
}

unsigned int codify_label(unsigned int label)
{
      unsigned int octal = 0;
      int n1, n2, n3;
      if (label <= 255)
      {
```

```
            //label has to be in [0, 255] interval
            //otherwise a 0 is returned
            n3 = label % 8;
            n2 = (label / 8) % 8;
            n1 = ((label / 8) / 8) % 8;

            if (debug == 3)
            {
                  printf("Label dec = %u, octal = %d%d%d \n",
                              label, n1, n2, n3);
                  printf("n1     =     "); showbits(n1);
                  printf("n2     =     "); showbits(n2);
                  printf("n3     =     "); showbits(n3);

                  printf("\nshift1 =     "); showbits_arincBCD(n1 << 30);
                  printf("\nshift2 =     "); showbits_arincBCD(n2 << 27);
                  printf("\nshift3 =     "); showbits_arincBCD(n3 << 24);
                  printf("\n            +__________________\
                        _____________________\n");
            }

            octal += n1 << 30;
            octal += n2 << 27;
            octal += n3 << 24;
      }
      return octal;
}

int get_parity(unsigned int n)
{
    int parity = 0;
    while (n)
    {
        parity = !parity;
        n = n & (n - 1);
    }
    return !parity;
}

int scheduling_ini(int argc, char ** argv)
{

      //The global var int* tiempo_ms is needed
      int i = 0;
      int status = 0;
      int num_elem = 0;
      int time_tot[atoi(argv[4])];
      if(argc < 3)
      {
            printf("Error:\n");
            printf("Usage: %s <num_elem> <p1> <p2> ... <p_n>\n", argv[0]);
            status = 1;
      }
      else
      {
            num_elem = atoi(argv[4]);
            if ( argc != num_elem + 5 )
            {
                  printf("Error: num_elem doesnt match \
                              with packet number\n");
                  status = 1;
            }
            else
            {
                  tiempo_ms = (int *) calloc( num_elem, sizeof(int));
                  if (tiempo_ms == NULL)
                  {
                        printf("Error: Could not reserve dynamic \
                              memory\n");
```

```
                        status = 1;
                }
                else
                {

                        for (i = 0; i < num_elem; i++)
                        {
                                time_tot[i] = atoi(argv[5+i]);
                        }

                        //transform time to elapsed time
                        *tiempo_ms = time_tot[0];
                        if (*tiempo_ms <= 0)
                        {
                                status = 1;
                                printf("Error: elapsed time \
                                                calculation\n");
                        }
                        for (i = 1; i < num_elem; i++)
                        {
                                *(tiempo_ms + i) = time_tot[i]
                                                        - time_tot[i-1];
                                *(tiempo_ms + i) *= 1000;
                                if( *(tiempo_ms + i) <= 0)
                                {
                                        printf("Error: elapsed time \
                                                        calculation\n");
                                        status = 1;
                                }
                        }
                }
        }
    }
    return status;
}
```

Código 6: arinc-Tx-func.c

arinc-Tx.h

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <arpa/inet.h>
#include <sys/time.h>
#include <signal.h>
#include <limits.h>

#define MAXBUF 4
#define VECTOR 3
#define BCD 0
#define BNR 1
#define FREE 10
#define WAITING_TX 11
/*ARINC-429 Packet*/
struct arinc429
{
       unsigned int label;
       unsigned int sdi;
       float data;
       unsigned int ssm;
       unsigned int parity;
};

/*Functions used*/
void tx_arinc();
void print_arinc429 (struct arinc429 * sp);
void serialize_int(unsigned char *buffer, int value);
int read_packt(FILE ** fp, unsigned int * label, float * value);
void showbits_arincBCD(unsigned int x);
void showbits_arincBNR(unsigned int x);
void showbits(unsigned int x);
int reverse_bits(int v);
unsigned int codify_bcd(struct arinc429 * sp);
unsigned int codify_bnr(struct arinc429 * sp, int nbit_used);
unsigned int codify_label(unsigned int label);
int get_parity(unsigned int n);
int scheduling_ini(int argc, char ** argv);
void read_db(unsigned int*                    label,
                          float*                              value,
             unsigned int*             sdi,
             unsigned int*             ssm,
             int*                      nbit_used,
             unsigned int*             codification);

//Gobal Vars
//Lookup table for bit reversing
static const unsigned char BitReverseTable256[] =
{
  0x00, 0x80, 0x40, 0xC0, 0x20, 0xA0, 0x60, 0xE0, 0x10, 0x90, 0x50, 0xD0, 0x30, 0xB0,
0x70, 0xF0,
  0x08, 0x88, 0x48, 0xC8, 0x28, 0xA8, 0x68, 0xE8, 0x18, 0x98, 0x58, 0xD8, 0x38, 0xB8,
0x78, 0xF8,
  0x04, 0x84, 0x44, 0xC4, 0x24, 0xA4, 0x64, 0xE4, 0x14, 0x94, 0x54, 0xD4, 0x34, 0xB4,
0x74, 0xF4,
  0x0C, 0x8C, 0x4C, 0xCC, 0x2C, 0xAC, 0x6C, 0xEC, 0x1C, 0x9C, 0x5C, 0xDC, 0x3C, 0xBC,
0x7C, 0xFC,
  0x02, 0x82, 0x42, 0xC2, 0x22, 0xA2, 0x62, 0xE2, 0x12, 0x92, 0x52, 0xD2, 0x32, 0xB2,
0x72, 0xF2,
  0x0A, 0x8A, 0x4A, 0xCA, 0x2A, 0xAA, 0x6A, 0xEA, 0x1A, 0x9A, 0x5A, 0xDA, 0x3A, 0xBA,
0x7A, 0xFA,
```

```
  0x06, 0x86, 0x46, 0xC6, 0x26, 0xA6, 0x66, 0xE6, 0x16, 0x96, 0x56, 0xD6, 0x36, 0xB6,
0x76, 0xF6,
  0x0E, 0x8E, 0x4E, 0xCE, 0x2E, 0xAE, 0x6E, 0xEE, 0x1E, 0x9E, 0x5E, 0xDE, 0x3E, 0xBE,
0x7E, 0xFE,
  0x01, 0x81, 0x41, 0xC1, 0x21, 0xA1, 0x61, 0xE1, 0x11, 0x91, 0x51, 0xD1, 0x31, 0xB1,
0x71, 0xF1,
  0x09, 0x89, 0x49, 0xC9, 0x29, 0xA9, 0x69, 0xE9, 0x19, 0x99, 0x59, 0xD9, 0x39, 0xB9,
0x79, 0xF9,
  0x05, 0x85, 0x45, 0xC5, 0x25, 0xA5, 0x65, 0xE5, 0x15, 0x95, 0x55, 0xD5, 0x35, 0xB5,
0x75, 0xF5,
  0x0D, 0x8D, 0x4D, 0xCD, 0x2D, 0xAD, 0x6D, 0xED, 0x1D, 0x9D, 0x5D, 0xDD, 0x3D, 0xBD,
0x7D, 0xFD,
  0x03, 0x83, 0x43, 0xC3, 0x23, 0xA3, 0x63, 0xE3, 0x13, 0x93, 0x53, 0xD3, 0x33, 0xB3,
0x73, 0xF3,
  0x0B, 0x8B, 0x4B, 0xCB, 0x2B, 0xAB, 0x6B, 0xEB, 0x1B, 0x9B, 0x5B, 0xDB, 0x3B, 0xBB,
0x7B, 0xFB,
  0x07, 0x87, 0x47, 0xC7, 0x27, 0xA7, 0x67, 0xE7, 0x17, 0x97, 0x57, 0xD7, 0x37, 0xB7,
0x77, 0xF7,
  0x0F, 0x8F, 0x4F, 0xCF, 0x2F, 0xAF, 0x6F, 0xEF, 0x1F, 0x9F, 0x5F, 0xDF, 0x3F, 0xBF,
0x7F, 0xFF
};

static FILE * fp = NULL;
static int stop = 0;
static unsigned int arincMsg = 0;
static int udp_s;
static unsigned char buf[MAXBUF] = "****";
static struct sockaddr_in udp_srv;

int debug;
int * tiempo_ms;
int sch_num;

int msg_id;
```

Código 7: arinc-Tx.h