Trabajo Fin de Grado
Grado en Ingeniería en Tecnologías Industriales

Librerías para identificación por radiofrecuencia y comunicación MODBUS TCP/IP en célula de fabricación flexible

Autor: Fernando Gutiérrez Arenas

Tutor: Dr. Luis Fernando Castaño Castaño

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

Sevilla, 2019

Trabajo Fin de Grado

Grado en Ingeniería en Tecnologías Industriales

Librerías para identificación por radiofrecuencia y comunicación MODBUS TCP/IP en célula de fabricación flexible

Autor:

Fernando Gutiérrez Arenas

Tutor:

Dr. Luis Fernando Castaño Castaño

Dpto. Ingeniería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2019

Trabajo Fin de Grado: Librerías para identificación por radiofrecuencia y comunicación MODBUS TCP/IP en célula de fabricación flexible

Autor: Fernando Gutiérrez Arenas
Tutor: Dr. Luis Fernando Castaño Castaño

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

A mi familia, por su apoyo incondicional.

A mis amigos, que me han ayudado y animado durante esta etapa.

A mi tutor, Fernando Castaño Castaño, por su paciencia, su ayuda y por la oportunidad de realizar este trabajo.

Fernando Gutiérrez Arenas

Sevilla, 2019

Resumen

Este Trabajo de Fin de Grado se ha desarrollado para comprobar la viabilidad del uso de la tecnología de identificación por radiofrecuencia o RFID, en el planteamiento de un método de gestión distribuido de las tareas que llevarán cabo cada una de las máquinas (estaciones de trabajo) de una célula de fabricación flexible.

Se ha escogido una célula de fabricación flexible compuesta por una serie de cintas transportadoras dispuestas en forma de rectángulo (sistema de transporte), por las que circulan unas bandejas metálicas que portan unos palets sobre los que se disponen las piezas objetivo de las operaciones a realizar por cada una de las estaciones de trabajo.

Tradicionalmente, la gestión de la automatización de la planta se realizada a partir de la gestión de un autómata central (autómata maestro) que se encarga de realizar la coordinación de la célula de fabricación mediante la comunicación con cada uno de los autómatas esclavos que se ocupan de las estaciones de trabajo. Esta centralización obliga a desarrollar una comunicación maestro-esclavo muy sofisticada si realmente se pretende dotar de flexibilidad a la aplicación de la célula de fabricación. Las operaciones de sincronización y gestión de fallos se complican de forma extraordinaria con este esquema.

Mediante la tecnología RFID con etiquetas incorporadas a las bandejas y puntos de lectura/escritura situados y controlados por las estaciones de trabajo se pretende simplificar las operaciones de coordinación y sincronización de las tareas. Así, la bandeja en su salida del almacén llevará incorporada como información las tareas que deben ser realizadas sobre la misma que le asigna el autómata maestro, al pasar la bandeja por cada estación de trabajo, un punto de lectura obtendrá las operaciones a realizar sobre el palet o piezas de la bandeja, realizará el trabajo especificado en la etiqueta RFID y posteriormente escribirá en la etiqueta la información de lo que se ha realizado, así como información de fallos o incidencias si no se ha podido llevar a cabo todas las operaciones de esa célula.

Con todo esto, se consigue un sistema más autónomo y en el que se tiene un mayor control de los procesos que se han realizado sobre la bandeja, simplificado enormemente la gestión del autómata maestro.

Para el manejo de las etiquetas RFID se ha hecho uso de unos módulos basados en el chip MIFARE MFRC522 que se encargan de la lectura y escritura de los mismos. El enlace entre estos módulos y el PLC Modicon M340 se ha hecho con placas Arduino las cuales mediante comunicación SPI, aparte de controlar los módulos RFID, también controlan un módulo de Ethernet que se encarga de la comunicación con el PLC por MODBUS TCP/IP.

Abstract

This End of Degree Project has been developed to verify the viability of using radio frequency identification technology or RFID, in the approach of a distributed management method of the tasks carried out by each of the machines (work stations) of a flexible manufacturing cell.

A flexible manufacturing cell composed of a series of conveyor belts has been chosen arranged in the shape of a rectangle (transport system), through which metal trays carry some pallets on which the objective pieces of the operations to be carried out by each of the work stations are arranged. Traditionally, the automation management of the plant was carried out based on the management of a central automaton (master automaton) which is responsible for coordinating the manufacturing cell by communicating with each of the slave automats that deal with the workstations.

This centralization requires the development of a very sophisticated master-slave communication if it is really intended to give flexibility to the application of the manufacturing cell. The synchronization and fault management operations are complicated in an extraordinary way with this scheme.

Through RFID technology with tags incorporated into trays and read / write points located and controlled by the work stations is intended to simplify the operations of coordination and synchronization of tasks. Thus, the tray in its output from the store will be incorporated as information the tasks that must be performed on the same as assigned by the master automaton, when passing the tray for each work station, a reading point will obtain the operations to be performed on the pallet or pieces of the tray, will perform the work specified in the RFID tag and later write on the label the information of what has been done, as well as information of failures or incidents if it has not been possible to carry out all the operations of that cell.

With all this, a more autonomous system is achieved and in which there is greater control of the processes that have been carried out on the tray, greatly simplifying the management of the master automaton.

For the handling of RFID tags, modules based on the MIFARE MFRC522 chip have been used, which are the responsibles for reading and writing them. The link between these modules and the Modicon M340 PLC has been made with Arduino boards which, by means of SPI communication, apart from controlling the RFID modules, also control an Ethernet module that is responsible for the communication with the PLC by MODBUS TCP / IP.

Índice Abreviado

Índice

1 Introducción

El objetivo de este trabajo ha sido comprobar si existía la posibilidad de que la gestión del sistema actual de cintas transportadoras que se encuentra en el laboratorio de robótica y control de la Escuela Técnica Superior de Ingeniería se haga de manera distribuida. Para probar la viabilidad de dicho objetivo, se ha recurrido a un sistema de identificación mediante radio frecuencia o RFID en sus siglas en inglés con el cual se consigue intercambiar información entre distintos subsistemas sin que haya contacto físico entre ellos.

1.1 Sistema actual

La célula de fabricación flexible sobre la que se ha desarrollado el proyecto se encuentra en el ala este de la planta baja del laboratorio 1 de la escuela técnica superior de ingeniería de Sevilla.

La célula de fabricación está compuesta por una serie de estaciones de trabajo y almacenes, cada uno controlado por un PLC Modicon M340 de Schneider Electric (PLC esclavos). Las piezas a tratar o fabricar se desplazan de una estación a otra a través de un sistema de transporte (manejado por el PLC maestro) formado por cintas transportadoras que portan bandejas sobre las que se fijan palets de piezas. En la figura 1.1 se muestra un plano de la célula completa con todas las estaciones de trabajo y almacenes.

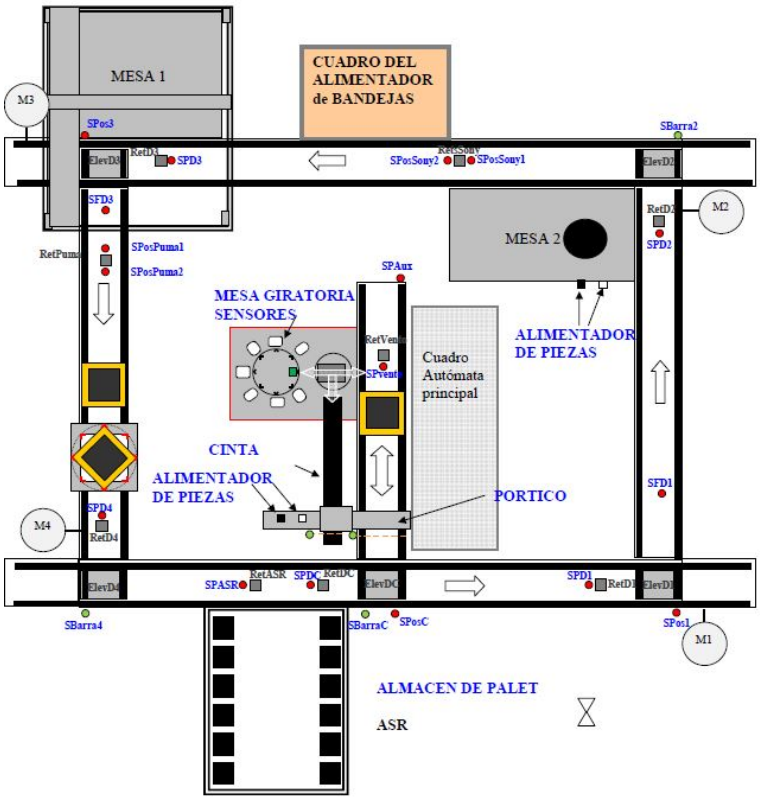

Figura 1.1 Esquema de la célula de fabricación flexible.

Las piezas disponibles para la célula de fabricación son cuatro, dependiendo de la forma y del material. Como se puede ver en la figura 1.2, las piezas pueden ser metálicas, en concreto de aluminio, o plásticas. La diferencia de forma está en la altura de cada una. La altura de las bajas es de 15mm y la de las altas 18mm. El diámetro es igual para todas, 35mm.

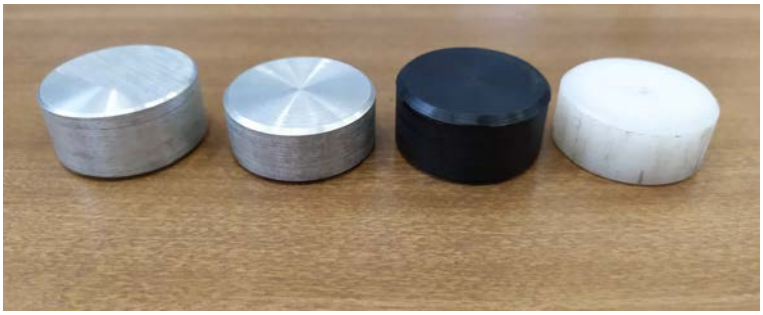

Figura 1.2 Tipos de piezas.

En la figura 1.3 se puede ver una bandeja con el palet 4 encima. Ese palet lleva unos delimitadores de aluminio que definen dos posiciones en los que puede llevar pieza. En el caso de la imagen, la pieza sería una pieza baja de plástico en la posición dos del palet. La posición libre sería la número uno.

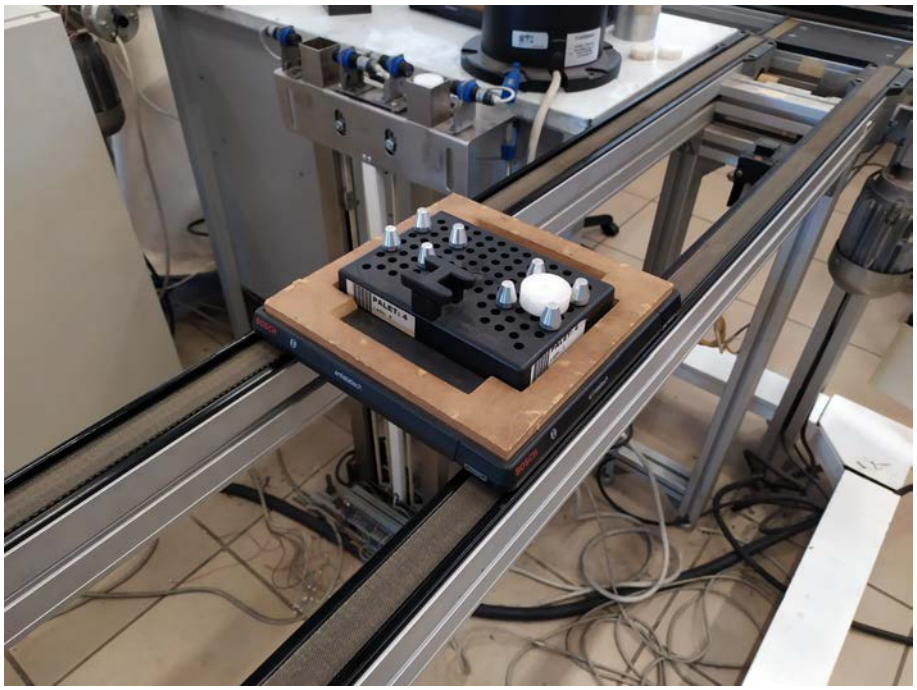

Figura 1.3 Bandeja con palet 4 y pieza en posición 2.

El almacén de bandejas o alimentador de bandejas, Figura 1.4, es el puesto que gestiona las bandejas vacías apiladas que se irán sirviendo o almacenando en función de la demanda de la instalación principal.

Figura 1.4 Almacén vertical de bandejas.

El almacén matricial está compuesto por 72 bahías de almacenamiento dispuestas en dos bancos de 6 x 6 enfrentados y un robot cartesiano con una pinza que trabaja entre los dos bancos sirviendo palets, Figura 1.5. Al robot original se le añadió un sensor de ocupación de las celdas.

Figura 1.5 Almacén matricial de palets.

El clasificador de piezas, Figura 1.7 es un plato giratorio sobre el que se coloca la pieza mediante una ventosa neumática. En el perímetro del plato se encuentran una serie de sensores ópticos y capacitivos para determinar el color, el material de que está hecha, etc. Para hacer llegar las piezas a este puesto se añadió la quinta cinta transportadora que compone el sistema, un pórtico para mover las piezas entre cintas y un alimentador de piezas.

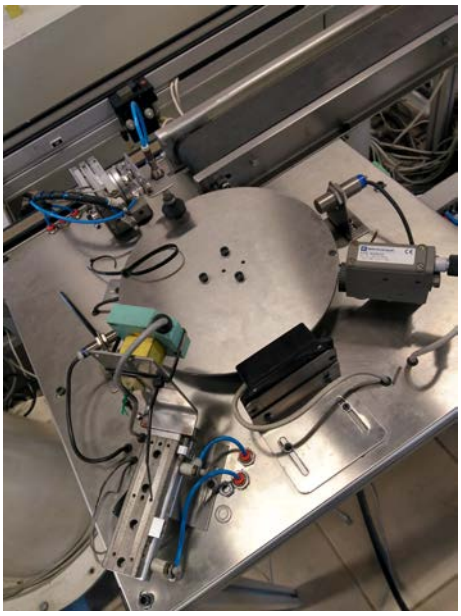

Figura 1.6 Clasificador de piezas.

El otro alimentador de piezas se encuentra junto al brazo robótico SCORBOT. Estos alimentadores están compuestos por dos columnas de piezas colocadas en vertical. Para subir y bajar la columna se utilizan motores paso a paso. Este alimentador es el encargado de suministrar o recibir piezas del robot SCORBOT, que las retira o las coloca en el palet detenido frente a él. Se puede ver el brazo Scorbot en azul y los dos

alimentadores de piezas verticales en la Figura 1.7.

Figura 1.7 Brazo robótico Scorbot y alimentador de piezas.

El último elemento del sistema es el robot SCARA, Figura 1.8. Este dispone de una cámara que permite la utilización de la visión artificial en el sistema. Además dispone de un actuador lineal que sube y baja para operar sobre los palets.

Figura 1.8 Robot Sony.

Todos estos sistemas están controlados individualmente por el PLC colocado en cada puesto, pero el PLC principal es el encargado de coordinarlos todos. Este PLC además, es el único encargado de mover las cintas, actuar sobre los retenedores, leer los sensores inductivos y mover las plataformas elevadoras. La idea es hacer que todos los puestos estén totalmente desacoplados y que solo dependan de si mismos al recibir qué hacer

de la propia bandeja.

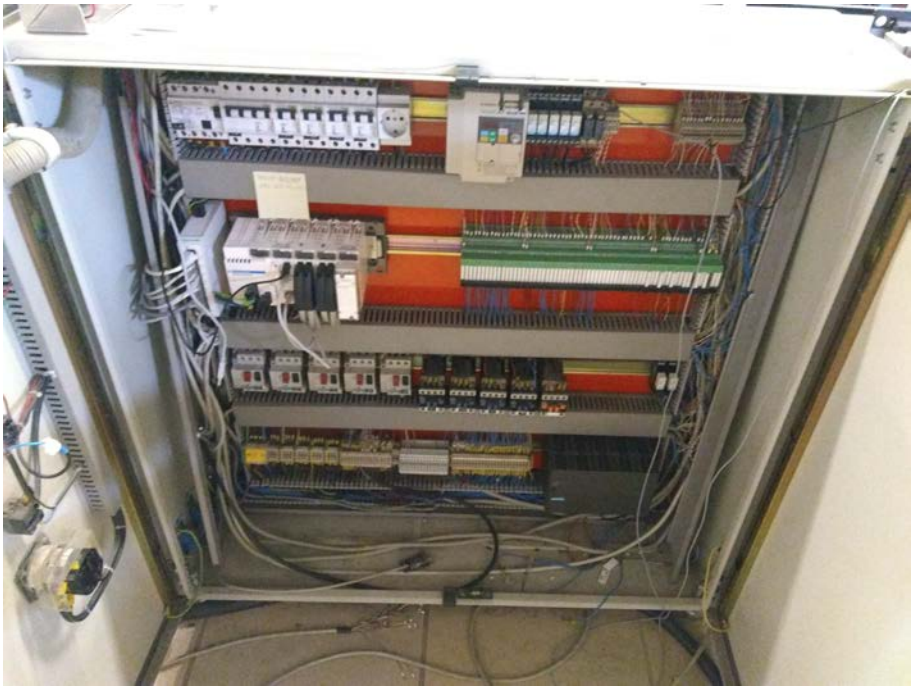

Figura 1.9 PLC Principal.

1.2 Sistema desarrollado

El sistema se ha ideado como un complemento para la célula de fabricación flexible detallada en 1.1. Su desarrollo se ha realizado sobre la plataforma Arduino gracias a la cantidad de periféricos disponibles, al coste de este sistema y a la cantidad de información disponible.

Uno de los múltiples periféricos disponibles es la placa MFRC522, la cual ha permitido que se realice una comunicación inalámbrica entre las bandejas y la placa Arduino gracias a la tecnología RFID. Sin embargo, solo con periféricos que se encarguen de la lectura y escritura mediante RFID no se puede realizar todo lo que se quería, ya que era necesario comunicarle esa información al PLC que se encarga de realizar las acciones que se han leído. Debido a que un PLC está pensado para un ambiente industrial, hace que sea imposible comunicarse con el con las comunicaciones nativas de la placa Arduino, por lo tanto hubo que buscar una nexo de unión entre estos dos sistemas. Un estándar de comunicación en la industria es la comunicación Modbus, la cual Arduino no puede realizarla tal cual, pero existen periféricos que permiten dicha comunicación. Se ha recurrido a una *shield* que permite la comunicación por Ethernet de la placa Arduino y así mediante un cable Ethernet se tiene un medio por el que realizar la comunicación MODBUS con el PLC. Con todo esto, el sistema estaría completo.

1.3 Ejemplo con ambos sistemas

En este apartado se pretende hacer una comparación entre el tratamiento de una bandeja mediante el sistema actual y el sistema propuesto para que se vean las diferencias entre uno y otro.

Se va a exponer un ejemplo en el que se ve como son los pasos de una tarea de la célula. En la figura 1.10 se pueden ver recuadradas en rojo las distintas zonas por las que pasará la bandeja.

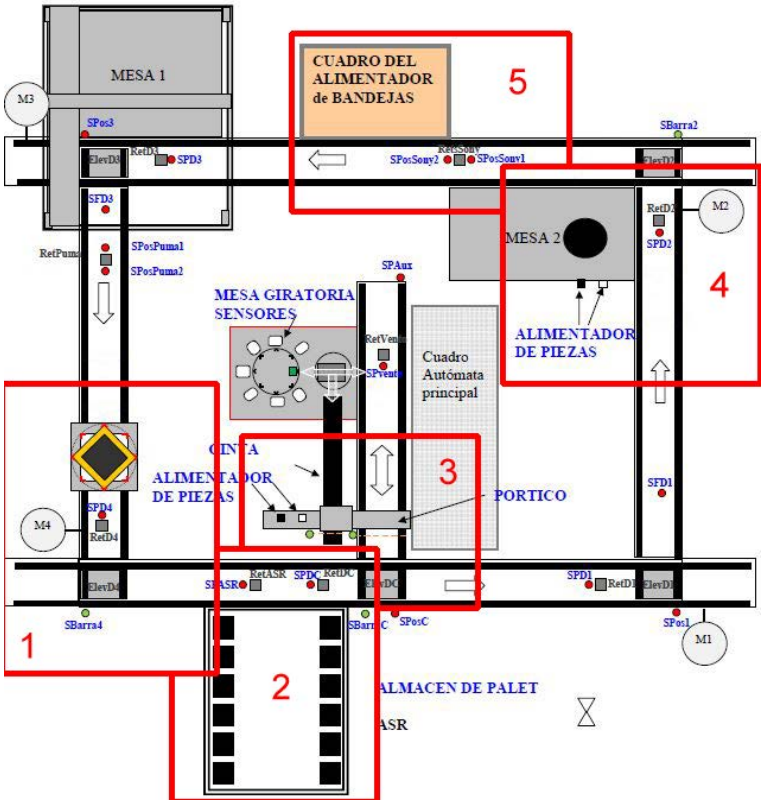

Figura 1.10 Esquema general de la planta por zonas.

En la figura 1.11 se puede ver como una bandeja acaba de salir del almacén de bandejas y se dirige al almacén de palets. Esta bandeja va sin palet ya que no se pueden guardar con nada encima.

Figura 1.11 Zona 1, almacén de bandejas.

En la figura 1.11 la bandeja ya está detenida frente al almacén de palets. Para este ejemplo se supondrá que se le carga el palet 5. Se puede ver en la imagen que el palet viene con unas piezas troncocónicas, las cuales delimitan las dos posibles posiciones dentro del palet. La posición más lejana y a la izquierda en la imagen, se define como posición uno y la más cercana a la derecha, como posición dos.

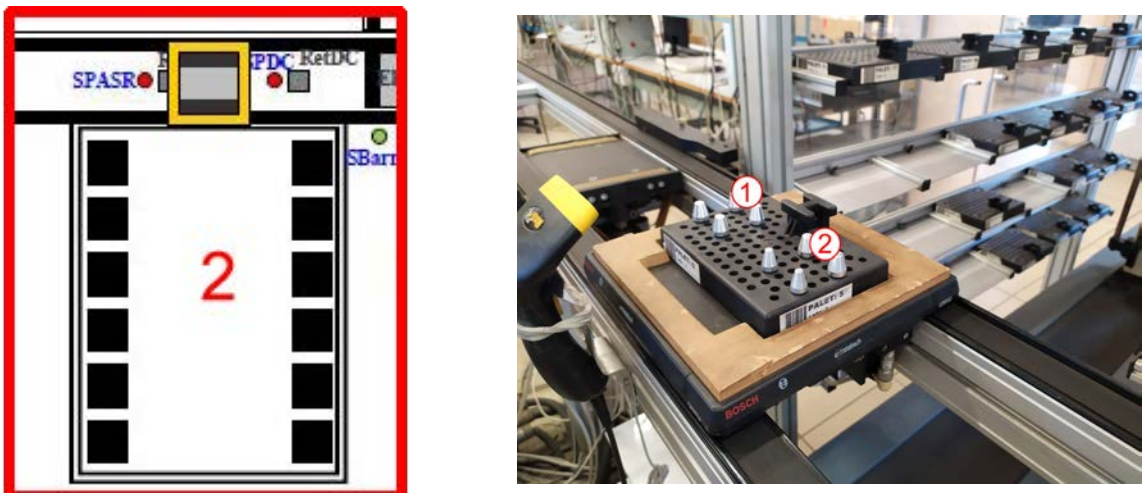

Figura 1.12 Zona 2, almacén de palets.

El siguiente paso que seguiría la bandeja sería parar en el pórtico para cargar una pieza. Esta situación es la representada en la figura 1.16. En este puesto se le pueden cargar y recoger piezas al palet. En este ejemplo se le carga una pieza de plástico de 15mm en la posición dos al palet.

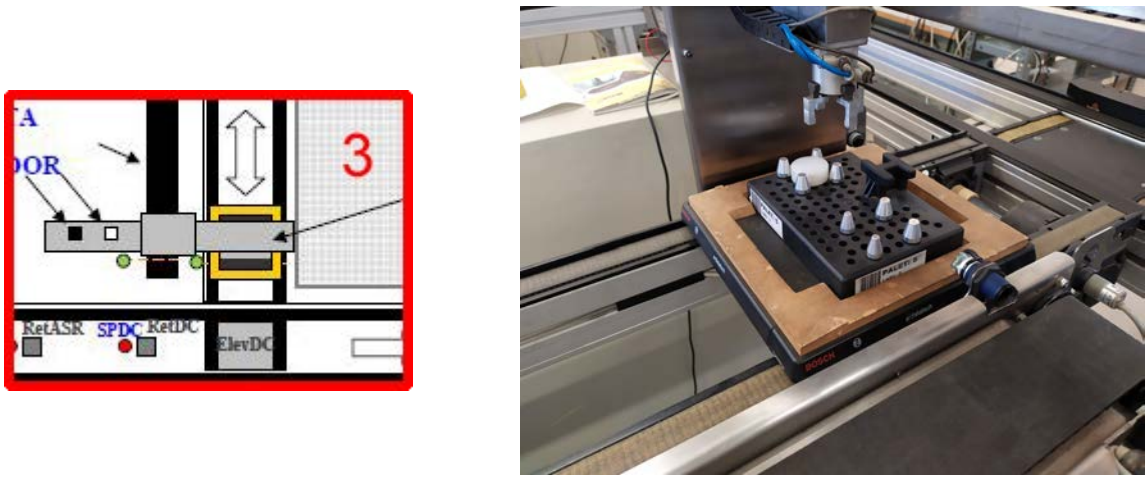

Figura 1.13 Zona 3, pórtico 1.

Una vez se le ha cargado la pieza a la bandeja, se devuelve esta a la cinta principal para que vaya al siguiente puesto, el que tiene el brazo robótico SCORBOT. En este punto se detallarán las diferencias que hay entre el sistema actual y el desarrollado en este documento.

Al llegar al SCORBOT en el sistema actual, el PLC principal debe llevar el seguimiento perfecto de la bandeja desde que sale hasta que llega al puesto. Este seguimiento se complica debido a que no tiene por qué haber una sola bandeja, sino que puede haber varias. Para saber donde está cada una, el PLC principal tiene que llevar el seguimiento del orden de salida de las bandejas y tener programado el orden de los sensores de la cinta para que según se vayan activando y desactivando, pueda calcular por donde va cada bandeja a lo

largo de la célula. En el ejemplo propuesto, el PLC principal tiene que llevar toda la información de lo que se ha hecho sobre la bandeja desde que se sacó. En este caso, que se colocó sobre ella el palet cinco y que a este se le ha cargado una pieza plástica baja en la posición uno. Según pase por los sensores, se detecta que la siguiente bandeja en pasar por el SCORBOT es de la que se quiere descargar una pieza. Una vez el PLC principal detecta que se activa el sensor de este puesto, levanta el retenedor.

Para ejemplificar un caso más real, se supondrá que la situación en la célula es la que se ve en la figura 1.14, con varias bandejas con palets y no solo una como se ha estado suoniendo hasta ahora. Al lado de cada bandeja está marcado el número de palet que se supondrá que llevan.

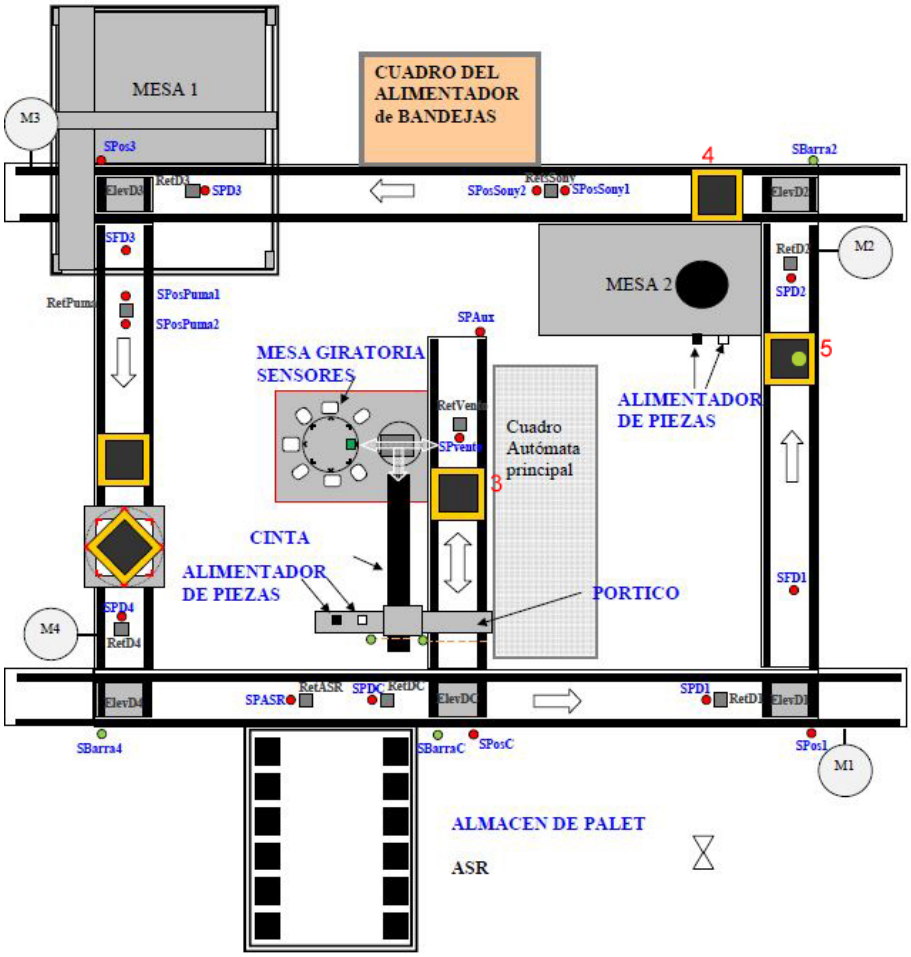

Figura 1.14 Esquema de seguimiento del palet 4.

En la tabla 1.1 se puede ver la información que tiene el PLC principal para llevar el seguimiento de las bandejas. Se considera el ejemplo planteado que se corresponde con la situación ilustrada en la figura 1.14.

Tabla 1.1 Seguimiento de bandejas del PLC principal.

Bandeja	Palet	Siguiente retenedor
1ª	Sin palet	Almacén de bandejas
2ª	4	SONY
3ª	5	SCORBOT
4ª	3	Comprobador

En el sistema propuesto, el PLC principal quedaría liberado de todo ese seguimiento, ya que al leerse la etiqueta RFID de la bandeja ya se sabe si se tiene que detener o no. Lo único que recibiría el PLC principal sería una orden en formato booleano para que activara el retenedor del puesto del SCORBOT. De esta forma ningún puesto hace el seguimiento de las bandejas, sino que cada una lleva guardadas sus intrucciones y va comunicandoselas a cada puesto.

El siguiente paso sería una vez parada la bandeja. En el caso del sistema actual, una vez se ha parado la bandeja el PLC principal tendría que comunicarle al PLC del puesto del SCORBOT lo que tiene que hacer sobre la bandeja, que sería descargar la pieza de la posición uno al almacén dos. A partir de este punto, la ejecución del programa del PLC del puesto sería igual que en el sistema propuesto, le pasaría la orden recibida al SCORBOT y cuando este terminara le daría la señal de que ya ha finalizado y puede liberar la bandeja. De nuevo, se ve como el PLC principal ha tenido que actuar para mandar la instrucción.

En el caso del sistema mediante RFID, al pararse la bandeja, el Arduino lee la etiqueta y le pasa la información directamente al PLC del puesto. Una vez el SCORBOT ha realizado la tarea, el PLC del puesto le pregunta a la etiqueta si tiene más instrucciones que se tengan que realizar en el SCORBOT, así hasta realizar todas las tareas consecutivas que haya desde la primera. Una vez se realizan todas, se le indica al PLC mediante el mismo booleano que indicaba que subiera el retenedor, que ahora lo tiene que bajar porque ya se han realizado todas las tareas en ese puesto.

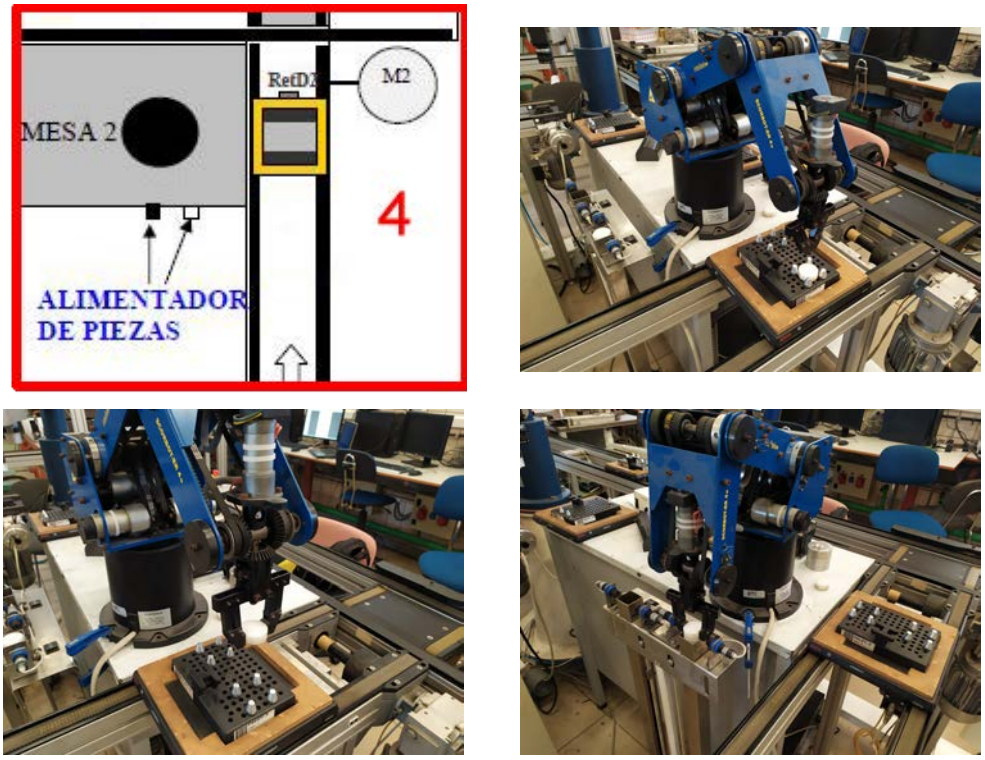

Figura 1.15 Zona 4, SCORBOT.

Una vez se ha descargado la pieza en el SCORBOT, habría que guardar el palet y en la siguiente vuelta a la célula guardar la bandeja en el almacén de bandejas. Antes de esto, la bandeja pasaría por el puesto del robot SONY, aunque no se realizaría ninguna acción. En este puesto está situado el sistema desarrollado en este documento.

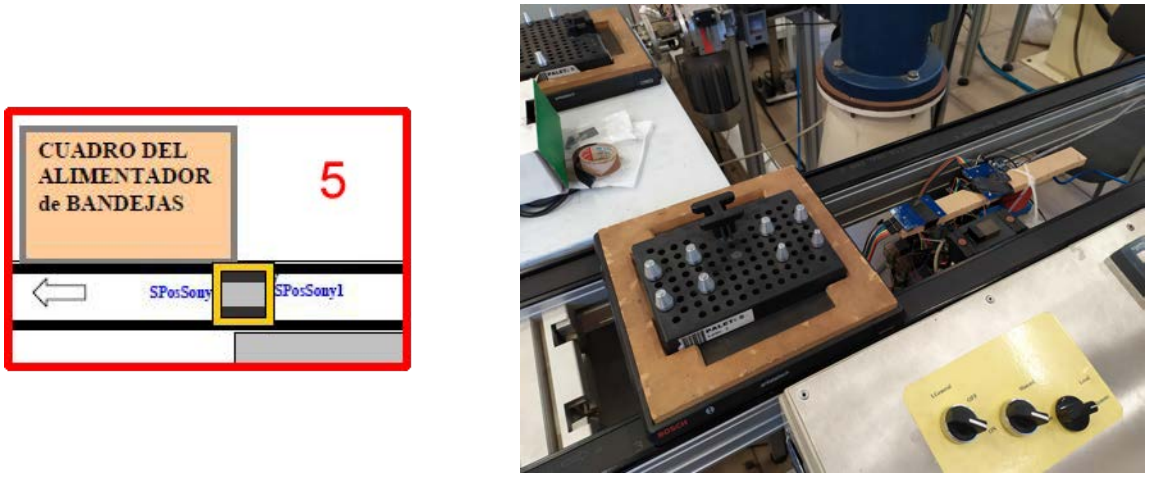

Figura 1.16 Zona 5, robot SONY.

2 Descripción de los elementos

2.1 Arduino Mega 2560 R3

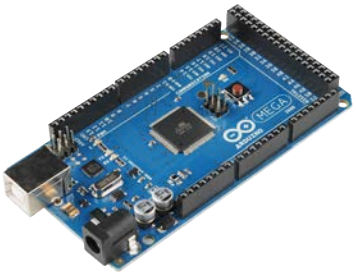

Figura 2.1 Arduino Mega 2560 R3.

La placa Mega 2560 R3 [1] es la tercera revisión de la placa de desarrollo creada por la plataforma Arduino basada en el chip de Atmel ATMega2560. Esta placa dispone de 54 pines de entrada/salida, 4 conexiones serie y soporta comunicación SPI e I2C. A parte la memoria de la placa se divide en, 256 KB de memoria flash, 8 KB de memoria SRAM y 4 KB de memoria EEPROM lo que permite el desarrollo de proyectos de mayor complejidad.

La disposición de las cuatro filas de pines de la placa Uno y del conector ICSP es idéntica en esta placa, por lo que la mayoría de placas módulares de la Uno son compatibles con la Mega 2560 R3. El mayor problema es que los pines para establecer la comunicación por SPI son distintos de la Uno, por lo que una placa modular que haga uso de esta comunicación y esté desarrollada para la Uno, deberá conectarse a través del conector ICSP para que sea compatible con la Mega 2560 R3.

La tensión de operación del microcontrolador es de 5V, pero gracias al convertidor reductor integrado, se puede alimentar la placa a través del conector de alimentación con una tensión de 6V hasta 20V, aunque lo recomendado sería alimentarla con una tensíon en el rango de los 7V a los 12V para que el convertidor funcione correctamente y no se caliente en exceso. Estos márgenes impiden trabajar con los 24V de alimentación de los PLC. Por otro lado, la corriente por pin es de 20mA.

2.2 Módulo Ethernet

Figura 2.2 Módulo Ethernet.

La Arduino Ethernet Shield V1 [2] permite conectar las placas Arduino a nuestra red local. Esta *shield* esta basada en el chip Wiznet W5100 [3] y dispone de un conector RJ45 en el que conectar el cable de ethermet desde nuestro Router o Switch. En el sistema, se conectará la *shield* a un switch para realizar la comunicación Modbus TCP con los distintos PLC.

Gracias a su diseño modular permite montarla sobre la placa Arduino y usarla sin necesidad de recurrir a más cables.

2.3 Módulo MFRC522

Figura 2.3 Módulo MFRC522.

El módulo RFID-RC522 está diseñado sobre el chip MIFARE MFRC522, de NXP, con los elementos necesarios para su funcionamiento (resistencias, condensadores y cristal de 27.12 MHz), una antena impresa en el propio PCB y ocho pines de los cuales dos son para alimentación (3.3V y GND), cinco se usan para la comunicación SPI (SDA, SCK, MOSI, MISO, RST) y un pin (IRQ) que actúa como interrupción para despertar a la placa arduino, pero que no será necesario en este proyecto.

El microcontrolador de este módulo está diseñado para realiza la comunicación a distancia a una frecuencia de 13.56 MHz. La comunicación con el Arduino a parte de por SPI también sería posible realizarla mediante I2C y UART. La distancia de lectura/escritura que se consigue con este módulo y las etiquetas MIFARE Classic es de aproximadamente 15 mm.

Para obtener información adicional se puede consultar el Datasheet correspondiente al chip [4]

2.4 Etiquetas MIFARE Classic 1K

Figura 2.4 Etiquetas RFID.

Las etiquetas RFID están compuestas por un chip y una antena que ocupa la mayor parte de la etiqueta. En el caso de las etiquetas activas, estas disponen además de una batería que permite un mayor rango de acción, pero limita su tiempo de vida a la duración de la batería. En cambio, las etiquetas RFID pasivas como las que se usarán en este sistema, no necesitan batería ya que obtienen la energía para funcionar del propio lector mediante radio frecuencia. Esto permite un menor mantenimiento y coste.

Las etiquetas MIFARE Classic 1K [5] disponen de 1024 bytes de memoria divididos en 16 sectores. Cada sector está protegido por dos claves (A y B) que van almacenadas en cada sector. Estas claves junto con las condiciones de acceso a la información ocupan 16 bytes por sector. A parte, el primer sector es de solo lectura y en él viene almacenada información del fabricante, como la ID de la tajeta. Debido a esto la capacidad real de almacenar información en la tarjeta se reduce a 752 bytes.

En la Figura 2.5 se puede ver la distribución de la memoria de la tarjeta MIFARE Classic.

		Número de byte dentro de cada bloque															
Sector	Bloque	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Descripción
0	0	Información del fabricante																Solo lectura
	1																	Información
	2																	Información
	3	Clave A						Bits de acceso				Clave B						Fin de bloque 0
1	0																	Información
	1																	Información
	2																	Información
	3	Clave A						Bits de acceso				Clave B						Fin de bloque 1
:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:
14	0																	Información
	1																	Información
	2																	Información
	3	Clave A						Bits de acceso				Clave B						Fin de bloque 14
15	0																	Información
	1																	Información
	2																	Información
	3	Clave A						Bits de acceso				Clave B						Fin de bloque 15

Figura 2.5 Memoria etiquetas RFID.

3 Conexionado de los elementos

3.1 Conexiones de los módulos de Arduino

El conexionado del sistema se reduce a la conexión del módulo RFID, ya que gracias a la modularidad de las *shield* de Arduino, el módulo de Ethernet va pinchado directamente sobre la placa Arduino.

Ambos módulos hacen uso de la conexión SPI, por lo que las señales SCK, MISO y MOSI son comunes. Los otros dos pines necesarios en SPI se usan para controlar cada módulo por separado e iniciar la comunicación con el que se quiere mientras el otro permanece a la espera. Estos pines vienen predefinidos en la librería del módulo Ethernet como el pin 5 para RST y el pin 10 para el SS. En el módulo RFID se pueden configurar libremente y para este sistema se han definido el RST del RFID como el pin 47 y el SS como el pin 46.

Debido a la diferencia de microcontrolador entre el Arduino Uno y el Mega, los pines de comunicación SPI están situados en distinto lugar. Por ello, el *shield* Ethernet comunicará a través del puerto ICSP y usará los pines 5 y 10 para las señales RST y SS como se dijo antes. El módulo RFID se conectará a los pines 50 (MISO), 51 (MOSI) y 52 (SCK).

El módulo RFID acepta que la tensión de las señales sea 5V, pero la alimentación del mismo debe hacerse a 3.3V por lo que se conectará a la salida regulada a 3.3V del Arduino.

Tabla 3.1 Asociación de pines.

	Pines Arduino Mega	
Señal SPI	Módulo MFRC522	Ethernet Shield
RST	47	5
SS/SDA	46	10
MISO	50	ICSP-1
MOSI	51	ICSP-4
SCK	52	ICSP-3

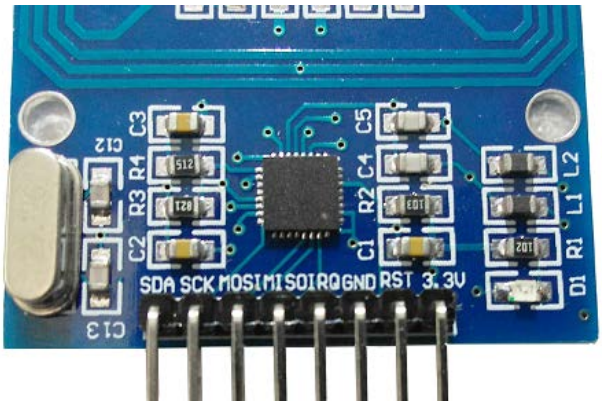

Figura 3.1 Pinout del módulo RFID.

La conexión final del Arduino con los módulos quedaría como se muestra en la 3.2.

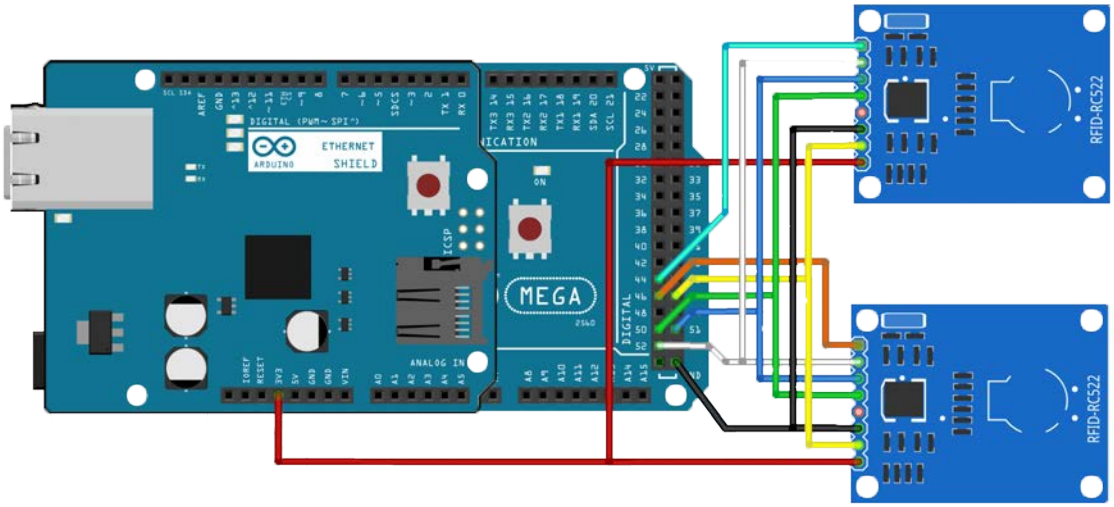

Figura 3.2 Conexión general.

3.2 Conexión entre el PLC y Arduino

La conexión entre el PLC y Arduino se hace a través del módulo Ethernet de este y el puerto Ethernet del PLC, no el de MODBUS. Esta conexión no es elemento a elemento, sino que PLC y el Arduino se conectan a una red local a través de un *switch*, con dos cables de Ethernet directos.

Hay que tener en cuenta la IP del PLC del puesto para que el Arduino sepa a quién se tiene que conectar de la red local y también la IP del PLC principal para poder comunicarle las instrucciones de activacion y desactivación de los retenedores.

4 Comunicación

Para la realización de este proyecto ha sido necesario la comunicación entre los distintos elementos del sistema para intercambiar la información necesarias. Debido a la diferente naturaleza de cada subsistema, las comunicaciones utilizadas son muy diferentes.

La placa Arduino tiene en este caso dos periféricos conectados por SPI, cuya configuración se puede ver en el punto 7.1.5. Este es un estandar de comunicación muy habitual entre los microcontroladores junto al I^2C y a la comunicación por puerto serie, que en este caso solo se ha usado a modo de busqueda de fallos, pero no es necesario su uso una vez implementado el sistema.

Se puede considerar a su vez, que la placa Arduino es un periférico del PLC Modicon M340 y para que se comuniquen se ha usado el protocolo MODBUS, ampliamente usado en la industria. En concreto, se ha recurrido al MODBUS TCP/IP, de manera que se pueda usar un simple cable de Ethernet para realizar la comunicación.

4.1 Estandar SPI

El bus SPI (Serial Peripheral Interface) fue desarrollado por Motorola en 1982 para sus primeros microcontroladores derivados de la misma arquitectura que su microprocesador 68000. El SPI fue definido como un bus externo al microcontrolador y era usado para conectar los periféricos del microcontrolador mediante cuatro cables. Hoy en día sigue siendo comúnmente utilizado para enviar datos entre microcontroladores y periféricos [6].

Su arquitectura es de tipo maestro-esclavo. El dispositivo maestro puede iniciar la comunicación con uno o varios dispositivos esclavos y enviar o recibir datos de ellos. De los cuatro cables necesarios, tres hilos son compartidos entre el maestro y todos los esclavos y el cuarto hilo es individual entre el maestro y cada esclavo. Las funciones de cada uno de los tres hilos comunes son las siguientes:

1. El primer hilo es la señal de reloj (SCLK o SCK) enviada desde el bus master a todos los esclavos. Todas las señales SPI son síncronas a esta señal de reloj.

2. El segundo hilo es la línea de datos del maestro a los esclavos (MOSI), en inglés Master Out-Slave In.

3. El tercer hilo es la línea de datos de los esclavos al maestro (MISO), en inglés Master In-Slave Out.

La función del cuarto hilo, único entre cada esclavo y el master es la siguiente:

4. Este último hilo es el de selección de esclavo (SS) y su propósito es conectar o desconectar la operación

del dispositivo con el que uno desea comunicarse. Para comenzar esa comunicación, se pone la señal del esclavo deseado a nivel bajo.

El bus SPI es "full duplex" (tiene líneas de envío y recepción separadas) y, por lo tanto, en ciertas situaciones, puede transmitir y recibir datos al mismo tiempo (por ejemplo, solicitar una nueva lectura del sensor al recuperar los datos de la el anterior).

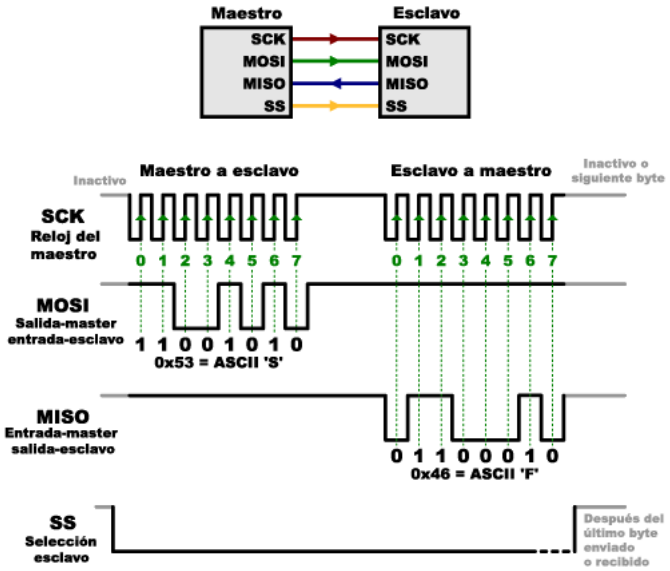

Figura 4.1 Esquema comunicación SPI.

Hay cuatro modos de comunicación disponibles: Modo 0, 1, 2 y 3 que definen:

1. El flanco de la señal de reloj en la que la línea MOSI cambia.

2. El flanco de la señal de reloj en la que el maestro muestrea la señal MISO.

3. La señal de reloj en reposo, es decir, el nivel de la señal de reloj, alto o bajo, cuando el la señal de reloj no está activa.

Cada modo está definido con un par de parámetros llamados polaridad de reloj (CPOL) y fase de reloj (CPHA). Ver Figura 4.2.

El maestro y el esclavo deben usar los mismos parámetros (frecuencia de reloj, CPOL y CPHA) para que sea posible la comunicación. Si hay varios esclavos con distintas configuraciones, es el propio maestro el que se deberá reconfigurar cada vez que tenga que comunicar con cada esclavo.

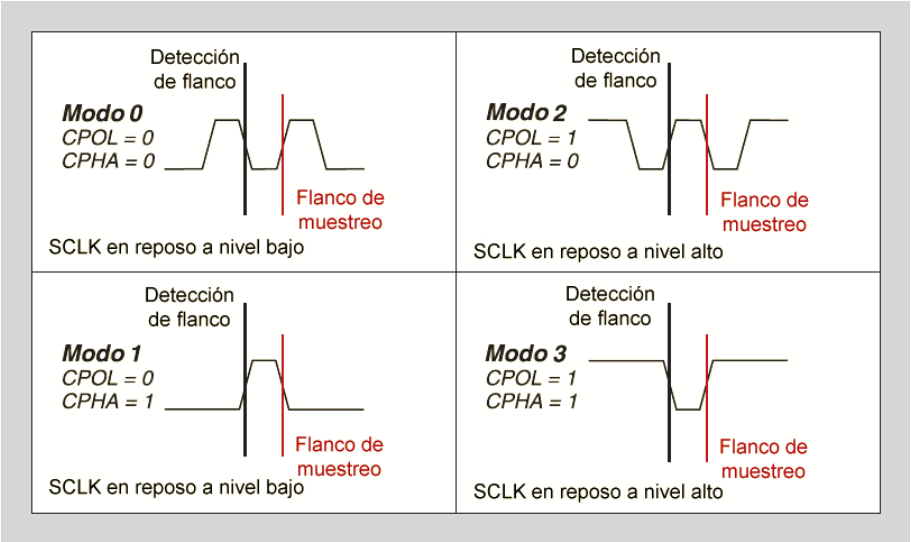

Figura 4.2 Modos de comunicación SPI.

4.2 Protocolo Modbus

El protocolo Modbus es una interfaz de comunicación electrónica desarrollada por Modicon en 1979, con carácter público y gratuito. Es usado para establecer una comunicación maestro-esclavo (UDP) / cliente-servidor (TCP/IP), desarrollada en 1999, entre dispositivos electrónicos industriales. Desde que fue desarrollado se ha convertido en la comunicación estándar en la industria ya que es fácil de implementar, es flexible y es muy fiable [7] [8].

El protocolo MODBUS se divide principalmente en tres modos: ASCII, RTU (basada en comunicación serie RS485, RS422, RS232) y TCP/IP (basada en Ethernet). MODBUS ASCII comunica a 7 bits de datos y el inicio y el final del mensaje se indica con caracteres ASCII, en concreto usa los dos puntos ':'como inicio de mensaje y el retorno de carro más el salto de línea para el fin del mensaje. En cambio en modo RTU comunica a 8 bits de datos y el inicio y el final del mensaje se indica con una pausa equivalente al tiempo de 3.5 caracteres, siendo la pausa entre caracteres menor a 1.5. RTU es capaz de enviar más información en el mismo periodo de tiempo que ASCII. El modo TCP es prácticamente un modo RTU dentro de una conexión TCP/IP, con una cabecera de 6 bytes.

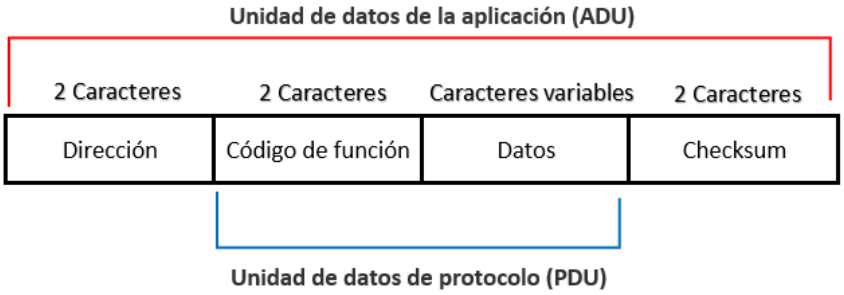

Figura 4.3 Unidad de datos MODBUS ASCII.

Los mensajes enviados se componen de una PDU o unidad de datos de protocolo, que contiene el código de la función y los datos del mensaje. Dependiendo del modo de MODBUS, a la PDU se le añaden otros bloques de información, de manera que se forma la unidad de datos de la aplicación o ADU. Estos bloques

de información pueden ser la dirección, el checksum o una cabecera para distinguir el paquete en la red TCP [9]. Se puede ver la distribución de la PDU de los distintos modos en las Figuras 4.3, 4.4 y 4.5.

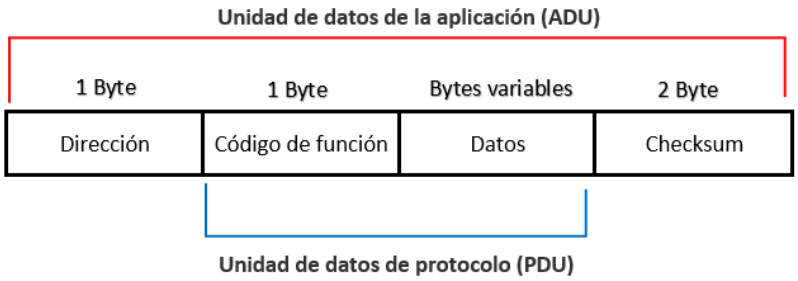

Figura 4.4 Unidad de datos MODBUS RTU.

Se observa en la Figura 4.5 como el bloque de la dirección ha sido sustituido por la cabecera MBAP y que ha desaparecido el checksum, ya que esta comprobación se realiza ya en otra de las capas de la comunicación. La cabecera MBAP es la encargada de identificar la ADU mientras la información es transportada por la red

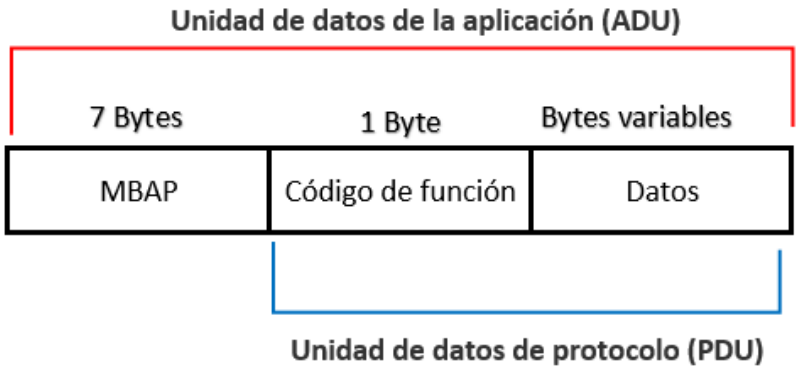

Figura 4.5 Unidad de datos MODBUS TCP/IP.

El formato de la PDU está definido como un código de función, seguido por un conjunto de datos asociados. El tamaño y el contenido de estos datos están definidos por el código de la función, y la PDU completa (código de función y datos) no puede ser mayor de 253 bytes. Cada código de función genera un comportamiento específico que puede implementarse de manera flexible en función del comportamiento deseado de la aplicación.

Bloque de memoria	Tipo de datos	Acceso de maestro	Acceso de esclavo
Bobinas	Booleano	Lectura/Escritura	Lectura/Escritura
Entradas discretas	Booleano	Solo Lectura	Lectura/Escritura
Registros de retención	Palabra sin signo	Lectura/Escritura	Lectura/Escritura
Registros de entrada	Palabra sin signo	Solo Lectura	Lectura/Escritura

Figura 4.6 Permisos PDU.

Los códigos de función y sus datos son definidos explícitamente por el estándar y están en las especificaciones. Cada función sigue un patrón. Primero, el esclavo valida entradas como el código de función, dirección de datos y el rango de datos. Después, ejecuta la acción solicitada y envía una respuesta adecuada al código. Si cualquier paso en este proceso falla, se regresa una excepción al solicitante. El transporte de datos para estas solicitudes es la PDU. Los principales códigos de función se pueden ver en la tabla 4.1

Tabla 4.1 Códigos y descripción PDU.

Código	Descripción
1	Leer bobinas
2	Leer entradas discretas
3	Leer múltiples registros
4	Leer registros de entrada
5	Escribir una bobina
6	Escribir un registro
16	Escribir múltiples registros

En este proyecto, la comunicación MODBUS al ser mediante TCP/IP, no hay maestros y esclavos, sino clientes y servidores.

Se ha configurado la comunicación de manera que los Arduinos funcionan como clientes, mientras que los PLC actúan como servidores. Cada Arduino se encarga de iniciar las comunicaciones y periódicamente pregunta si hay alguna instrucción nueva para escribir en las tarjetas. También puede iniciar la comunicación para escribir información en la memoria del PLC.

El PLC tiene unas direcciones de memoria asociadas según el tipo de dato que se va a escribir. Al elegir el modo 16, se está indicando que se quieren escribir registros, por lo que en el PLC hay que indicar que las direcciones de memoria correspondientes a registros serán las utilizadas. Esto se hace mediante el comando %MW, pero este comando solo indica el apartado de memoria que esta dedicado a los registros. Hay que definir a partir de qué dirección se escribirá y se leerá. Esta configuración se hace desde Arduino.

Por ejemplo, el comando de Arduino para leer un registro es el que se puede ver en el código 4.1. Con *Mb.Req* se hace un requerimiento al elemento *Mb* previamente definido. El primer campo de esta función

es lo que se quiere hacer, como se quiere leer, el comando *MB_FC_READ_REGISTERS* establece una comunicación en modo 3. Los siguientes parámetros que se le pasan a la función indican de que dirección se leerá del PLC, *DLP*, cuantos registros se van a leer, *tam* y en que dirección del registro de arduino se escribirá, *DEA*. La dirección del PLC de la que lee Arduino se ha establecido que sea desde la 0 y la cantidad de registros 8, por lo que del 0 al 7 son los registros en los que lee Arduino.

Código 4.1 Configuración de las direcciones de lectura de los registros.

```
Mb.Req(MB_FC_READ_REGISTERS, DLP , tam , DEA );
```

Para escribir un registro, el comando de Arduino es el que se puede ver en el código 4.2. Como en este caso se quiere escribir, el primer campo de esta función es el comando *MB_FC_WRITE_MULTIPLE_REGISTERS* que establece una comunicación en modo 16. Los siguientes parámetros que se le pasan a la función indican en que dirección del PLC se escribirá, *DEP*, cuantos registros se van a escribir, *tam* y de que dirección del registro de arduino se leerá la información a escribir, *DEA*. La dirección del PLC en la que escribe Arduino se ha establecido que sea desde la 8 y la cantidad de registros 8, por lo que del 8 al 15 son los registros en los que escribe Arduino.

Código 4.2 Configuración de las direcciones de escritura de los registros.

```
Mb.Req(MB_FC_WRITE_MULTIPLE_REGISTERS, DEP , tam , DLA );
```

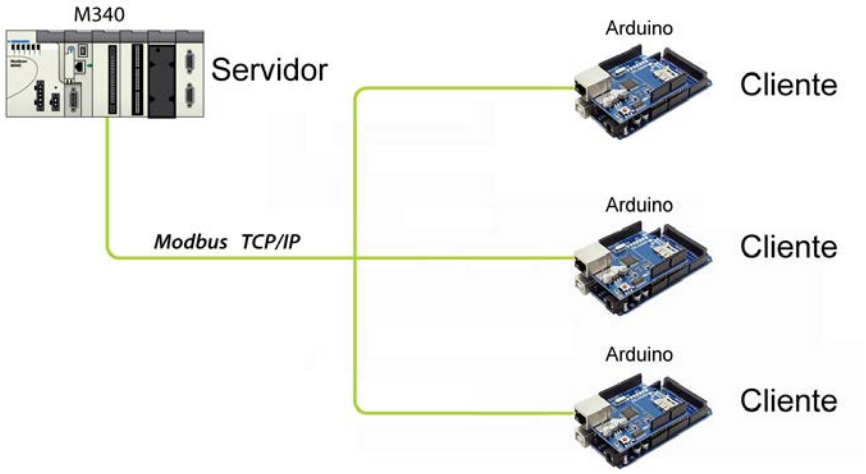

Figura 4.7 Esquema Cliente/Servidor.

4.3 Esquema de comunicaciones entre equipos

En los apartados anteriores se han explicado los dos tipos de comunicaciones usados, de forma aislada. En este apartado se mostrará como se usan esas comunicaciones de una forma más general.

Como se ha visto en el apartado 4.2, en la comunicación MODBUS se usa tanto la lectura como la escritura. En el caso de la lectura para comprobar si el usuario ha introducido nuevas instrucciones, esta se hace de forma cíclica. Arduino lee de la memoria del PLC y en caso de que haya alguna instrucción nueva, la copia en su memoria. También de forma cíclica se comprueba, leyendo a través de SPI el lector RFID, si hay una

nueva tarjeta sobre el lector. En caso necesario, el Arduino comunica al PLC principal mediante MODBUS la orden de levantar el retenedor de su puesto. Esta situación es la representada en la figura 4.8.

Figura 4.8 Diagrama cíclico de lectura de datos.

De forma no cíclica se comunica mediante SPI con el lector RFID para leer información de las etiquetas y escribirle esta al PLC mediante MODBUS. A su vez el PLC se comunicaca con los elementos de su puesto según las comunicaciones que estos tengan disponibles. Finalmente el proceso se invierte, el elemento del puesto se comunica con el PLC, este escribe en su memoria la información recibida y la necesaria para el Arduino, el Arduino lee por MODBUS la información de memoria del PLC y a través de SPI le manda las instrucciones al lector RFID para escribir la información o para extraer nuevos datos, según lo que sea necesario. Esta situación es la representada en la figura 4.9.

Figura 4.9 Diagrama de envío de datos.

5 Implantación del sistema RFID

En este capítulo se van a plasmar los imprevistos encontrados durante la realización de este proyecto y que se deberían evitar al realizar el montaje en los distintos puestos.

5.1 Colocación de la etiqueta

La primera duda que resultó de los distintos elementos del sistema era donde colocar las etiquetas RFID. Evidentemente estas debían ser las que se movieran y los lectores los que se quedaran en estático al necesitar una alimentación y ser mas voluminosos y costosos.

Se pensó en primer lugar que cada palet llevara una etiqueta asociada con su número de palet. El problema de esta solución era que si se colocaba la etiqueta en la parte inferior del palet, una vez colocado este sobre la bandeja no habría acceso para la lectura de la información. En cambio, si se colocaba en la parte superior entorpecía el manejo de las piezas que se cargaran sobre el. Los laterales tampoco eran una posibilidad al ser de difícil acceso para el lector y poder interferir con alguno de los otros sistemas.

Solo quedaba la opción de que fuera la bandeja la que llevará la etiqueta, y que una vez se le cargaba un palet, simplemente se definía en la etiqueta qué palet llevaba cargado. De nuevo se encontró el problema de que en la parte superior no sería accesible a la lectura al llevar un palet sobre la etiqueta. La parte inferior era la única solución que quedaba, pero de nuevo había sistemas con los que interfería esto. Se pensó en colocar la etiqueta en una esquina de la parte inferior, pero si alguien quisiera colocar el palet sobre la cinta tendría que mirar antes cómo colocarlo y esto ralentizaría las operaciones, a parte que la bandeja podía salir en una dirección distinta de los distintos puestos. La opción finalmente fue colocar la etiqueta centrada en la parte inferior. Esta solución solo afectaba al alimentador de bandejas, pues utiliza un cilindro que eleva empujando por la parte inferior a la bandeja para colocarla en el almacén. Esto era fácilmente solventable colocando una corona en la parte interior de cada palet, alrededor de la etiqueta o más fácil aún colocando la corona solo en el elevador de bandejas y bajando este un poco para compensar la nueva altura.

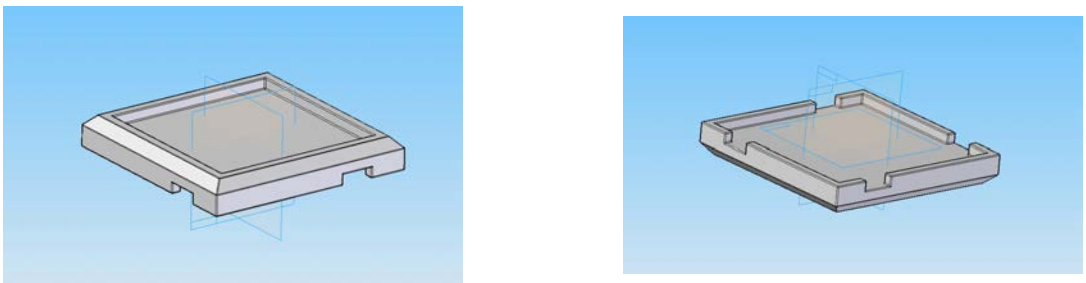

Figura 5.1 Bandeja.

Una vez decidida la posición de la etiqueta se probó a pegar una en la posición elegida y comprobar si podía ser la solución definitiva. Se colocó la bandeja en la cinta, se puso un lector en una posición que no tocara la bandeja y se probó si se podía leer la información contenida en la tarjeta. Después de pasar varias veces la bandeja sobre el lector, no se consiguió ninguna lectura. Probando los lectores fuera del sistema, se comprobó que funcionaban sin problema, pero se vio que la distancia de lectura era bastante pequeña. A parte de esto, también se averiguó, que la propia bandeja, al ser metálica, hacía de pantalla al tener colocada la etiqueta justo pegada al metal. Los dos problemas se solucionaron colocando una pieza plástica que alejara la etiqueta de la bandeja y que además la acercara al lector.

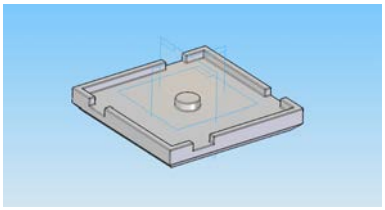

Figura 5.2 Inferior bandeja con tag.

5.2 Zona de lectura

Como se vio con las etiquetas, mientras más cerca estuviera del lector, menos problemas para la lectura había, de manera que se ajustó la altura del lector hasta que la distancia entre este y la etiqueta fuera de unos 2 - 3 mm.

La primera idea para la zona de lectura fue colocar un único lector RFID. Para ello hubo que mover el lector según la distancia a los retenedores. La primera idea fue colocar la placa de Arduino junto con el lector RFID en la célula de fabricación flexible de tal manera que una vez estaba parada la bandeja por el retenedor, la etiqueta estuviera en el centro del lector RFID como se ve en la Figura 5.3.

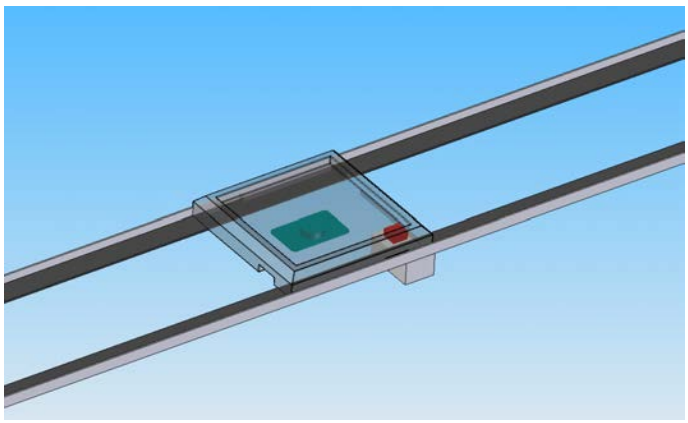

Figura 5.3 Bandeja parada sobre sensor centrado.

Esta opción se descartó al no dar tiempo a hacer una lectura de la tarjeta, comprobar si había que parar la bandeja y en caso de que sí, dar la orden al PLC para que activara el retenedor.

Se probó a retrasar un poco el lector, de manera que cuando estuviera parada la bandeja, la etiqueta estuviera en el borde del lector pero aún fuera alcanzable como se ve en la Figura 5.4, se conseguía más tiempo para poder leer y comprobar la información de la tarjeta, de manera que se pudiera decidir si levantar o no el retenedor para actuar sobre la bandeja y una vez detenida, se pudiera seguir trabajando sobre ella.

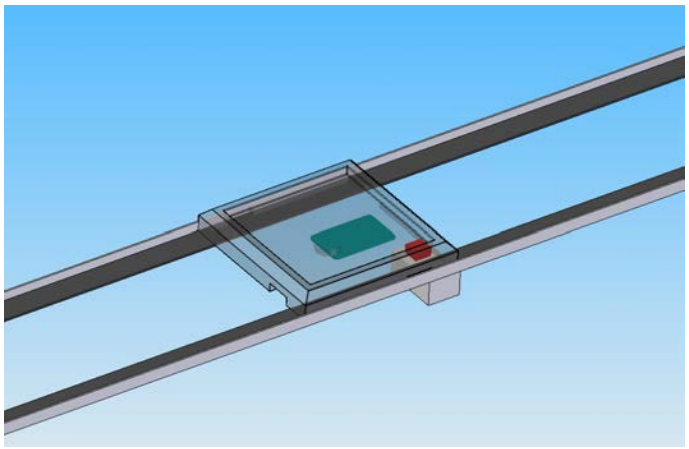

Figura 5.4 Bandeja parada sobre sensor desplazado.

De nuevo, esta opción finalmente se descartó, porque aunque se conseguía parar el palet, el margen para colocar el lector dada la velocidad de la cinta, era de muy pocos milímetros y con cualquier roce ya no se leía la información una vez parado el palet o no se llegaba a parar.

Finalmente se optó por colocar dos lectores RFID. El primero de ellos estaría junto al retenedor y se encargaría de detectar si había que parar el palet, de manera que el arduino mandara la orden de levantar el retenedor. El segundo lector se colocaría de tal forma que coincidiera con el centro del palet una vez retenido y unas décimas de segundo después de pararlo, para darle tiempo a colocarse completamente sobre el lector, leería o escribiría la tarjeta, según lo que se necesitara.

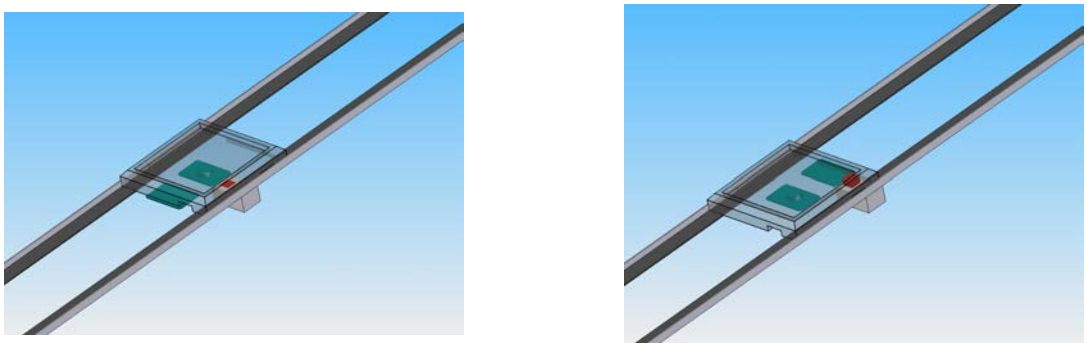

Figura 5.5 Bandeja sobre el detector y sobre el lector.

5.3 Interrupción por detección

Para hacer la ejecución del programa lo más rápida posible, se intentó que la detección de la tarjeta activara una interrupción en el código de arduino. Para ello, el lector RFID tiene un pin dedicado, el llamado IRQ. Después de varios intentos fallidos, se buscó información extra en la librería y se advirtió, que los comandos para las interrupciones estaban solo a modo de prueba de concepto, pero estaba especificado que esa opción no estaba implementada en la librería [10].

6 Funcionamiento del sistema

El objetivo de este proyecto ha sido dotar de más flexibilidad a la célula de fabricación flexible de la Escuela Técnica Superior de Ingeniería de Sevilla. Para cumplir dicho objetivo se ha desarrollado un sistema de seguimiento de los palets a través de la planta mediante radio frecuencia.

Se valoraron varias formas de darle las órdenes al sistema.

Una de las más interesantes era almacenar en la etiqueta el estado final al que se quería llegar, por ejemplo, "almacena el palet 4 con una pieza alta de aluminio en la posición 2 en el hueco 31 del almacén. Además, lleva una pieza plástica baja en la posición 1 hasta el segundo almacén de piezas". A partir de ese estado final, cada PLC debería desgranar ese estado final para ver si tenía que actuar en el palet, teniendo en cuenta las tareas que ya se habían realizado.

En el ejemplo propuesto, al llegar la bandeja al almacén de palets, el arduino de ese puesto leería el tag y le comunicaría la información el PLC. El PLC tendría que interpretarlo y saber que como no hay aún palet encima de la bandeja, tendría que servir el 4 e indicarle al arduino que ya lo ha colocado, para que este puesda escribir esa información en la etiqueta de la bandeja. En el siguiente puesto, el almacén uno, el arduino leería y le comunicaría el mismo estado final al PLC, pero en este caso diciendole ademas que la bandeja ya lleva el palet 4 encima. El PLC interpretaría la instrucción y sabría que como aún no están las piezas necesarias colocadas, él es el siguiente. Por lo tanto se pararía la bandeja y se descargaría una pieza plástica baja en la posición 1 y una metálica alta en la 2. La misma rutina de comprobación y actuación se realizaría en el SCORBOT, cargando y descargando las piezas. Al pasar por el Sony y por el pórtico, al no intervenir esos puestos en la tarea encargado a la bandeja, esta pasaría de largo. El siguiente puesto sería el almacén de bandejas. La bandeja hay que almacenarla, pero aún lleva encima un palet. Al interpretar la orden final y comprobar las realizadas, el PLC vería que aún tiene encima el palet, por lo que dejaría pasar la bandeja y en la siguiente vuelta, si no se le ha cargado otro palet, sí se guardaría.

La cantidad de información a interpretar por cada PLC sería bastante grande y se perdería mucha flexibilidad ya que solo podría haber una serie de tareas predefinidas en los PLC. Además, una vez escrita la orden no había más control sobre las tareas a realizar hasta que el palet acabara de realizarlas todas.

Por otro lado de esta manera se liberaba al operario de saber que operaciones intermedias había en el proceso, quitando posibles errores humanos en el proceso.

Otra forma, y la implementada en este proyecto era que el operario escribiera en la etiqueta todas las tareas que quería que se realizaran sobre la bandeja y el orden en que quería que se realizaran. De esta manera, los arduinos al leer la etiqueta sabrían cual es la siguiente tarea a realizar y la única información a interpretar era si al leer la etiqueta, la siguiente tarea a realizar correspondía al puesto desde el que se leía o no. En caso de que se tuviera que parar, después de activar el retenedor, se leería solamente la primera instrucción a realizar, que correspondería al puesto que lo paró. Una vez realizada la tarea, se comprobaría si la siguiente es en el mismo puesto y en caso afirmativo seguiría así hasta que la siguiente tarea a realizar fuera en otro puesto,

momento en el que se bajaría el retenedor, dejando pasar la bandeja.

Para el ejemplo anterior, "almacena el palet 4 con una pieza alta de aluminio en la posición 2 en el hueco 31 del almacén. Además, lleva una pieza plástica baja en la posición 1 hasta el segundo almacén de piezas", habría que introducir las órdenes una a una, pero se liberaría de mucha carga al PLC.

La primera orden que habría que escribir seria que del almacén de bandejas hay que sacar el palet 4. Pero podría cometerse un error y antes de escribir la orden de sacar el palet, escribir en la etiqueta que hay que cargar la pieza plástica baja en la posición 1 del almacén uno. No pasaría nada, ya que antes de dejar marchar la bandeja, se podría introducir la orden de sacar el palet 4 con orden de realización 1, lo que incrementaría el orden de todas las tareas almacenadas de orden igual o superior a la introducida. Se continuaría escribiendo que también se quiere que en la posición 2 se cargue una pieza metálica alta del almacén 1, orden 3, después se escribiría que en el SCORBOT hay que descargar la pieza de la posición 1 en el almacén 2. La última tarea que habría que escribir antes de la de almacenar la bandeja, es almacenar el palet 4 en el hueco 31 del almacén. Cada vez que se escriba una instrucción, el arduino actualiza el bloque de comprobación de la etiqueta, en el que se guarda cual es el número de palet que lleva y en que puesto se va tiene que realizar la tarea con orden 1. Además se van contando el número de instrucciones escritas para no sobrepasar la capacidad de la etiqueta y que se pierda información.

Una vez liberado el palet con su lista de instrucciones escritas, este solo pararía en el puesto que dice su bloque de comprobación o en caso de que un operario decida escribir una nueva orden en medio del proceso en el palet. Para parar la bandeja, el arduino manda una señal al PLC principal para indicarle que levante el retenedor X al ser él el encargado de los retenedores y la comunicación a través de ethernet no ser suficientemente rápida para pasarle la orden antes de que la bandeja se pase el retenedor.

Cada vez que el palet para en un puesto, el arduino leería la etiqueta pero sólo le diría al PLC lo que tiene que hacer, por lo que este no sabría nada de las tareas que ya se han realizado o de las que aún están pendientes de realizarse. Una vez que el PLC recibe la instrucción procede a realizarla y cuando acaba se lo comunica al arduino para que este le envíe la siguiente orden en caso de que se haya que realizar más de una en el puesto. Si arduino detecta que no hay más intrucciones que enviar al PLC, le dice al PLC principal que puede bajar el retenedor y se deja pasar la bandeja.

6.1 Ordenamiento de la información en la tarjeta

Las etiquetas MIFARE Classic 1K como se vio en la sección 2.3, disponen de 1024 bytes repartidos en 16 sectores de 4 bloques y cada uno de estos con 16 bytes. A continuación se detalla la información contenida en algunos bloques significativos y cómo se guarda la información en etiqueta una vez se recibe una orden desde el PLC.

6.1.1 Bloque de identificación

El bloque 0 del sector 0 contiene toda la información que el fabricante ha escrito en la tarjeta. Entre esa información se encuentra el identificar único de la etiqueta y su modelo. Este bloque está protegido de fabrica contra escritura y solo permite leerlo. Se puede identificar este bloque en la figura 6.1 como el bloque con fondo gris.

En este proyecto, este bloque no se ha utilizado al no necesitar saber el identificador de cada etiqueta para nada.

6.1.2 Bloque de comprobación

Este bloque no está definido por el fabricante, sino que se ha aislado para este proyecto con el fin de almacenar en él información relevante para el almacenamiento de instrucciones. Se puede identificar este bloque en la figura 6.1 como el bloque con fondo rojo.

En este bloque solo se escribe en tres bytes. En el primer byte, el 0 según la figura 6.1, se almacena el número del puesto en que se va a realizar la siguiente tarea sobre la bandeja. En el segundo byte, el 1 según la figura 6.1, se almacena el número del palet que lleva la bandeja en caso de que lleve alguno. Por último, en el byte sexto, el 5 según la figura 6.1, se almacena el número de tareas sin realizar que hay escritas en la etiqueta. Este número es útil para saber si hay hueco para escribir más instrucciones. Los bloques usados están resaltados en la figura 6.1 con un fondo rojo oscuro.

6.1.3 Bloques de claves

Este es el cuatro bloque de todos los sectores, el 3 según la figura 6.1, y viene definido por el fabricante, aunque no viene protegido contra escritura.

En este bloque están almacenadas las claves de escritura y lectura de la tarjeta, cada una de 6 bytes. Cada vez que se quiere acceder a un bloque, antes hay que autentificarse con las claves almacenadas en el bloque de claves del final del sector sobre el que se está actuando. Estas claves vienen predefinidas por el fabricante, pero se pueden cambiar para dotar al sistema de más seguridad. En el caso de este proyecto las claves en hexadecimal, tanto para lectura como para escritura, son [0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF]. Esta es la clave por defecto de estas etiquetas.

Como se ha dicho, estos bloques no están protegido contra escritura, pero si hay que identificarse antes si se quiere cambiar alguna de las claves. Durante el desarrollo del proyecto se detectó un problema al respecto. Si se intentaba escribir sobre un bloque sin identificarse antes, no hay nada que te lo impida, pero el bloque queda inutilizado para siempre. Aunque se intente escribir de nuevo autentificándose antes, solo escribe números aleatorios en el bloque.

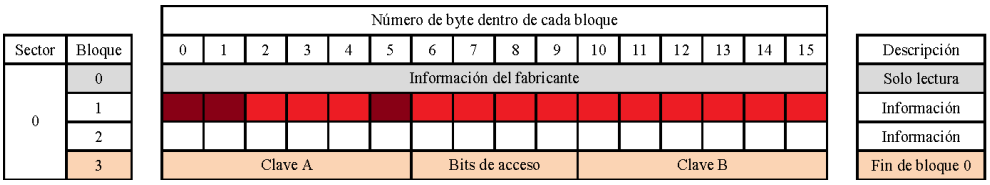

Figura 6.1 Detalle de información almacenada en sector 0.

6.1.4 Bloques de información y su estructura

El último elemento que queda por explicar de las etiquetas es el almacenamiento de la información que se necesita para el funcionamiento del sistema desarrollado.

Como se ha visto anteriormente, hay ciertos bloques que no sirven para almacenar información, por lo que de los 1024 bytes no todos son útiles. Se pueden calcular cuantos quedan libres:

- El bloque 0 del sector 0 no está libre.
- El bloque 1 del sector 0 se usa para la información de comprobación.

- El bloque 2 del sector 0 se ha dejado sin usar para futuras mejoras
- El bloque 3 de cada sector está ocupado por las claves.

Por lo tanto, de los 16 sectores, solo hay 3 de los 4 bloques libres. Esto deja 48 bloques. A estos hay que restarles el bloque 0, 1 y 2 del sector 0. Por lo que finalmente quedan solo 45 bloques.

Además se vio que cada bloque contiene espacio para 16 bytes, haciendo un total de 720 bytes libres de los 1024 de la etiqueta.

A la hora de ver la información que tenía que llevar, se llegó a la conclusión de que había 7 paquetes de datos que había que almacenar; número de palet, número de tarea, orden a realizar, posibles errores, puesto, estado de la tarea y detalles de la tarea. La primera idea fue almacenar estas instrucciones en un byte cada una, pero al ser cada bloque de 16 bytes, cabrían dos instrucciones y se dejarían sin usar dos bytes. Se decidió que esa información podía comprimirse más y ajustarse a las necesidades del sistema. Por ejemplo, para saber si está realizada o no la tarea basta con un bit, puestos solo hay 7, por lo que con 3 bits era suficiente, etc. Así se llevo a la distribución de bits para los siete paquetes de datos que se ve en la figura 6.2 . Con esta distribución se consigue almacenar cuatro órdenes por bloque, sin desperdiciar ningún byte. Con un total de 45 bloques disponibles, se puede llegar a 180 ordenes en la tarjeta.

Byte	0								1								2								3							
Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Tarea	Palet								Tarea								Orden				Error				Puesto			Realizado	Detalle			

Figura 6.2 Información en la tarjeta bit a bit.

Sector	Bloque	Número de byte dentro de cada bloque															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0																
	1																
	2																
	3	Clave A						Bits de acceso				Clave B					

Descripción
Información
Información
Información
Fin de bloque 1

Figura 6.3 Detalle de información almacenada en los bloques de información.

6.1.5 Ordenamiento de la información según orden de realización

Un aspecto importante a la hora de desarrollar este proyecto fue decidir como se ordenarían las instrucciones dentro de la tarjeta. El ordenamiento se podría hacer por secciones, según el puesto sobre el que se debía hacer cada instrucción, por orden de realización de la instrucción, por orden de escritura de la instrucción o escribirla en el primer hueco libre de la etiqueta. Para este proyecto se ha decidido realizar el ordenamiento según el último criterio. Las instrucciones se van escribiendo en la tarjeta en el primer hueco libre que se encuentre. La decisión de si un hueco está libre o no se hace comprobando el bit 27, que indica si la tarea está realizada o no. En caso de que esté realizada, este bien está a 0. Pero esta no es la única información que hay que comprobar. Una tarea puede estar realizada pero haber ocurrido algún error. Por ello hay que mirar también que los bits 24, 25 y 26 estén a 0. Por lo tanto, los 4 bits más significativos del 4 byte deben estar a 0 para que se puede sobre escribir la tarea.

6.2 Pantalla de explotación en Unity y puerto serie arduino

La ventana de explotación de Unity Pro, figura 6.4 está formada por una serie de botones y cuadros de escritura, los cuales permiten al usuario mandar las órdenes de una manera más sencilla al Arduino.

A la izquierda están los botones que definen la instrucción que se quieren que se realicen. Están predefinidas 9, comprobar pieza, dejar pieza en alimentador, recoger pieza del alimentador, dejar en pieza en el palet, recoger pieza del palet, recoger bandeja, sacar bandeja, recoger palet de la bandeja y poner palet en la bandeja.

A continuación hay unos cuadros de texto que permiten introducir datos extra. El primero es el del detalle. Cada instrucción tiene unos detalles concretos. Por ejemplo, la instrucción comprueba pieza puede llevar el detalle de que se compruebe la altura o el material, al dejar la pieza en el palet hay que decirle en que posición, al recoger pieza del alimentador hay que decirle que tipo de pieza se quiere, etc.

El segundo cuadro de texto hace referencia al puesto en el que se quiere que se realice la tarea. Se puede querer recoger una pieza, pero hay dos almacenes. Esta información podría extraerse directamente de la instrucción, pero para facilitar la detección por parte del Arduino se decidió hacerlo así.

El tercer cuadro indica el palet y solo es útil para cuando se extrae o se almacena el palet, ya que es el único puesto en el que se permite cambiar el número de palet que se lleva. Como se vio en el apartado 6.1.5, la instrucción Palet es una de las que mayor tamaño tienen asignado, esto es debido al número de palets que hay en el almacén. Con un bit menos no sería suficiente el espacio para trabajar con todos el almacén de palets.

El último que se encuentra alineado con los anteriores es Tarea. En este recuadro se ha de introducir el orden de realización de la instrucción que se va a introducir. Si se introduce la 5 y solo hay 3 instrucciones almacenadas en la etiqueta, se almacenará como la número 4. Sin embargo, si se introduce la 2 y hay 3 almacenadas, la que está almacenada con el número 2 pasará a ser la 3 y la que hay almacenada con el número 3 pasará a ser la número 4, para posteriormente introducirse la nueva instrucción con el número 2.

Una vez se han rellenado todos los cuadros necesarios hay que darle al botón Confirmar para que la orden sea enviada al Arduino y este las vaya almacenando en su memoria hasta que pase el palet para el que esa instrucción está dirigida.

En la parte superior se ve otro recuadro con un círculo rojo y que a su derecha pone error. En este recuadro se mostrarán los distintos errores que vayan apareciendo durante el funcionamiento del sistema. El círculo rojo solo aparece cuando el programa está corriendo y aparece algún error. Hasta que no se pulse el botón Reiniciar error de instrucción, no se apagará el indicador de error ni se podrán introducir nuevas instrucciones.

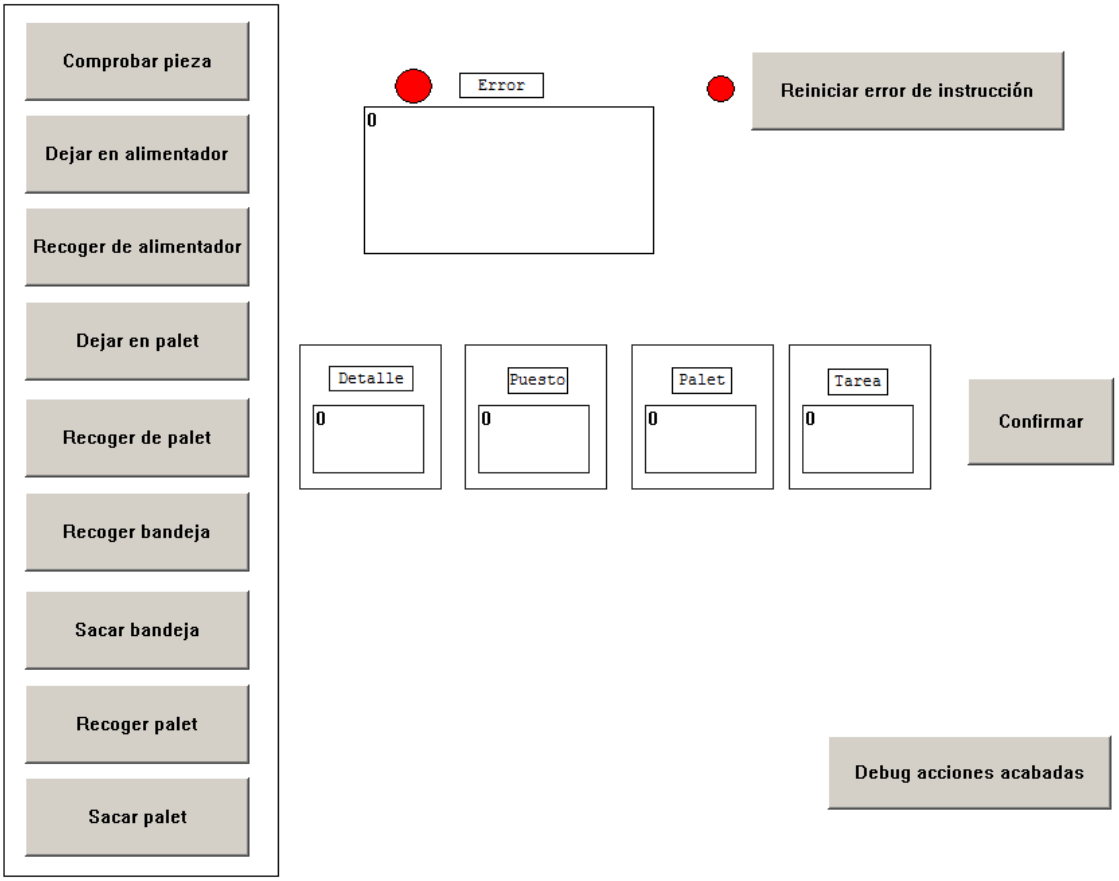

Figura 6.4 Pantalla de explotación de Unity.

6.3 Ventajas e inconvenientes de este sistema

La principal ventaja de este sistema es la descentralización de la gestión que se consigue, que salvo por el control de los retenedores, que está físicamente impuesto que sea el PLC principal el encargado de controlarlos, todos los demás puestos son independientes y no necesitan comunicarse entre ellos para realizar ninguna tarea. Con la información almacenada en cada etiqueta cada equipo sabe lo que tiene que hacer y una vez hecho, deja pasar la bandeja para que sea cual sea la siguiente tarea se realice en su puesto correspondiente.

Los inconvenientes de este sistema son principalmente, la introducción de nuevos sub sistemas que pueden dar lugar a fallos, especialmente si son de baja calidad.

Por último, una característica que puede ser tanto ventaja como inconveniente es la necesidad de introducir las instrucciones una a una. Esto tiene la ventaja de añadir gran flexibilidad al sistema, pero en caso de necesitar realizar muchas instrucciones, ya sea en una misma bandeja o en muchas, se hace muy tedioso.

6.4 Diagrama de ejemplo de recepción de orden

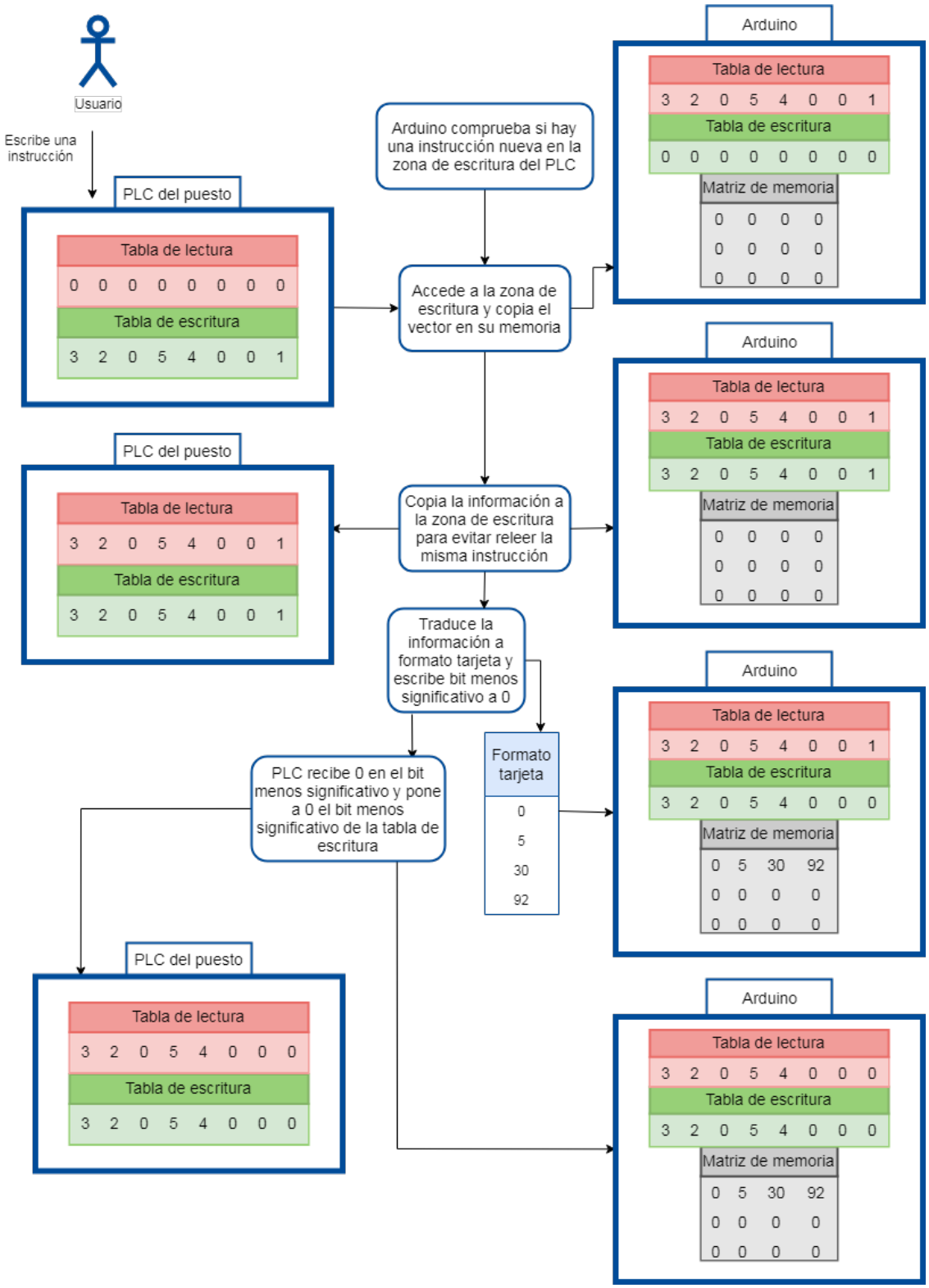

Figura 6.5 Gestión de orden recibida.

7 Códigos

En esta sección se va a mostrar el código desarrollado para el manejo de los distintos elementos del proyecto. Ha sido necesario desarollar un código en Arduino para la gestión de la información de la etiqueta RFID y un código en Unity para enviar y recibir las órdenes necesarias.

El código en Arduino se ha desarrollado a partir de las librerías del chip MFRC522, encargado del funcionamiento del lector RFID y de una librería que permite la comunicación MODBUS de Arduino con un PLC mediante la *shield* de Ethernet de Arduino.

Para la parte del PLC se ha creado con Unity Pro un programa que mediante MODBUS, reciba las órdenes de Arduino sobre qué hacer en ese puesto sobre la bandeja o el palet y además, se ha creado una pantalla de manejo que, permita al usuario escribir órdenes sobre las bandejas.

7.1 Código en Arduino

El código de Arduino se ha dividido en varias funciones para que sea más fácil depurar el código a medida que se desarrollaba, para permitir que en un futuro un usuario pueda modificar una parte de él de manera sencilla y también para facilitar la lectura del mismo.

El código principal hace uso de las funciones desarrolladas y gracias a estas se hace muy sencillo el comprender el funcionamiento del programa. En las siguientes subsecciones se verán las declaraciones de variables, las inicializaciones de las mismas y de los periféricos, las distintas funciones y finalmente el código principal haciendo uso de estas.

7.1.1 Declaración de librerías

Como todos los códigos, las primeras líneas están dedicadas a la declaración de las librerías necesarias para la correcta ejecución del programa. En el caso de este programa se ha recurrido a cuatro librerías distintas.

La primera librería es la de la comunicación por el estándar SPI. Esta librería es la encargada de definir las funciones para la comunicación de los periféricos por SPI [11].

La segunda librería es la *Ethernet.h*, que define las funciones necesarias para hacer uso de la *shield* de Ethernet [12]. Esta librería define gran parte de los parámetros de configuración de este módulo. Hay que tener en cuenta que viene codificado por defecto el conector de esclavo para la comunicación SPI en el pin 10 y no se puede cambiar en nuestro código a menos que se modifique la librería. Otro problema asociado a esta librería es que no tiene definida la configuración de la comunicación SPI, por lo que cada vez que se vaya a

operar con el módulo Ethernet, hay que mandar una orden con la configuración que necesita el módulo para funcionar.

La tercera la que permite el uso del módulo lector de RFID basado en el chip Mifare RC522 [13]. Esta librería está desarrollada por Miguel Balboa y como se ha visto, en la información de la propia librería vienen las casas que funcionan y las que no. Por ejemplo una de las características que no están desarolladas, mas que como una prueba de concepto, es la de las interrupciones por el pin IRQ del módulo.

Por último, la cuarta librería define las funciones para la comunicación MODBUS TCP/IP [14]. Dentro de esta librería se ven que están desarrolladas dos formas de realizar la comunicación por MODBUS, tanto definiendo Arduino como esclavo, como master. En un principio se definió a Arduino como esclavo, de manera que cada PLC fuera master de su Arduino. Esta jerarquía dio muchos problemas de comunicación, sobre todo de la parte del PLC, que aleatoriamente dejaba de leer o cuando se le decía que escribiera tardaba minutos en hacerlo. Finalmente se optó por definir al Arduino como master y usar los comandos asociados a esta jerarquía. Hubo que modificar un parámetro de la librería, porque aunque existía un comando para identificar la IP del esclavo, en la librería había una parte en que la IP venía ya definida. Cada Arduino habrá que programarlo con la librería modificada según el puesto al que va a estar asociado.

Código 7.1 Inclusión de librerías.

```
/* Librería SPI que nos permitirá comunicarnos con el módulo Ethernet
    y el lector RFID */
#include <SPI.h>
/* Librería Ethernet que nos permitirá usar los comandos necesarios
   para conectarnos a la red */
#include <Ethernet.h>
/* Librería MFRC522 que nos permite usar los modulos RFID basados en
   el chip Mifare RC522 */
#include <MFRC522.h>
/* Librería MgsModbus, la cual nos permitirá comunicarnos mediante
   Modbus con los PLC */
#include "MgsModbus.h"
```

7.1.2 Declaración de las constantes

En esta sección están declaradas las constantes necesarias para el correcto funcionamiento del programa.

La primera es la constante Puesto, en la que se declara el número de puesto en el que va colocado el Arduino. Este es uno de los pocos datos que hay que modificar al asociar el Arduino a uno de los PLC.

Otra de las constantes declaradas es tam, que es el tamaño del vector de información que intercambia Arduino con el PLC. Como se vio solo se necesitan siete posiciones, pero se ha sobredimensionado por si fuera necesario pasar información adicional. Del mismo modo que tam es el tamaño del vector que maneja el PLC, bytes_RFID indica el tamaño del vector con el que trabaja Arduino con el lector RFID.

Las constantes Bloque_Comprobacion y Fin_Bloque_Comprobacion indican el bloque donde está localizado el bloque de comprobación y el bloque de las claves de este. Estas posiciones no variaran durante toda la ejecución del programa.

También se tiene la constante Inter que define el pin por el cual, a través de un optoacoplador, se le indica al PLC maestro que levante el retenedor del puesto en el que está situado el Arduino. El número del retenedor hay que definirlo en el propio PLC maestro.

A continuación se encuentran las constantes RST_PIN, RFID y RFIDD. Estas permiten configurar la

conexión de algunos de los pines del lector RFID. La primera define el pin donde se conecta la línea de reset para la comunicación SPI y tanto RFID como RFIDD definen los pines de activación de cada uno de los módulos MFRC522 esclavos. RFID hace referencia al módulo lector de la información y RFIDD al modulo detector y encargado únicamente de leer el bloque de comprobación.

Por último están declarados los pines para la comunicación SPI del módulo de Ethernet. La constante Ether hace referencia al pin de esclavo de este módulo. Como se dijo en la sección 7.1.1, el pin de esclavo de este módulo está por defecto en la librería como el 10, por lo que no queda otra que definirlo como el pin 10. Otra de las funcionalidades de la *shield* de Ethernet es que trae un lector de tarjetas,que aunque no se vaya a usar en este proyecto, se le ha asignado el pin 4 para la comunicación SPI. Está ultima constante es SD_Ether.

Código 7.2 Definición de constantes.

```
/*** Definimos información que podremos/deberemos modificar ***/
constexpr uint8_t Puesto = 4;                       // Puesto en el
   que irá colocado el equipo
constexpr uint8_t tam = 8;                          // Definimos un
   tamaño para el vector de información que recibiremos del PLC
constexpr uint8_t bytes_RFID = 4;                   // Número de
   bytes usados para almacenar la información en la tarjeta RFID
constexpr uint8_t Bloque_Comprobacion = 1;          // Bloque en el
   que guardaremos el siguiente puesto y el palet en el que escribir
constexpr uint8_t Fin_Bloque_Comprobacion = 3;      // Fin del bloque
    Bloque_comprobacion para leer las keys
constexpr uint8_t Fin_IP_Servidor = 16;         // Aquí pondremos el ú
   ltimo campo de la IP del PLC del puesto
constexpr uint8_t Fin_IP_Ppal = 12;                 // Aquí pondremos
    el último campo de la IP del PLC principal

/*** Definimos los pines que usaremos, dandoles un nombre ***/
// Pines generales
constexpr uint8_t Inter = 22;                       // Pin por donde
   mandar la señal de interrupción para decirle al PLC que active el
    retenedor

// Pines configurables del lector RFID
constexpr uint8_t RST_PIN = 47;                     // Pin de reset
   del módulo RFID
constexpr uint8_t RFID = 46;                        // Pin de slave
   del módulo RFID
constexpr uint8_t RFIDD = 44;                       // Pin de slave
   del módulo RFID detector

// Pines del módulo Ethernet
constexpr uint8_t Ether = 10;                       // Pin de slave
   del módulo Ethernet
constexpr uint8_t SD_Ether = 4;                     // Pin de slave
   del lector de tarjetas del módulo Ethernet
```

7.1.3 Definiciones necesarias para los periféricos

En esta sección están las declaraciones necesarias para que funcione el lector RFID, la *shield* Ethernet y la comunicación Modbus.

Se empieza declarando los objetos necesarios para el funcionamiento de los módulos RFID. A los módulos se les asignan nombres distintos y sus respectivos pines de esclavo y de reset. Los números de estos pines ya se declararon en las constantes. También se declara la variable key, que almacenaré las claves de acceso a las etiquetas.

Respecto al módulo Ethernet, hay que definir la mac que se le va a asignar, ya que no trae una asociada. También hay que definir varias IP. Con la variable ip se declara la IP del Arduino y del módulo, gateway define la ip privada de la red local, subnet define la máscara de red que sirve para delimitar el ámbito de una red de ordenadores y por último está ip_PLC que es la IP del PLC que hay asociado al Arduino. Esta dirección habrá que modificarla en los distintos puestos, junto con la de la librería como se comentó en la sección 7.1.1

Código 7.3 Definiciones para los módulos.

```
/*** Creamos los objetos necesarios ***/
MgsModbus Mb;                                    // Objeto
   MgsModbus llamado Mb
MFRC522 mfrc522(RFID, RST_PIN);                  // Objeto MFRC522
    llamado mfrc522 al que le pasamos los pines que hemos definido
MFRC522 mfrc522D(RFIDD, RST_PIN);                // Objeto MFRC522
    llamado mfrc522 al que le pasamos los pines que hemos definido
   para el dectector

MFRC522::MIFARE_Key key;                         // Estructura
   tipo key para las claves de acceso a las etiquetas RFID

/*** Configuración de la red ***/
uint8_t mac[] = {0x90, 0xA2, 0xDA, 0x0E, 0x94, 0xB5 }; // Dirección
   MAC de nuestro módulo
IPAddress ip(192, 168, 0, 8);                    // Dirección IP
   del módulo
IPAddress gateway(192, 168, 0, 1);               // Dirección IP
   principal de la red
IPAddress subnet(255, 255, 255, 0);              // Máscara de red
IPAddress ip_PLC(192, 168, 0, Fin_IP_Servidor);  // Dirección IP
   del PLC del puesto
IPAddress ip_PLC_P(192, 168, 0, Fin_IP_Ppal);    // Dirección IP
   del PLC principal
```

7.1.4 Definición de las variables del programa

En esta sección se encuentran todas las variables necesarias para el funcionamiento del programa.

tiempo: Variable que se usa para almacenar el valor de millis() en un determinado momento y luego poder comprobar si ha pasado suficiente tiempo para ejecutar la función que pide instrucciones al PLC. Al medir tiempo, tiene que ser unsigned long para que quepa el valor del tiempo en milisegundos.

tiempo2: Variable que al igual que *tiempo* sirve para medir periodos de tiempo. También tiene que ser un *long* sin signo.

retardo_pide_inst: Tiempo, en milisegundos, entre cada petición de instrucción al PLC.

bloque: Valor en el que se guarda el bloque en el que hay una posición libre para guardar una instrucción.

bytes_libres: En esta variable se guarda la información del lugar del bloque que tiene huecos libres en los que se puede escribir.

bytes_sector: Número de bytes de cada sector de la tarjeta. Como se ha visto, solo hay 16 bytes por bloque, pero cuando se lee un bloque hay que guardarlo en 18 bytes, ya que hay dos bytes extra que son necesarios guardar aunque no se usen.

dato_escribir[4]: Vector para almacenar la información ya en formato tarjeta. Como se vio, solo son necesarios 4 bytes.

DEA: Dirección de escritura en Arduino. Hay que definir un bloque de memoria en el Arduino en el que escribir la información procedente del PLC. Con esta variable se indica donde empieza ese bloque.

DEP: Dirección de escritura en el PLC. Hay que definir un bloque de memoria en el PLC en el que escribir la información procedente del Arduino. Con esta variable se indica donde empieza ese bloque.

DLA: Dirección de lectura en Arduino. Hay que definir un bloque de memoria en el Arduino en el que leer la información a enviar al PLC. Con esta variable se indica donde empieza ese bloque.

DLP: Dirección de lectura en el PLC. Hay que definir un bloque de memoria en el PLC en el que leer la información a enviar al Arduino. Con esta variable se indica donde empieza ese bloque.

inst_tarj[4]: Variable intermedia que se usa al pasar los datos de formato PLC a formato tarjeta y viceversa

instruccion[255][bytes_RFID]: Matriz que contiene la lista de las instrucciones recibidas del PLC, ya en formato tarjeta, para escribir en las etiquetas.

n_instruccion: Contiene la cantidad de instrucciones que hay en la tarjeta, una vez se ha leído su bloque de comprobación.

orden[tam]: Instrucciones entrante del PLC ya guardadas en formato PLC.

Palet_actual: Número de palet retenido en el puesto de trabajo.

palets_parar[255]: Lista de palets a parar. El valor almacenado en cada posición del vector es el número de acciones a realizar sobre dicho palet. Por ejemplo, si hay 5 tareas a realizar sobre el palet número 3, en la posición 3 de este vector habrá un 5.

retenedor: Variable que indica si hay que parar la bandeja

tarjeta[64][16]: Matriz en la que se vuelva la información de la tarjeta completa para trabajar con esta matriz en Arduino en vez de ir trabajando bloque a bloque de la tarjeta.

valor_1: Será igual a uno si está el retenedor parado y por lo tanto hay un palet parado. Vale como variable bandera para entrar a realizar tareas sobre la etiqueta.

debug_tarj_stop: La única de esta utilidad es para mostrar por pantalla un mensaje si la bandeja está parada.

encontrado: Bandera para indicar donde hay un hueco libre para escribir.

PPC: 'Paso por cero'Variable para no leer la misma instrucción dos veces, hay que pasar por información recibida antes de volver a recibir algo.

sector: Número del sector sobre el que se trabaja.

trailerBlock: Contiene el número del último bloque del sector sobre el que se está trabajando. Este dato servirá para saber que grupo de claves usar para autentificar el bloque deseado.

blockAddr: En esta variable irá indicando cual es el bloque sobre el que se está trabajando, tanto para escritura como para lectura.

Código 7.4 Definición de las variables del programa.

```
/**** Variables del programa ***/
unsigned long tiempo;                                   // Variable para
    medir tiempo, tiene que ser unsigned long para que quepa el valor
     del tiempo en milisegundos
unsigned long tiempo2;                                   // Variable para
     medir tiempo, tiene que ser unsigned long para que quepa el
    valor del tiempo en milisegundos
unsigned long int retardo_pide_inst = 200;
uint8_t bloque;                                         // Valor en el
    que guardamos el bloque en el que hay una posición libre para
    guardar una instrucción
uint8_t bytes_libres;                                   // Lugar de los
    bytes dentro del bloque que están libres para ser sobre escritos
uint8_t bytes_sector = 18;                              // Número de
    bytes de cada sector de la tarjeta
uint8_t dato_escribir[4];                               // Datos que
    queremos guardar en la tarjeta
uint8_t DEA = 0;                                        // Dirección de
    escritura del vector en Arduino
uint8_t DEP = 8;                                        // Dirección de
    escritura de la memoria del PLC
uint8_t DLA = 8;                                        // Dirección de
    lectura del vector en Arduino
uint8_t DLP = 0;                                        // Dirección de
    lectura de la memoria del PLC
uint8_t inst_tarj[4];                                   // Orden a
    ejecutar en el PLC leida de la tarjeta
uint8_t instruccion[255][bytes_RFID];                   // Lista de las
    instrucciones recibidas del PLC, ya en formato tarjeta
uint8_t n_instruccion = 0;                              // Número de
    instrucciones almacenadas para saber donde escribir otra
uint8_t Nueva_orden[4];
uint8_t orden[tam];                                     // Instrucciones
    entrante del PLC en formato PLC
uint8_t Palet_actual = 0;                               // Almacenamos el
     palet que tenemos parado
uint8_t palets_parar[255];                              // Lista de
    palets a parar. El valor almacenado en cada posición del vector
    es el número de acciones a realizar sobre dicho palet
uint8_t retenedor = 0;                                  // Variable que
    nos indica si hay que parar la bandeja
uint8_t tarjeta[64][16];                                // Variable en la
     que volcamos la información de la tarjeta
uint8_t valor_1;
bool a = 0, b = 0;                                      // Indicadores
    para cuando esté todo hecho sobre la tarjeta se pueda bajar el
    retenedor
```

```
bool debug_tarj_stop = 1;
bool encontrado;                                    // Flag para
   indicar donde podemos escribir
bool PPC = 1;                                       // 'Paso por cero
   ' Variable para no leer la misma instrucción dos veces, hay que
   pasar por información recibida antes de volver a recibir algo

/*** Variables necesarias para la lectura de la tarjeta RFID ***/
uint8_t sector    = 1;                              // Sector que
   usaremos.
uint8_t trailerBlock = 7;                           // Último bloque
   del sector que usaremos, en el que están las claves
uint8_t blockAddr = 4;                              // Primer bloque
   del sector que usaremos. Los bloques se cuenta de manera continua.
```

7.1.5 Configuración del puerto SPI

La comunicación SPI hay que configurarla independientemente para cada periférico que se comunique utilizando este protocolo. Para la configuración se usa la función SPISettings(A,B,C), donde la A es la frecuencia de transmisión, la B hace referencia al orden de transmisión, pudiendo ser LSBFIRST, que envía en primer lugar los bits menos significativos o MSBFIRST, que envía en primer lugar los bits más significativos y por último la C que indica el modo de transmisión. En este caso, ambos periféricos se comunican a 16Mhz, que es a lo que equivale el valor 16000000. En el orden de envío de la información, el módulo RFID envía primero los bits menos significativos, mientras que el módulo Ethernet envío primero los bits más significativos. Por último, el modo de transmisión en ambos periféricos es el modo 0.

Se dejan las dos configuraciones a modo de ejemplo, pero solo es necesario definir el modo al ir a iniciar la comunicación SPI con el módulo, ya que en su librería no está este comando. En cambio, en la librería del MFRC522 si está declarado el modo de la comunicación SPI para que se configure esta antes de establecer la comunicación.

Código 7.5 Configuración del puerto SPI.

```
/***     Configuración del puerto SPI para los distintos módulos ***/
/***          En este caso, la velocidad será de 16Mhz  ***/
/*** El módulo RFID transmite primero el bit menos significativo (LSB
   ) ***/
/*** El módulo Ethernet transmite primero el bit más significativo (
   MSB) ***/
/***        El modo del SPI en ambos casos será el 0    ***/
SPISettings Config_RFID(SPISettings(16000000,LSBFIRST,SPI_MODE0));
SPISettings Config_Ethernet(SPISettings(16000000,MSBFIRST,SPI_MODE0))
   ;
```

7.1.6 Inicialización de elementos necesarios y de las comunicaciones

Esta función es la primera en ejecutarse en el código de Arduino y solo se ejecuta una vez cuando se enciende o reinicia el Arduino.

Le primera linea después de la cabecera de la función es la asignación de el número de IP a la variable de a librería MODBUS master encargada de almacenar este valor.

A continuación está la declaración de los pines de los distintos módulos conectados al Arduino. Para la comunicación por SPI hay que declarar el pin de esclavo de cada módulo como salida. Como ya se asoció el número del pin de esclavo con su variable, basta con que se indiquen estas como OUTPUT.

Para indicarle a un módulo que se quiere iniciar la comunicación con él, se pone su pin es esclavo a nivel bajo. En este caso no se va a comunicar con ellos, por lo que se ponen todas las salidas a nivel alto.

La comunicación por puerto seria está habilitada para poder mostrar mensajes por pantalla en caso de que se quiera tener un mayor control sobre el flujo del programa. La velocidad se ha definido en 115200 baudios. Se muestra un mensaje de bienvenida al iniciar el programa que indica la versión del programa y la IP del esclavo con el que se va a trabajar.

El siguiente paso es iniciar la comunicación SPI con el comando SPI.begin() y si se quiere iniciar la comunicación con uno de los módulos, basta con llamar a la función Seleccion_Modulo pasándole el nombre del pin del esclavo que se quiere activar.

Se han de iniciar los dos módulos MFRC522 y definir las claves de los bloques.

También se ha de configurar el módulo Ethernet. Para ello se selecciona el módulo y se le pasan todas las direcciones IP necesarias.

Por último se inicializan a cero todos los vectores de datos intercambiados entre el PLC y las etiquetas. Con la orden Mb.Req se manda por modbus el vector de órdenes inicializado a 0. Como se vio, se comunica por modbus según el modo 16, escribir múltiples registros al usar la orden MB_FC_WRITE_MULTIPLE_-REGISTERS.

Código 7.6 Inicialización de elementos.

```
/*** Programa de ejecución única ***/
void setup() {                                  // Programa de
    ejecución única

   Mb.remSlaveIP = ip_PLC;                      // Dirección IP
      del dispositivo esclavo de la comunicación SPI
   MbP.remSlaveIP = ip_PLC_P;                   // Dirección IP
      del PLC principal
   /*** Habilitamos los pines de salida necesarios ***/
   pinMode(SD_Ether, OUTPUT);                   //
   pinMode(Ether, OUTPUT);                      // Pin de Slave
      del módulo Ethernet
   pinMode(Inter, OUTPUT);                      // Pin mediante
      el cual le decimos al PLC maestro que debe accionar el
      retenedor del puesto en el que estamos
   pinMode(RFIDD, OUTPUT);                      // Pin de Slave
      del módulo RFID-MFRC522 detector
   pinMode(RFID, OUTPUT);                       // Pin de Slave
      del módulo RFID-MFRC522
   pinMode(53, OUTPUT);                         // Se debe
      definir el pin 53 como salida para el correcto funcionamiento
      del SPI
   digitalWrite(SD_Ether, HIGH);
   digitalWrite(Ether, HIGH);                   // Desactivamos
      el SPI del módulo Ethernet
   digitalWrite(RFIDD, HIGH);                   // Desactivamos
      el SPI del módulo RFID
   digitalWrite(RFID, HIGH);                    // Desactivamos
      el SPI del módulo RFID
```

```
  Serial.begin(115200);                              // Inicialización
      de la comunicación serie para mostrar por pantalla informació
     n
  Serial.println("Programa Simple v5.3");
  Serial.println(Mb.remSlaveIP);

  SPI.begin();                                       // Inicialización
      del bus SPI estableciendo los pines correspondientes como
     salidas y al nivel requerido

  /*** Inicialización del módulo RFID ***/
  Seleccion_Modulo(RFID);                            // Desactivación
     del pin de slave correspondiente al módulo RFID
  mfrc522.PCD_Init();                                // Inicializamos
     el módulo RFID
  for (byte i = 0; i < 6; i++)                       // Inicialización
      de los valores de las keys que vienen por defecto en el chip
     para poder acceder a la tarjeta
     key.keyByte[i] = 0xFF;
  delay(100);
  for (byte i = 0; i < 6; i++)
     key.keyByte[i] = 0xFF;                          // No escribe
        bien la primera vez, asi que hay que volver a escribir el
        primer numero de la clave para que no de error

  delay(100);

  /*** Inicialización del módulo RFID ***/
  Seleccion_Modulo(RFIDD);                            // Desactivación
      del pin de slave correspondiente al módulo RFID
  mfrc522D.PCD_Init();
  for (byte i = 0; i < 6; i++)                       // Inicialización
      de los valores de las keys que vienen por defecto en el chip
     para poder acceder a la tarjeta
     key.keyByte[i] = 0xFF;
  delay(100);
  for (byte i = 0; i < 6; i++)
     key.keyByte[i] = 0xFF;                          // No escribe
        bien la primera vez, asi que hay que volver a escribir el
        primer numero de la clave para que no de error

  delay(100);                                         // Probamos a
     darle tiempo para que cierre una conexión antes de activar la
     siguiente

  /*** Inicialización del módulo Ethernet ***/
//  Seleccion_Modulo(Ether);                           // Desactivació
    n del pin de slave correspondiente al módulo RFID
  Seleccion_Modulo(Ether);                           // Desactivación
     del pin de slave correspondiente al módulo RFID

  tiempo2 = millis();
  while (millis() - tiempo2 > 200) {}

  SPI.beginTransaction(Config_Ethernet);          // Inicializamos
     la comunicación por SPI con la configuración del RFID
```

```
    Ethernet.begin(mac, ip, gateway, subnet);        // Paso de los
       parámetros que hemos definido de la red para inicializar el mó
       dulo

    tiempo2 = millis();
    while (millis() - tiempo2 > 200) {}

    for (int i = 0; i < tam << 1; i++)                  // Ponemos a
       cero todos los registros y el vector de instrucciones
    {
       Mb.MbData[i] = 0;
       orden[i] = 0;
    }
    for (int i = 0; i < 256; i++) {                  // Ponemos a 0
       todos los elementos del vector palets_parar para no encontrar
       basura
       palets_parar[i] = 0;
    }
    Mb.Req(MB_FC_WRITE_MULTIPLE_REGISTERS, 0 , tam << 1 , 0,
       Fin_IP_Servidor);
    Mb.MbmRun();
    SPI.endTransaction();

    SPI.beginTransaction(Config_Ethernet);
    MbP.MbData[0] = 0;
    MbP.Req(MB_FC_WRITE_COIL, 0 , 0 , 0, Fin_IP_Ppal);
    MbP.MbmRun();

    SPI.endTransaction();
}
```

7.1.7 Código de ejecución ciclica o principal

La función *loop* se ejecuta cíclicamente sin que haya que llamarla. En ella va el cuerpo principal del programa.

En el *loop* de este proyecto hay dos funciones que se llaman siempre en cada ciclo. La primera es la encargada de preguntar al PLC si tiene algún mensaje que entregar. Esto se hace en cada ciclo para que se le puedan mandar instrucciones siempre al Arduino. La segunda función es la encargada de comprobar si hay alguna bandeja sobre el módulo detector que haya que parar.

La siguiente parte de la función solo se ejecuta si la variable valor_1 es igual a uno. Esta variable es modificada por la función Comprueba_tarjeta, que si detecta que hay que parar la bandeja, la pone a uno. Una vez se ha cumplido la condición y se ha entrado en esta sección, la siguiente tarea es esperar hasta que el segundo módulo RFID detecta la tarjeta. Una vez se ha detectado la tarjeta, se pasa a leerla hasta que no quede ningún dato referente al puesto en que se encuentra el Arduino. Las órdenes leidas se le pasan al PLC para que las ejecute. Después de leer y mandar las ordenes leidas al PLC, se procede a escribir en la tarjeta las nuevas instrucciones que tenía almacenadas el Arduino. Cuando se terminan de escribir las instrucciones, se manda la orden al PLC maestro de que baje el retenedor y se vuelve a empezar un ciclo del *loop*

Código 7.7 Código principal.

```
void loop() {                                        // Programa de
    ejecución cíclica
```

```
/*
 * 'Pregunta' al PLC si quiere escribir información en una tarjeta
    , si hay información disponible, se va almacenando en un
    vector para cuando llegue el palet,
 * poder realizar la escritura de una vez, sin esperar las
    instrucciones del PLC una a una con el palet parado y
    ralentizando el sistema
 */
Pide_instruccion();                                 // 'Pregunta' al
   PLC si quiere escribir/comprobar información en una tarjeta (
   en cual y qué)

Comprueba_tarjeta(palets_parar);                    // Le pasamos el
   vector con los palets que queremos parar

if(valor_1) {                                       // Si está el
   retenedor parado y por lo tanto hay un palet parado,
   procedemos a realizar las acciones necesarias
   /* Hay que leer la tarjeta o escribir algo, sino no estaría el
      retenedor activo. Hay que comprobar tambien la siguiente
      accion por si se realiza en el mismo puesto a
    * continuación de la que acabamos de realizar, debemos seguir
      en bucle hasta que la siguiente acción sea en otro PLC o no
       haya mas información para escribir en la tarjeta.
    * Cada vez que se realice una acción hay que actualizar la
      matriz 'instruccion'.
    */

   uint8_t detector = 1;
   Seleccion_Modulo(RFID);
   while ( detector ) {
      if ( mfrc522.PICC_IsNewCardPresent())      // Busqueda de
         tarjetas cercanas
      {
         if (mfrc522.PICC_ReadCardSerial()) detector = 0; //
            Selección de la tarjeta de la que leeremos y
            escribiremos
      }
   }

   delay(100);
   while(a) {                                       // Vamos leyendo
      la tarjeta y mandandole la información al PLC de la acción
      a realizar en el puesto
      Serial.println("Leyendo tarjeta");
      a = Lee_tarjeta();
      Manda_instruccion();
      /*
       * Hay que meter un flag para que el PLC nos indique cuando
         ha terminado de realizar la acción y podamos mandarle la
          siguiente
       */
   }
   Serial.println("Tarjeta leida");
   if (b){
      Escribe_tarjeta();
      Serial.println("Tarjeta escrita");
```

```
        }
        /*
         * Hay que ver si realizar una pasada más por si la información
             que acabamos de escribir queremos que se realice YA.
            Podemos seguir pidiendo instrucciones y usar un
         * byte más para saber cuando podemos dejar pasar el palet,
            pero habria que asegurarse que dicho byte esté normalmente
            a 0 y solo se ponga a 1 cuando se quiera esperar
         */
        Retenedor(0);                                  // Señal de activaci
           ón del retenedor a nivel bajo
        valor_1 = 0;
        delay(1500);                                   // Damos un
           segundo y medio de margen para que salga la bandeja
        debug_tarj_stop = 1;
    }
    else
    {
        if (debug_tarj_stop == 1) {
            Serial.println("Tarjeta parada");
            debug_tarj_stop = 0;
        }
        Seleccion_Modulo(RFID);                        // Desactivación
           del pin de slave correspondiente al módulo RFID
        // Halt PICC
        mfrc522.PICC_HaltA();                          // Paramos la
           tarjeta
        // Stop encryption on PCD
        mfrc522.PCD_StopCrypto1();
    }
}
```

7.1.8 Funciones para la selección del módulo SPI

La función *Seleccion_Modulo* es la encargada de seleccionar el esclavo con el que se quiere iniciar la comunicación SPI.

Esta función no devuelve nada, pero recibe el nombre de la variable que tiene asociado el pin de selección de cada esclavo. Aunque se le pase el nombre de la variable, lo que recibe la función es un entero.

El funcionamiento es muy simple, pone todos los pines de los esclavos a nivel alto. De esta manera es seguro que todos van a estar desconectados. Y la última instrucción es pone a nivel bajo el pin que ha recibido la función, habilitando el esclavo asociado a dicho pin.

Código 7.8 Selección del periférico SPI.

```
/* Función que recibe el pin del módulo SPI que queremos activar */
void Seleccion_Modulo (int Pin) {
    digitalWrite(Ether, HIGH);
                                                                    //
       Desactivamos el SPI del módulo Ethernet
```

```
    digitalWrite(RFIDD, HIGH);
                                                                    //
        Desactivamos el SPI del módulo RFID
    digitalWrite(RFID, HIGH);
                                                                     //
        Desactivamos el SPI del módulo RFID
    digitalWrite(Pin, LOW);

        // Activamos el SPI del módulo requerido
}
```

7.1.9 Autentificación de los bloques

La autentificación de los distintos bloques de las tarjetas es una acción que se repite mucho durante el programa, por ello se ha decidido separarla en una función a parte.

Esta función recibe el número del bloque de claves correspondiente al bloque sobre el que se quiere trabajar y no devuelve nada. Como se ve, se autentifica el bloque con la clave A y la B para que sea posible tanto la lectura como la escritura del bloque. En caso de que haya algún problema en la autentificación se muestra un mensaje por pantalla diciendo con qué clave se ha producido el error.

Código 7.9 Autentificación de tarjetas.

```
/*** Autentificamos la tarjeta usando las claves A y B ***/
void Autentificar_Bloque(uint8_t bl_aut) {

      MFRC522::StatusCode status;
      status = (MFRC522::StatusCode) mfrc522.PCD_Authenticate(MFRC522
          ::PICC_CMD_MF_AUTH_KEY_A, bl_aut, &key, &(mfrc522.uid));
      if (status != MFRC522::STATUS_OK) {
         Serial.print(F("Problema al autentificar la tarjeta en
             lectura A, código: "));
         Serial.println(mfrc522.GetStatusCodeName(status));
         return false;
      }
      status = (MFRC522::StatusCode) mfrc522.PCD_Authenticate(MFRC522
          ::PICC_CMD_MF_AUTH_KEY_B, bl_aut, &key, &(mfrc522.uid));
      if (status != MFRC522::STATUS_OK) {
         Serial.print(F("Problema al autentificar la tarjeta en
             lectura B, código: "));
         Serial.println(mfrc522.GetStatusCodeName(status));
         return;
      }
}
```

7.1.10 Autentificación de los bloques del detector

Esta función cumple el mismo cometido que la función de la sección 7.1.9, pero en este caso se usa para el módulo detector y poder leer el bloque de comprobación de la etiqueta.

Se usa una función distinta porque los objetos a los que van dirigidos son distintos, aunque la función en sí sea igual.

Código 7.10 Autentificación de tarjetas.

```
/*** Autentificamos la tarjeta usando las claves A y B ***/
void Autentificar_Bloque_D(uint8_t bl_aut) {

      MFRC522::StatusCode status;
      status = (MFRC522::StatusCode) mfrc522D.PCD_Authenticate(
         MFRC522::PICC_CMD_MF_AUTH_KEY_A, bl_aut, &key, &(mfrc522D.
         uid));
      if (status != MFRC522::STATUS_OK) {
         Serial.print(F("Problema al autentificar la tarjeta en
            lectura A, código: "));
         Serial.println(mfrc522D.GetStatusCodeName(status));
         return false;
      }
      status = (MFRC522::StatusCode) mfrc522D.PCD_Authenticate(
         MFRC522::PICC_CMD_MF_AUTH_KEY_B, bl_aut, &key, &(mfrc522D.
         uid));
      if (status != MFRC522::STATUS_OK) {
         Serial.print(F("Problema al autentificar la tarjeta en
            lectura B, código: "));
         Serial.println(mfrc522D.GetStatusCodeName(status));
         return;
      }

}
```

7.1.11 Detección y comprobación de tarjetas

Esta función es la encargada de comprobar si la tarjeta que pasa por encima del detector tiene que pararse para realizar alguna acción sobre ella.

Como el único cometido de esta función es comprobar si hay que parar el palet, solo tiene que actuar el módulo RFID detector. Por lo tanto, todas las llamadas a funciones relacionadas con el RFID se hacen con los parámetros del detector.

En la función se crea un vector para almacenar la información del bloque que se va a leer. Como se dijo en la sección 7.1.4, este vector de lectura tiene que ser dos bytes mayor que el tamaño del bloque. A continuación se llama a la función *Autentificar_Bloque_D* a la que se le pasa el número del bloque de claves correspondiente al bloque de comprobación. Una vez autentificado, se pasa a leer el bloque de comprobación. La información leída se guarda en el vector declarado al principio de la función.

Una vez leído el bloque de comprobación, si comprueba si el primer byte es igual al número del puesto en que se encuentra situado el Arduino o si la posición del vector palet igual al segundo byte del bloque de comprobación es distinto de 0. En caso de que una de estas dos condiciones se cumpla, se activa el retenedor y la variable bandera *valor_1* se pone a 1, para indicar que se ha activado el retenedor. A continuación, dependiendo de cual de las dos condiciones se haya cumplido, se pone a 1 la variable a o la b. Por último se guarda el número del palet que lleva la bandeja.

Los dos últimos comandos de esta función se encargan de desvincular la etiqueta, de manera que si vuelve a entrar en su rango de lectura la pueda volver a leer. Si la tarjeta no sale del rango de lectura no se volverá a leer.

Código 7.11 Detección y comprobación de tarjetas.

```
/* Vamos comprobando las tarjetas que pasan a ver si tenemos que
   parar una porque queramos escribir en ella o porque haya que
   realizar una acción sobre ella
 * Le mandamos el número de palet que tenemos que parar, si no
    queremos parar ninguno, le pasamos palet 0.
 */
void Comprueba_tarjeta(uint8_t Palet[]) {

   Seleccion_Modulo(RFIDD);

   if ( ! mfrc522D.PICC_IsNewCardPresent())
                                                    // Busqueda de
      tarjetas cercanas
      return;
   //Serial.println("Tarjeta detectada");
   if ( ! mfrc522D.PICC_ReadCardSerial())
                                                      // Selección de
      la tarjeta de la que leeremos y escribiremos
      return;
   //Serial.println("Tarjeta seleccionada");
   MFRC522::StatusCode status;
   /* Hay que probar a sacarlo de aqui a ver si funciona y se gana
      algo de velocidad */
   byte buffer[bytes_sector];
                                                                  //
      Vector en el que guardaremos los valores leidos
   byte size = sizeof(buffer);
   Serial.println();

   Autentificar_Bloque_D(Fin_Bloque_Comprobacion);

   /*** Leemos de la tarjeta la direccion del bloque definida ***/
   status = (MFRC522::StatusCode) mfrc522D.MIFARE_Read(
      Bloque_Comprobacion, buffer, &size);
   if (status != MFRC522::STATUS_OK) {
                                                        // Si se ha
      leido correctamente continuamos
         Serial.println("Fallo de lectura");
         Serial.println(mfrc522D.GetStatusCodeName(status));
         return;
   }
   //Serial.println("Tarjeta comprobada");
   //Serial.println(buffer[1]);                    /***Debugger***/
   //Serial.println(buffer[0]);

   /*
    * Comprobamos si se cumple alguna de las dos condiciones
       siguientes (condición OR), para detener la bandeja:
    * - El primer byte corresponde con el puesto en que nos
       encontramos, lo que querrá decir que sobre el palet debemos
       realizar algún trabajo en dicho puesto
    * - El segundo byte corresponde, que indica el número de nuestro
       palet, corresponde con uno de los que el PLC quiere parar para
        trabajar sobre el palet
    */
```

```
    if (buffer[0] == Puesto || Palet[buffer[1]]) {
                                                    // **** Debería funcionar
        , sino añadir un != 0 despues de Palet[buffer[1]] ***
        digitalWrite(Inter,HIGH);
                                                                          //
            Si se cumple alguna de las condiciones anteriores activamos
             el retenedor
        valor_1 = 1;

            // Flag para indicar que se ha activado el retenedor
        Serial.println("Retenedor activado");
        delay(1000);
        if (buffer[0] == Puesto)
            a = 1;

                // Indicamos que debemos leer de la tarjeta
        if (Palet[buffer[1]] != 0)
            b = 1;

                // Indicamos que debemos escribir en la tarjeta
        Palet_actual = buffer[1];
    }
    // Halt PICC
    mfrc522.PICC_HaltA();                          // Paramos la
        tarjeta
    // Stop encryption on PCD
    mfrc522.PCD_StopCrypto1();

}
```

7.1.12 Petición de instrucciones al PLC

Esta función es la encargada de leer las instrucciones enviadas por el PLC medante MODBUS. No recibe ni devuelve ningún dato.

Como la introducción por parte del usuario no sería posible hacerla cada pocos milisegundos, se ha introducido un retardo entre cada petición de instrucciones. Este retardo se puede controlar con la variable *retardo_pide_inst*.

Una vez se ha entrado completamente en la función, se activa el módulo Ethernet y se inicia la comunicación SPI. Una vez inicializado el módulo Ethernet, ya que es el encargado de la comunicación MODBUS con el PLC, se lanza el comando *Mb.MbmRun()* que es el encargado de poner en modo master al Arduino. Con el modo master ya iniciado se leen de las posiciones de memoria del PLC que se han indicado en la declaración de variables.

Se tiene que comprobar si el mensaje que se ha leido es nuevo o es uno antiguo. Para eso se lee el ultimo byte del vector leido y el valor de *PPC*. Si los dos valores son iguales a 1 quiere decir que la instrucción es nueva, por lo que se procede a guardarla en una variable interna de Arduino. A la vez que se va guardando la instrucción, se va copiando en el bloque de memoria dedicado al envío hacia el PLC. De esta manera el PLC puede comprobar si la información que se ha enviado corresponde con la que esta enviandole el Arduino. Una vez está toda la información copiada, esta se escribe en la dirección de memoria dirigida al PLC. Se pone a 0 *PPC* para que no se vuelva a leer esta instrucción. Toda esta información se va mostrando por pantalla mediante el puerto serie para que se pueda ver en Arduino que vector se ha recibido.

El siguiente paso, es transformar la información recibida en formato PLC a formato tarjeta. Esto no sería

necesario si los bloques de información fueran del mismo tamaño, pero en este caso hay que pasar de 8 bytes a 4 bytes. Para la transformación de los datos se ha hecho uso del desplazamiento y de las operaciones lógicas con bits. Por ejemplo, la instrucción *orden[0] « 4* desplaza *orden[0]* cuatro bits a la izquierda. Con esto se consigue que al guardar el dato en otra variable, los 4 bits más significativos contengan la información de *orden[0]* y los 4 bits menos significativos sean cero. La siguiente instrucción hace uso de la operación lógica OR. Con esto por ejemplo se consigue que al hacer un OR con el número 16 (0001000 en binario) se ponga el bit de realizado a uno, que es el cuarto bit menos significativo.

La información pasada a formato tarjeta también es mostrada en pantalla, aunque es menos intuitiva que la recibida por el PLC.

Antes de salir de la función, hay que volver a lanzar el comando que establece al Arduino como master en la comunicación MODBUS y cerrar la comunicación por SPI para que no haya conflictos con el modo del mismo en futuras comunicaciones.

Código 7.12 Petición de instrucciones al PLC.

```
/* 'Pregunta' al PLC si quiere escribir/comprobar información en una
   tarjeta (en cual y qué), devuelve la orden a escribir
 */
void Pide_instruccion() {
   if (millis() - tiempo > retardo_pide_inst) {
                                                    // Esperamos unas dé
      cimas de segundo entre cada ejecución de esta función
      tiempo = millis();

         // Vamos guardando el tiempo del reloj interno del Arduino
         para saber el tiempo entre una ejecución y otra de esta
         función
      Seleccion_Modulo(Ether);

         // Activamos el pin slave del shield Ethernet
      SPI.beginTransaction(Config_Ethernet);
                                                          // Activamos
         la comunicación SPI con el módulo Ethernet
      Mb.MbmRun();

         // Lanzamos la instrucción para que funcione el Modbus
         master
      Mb.Req(MB_FC_READ_REGISTERS, DLP , tam , DEA , Fin_IP_Servidor)
         ;                                  // Indicamos de que
         direccion de memoria del PLC queremos leer, cuantos
         registros y en que ubiación guardar la información en
         Arduino

      if (Mb.MbData[tam - 1] == 1 && PPC == 1) {
                                                      // Comprobamos si
         hay algo que escribir o si se quiere comprobar la informaci
         ón de la etiqueta
         Serial.print("Instrucción en formato PLC: [ ");
         orden[0] = Mb.MbData[0];

            // Pasamos el primer byte que hemos recibido del PLC a
            la primera posición de un vector para trabajar con esta
            variable interna
         Serial.print(orden[0]);
```

```
        Mb.MbData[tam] = orden[0];
                                                                    //
            Pasamos el dato recibido a los datos a enviar al PLC
        for(uint8_t i = 1; i < tam; i++) {
                                                                //
           Mediante el bucle seguimos extrayendo las posiciones del
            vector leido por modbus del PLC
           Serial.print(" , ");
           orden[i] = Mb.MbData[i];

              // Copiamos el vector en una variable propia
           Serial.print(orden[i]);
           /*
            * Guardamos la información en las posiciones que van a
              continuación de las usadas para recibir la informació
              n
            * Con esto conseguimos pasarle al PLC la información
              recibida para que él compruebe si lo enviado y lo
              recibido es lo mismo
            */
           Mb.MbData[i + tam] = orden[i];
        }
        Mb.Req(MB_FC_WRITE_MULTIPLE_REGISTERS, DEP , tam , DLA ,
           Fin_IP_Servidor);        // Escribimos los datos en la
           dirección de memoria indicada por los parámetros, en
           modo 16, es decir, escribir en múltiples registros
        Serial.println(" ]");
        PPC = 0;

           // Para indicar que ya hemos leido la información y no
           vovler a leerla, ponemos PPC a 0
        /* Traducimos la orden al formato en que guardamos la
           información en el tag y la vamos guardando en una matriz
            para poder
         * realizar varias acciones cuando se pare el palet. También
             vamos almacenando los palets que queremos parar en el
            puesto.
         */
        instruccion[n_instruccion][0] = orden[5];
                                                    // Número de
           palet sobre el que actuar
        instruccion[n_instruccion][1] = orden[3];
                                                    // Número de
           tarea, para orden de realización
        instruccion[n_instruccion][2] = orden[0] << 4;
                                                // Hacemos un
           desplazamiento de bits, ya que el byte lo comparten la
           orden a realizar con los posibles errores que ocurran.
           Los 4 primeros bits son para la orden y los 4 últimos
           son para grabar el error
        instruccion[n_instruccion][3] = orden[4] << 5 | 16 | orden
           [1];                    // Volvemos a hacer un
           desplazamiento para guardar el número de puesto en el
           que queremos que se realice la tarea, introducimos los
           detalles y ponemos el realizado a 1, para saber que no
           se ha realizado aún
```

```
            n_instruccion++;

               // Incrementamos el número de acciones para escribir en
               la siguiente fila
            palets_parar[orden[5]] = palets_parar[orden[5]] + 1;
                                          // Actualizamos el número de
             acceiones a realizar sobre el palet que queremos parar

               /*** Debugging ***/
            Serial.print("Instrucción en formato tarjeta: [ ");
            Serial.print(instruccion[n_instruccion - 1][0],HEX);
            Serial.print(" , ");
            Serial.print(instruccion[n_instruccion - 1][1],HEX);
            Serial.print(" , ");
            Serial.print(instruccion[n_instruccion - 1][2],HEX);
            Serial.print(" , ");
            Serial.print(instruccion[n_instruccion - 1][3],HEX);
            Serial.println(" ]");
            //return 1;
         }
         else {
            if (Mb.MbData[tam - 1] == 0) {
               Mb.MbData[tam * 2 - 1] = 0;

                  // Hacemos 0 la última posición del vector de envío
                  al PLC por si recibimos una nueva petición
               PPC = 1;
            }
            //return 0;
         }
         Mb.MbmRun();
         SPI.endTransaction();
      }
      //SPI.endTransaction();
}
```

7.1.13 Envío de instrucciones al PLC

Esta función es la encargada de mandar las instrucciones al PLC. No recibe ni devuelve ningún parámetro.

Al igual que la función de la sección anterior (7.1.12, hay que establecer al Arduino como maestro para la comunicación por MODBUS y hay que definir para en este caso, no mandar dos veces la misma información. Para eso se usa la bandera *PPC_M*. Aunque en este caso se pretenda mandar información al PLC, se leen los posibles mensajes que haya en la memoria, para evitar sobreescribir alguna orden no realizada aún.

Si se comprueba que no hay ninguna orden pendiente, se procede a escribir la primera orden de la cola en el vector a enviar al PLC. Esta información se va mostrando por pantalla también a través del puerto serie. Para indicar que la información va desde el Arduino al PLC, el octavo byte del vector se pone a 2.

Cuando se han pasado todos los datos al vector, se escribe esta en los bloques de memoria definidos mediante la escritura de múltiples registros.

Para no reenviar varias veces el mismo mensaje, se comprueba que el PLC responde con el mismo vector. Entonces se puede salir de esta función para mandarle otra nueva instrucción o dejar pasar la bandeja.

Código 7.13 Envío de instrucciones al PLC.

```
oid Manda_instruccion() {
   uint8_t PPC_M = 1;
   Seleccion_Modulo(Ether);
                                                                    //
      Activamos el pin slave del shield Ethernet
   SPI.beginTransaction(Config_Ethernet);
                                                  // Inicializamos
      la comunicación por SPI con la configuración de Ethernet
   Mb.MbmRun();
   tiempo2 = millis();
   while (millis() - tiempo2 > 200) {}
   Serial.println("Esperando para mandar otra instrucción ");
   Mb.Req(MB_FC_READ_REGISTERS, DLP , tam , DEA ,Fin_IP_Servidor);
   if (Mb.MbData[tam - 1] == 0 && Mb.MbData[tam - 2] != 2) {

      tiempo2 = millis();
      while (millis() - tiempo2 > 200) {}

      Mb.MbmRun();
      for(uint8_t i = 0; i < tam; i++) {
          Mb.MbData[i + tam] = inst_tarj[i];
                                                                //
             Guardamos la información en las posiciones que van a
             continuación de las usadas para recibir la información
          Serial.print("Instrucción de tarjeta: "); Serial.print(
             inst_tarj[i]); Serial.print(" ");
      }
      Serial.println();
      Mb.MbData[(tam << 1) - 1] = 2;
   }

   tiempo2 = millis();
   while (millis() - tiempo2 > 200) {}
   Mb.MbmRun();
   Mb.Req(MB_FC_WRITE_MULTIPLE_REGISTERS, DEP , tam , DLA ,
      Fin_IP_Servidor);
   Serial.println("Nueva orden escrita");
   tiempo2 = millis();
   while (millis() - tiempo2 > 1050) {}
   Mb.MbmRun();
   Mb.Req(MB_FC_READ_REGISTERS, DLP , tam , DEA , Fin_IP_Servidor);
   Mb.MbmRun();

   do {
      if (millis() - tiempo2 > 800) {
         Mb.MbmRun();
         Mb.Req(MB_FC_READ_REGISTERS, DLP , tam , DEA ,
            Fin_IP_Servidor);
         Mb.MbmRun();
         if(Mb.MbData[tam - 2] == 2) {
            PPC_M = 0;
            Mb.MbData[(tam << 1) - 2] = 1;
            Mb.MbData[tam - 2] = 0;
            Mb.Req(MB_FC_WRITE_MULTIPLE_REGISTERS, DEP , tam , DLA ,
               Fin_IP_Servidor);
```

```
            Mb.MbmRun();
         }
         else {
            if(Mb.MbData[tam - 2] == 1) {
               Mb.MbData[(tam << 1) - 2] = 0;
               Mb.MbmRun();
               Mb.Req(MB_FC_WRITE_MULTIPLE_REGISTERS, DEP , tam , DLA
                    , Fin_IP_Servidor);
               Mb.MbmRun();
            }
         }

         tiempo2 = millis();
      }
   }
   while(PPC_M == 1);

   Serial.println("Salimos del bucle");

   tiempo2 = millis();
   while (millis() - tiempo2 > 200) {}

   Mb.MbmRun();
   SPI.endTransaction();
}
```

7.1.14 Lectura de tarjeta

Esta función es de las más extensas del programa. Se encarga de leer todos los bloques de la tarjeta y actualizar el bloque de comprobación. No recibe ningún parámetro, pero devuelve un 1 en caso de que haya extraído algún dato de la tarjeta o un 0 si no ha extraído nada.

Para la lectura, se define un vector local y para la escritura, por si fuera necesario, otro. Se crean también un vector de dos posiciones por si se detecta que la segunda tarea a realizar es en el mismo puesto.

La ejecución de instrucciones dentro de esta función comienza con un bucle que va recorriendo todos los bloques de la tarjeta, evitando el primer sector y el último bloque de cada sector. Según el sector, va cambiando el número del bloque de autentificación.

El primer paso dentro del bucle es leer cada bloque completo y en caso de ocurrir algún error mostrar un mensaje de error. La información leída se pasa a una matriz de Arduino que imita la distribución de la información dentro de la tarjeta.

El siguiente paso es con la información recién guardada en la matriz de Arduino que imita la distribución de la tarjeta, buscar si alguno de los cuatro bloques de cuatro bytes contiene tareas sin realizar. Una vez localizadas las tareas sin realizar del bloque, se comprueba si alguna de ellas es la segunda tarea a realizar. En caso de que se encuentre, se guarda en el vector *marca_lectura[]* en que puesto se realiza y sobre que palet. También se comprueba si alguna de las tareas es la número 1. En caso de encontrarla, se transforma la información e formato tarjeta a formato PLC y se marca la tarea como realizada. En cada bloque que se encuentra una instrucción que no es la número uno, se va decrementando en una unidad el número de la instrucción, porque si se entra en el bucle es porque hay una instrucción número uno que en algún momento se va a extrae. Así se consigue actualizar la lista completa.

Cada bloque que se extrae y se modifica al buscar tareas en él, se vuelve a escribir en la tarjeta una vez se ha modificado.

Una vez se ha acabado de recorrer la tarjeta y se ha salido del bucle hay que actualizar el bloque de comprobación.

Como se han extraído tareas hay que actualizar el bloque de comprobación con el número de tareas que hubiera, menos las que se haya extraído. También se actualiza al número de palet y de puesto de la siguiente instrucción en caso de que se haya encontrado.

Por último, se devuelve un cero o un uno dependiendo de si la siguiente tarea a realizar es en el mismo puesto.

Código 7.14 Lectura de tarjeta.

```
/* La información que saquemos de la tarjeta la guardaremos en la
   variable inst_tarj. Esta será la variable que tendremos que
   mandar al PLC para que la ejecute.
 * Devuelve un 0 si la siguiente acción es en otro puesto o un uno si
     hay que seguir en el mismo puesto para la siguiente acción.
 * Leemos las instrucciones hasta encontrar la que debemos ejecutar,
    recibimos la información del bloque 1
 */
bool Lee_tarjeta() {

   Seleccion_Modulo(RFID);
   SPI.beginTransaction(Config_RFID);
                                                    // Inicializamos
      la comunicación por SPI con la configuración del RFID

   MFRC522::StatusCode status;

   uint8_t buffer_lectura[bytes_sector];
                                                  // Vector en el que
      guardaremos los valores leidos
   uint8_t buffer_escritura[bytes_sector - 2];
                                                 // Vector en el que
      guardaremos los valores a escribir
   uint8_t size = sizeof(buffer_lectura);
   uint8_t size_escr = sizeof(buffer_escritura);
                                               // Variable que almacena el
      tamaño de buffer de lectura
   bool cuenta_acc_borrar = 0;
                                                                // Nos
      indica si tenemos que borrar una acción del contador del
      bloque de comprobación
   /* En este vector almacenamos el número de puesto en el que se
      realizará la segunda acción y el número
    * de palet sobre el que se actuará una vez se haya extraido la
       información de la primera acción a realizar */
   uint8_t marca_lectura[2] = {0, 0};

   uint8_t Fin_Block = 7;
                                                                    //
      Variable en la que vamos guardando la posición del último
      bloque del sector actual

   /* Hay que ir leyendo fila a fila hasta encontrar la tarea numero
      uno para mandarsela al PLC */
```

```
for (uint8_t i = 4; i < 63; i++) {
                                                    // Recorremos los
    bloques, evitando el primer sector y los bloques de las claves
    if(i == 7 || i == 11 || i == 15 || i == 19 || i == 23 || i ==
       27 || i == 31 || i == 35 || i == 39 || i == 43 || i == 47
       || i == 51 || i == 55 || i == 59) {
       i++;
       Fin_Block = i + 3;
    }

    uint8_t Block = i;
                                                                    //
        Vamos recorriendo los bloques de cada sector
    Autentificar_Bloque(Fin_Block);

    status = (MFRC522::StatusCode) mfrc522.MIFARE_Read(Block,
       buffer_lectura, &size);
    if (status != MFRC522::STATUS_OK) {
                                                  // Si se ha leido
       correctamente continuamos
        Serial.print(F("Problema al leer la tarjeta, código: "));
        Serial.println(mfrc522.GetStatusCodeName(status));
                                  // Código de error
        SPI.endTransaction();
        return 0;
    }
    for (uint8_t j = 0; j < 16; j++)
        tarjeta[Block][j] = buffer_lectura[j];
    for (uint8_t c = 0; c < 15; c = c + 4) {
                                                 // Recorremos los
       bytes de la tarjeta, como almacenamos la información en
       conjuntos de 4 bytes, en un bloque caben 4 instrucciones
       if ((tarjeta[Block][c + 3] >> 4 & 1) == 1) {
                                          // Buscamos las tareas que
           aún no se hayan realizado
          /* Si la tarea es la segunda, la guardamos para
             posteriormente actualizar el bloque de
           * comprobación
           */
          if(tarjeta[Block][c + 1] == 2) {
             // Guardamos la posición de la siguiente acción para
                actualizarla posteriormente
             marca_lectura[0] = tarjeta[i][c + 3] >> 5 & 7;
                                       // Número de puesto
             marca_lectura[1] = tarjeta[i][c];
                                                    // Número de
                palet
             a = 1;
          }
          /* Si la tarea es la primera, se la pasamos a la variable
              inst_tarj para pasarle la instrucción
           * al PLC. Descodificamos la instrucción almacenada en la
              tarjeta de manera que el PLC la
           * reciba claramente al no tener problemas de espacio
           */
          if(tarjeta[Block][c + 1] == 1 && cuenta_acc_borrar == 0)
             {
```

```
                Serial.print("Accion 1 encontrada: "); Serial.print(c
                    + 1); Serial.print(" , "); Serial.println(Block);
                inst_tarj[0] = tarjeta[Block][c + 2] >> 4 & 15;
                                        // Orden que tiene que
                    realizar el PLC
                inst_tarj[1] = tarjeta[Block][c + 3] & 15;
                                                // Detalle de la orden a
                     realizar (posición, color, ...)
                inst_tarj[2] = 0;

                    //
                inst_tarj[3] = 1;

                    // Número de tarea actualizada, siempre será 1
                inst_tarj[4] = tarjeta[Block][c + 3] >> 5 & 7;
                                        // Puesto en el que estamos
                    realizando la tarea
                inst_tarj[5] = tarjeta[Block][c];
                                                        // Palet sobre
                    el que estamos actuando
                inst_tarj[6] = tarjeta[Block][c + 2] & 15;
                                                // Posible error que se
                    haya almacenado
                inst_tarj[7] = 1;

                    // Indicador de nueva orden
                tarjeta[Block][c + 3] = tarjeta[Block][c + 3] - 16;
                                        // Marcamos la acción como
                    realizada o "sacada" de la tarjeta
                cuenta_acc_borrar = 1;
             }
             else {
                if(tarjeta[Block][c + 1] != 0)
                   tarjeta[Block][c + 1] = tarjeta[Block][c + 1] - 1;
             }
          }
       }
//      Serial.println(Block);
       for (uint8_t j = 0; j < 16; j++) {
           buffer_escritura[j] = tarjeta[Block][j];
//           Serial.print(buffer_escritura[j]);
//           Serial.print(" : ");
       }
//      Serial.println();
       Autentificar_Bloque(Fin_Block);
       status = (MFRC522::StatusCode) mfrc522.MIFARE_Write(Block,
          buffer_escritura, 16); // Multiplicamos por cuatro y le
          sumamos n para ir recorriendo los bloques de los sectores
       if (status != MFRC522::STATUS_OK) {
                                                        // Si se ha leido
           correctamente continuamos
           Serial.print("Problema al actualizar la tarjeta, código: "
              );
           Serial.println(mfrc522.GetStatusCodeName(status));
           SPI.endTransaction();
           return 0;
       }
```

```
    }
    Serial.println("Instruccion actualizada en la tarjeta");

    Autentificar_Bloque(Fin_Bloque_Comprobacion);

    /* Leemos el bloque de comprobación para actualizarlo con la nueva
        orden */
    status = (MFRC522::StatusCode) mfrc522.MIFARE_Read(
       Bloque_Comprobacion, buffer_lectura, &size);
    if (status != MFRC522::STATUS_OK) {
                                                          // Si se ha
       leido correctamente continuamos
           SPI.endTransaction();
           return 0;
    }
    if (marca_lectura[0] != 0) buffer_lectura[0] = marca_lectura[0];
                            // Guardamos el número de puesto donde se
       realizará la siguiente acción
    if (marca_lectura[1] != 0) buffer_lectura[1] = marca_lectura[1];
                            // Guardamos el número de palet sobre el
       que se realizará la siguiente acción
    if (cuenta_acc_borrar == 1) buffer_lectura[5]--;
                                              // Decrementamos el número
        de acciones almacenadas en 1
    if (buffer_lectura[5] == 0) buffer_lectura[0] = 0;
    for (uint8_t i = 0; i < 16; i++) {
       buffer_escritura [i] = buffer_lectura[i];
    }
    status = (MFRC522::StatusCode) mfrc522.MIFARE_Write(
       Bloque_Comprobacion, buffer_escritura, 16); // Actualizamos la
        información de comprobación
    if (status != MFRC522::STATUS_OK) {
                                                          // Si se ha
       leido correctamente continuamos
           SPI.endTransaction();
           return 0;
    }
    Serial.println("Bloque de comprobación actualizado después de
       escribir en la tarjeta");

    Serial.print("Marca lectura 0: "); Serial.println(marca_lectura
       [0]);
    /* Comprobamos si la siguiente instrucción es en este mismo puesto
        para volver a leer la tarjeta en la siguiente iteración o
       dejar pasar la bandeja */
    if (marca_lectura[0] == Puesto)
       return 1;
    else
       return 0;



    SPI.endTransaction();
       /*
    // Halt PICC
    mfrc522.PICC_HaltA();                              // Paramos la
       tarjeta
```

```
    // Stop encryption on PCD
    mfrc522.PCD_StopCrypto1();
*/

}
```

7.1.15 Escritura en tarjeta

Esta función es la encargada de escribir paquetes de información en las tarjetas. No recibe ni devuelve ningún valor.

Se definen dos vectores locales, uno para la lectura y otro para la escritura.

Como en la sección 7.1.14, se selecciona el módulo RFID lector, pero en este caso lo primero que se hace es autentificarse en el bloque de comprobación y leerlo. La información que se busca en este bloque es la correspondiente al número de tareas guardadas. Este dato es importante porque así se puede saber si hay hueco para escribir nuevas órdenes.

Si se está ejecutando esta función es porque hay una bandeja parada. El siguiente paso es recorrer la matriz de instrucciones pendientes de escribir en una tarjeta. Luego se va comprobando cuales de estas instrucciones están pendientes de ser escritas en el palet parado y la que se encuentre se pasa a un vector que será en la tarjeta. En este punto es cuando se comprueba si hay huecos libres en la tarjeta. Si se detecta que hay hueco, se comprueba si el número de instrucción que se quiere escribir es superior al número de instrucciones escritas en la tarjeta. Si es así, se guardará con el número inmediatamente superior al número de tareas guardadas. Por ejemplo, si hay 5 tareas guardadas y se intenta guardar una tarea como la 8, se cambiará ese 8 por un 6, que sería el número de tarea inmediatamente superior al número de tareas guardadas.

El siguiente paso es recorrer los sectores en busca de un hueco donde escribir la instrucción. Para ello se va accediendo a cada bloque y se va comprobando si las instrucciones están realizadas y si el orden de realización que tienen asignado es igual o superior al de la instrucción que se desea guardar. En caso de que sea igual o superior, se aumentará en uno el número de esa instrucción. Por ejemplo, hay tareas guardadas de la uno a la siete y se quiere almacenar la tarea número tres. Desde la tarea tres a la siete se aumentarán en una unidad, pasando a ser de la cuatro a la ocho. La comprobación seguirá hasta encontrar un hueco libre, momento en que guardará la instrucción con el número que se le ha dicho, pero seguirá aumentando el número de las tareas que sea necesario aumentar. Cada vez que se aumenta el número de tarea hay que modificar el bloque y volverlo a escribir con la información modificada.

Una vez se ha escrito la instrucción en la tarjeta, hay que poner a cero la fila de la matriz de ordenes pendientes de escribir.

Si se ha escrito una nueva instrucción, hay que actualizar el bloque de comprobación, por lo que se lee y se aumenta en un el número de tarea. En caso de que la instrucción introducida sea la primera, también hay que actualizar el bloque de comprobación con el número de palet y puesto de la tarea.

Si no se ha realizado nada de lo anterior porque no se ha encontrado hueco en la tarjeta, se inicia la comunicación con el PLC para mandarle un mensaje de error indicándole que no hay huecos en la tarjeta para escribir la instrucción deseada.

Código 7.15 Escritura en tarjeta.

```
/* Escribimos en la tarjeta la información deseada, debemos recibir
   la información a escribir
```

```
 * En este bloque escribimos en la tarjeta las ordenes que recibimos.
     Para ello debemos comprobar si hay espacio donde escribir la
    nueva orden,
 * si hay hueco, hay que ver dónde, y una vez encontrado el lugar,
    guardar la información y actualizar el orden de las tareas que
    había previamente.
 * También hay que actualizar el número de ordenes que tenemos
    almacenadas, para llevar el control de los espacios disponibles.
 * Si la tarea a escribir es la que primero queremos que se realice,
     debemos actualizar también esta información en el bloque de
    comprobación.
 */
void Escribe_tarjeta() {

  Seleccion_Modulo(RFID);
  SPI.beginTransaction(Config_RFID);

  Serial.println("Escribimos en la tarjeta");

  MFRC522::StatusCode status;

  uint8_t buffer_lectura[bytes_sector];
                                                    // Vector en el
     que guardaremos los valores leidos
  uint8_t buffer_escritura[bytes_sector];
                                                   // Vector en el
     que guardaremos los valores a escribir
  uint8_t size = sizeof(buffer_lectura);
                                                    // Variable que
     almacena el tamaño de buffer

  Autentificar_Bloque(Fin_Bloque_Comprobacion);

  /* Leemos el bloque de comprobación para ver si hay hueco donde
     escribir */
  status = (MFRC522::StatusCode) mfrc522.MIFARE_Read(
     Bloque_Comprobacion, buffer_lectura, &size);
  uint8_t num_inst = buffer_lectura[5];
  if (status != MFRC522::STATUS_OK) {
                                                      // Si se ha
     leido correctamente continuamos
      SPI.endTransaction();
      return 0;
  }
  Serial.println("Autentificado para escribir en la tarjeta");
  /* Buscamos las instrucciones del PLC que queremos pasar a la
     tarjeta
   * Hay que entrar varias veces para sacar todas las instrucciones,
       ya que como está, cuando encuentra una instrucción se sale a
      escribirla
   * Habría que cerrar el primer for después de que se cierra el if
      que comprueba si hay hueco
   */
  for (uint8_t k = 0; k < 255; k++) {
     if (instruccion[k][0] == Palet_actual) {
        Serial.print("Linea: ");Serial.println(k);
        for (uint8_t j = 0; j < 4; j++){
```

```
            dato_escribir[j] = instruccion[k][j];
            Serial.print(" Posición ");Serial.print(j);Serial.print("
                : ");Serial.print(dato_escribir[j],HEX);
        }
        Serial.println();
        Serial.print("Instrucción extraida para ser escrita");
        Serial.println();
        /* Si hay menos de 256 ordenes escritas, es que al menos hay
            un hueco libre y podemos escribir en la tarjeta*/
        if (num_inst < 179) {
                                                                    //
            Tenemos 3 bloques por sección, en cada uno caben 4
            instrucciones y hay 15 secciones, ya que la primera es
            información del palet
            Serial.println("Hay hueco para escribir la instrucción en
                la tarjeta");
            /* Buscamos donde podemos escribir */
            encontrado = 0;

                // Utilizamos esta variable por si no queda espacio
                en la tarjeta, poder mostrar un error

            if(dato_escribir[1] > num_inst) {
                                                                //
                Buscamos si queremos introducir una instrucción con
                un orden superior al numero de tareas que hay
                dato_escribir[1] = num_inst + 1;
                                                                 // Si es
                    mayor, le asignamos el numero de tareas mas uno,
                    para asignarle el ultimo puesto de la cola
                Serial.print("Disminuida la instrucción a la ");Serial
                    .println(dato_escribir[1]);
            }

            uint8_t Fin_Block = 7;
            for (uint8_t i = 4; i < 63; i++) {
                                                               // Vamos
                recorriendo los bloques de cada sector
                if(i == 7 || i == 11 || i == 15 || i == 19 || i == 23
                    || i == 27 || i == 31 || i == 35 || i == 39 || i
                    == 43 || i == 47 || i == 51 || i == 55 || i == 59)
                     {
                    i++;
                    Fin_Block = i + 3;
                }
                uint8_t Block = i;

                    // Vamos recorriendo los bloques de cada sector

                Autentificar_Bloque(Fin_Block);

                status = (MFRC522::StatusCode) mfrc522.MIFARE_Read(
                    Block, buffer_lectura, &size);
                if (status != MFRC522::STATUS_OK) {
                                                              // Si se ha
                     leido correctamente continuamos
                     SPI.endTransaction();
```

```
        return 0;
    }

    /* Buscamos los huecos libres en la tarjeta para
       escribir nuevas instrucciones y vamos reordenando
       las instrucciones que estén en la tarjeta pero que
     * vayan después de la que vamos a introducir. Si
        tenemos que introducir una tarea en la posición 3,
         la que hay en la 3 pasará a la 4 antes de
        introducir
     * la nueva, la 4 a la 5 y así sucesivamente
     */
    for (uint8_t c = 0; c < 15; c = c + 4) {
                                          // Recorremos
       los bytes de la tarjeta, como almacenamos la
       información en conjuntos de 4 bytes, en un bloque
       caben 4 instrucciones
        if( (buffer_lectura[c + 3] >> 4 & 1 == 1) && (
           buffer_lectura[c + 1] >= dato_escribir[1])) {
           // Vamos buscando tareas por realizar y si la
           tarea actual es mas alta que la que queremos
           escribir
            buffer_lectura[c + 1] = buffer_lectura[c + 1]
                + 1;                // Si la tarea a
               escribir es anterior a la que hay, esta
               habrá que aumentarla 1 posición para
               cuando introduzcamos la nueva orden se
               queden las demás ordenadas
            Serial.print("Aumentada la instrucción
               existente a la ");Serial.println(
               buffer_lectura[c + 1]);
            status = (MFRC522::StatusCode) mfrc522.
               MIFARE_Write(Block, buffer_lectura, 16);
            if (status != MFRC522::STATUS_OK) {
                                          // Si se ha
               leido correctamente continuamos
                SPI.endTransaction();
                return 0;
            }
            Serial.println("Aumentado el orden de la
               tarea");
        }
        if( !(buffer_lectura[c + 3] >> 4 & 1 == 1) && (
           encontrado == 0)) {                  //
           Buscamos tareas que se hayan realizado
            //num_tarea = buffer_lectura[c + 1];
                                         //
               Actualizamos num_tarea solo cuando hemos
               encontrado una tarea más antigua
            bytes_libres = c;

               // Guardamos en qué byte empieza el
               espacio donde podemos escribir
            bloque = Block;

               // Almacenamos la dirección del bloque en
               el que queremos escribir
```

```
                    Serial.print("Hueco encontrado en bloque ");
                       Serial.print(bloque); Serial.print(",
                       posición "); Serial.println(bytes_libres);
                    for (uint8_t m = 0; m < size ; m++)
                        buffer_escritura[m] = buffer_lectura[m];
                                                // Guardamos la
                           información que contiene el bloque
                           para que cuando tengamos que
                           sobreescribirlo, no borremos las otras
                            instrucciones del bloque
                    encontrado = 1;

                       // Marcamos como que se ha encontrado un
                       espacio libre
                    i = 64; c = 16;

                       // Marcamos para no seguir buscando huecos
                        libres
                }
            }
        }

        if (encontrado == 0) {
          /* Mostramos error por no encontrar el hueco.
             Esto no debería pasar a menos que se haya programado
                mal lo anterior
          */
          }
        for (uint8_t i = 0; i < 4 ; i++) {
//            Serial.print(" dato_escribir[i]: ");Serial.println(
    dato_escribir[i]);
//            Serial.print(" bytes_libres + i: ");Serial.println(
    bytes_libres + i);
            buffer_escritura[bytes_libres + i] = dato_escribir[i];
        }
//         for (uint8_t m = 0; m < size ; m++) {
//            Serial.print(" Buffer_escritura: ");Serial.println(
    buffer_escritura[m]);}
//
        uint8_t fin_bloque = bloque - bloque % 4 + 3;
        //Serial.println(fin_bloque);      // Comprobamos si está
            bien calculado el final del bloque

        /* Le pasamos la dirección del bloque de comprobación */
        Autentificar_Bloque(fin_bloque);

        /*** Escribimos en la tarjeta en la direccion del bloque
           definida ***/
        status = (MFRC522::StatusCode) mfrc522.MIFARE_Write(
           bloque, buffer_escritura, 16);
        if (status != MFRC522::STATUS_OK) {
                                                              //
           Si se ha leido correctamente continuamos
           SPI.endTransaction();
           return;
        }
```

```
            /* Si hemos encontrado hueco y ya se ha escrito la
               instrucción en el espacio libre de la tarjeta, la
               información almacenada en la matriz 'instruccion'
             * la podemos borrar para no volver a leerla
             */
            for (uint8_t j = 0; j < 4; j++) {
               instruccion[k][j] = 0;
               Serial.print("Borrando instrucción [");Serial.print(k)
                  ;Serial.print(", ");Serial.print(j);Serial.println
                  ("]");
            }

            /* Ahora tenemos que actualizar el número de tareas que
               tenemos almacenadas. Para ello cargamos el bloque 1
               para actualizar su información.
             * A parte, si la tarea a realizar es la primera, debemos
                 colocarla como la primera de la lista, modificndo
                también la información del bloque 1.
             */
             Autentificar_Bloque(Fin_Bloque_Comprobacion);

            status = (MFRC522::StatusCode) mfrc522.MIFARE_Read(
               Bloque_Comprobacion, buffer_lectura, &size); //
               Cargamos el bloque de la información principal
            if (status != MFRC522::STATUS_OK) {
                                                              // Si
               se ha leido correctamente continuamos
               SPI.endTransaction();
               return;
            }

            buffer_lectura[5] = buffer_lectura[5] + 1;
                                                    // Aumentamos
               en uno el contador del registro de tareas
            num_inst = buffer_lectura[5];
            Serial.print("El número de tareas guardadas es: ");Serial
               .println(buffer_lectura[5]);
            if (dato_escribir[1] == 1) {

               // Comprobamos si la tarea que acabamos de escribir
               es la primera en realizarse, si es así actualizamos
               el bloque 1
               buffer_lectura[0] = dato_escribir[3] >> 5 & 7;
                                                 // Guardamos el nú
                  mero del puesto en que la bandeja tiene que parar
               if (dato_escribir[2] >> 2 == 5)
                                                                 //
                  Comprobamos si la acción a realizar es recoger un
                  palet, en cuyo caso tendremos que escribir el nú
                  mero de palet que llevamos
                  buffer_lectura[1] = dato_escribir[0];
                                                               //
                     Guardamos el número de palet en que queremos
                     actuar, en caso de que lo que queramos sea
                     actuar sobre un palet
            }
```

```
            status = (MFRC522::StatusCode) mfrc522.MIFARE_Write(
               Bloque_Comprobacion, buffer_lectura, 16); //
               Actualizamos la información de comprobación
            if (status != MFRC522::STATUS_OK) {
                                                              // Si
               se ha leido correctamente continuamos
               SPI.endTransaction();
               return;
            }
         }
         else {
            SPI.endTransaction();

               // Cerramos la comunicación SPI con el módulo RFID
            for (uint8_t j = 0; j < 4; j++)
               instruccion[k][j] = dato_escribir[j];
            Serial.println("No hay huecos disponibles en la tarjeta
               para escribir la información");

            /* Mostramos un error de que no hay espacio disponible
               para escribir en la tarjeta, habría que definir unos
               códigos de error */
            Seleccion_Modulo(Ether);

               // Activamos el pin slave del shield Ethernet
            SPI.beginTransaction(Config_Ethernet);
                                                              //
               Inicializamos la comunicación por SPI con la
               configuración del RFID

            Mb.MbsRun();

               // Iniciamos la comunicación Modbus TCP con el PLC
            Mb.MbData[2 + tam] = 10;

               // Le damos el valor 10 al error por quedarnos sin
               huecos en la tarjeta

            SPI.endTransaction();
            break;
         }
      }
   }
   Serial.println("Tarjeta escrita");
   // Halt PICC
   mfrc522.PICC_HaltA();

      // Paramos la tarjeta
   // Stop encryption on PCD
   mfrc522.PCD_StopCrypto1();
   SPI.endTransaction();
}
```

7.1.16 Gestión del retenedor por Arduino

Esta última función del código de Arduino es la encargada de enviarle la señal al PLC principal para que actúe sobre el retenedor del puesto. Esta función recibe un uno cuando se quiere subir el retenedor y un cero cuando se quiere bajar.

Para el envío de vectores de bytes se usaba la función *MB_FC_WRITE_MULTIPLE_REGISTERS*, pero en este caso solo es necesario enviar una señal de tipo booleano, ya que los posibles estados del retenedor son solo 1 o 0, por lo que se usa la función *MB_FC_WRITE_COIL*.

Código 7.16 Gestión del retenedor.

```
void Retenedor(bool rete){
   SPI.beginTransaction(Config_Ethernet);          // Inicializamos
      la comunicación por SPI con la configuración del RFID

   tiempo2 = millis();
   while (millis() - tiempo2 > 200) {}

   MbP.MbData[0] = rete;
   MbP.Req(MB_FC_WRITE_COIL, 0 , 0 , 0, Fin_IP_Ppal);
   MbP.MbmRun();

   SPI.endTransaction();
}
```

7.2 Código en Unity Pro

En la parte del PLC se han creado dos módulos para el tratamiento de las instrucciones y la pantalla de explotación.

7.2.1 Gestión de las instrucciones mediante botón único

Este módulo de Unity tiene como objetivo que no se intenten enviar varias ordenes a la vez.

Como se ha visto, hay una serie de botones encargados de indicar qué acción hay que realizar. Por defecto, los botones no son excluyentes unos de otros. Con este módulo lo que se ha hecho es que al pulsar uno, los demás se desactiven, de forma que solo sea posible que haya una instrucción seleccionada.

Código 7.17 Gestión de las instrucciones en Unity Pro.

```
(* Con Flanco [] detectamos cambios en las variables para detectar
   cmbios en los botones y resetear los demás *)
IF Recoge_de_Palet = 1  AND Flanco[6] = 0   THEN

  (* Cada vez que se produce una pulsación se resetean los
     indicadores de flanco *)
  FOR i :=1 TO 9 BY 1 DO
    Flanco[i] := 0;
```

```
  END_FOR;

  (* Se activa solo el de la pulsación última *)
  Flanco [6] := 1;

  (* Desactivamos los demás botones *)
  Recoge_de_Alimentador := 0;
  Deja_en_Palet := 0;
  Deja_en_alimentador := 0;
  Saca_Palet := 0;
  Saca_Bandeja := 0;
  Recoge_Palet := 0;
  Recoge_Bandeja := 0;
  Comprueba_Pieza := 0;

ELSIF Recoge_de_Alimentador = 1 AND Flanco[5] = 0  THEN
  FOR i :=1 TO 9 BY 1 DO
    Flanco[i] := 0;
  END_FOR;
  Flanco [5] := 1;
  Recoge_de_Palet := 0;
  Deja_en_Palet := 0;
  Deja_en_alimentador := 0;
  Saca_Palet := 0;
  Saca_Bandeja := 0;
  Recoge_Palet := 0;
  Recoge_Bandeja := 0;
  Comprueba_Pieza := 0;
ELSIF Deja_en_Palet = 1  AND Flanco[3] = 0  THEN
  FOR i :=1 TO 9 BY 1 DO
    Flanco[i] := 0;
  END_FOR;
  Flanco [3] := 1;
  Recoge_de_Palet := 0;
  Recoge_de_Alimentador := 0;
  Deja_en_alimentador := 0;
  Saca_Palet := 0;
  Saca_Bandeja := 0;
  Recoge_Palet := 0;
  Recoge_Bandeja := 0;
  Comprueba_Pieza := 0;
ELSIF Deja_en_alimentador = 1  AND Flanco[2] = 0  THEN
  FOR i :=1 TO 9 BY 1 DO
    Flanco[i] := 0;
  END_FOR;
  Flanco [2] := 1;
  Recoge_de_Palet := 0;
  Recoge_de_Alimentador := 0;
  Deja_en_Palet := 0;
  Saca_Palet := 0;
  Saca_Bandeja := 0;
  Recoge_Palet := 0;
  Recoge_Bandeja := 0;
  Comprueba_Pieza := 0;
ELSIF Saca_Palet = 1  AND Flanco[9] = 0   THEN
  FOR i :=1 TO 9 BY 1 DO
    Flanco[i] := 0;
```

```
  END_FOR;
  Flanco [9] := 1;
  Recoge_de_Palet := 0;
  Recoge_de_Alimentador := 0;
  Deja_en_Palet := 0;
  Deja_en_alimentador := 0;
  Saca_Bandeja := 0;
  Recoge_Palet := 0;
  Recoge_Bandeja := 0;
  Comprueba_Pieza := 0;
ELSIF Saca_Bandeja = 1  AND Flanco[8] = 0   THEN
  FOR i :=1 TO 9 BY 1 DO
    Flanco[i] := 0;
  END_FOR;
  Flanco [8] := 1;
  Recoge_de_Palet := 0;
  Recoge_de_Alimentador := 0;
  Deja_en_Palet := 0;
  Deja_en_alimentador := 0;
  Saca_Palet := 0;
  Recoge_Palet := 0;
  Recoge_Bandeja := 0;
  Comprueba_Pieza := 0;
ELSIF Recoge_Palet = 1  AND Flanco[7] = 0  THEN
  FOR i :=1 TO 9 BY 1 DO
    Flanco[i] := 0;
  END_FOR;
  Flanco [7] := 1;
  Recoge_de_Palet := 0;
  Recoge_de_Alimentador := 0;
  Deja_en_Palet := 0;
  Deja_en_alimentador := 0;
  Saca_Palet := 0;
  Saca_Bandeja := 0;
  Recoge_Bandeja := 0;
  Comprueba_Pieza := 0;
ELSIF Recoge_Bandeja = 1  AND Flanco[4] = 0 THEN
  FOR i :=1 TO 9 BY 1 DO
    Flanco[i] := 0;
  END_FOR;
  Flanco [4] := 1;
  Recoge_de_Palet := 0;
  Recoge_de_Alimentador := 0;
  Deja_en_Palet := 0;
  Deja_en_alimentador := 0;
  Saca_Palet := 0;
  Saca_Bandeja := 0;
  Recoge_Palet := 0;
  Comprueba_Pieza := 0;
ELSIF Comprueba_Pieza = 1  AND  Flanco[1] = 0 THEN
  FOR i :=1 TO 9 BY 1 DO
    Flanco[i] := 0;
  END_FOR;
  Flanco [1] := 1;
  Recoge_de_Palet := 0;
  Recoge_de_Alimentador := 0;
  Deja_en_Palet := 0;
```

```
  Deja_en_alimentador := 0;
  Saca_Palet := 0;
  Saca_Bandeja := 0;
  Recoge_Palet := 0;
  Recoge_Bandeja := 0;
END_IF;
```

7.2.2 Envío de órdenes

Este módulo es el encargado del tratamiento de las instrucciones introducidas en la ventana de explotación.

El código se encarga de recoger la información introducida en cada recuadro de entrada de texto y almacenarlo en una posición del vector que se le enviará a Arduino. También guarda la orden correspondiente al botón seleccionado.

La información no es almacenada en el vector que se le enviará a Arduino hasta que se pulse el botón *Confirmar*. Una vez se han guardado todos los datos, se reinician los casilleros y la botonera a al espera de introducir otra orden.

Código 7.18 Envío de órdenes en Unity Pro.

```
IF Confirmar = 1 THEN


  (* Comprobamos si no hay ninguna asignación de orden. En caso
     contrario escribimos la orden requerida en la primera posicion
     del vector Orden *)
  IF  (Recoge_de_Palet OR Recoge_de_Alimentador OR Deja_en_Palet OR
     Deja_en_alimentador OR Saca_Palet OR Saca_Bandeja OR
     Recoge_Palet OR Recoge_Bandeja OR Comprueba_Pieza) = 0 THEN
     Error_Instruccion := 1;
   ELSIF Recoge_de_Palet  = 1  THEN
     Orden[0] := 1;
   ELSIF Recoge_de_Alimentador = 1 THEN
     Orden[0] := 2;
   ELSIF Deja_en_Palet  = 1  THEN
     Orden[0] := 3;
   ELSIF Deja_en_alimentador = 1  THEN
     Orden[0] := 4;
   ELSIF Saca_Palet  = 1 THEN
     Orden[0] := 5;
   ELSIF Saca_Bandeja  = 1 THEN
     Orden[0] := 6;
   ELSIF Recoge_Palet  = 1 THEN
     Orden[0] := 7;
   ELSIF Recoge_Bandeja  = 1  THEN
     Orden[0] := 8;
   ELSIF Comprueba_Pieza  = 1  THEN
     Orden[0] := 9;
  END_IF;

  IF Error_Instruccion = 0 THEN
    Orden[1] := Detalle;
    (*Orden[2] := 0;*)
    Orden[3] := Numero_Tarea;
```

```
    Orden[4] := Puesto;
    Orden[5] := Numero_Palet;
    (*Orden[6] := 0;*)
    Orden[7] := 1;
  END_IF;

  (* Reiniciamos la botonera *)
  Confirmar    := 0;
  Recoge_de_Palet   := 1;
  Recoge_de_Alimentador   := 0;
  Deja_en_Palet    := 0;
  Deja_en_alimentador   := 0;
  Saca_Palet    := 0;
  Saca_Bandeja    := 0;
  Recoge_Palet    := 0;
  Recoge_Bandeja    := 0;
  Comprueba_Pieza   := 0;
  Detalle    := 0;
  Numero_tarea    := 0;
  Numero_Palet   := 0;
  Puesto     := 0;


END_IF;

(* Cuando recibimos un 1 quiere decir que el Arduino ha recibido la
   orden correctamente y ya podemos dejar de indicarsela *)
IF Recibido [7] = 1 THEN
  Orden [7] := 0;
END_IF;

IF Recibido [6] = 1 THEN
  Orden [6] := 0;
END_IF;

(* Si recibimos un error del Arduino lo pasamos a la variable
   Error_Arduino para mostrarla por pantalla *)
IF Recibido [2] <> 0 THEN
  Error_Arduino := Recibido[2];
END_IF;
```

7.2.3 Otros datos

Cuando se recibe una orden de Arduino, el PLC la procesa y se la envía a las máquinas de su puesto para que realicen dicha orden. Una vez se ha acabado de realizar la orden, hay que enviarle algún dato a Arduino para que sepa que puede mandar la siguiente orden, o dejar pasar el palet. Para enviar esta información se ha creado el siguiente programa en Ladder.

Como se puede ver en la figura 7.1, para indicar que se ha realizado la orden, se envía un 2 en la posición 6 del vector *Orden* cuando se pone *Accion_Acabada* a 1. Cuando se asocie un programa en el PLC a este nuevo sistema, basta con poner *Accion_Acabada* a 1.

En la figura 7.2 aparecen las variables necesarias para el funcionanmiento del programa de Unity Pro. Se puede ver como los vectores de envío y recepción de datos tienen asociadas las direcciones de memoria en las que escribirá Arduino.

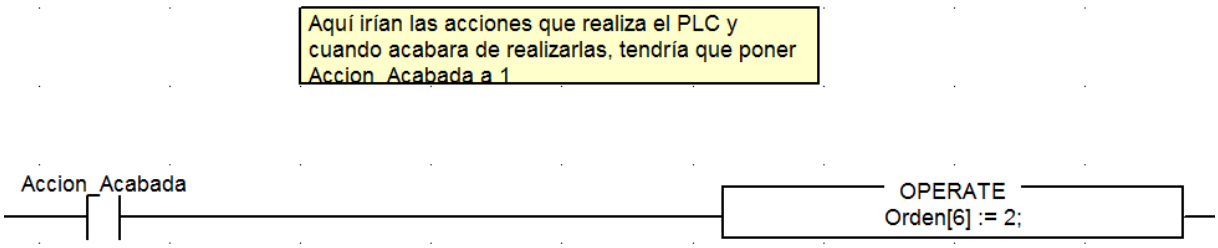

Figura 7.1 Código en Ladder para identificar la realización de la orden.

Name	Type	Address	Value	Comment
Error_Arduino	INT			
Error_Instruccion	EBOOL		0	
Flanco	ARRAY[1..9] OF EBO...			
Nueva_Instruccion	BOOL		0	
Nueva_Orden	EBOOL			
Numero_Palet	INT			
Numero_Tarea	INT			
Orden	ARRAY[0..7] OF INT	%MW8		
Recibido	ARRAY[0..7] OF INT	%MW0		
Estado_Tarea	EBOOL		0	#0 - No realizado, #1 - Realizado
Puesto	INT			#1 Almacén, #2 Pórtico, #3 Scorbot, #4 Sony, #5 Alimentador de bandejas
Comprueba_Pieza	EBOOL		0	*** De las ordenes con 3 asteriscos solo puede ir una por tarea
Deja_en_Alimenta...	EBOOL		0	*** De las ordenes con 3 asteriscos solo puede ir una por tarea
Deja_en_Palet	EBOOL		0	*** De las ordenes con 3 asteriscos solo puede ir una por tarea
Recoge_Bandeja	EBOOL		0	*** De las ordenes con 3 asteriscos solo puede ir una por tarea
Recoge_de_Alime...	EBOOL		0	*** De las ordenes con 3 asteriscos solo puede ir una por tarea
Recoge_de_Palet	EBOOL		0	*** De las ordenes con 3 asteriscos solo puede ir una por tarea
Recoge_Palet	EBOOL		0	*** De las ordenes con 3 asteriscos solo puede ir una por tarea
Saca_Bandeja	EBOOL		0	*** De las ordenes con 3 asteriscos solo puede ir una por tarea
Saca_Palet	EBOOL		0	*** De las ordenes con 3 asteriscos solo puede ir una por tarea
Confirmar	EBOOL		0	Confirmación de envío de la tarea al palet
i	INT			Contador
Accion_Acabada	EBOOL			Cuando acabamos de realizar la acción solicitada ponemos a 1 esta variable
Detalle	INT			Detalle de la posición de la que recoger o dejar la pieza, donde almacenar el palet, color de la pieza, etc. Solo valores numéricos
Error	UINT		0	Error por falta de pieza en posición requerida, orden no definida, imposible realizar la acción por estado actual del equipo, etc
PPC	EBOOL		1	Paso por cero para la botonera

Figura 7.2 Listado de variables necesarias en Unity.

7.2.4 Gestión de retenedores por el PLC

Para el correcto funcionamiento de este sistema es necesario que los retenedores funcionen correctamente. Del funcionamiento de estos, se encarga el PLC principal. En la figura7.3 se ven las variables necesarias para controlarlos.

Por un lado es necesaria una salida física que actúe sobre el propio retenedor. En este caso la variable *Retenedor_Sony* que corresponde con la salida 12 del segundo módulo. Y por otro lado es necesaria una señal que activa dicha salida. Esta señal es *Signal_Retenedor_Sony*, la cual lleva asociada una posición de memoria de tipo *coil*, para este caso concreto, la posición 0.

Con estos datos, Arduino le manda una señal a través de MODBUS al PLC principal, que se almacena en *Signal_Retenedor_Sony* y según esta sea un uno o un cero, se sube o se baja el retenedor.

Name	Type	Address
Retenedor_Sony	EBOOL	%Q0.2.12
Signal_Retenedor_Sony	EBOOL	%M0

Figura 7.3 Variables del PLC principal para la gestión de un retenedor.

8 Anexo

Para el correcto funcionamiento del código de Arduino ha sido necesario el uso de una librería externa, es decir, no una de las oficiales. La librería *MgsModbus* ha sido la encargada de la comunicación por MODBUS TCP/IP.

Esta librería se ha podido comprobar que era muy completa, permitiendo el envío de información en distintos modos, establecer el cliente y el servidor, el puerto 502 estaba configurado para la comunicación, etc. El único problema que se ha encontrado ha sido a la hora de, desde el Arduino, intentar acceder a más de un servidor. Esta librería tiene la dirección IP del servidor escrita y sin posibilidad de modificarla al llamar a la función desde el propio código de Arduino. Por ello, se han tenido que modificar varias líneas de las librerías, para que al llamar a la función, se le pueda enviar la IP del servidor.

Las líneas modificadas pertenecen a la función *MgsModbus::Req*. La primera línea modificada es la cabecera de la función, a la que se le ha añadido un parámetro más de entrada, al que se le ha llamado *IP_Rec*. Este parámetro corresponde con el último octeto de la dirección IP. La segunda línea modificada es la que asigna la IP a una variable. En esta línea los tres primeros octetos se han dejado fijos, pero el último es el parámetro recibido por la función. De esta manera se establece la dirección del servidor según se quiera conectar con el PLC del puesto o el PLC principal.

Código 8.1 Envío de instrucciones al PLC.

```
void MgsModbus::Req(MB_FC FC, word Ref, word Count, word Pos,word
    IP_Rec)
{
  MbmFC = FC;
  byte ServerIp[] = {192,168,0,IP_Rec};       // Anterior
     192.168.0.12
  MbmByteArray[0] = 0; // ID high byte
  MbmByteArray[1] = 1; // ID low byte
  MbmByteArray[2] = 0; // protocol high byte
  MbmByteArray[3] = 0; // protocol low byte
  MbmByteArray[5] = 6; // Lenght low byte;
  MbmByteArray[4] = 0; // Lenght high byte
  MbmByteArray[6] = 1; // unit ID
  MbmByteArray[7] = FC; // function code
  MbmByteArray[8] = highByte(Ref);
  MbmByteArray[9] = lowByte(Ref);

  /* Continúa el código */

  .
```

```
  .
  .

  /* Fin el código */
}
```

Índice de Figuras

Índice de Códigos

Bibliografía

[1] Arduino. *Arduino Mega 2560 R3*. URL: *https://www.arduino.cc/en/Main/ArduinoBoardMega2560*.

[2] Arduino. *Arduino Ethernet Shield V1*, URL: *https://www.arduino.cc/en/Main/ArduinoEthernetShieldV1*.

[3] Wiznet. *Wiznet W5100 Rev. 1.2.7*. URL: *http://www.wiznet.io/wpcontent/%20uploads/wiznethome/Chip/W5100/Document/W5100_Datasheet_v1.2.7.pdf*.

[4] NXP Semiconductors. *MFRC522 Rev. 3.9*. URL: *https://www.nxp.com/docs/en/data-sheet/MFRC522.pdf*.

[5] NXP Semiconductors. *MIFARE Classic EV1 1K Rev. 3.0*. URL: *http://www.nxp.com/docs/en/data-sheet/MF1S50YYX_V1.pdf*.

[6] Frederic Leens. «An introduction to I2C and SPI protocols». En: *IEEE Instrumentation & Measurement Magazine* 12.1 (ene. de 2009), págs. 8-13. DOI: *10.1109/MIM.2009.4762946*. URL: *https://ieeexplore.ieee.org/document/4762946/*.

[7] National Instruments. *The Modbus Protocol In-Depth*. URL: *http://www.ni.com/white-paper/52134/en/*.

[8] Siemens. *History of the Modbus protocol*. URL: *https://w3.usa.siemens.com/us/internet-dms/btlv/CircuitProtection/MoldedCaseBreakers/docs_MoldedCaseBreakers/Modbus%20Information.doc*.

[9] Aamir Shahzad y col. «Real Time MODBUS Transmissions and Cryptography Security Designs and Enhancements of Protocol Sensitive Information». En: *Symmetry* 7 (jul. de 2015), págs. 1176-1210. DOI: *10.3390/sym7031176*. URL: *http://www.mdpi.com/2073-8994/7/3/1176/htm*.

[10] Miguel Balboa. *Readme de la librería MFRC522 Arduino*. URL: *https://github.com/miguelbalboa/rfid/blob/master/README.rst*.

[11] Arduino. *Librería SPI Arduino*. URL: *https://www.arduino.cc/en/Reference/SPI*.

[12] Arduino. *Librería Ethernet Arduino*. URL: *https://www.arduino.cc/en/Reference/Ethernet*.

[13] Miguel Balboa. *Librería MFRC522 Arduino*. URL: *https://github.com/miguelbalboa/rfid*.

[14] Autor desconocido. *Librería MODBUS Arduino*. URL: *http://myarduinoprojects.com/modbus.html*.

Siglas

ASCII Código estándar estadounidense para el intercambio de información.

EEPROM Memoria de sólo lectura programable y borrable eléctricamente.

I2C Circuito inter-integrado.

ICSP Programación serial en circuito.

IP Protocolo de internet.

IRQ Solicitud de interrupción.

MBAP MODBUS Application Protocol header.

MISO Salida de esclavo, entrada de master.

MOSI Salida de master, entrada esclavo.

PDU Unidad de datos de protocolo.

RFID Identificación por radiofrequencia.

RTU Unidad terminal remota.

SCK Línea de reloj.

SDA Línea de datos.

SPI Bus de interfaz de periféricos serie.

TCP Protocolo de control de transmisión.

UDP Protocolo de datagramas de usuario.