Universidad
de La Laguna

Escuela Superior de Ingeniería y Tecnología

Trabajo Fin de Grado

“Construcción de un PLC mediante un dispositivo de bajo coste”

Titulación: Grado en Ingeniería Electrónica Industrial y Automática

Alumno: Sergio Heras Aguilar

Tutor: Pedro Antonio Toledo Delgado

Julio 2015

D. **Pedro Antonio Toledo Delgado**, con N.I.F. 45.725.874- B profesor Ayudante adscrito al Departamento de Ingeniería Informática de la Universidad de La Laguna

C E R T I F I C A

Que la presente memoria titulada:

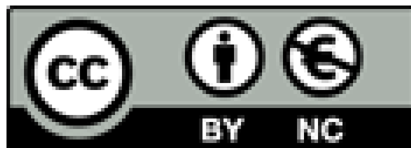
"Construcción de un PLC mediante un dispositivo de bajo coste".

Ha sido realizada bajo su dirección por D. Sergio Heras Aguilar, con N.I.F. 43.382.301-T.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 10 de julio de 2015.

Licencia Creative Commons

Reconocimiento – NoComercial (by-nc): Se permite la generación de obras derivadas siempre que no se haga un uso comercial. Tampoco se puede utilizar la obra original con finalidades comerciales.



Agradecimientos

A toda mi familia, por apoyarme durante estos cuatro años de carrera.

A los profesores que me han ayudado durante la misma.

Y a mi tutor, por ayudarme a sacar este proyecto adelante.

Resumen

Con el desarrollo de este proyecto se consigue replicar el funcionamiento de un Controlador Lógico Programable (PLC por sus siglas en inglés) mediante la programación de un dispositivo de bajo coste.

La implementación de este proyecto se lleva a cabo estudiando el funcionamiento de un PLC y de su programación interna, teniendo así una base sobre la que trabajar.

A través de un programa ensamblador, se procesan las instrucciones en el lenguaje de programación propio del autómata a un nuevo entorno de programación, que contiene la funcionalidad de simular el comportamiento de un PLC. De esta forma, el dispositivo de bajo coste que se elige, una placa Arduino, se comporta como un autómata.

Se usan dos lenguajes de programación en el proyecto. El propio del entorno del dispositivo de bajo coste, con el que se realizará la simulación del autómata y un segundo para hacer el compilador de instrucciones.

Dado que los PLC están concebidos para trabajar en ambientes industriales, tienen un elevado coste. Con el uso extendido de la automática hoy día, la posibilidad de implementar procesos de este tipo para aplicaciones que no requieran de la durabilidad y resistencia de los PLC, el uso de dispositivos de bajo coste, es una alternativa viable y económica.

Además de la diferencia notable de precio, el uso de este tipo de dispositivos en entornos académicos puede beneficiar a los alumnos dada la accesibilidad y sencillez que proporciona un dispositivo de estas características.

Palabras clave

PLC, dispositivo de bajo coste, programa ensamblador, Arduino, lenguajes de programación.

Abstract

The goal of this Project is to imitate the behavior of a Programmable Logic Controller (PLC) using a low cost device. This has been achieved after the analysis of the PLC general structure, operating mode and internal programming.

With an assembler program, the low level PLC instructions are processed and translated to the native programming language of the low cost device. This device has the objective of supporting the reproduction of the behavior of a PLC. Consequently, the low cost device chosen, an Arduino One board, works like a PLC.

Two programming languages has been used: the native language of the low cost device (where the simulation of the PLC will be processed) and a second one to create the instruction's compiler.

PLCs have been made to work in industrial environments, and consequently they have a high price. With the extended use of automation nowadays and, the possibility of implementing automatic processes for applications which do not require the durability and toughness of a PLC, the use of low cost devices is an economic and viable alternative.

Besides the price difference, the use of these devices on academic environments can benefit students due the accessibility and simplicity that these devices provide.

Keywords

PLC, low cost device, assembler program, Arduino, programming languages.

Índice:

1 **Introducción** ... 12
1.1 Objetivos ... 13
1.1.1 Objetivo general ... 13
1.1.2 Objetivos específicos ... 13
1.2 PLC ... 14
1.2.1 Breve Historia de los PLCs ... 14
1.2.2 Controlador lógico programable ... 14
1.2.3 Estructura interna ... 15
1.2.4 Estructura externa ... 16
1.2.5 Modo de funcionamiento de un PLC ... 16
1.2.6 Lenguajes de programación ... 18

2 **Análisis de dispositivos a utilizar** ... 22
2.1 Autómatas programables ... 22
2.1.1 Elección del PLC ... 22
2.1.2 Lenguaje de programación S7-200 ... 23
2.2 Dispositivos de bajo coste ... 23
2.2.1 Elección del dispositivo ... 25
2.3 Introducción al entorno de programación Arduino ... 26
2.3.1 Funciones ... 26
2.3.2 Estructura del programa ... 26
2.3.3 Librería Arduino ... 27

3 **Propuesta general de la solución realizada** ... 30
3.1 Estructura general ... 30
3.2 Solución Hardware: Arduino más módulo de relés ... 31
3.3 Software: Repertorio de instrucciones AWL soportado ... 31

4 **Librería PLC** ... 35
4.1 Pila lógica ... 36
4.2 Declaración de variables en la librería PLC ... 36

4.3 Unidad Aritmético/Lógica 38
4.3.1 Operaciones de lógica con bits 38
4.3.2 Operaciones lógicas de pila 42
4.3.3 Operaciones de comparación 43
4.3.4 Operaciones aritméticas 44
4.4 Contadores 45
4.4.1 Contador ascendente 45
4.4.2 Contador descendente 46
4.4.3 Contador ascendente y descendente 46
4.5 Temporizadores 47
4.5.1 Temporizador de retardo a la conexión 47
4.5.2 Temporizador de retardo a la desconexión 47
4.5.3 Temporizador de retardo a la conexión con memoria 48
4.6 Marcas especiales 48
4.7 Operaciones adicionales 50
4.7.1 Operación de guardado de variable 50
4.7.2 Instrucciones de control de flujo 50

5 **Ensamblador** 52
5.1 Introducción 52
5.2 Programación en Python 53
5.2.1 Diccionario de declaración 54
5.2.2 Diccionario de instrucciones 55

6 **Prueba de funcionamiento** 60
6.1 Estación 1: Medida de piezas 60
6.2 Circuito de conexión 64
6.3 Programación 67

7 **Conclusiones** 72
7.1 Conclusiones sobre el proyecto 72
7.2 Líneas futuras 73

8 **Conclusions** 74
8.1 Conclusions 74
8.2 Future lines 75

9 **Bibliografía** 77

10 **Anexos** 65
10.1 Librería Arduino 67
10.1.1 Archivo .h 67
10.1.2 Archivo.Cpp 69
10.2 Programa ensamblador en Python 85
10.3 Programa de prueba de funcionamiento 97
10.3.1 ArduinoIn.txt 97
10.3.2 ArduinoOut.txt 99

Índice de figuras:

Figura 1: Estructura interna de un PLC 15
Figura 2: Ciclo de un PLC 17
Figura 3: Ejemplo de programación en lenguaje AWL 18
Figura 4: Ejemplo de programación en lenguaje de texto estructurado 19
Figura 5: Ejemplo de programación en lenguaje KOP 19
Figura 6: Ejemplo de programación en lenguaje FUP 20
Figura 7: Placa Pingüino 45k50 24
Figura 8: Placa Raspberry 24
Figura 9: Placa Arduino Uno 25
Figura 10: Flujo de programa de la solución realizada 30
Figura 11: Módulo de 8 relés 31
Figura 12: Detalle de la etapa de librería PLC 35
Figura 13: Bloques de programación 35
Figura 14: Bits de la pila lógica 36
Figura 15: Detalle de la etapa de programación del ensamblador 52
Figura 16: Estación 1 de la planta Festo 60
Figura 17: Caja de conexiones y módulo de relés 64
Figura 18: Circuito de entrada a la placa Arduino 65
Figura 19: Circuito de salida de la placa Arduino 65
Figura 20: Circuito para la simulación de comunicación 66
Figura 21: Conexiones placa Arduino y botón de simulación 66

Índice de tablas:

Tabla 1: Comparativa de PLCs del mercado 22
Tabla 2: Comparativa de dispositivos de bajo coste del mercado 24
Tabla 3: Instrucciones soportadas por la librería Arduino 32
Tabla 4: Marcas especiales soportadas por la librería Arduino 33
Tabla 5: Operaciones de cargar y asignar bits 38
Tabla 6: Operaciones And y Or 40
Tabla 7: Operaciones con flancos 41
Tabla 8: Operaciones de Set y Reset 41
Tabla 9: Operaciones lógicas con la pila 43
Tabla 10: Operaciones de comparación 44
Tabla 11: Operaciones aritméticas 45
Tabla 12: Resolución de los temporizadores 48
Tabla 13: Lista de marcas especiales soportadas 49
Tabla 14: Lista de sensores de la estación 1 62
Tabla 15: Lista de actuadores de la estación 1 63

Introducción

1 Introducción

Actualmente los sistemas de automatización juegan un importante papel en los procesos industriales. La automatización de estos procesos conlleva una reducción de costes y de personal y un incremento de productividad y eficacia que se ven reflejados en la producción final.

El uso de autómatas programables para estas tareas en entornos industriales está justificado debido a la fiabilidad que tienen estos dispositivos en condiciones de trabajo extremas, pero estos dispositivos tienen un elevado coste.

Esta clase de dispositivos está muy extendida en cualquier tipo de industria. Por ello, el poder de replicar el comportamiento de estos dispositivos resulta útil para procesos menos críticos o soluciones sencillas. La automatización de elementos en el hogar (domótica) o el uso para docencia hace que no se necesite un dispositivo tan robusto como un PLC.

Para este tipo de aplicaciones, no sería necesario el uso de un autómata industrial. Una solución de bajo coste, que una persona con conocimiento sobre como programar un PLC pueda operar, sería suficiente para resolver un problema en un entorno académico o domótico.

En el desarrollo de este proyecto se quiere conseguir replicar el funcionamiento de un Controlador Lógico Programable (PLC por sus siglas en inglés) mediante la programación de un dispositivo de bajo coste. Y, a través de un programa ensamblador, conseguir traducir las instrucciones del lenguaje de programación propio del autómata al nuevo entorno de programación.

La implementación de este proyecto se llevará a cabo estudiando el funcionamiento de un PLC, su estructura y su programación interna, para simular su comportamiento en el dispositivo elegido.

Para ello, se van a usar dos lenguajes de programación. El propio del entorno del dispositivo de bajo coste, con el que se realizará la simulación del autómata y un segundo para hacer el compilador de instrucciones.

1.1 Objetivos

1.1.1 Objetivo general

El principal objetivo del proyecto es lograr simular el comportamiento de un Controlador Lógico Programable (PLC por sus siglas en inglés) mediante un soporte electrónico de bajo coste. En la programación y simulación de estos autómatas se utilizarán programas de software abierto.

1.1.2 Objetivos específicos

1. Se debe encontrar un dispositivo que se adecue a las necesidades de programación y que su precio de mercado sea el menor posible, para obtener una solución eficiente y de bajo coste.

2. Se estudiará la estructura interna de un PLC y su modo de funcionamiento, para posteriormente elegir el controlador que se replicará.

3. Se realizará un programa que implemente mediante su programación en un dispositivo las instrucciones más representativas y comunes de un autómata, como operaciones con bits, contadores, temporizadores, marcas especiales, etc.

4. Se realizará un programa que traduzca el lenguaje del autómata a un lenguaje legible por el dispositivo de bajo coste, un programa por el cual un usuario externo solo tenga que tener conocimientos de programación de autómatas para realizar una tarea de automatización con él.

5. Una vez realizadas las funciones del autómata se comprobará el funcionamiento mediante la conexión a un hardware sencillo. Se requerirá que tanto el software como el hardware funcionen correctamente simulando el funcionamiento de un PLC.

1.2 PLC

1.2.1 Breve Historia de los PLCs

El Controlador Lógico Programable (PLC) nació como solución al control de circuitos complejos de automatización. Por lo tanto se puede decir que un PLC no es más que un aparato electrónico que sustituye los circuitos auxiliares o de mando de los sistemas automáticos. A él se conectan los captadores (finales de carrera, pulsadores, etc.) por una parte, y los actuadores (bobinas de contactores, lámparas, pequeños receptores, etc.) por otra.

Los PLC's tienen su origen en las industrias de fabricación de coches. Los procesos llevados a cabo estaban parcialmente automatizados gracias al uso de circuitos de control rígidos, tanto eléctricos, como hidráulicos y neumáticos. En los que cualquier cambio por pequeño que fuera suponía la reconfiguración del sistema.

Con el desarrollo de la tecnología de computación, los procesos pasaron a ser controlados por un ordenador, el cual podía cambiar el estado de señales de salida en respuesta a señales de entrada, de forma que los cambios se podían hacer con la reprogramación del ordenador. Así nacieron los PLC.

1.2.2 Controlador lógico programable

Como se adelantaba bajo el epígrafe anterior, un PLC es un dispositivo electrónico que posee las herramientas necesarias, tanto de hardware como de software, para comprobar el estado de sensores y controlar dispositivos externos como actuadores a través de un programa elaborado por el usuario.

Dadas las características de diseño que tiene un PLC su campo de aplicación es muy extenso. La constante evolución del hardware y software amplía constantemente este campo para poder satisfacer las necesidades que se detectan en el espectro de sus posibilidades reales.

Su utilización se da fundamentalmente en aquellas instalaciones en donde es necesario realizar procesos de fabricación industriales de cualquier tipo, transformaciones industriales, control de instalaciones, etc.

Sus reducidas dimensiones, su facilidad de su montaje, la posibilidad de almacenar programas para su posterior utilización, la modificación o alteración de los mismos, etc., hace que su eficacia se aprecie fundamentalmente en procesos en que se producen necesidades tales como:

- Espacio reducido.
- Procesos de producción periódicamente cambiantes.
- Procesos secuénciales.
- Maquinaria de procesos variables.
- Instalaciones de procesos complejos y amplios.
- Comprobación de programación centralizada de las partes del proceso.
- Maniobra de máquinas e instalaciones.
- Señalización y control.

1.2.3 Estructura interna

La figura 1 muestra la estructura interna de un PLC, donde aparecen los distintos bloques que requiere un autómata. La Unidad Central de Procesos (CPU) es el cerebro del sistema. En ella se ejecuta el programa de control del proceso, el cual se carga por medio de la consola de programación.

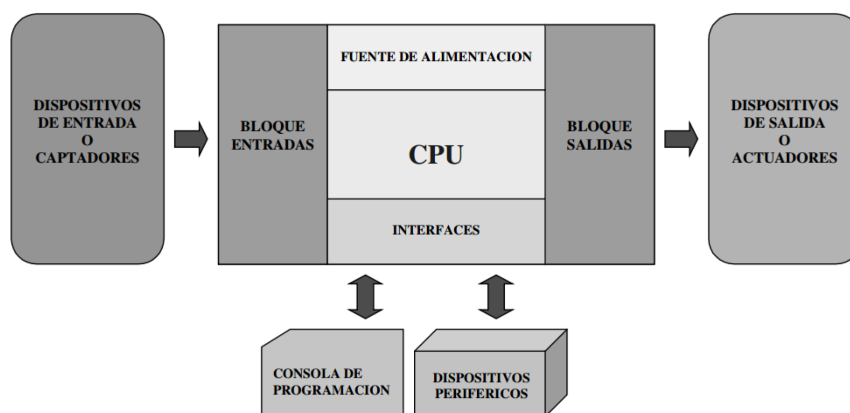


Figura 1: Estructura interna de un PLC

Los autómatas poseen dos sistemas de conexión diferentes a la entrada y salida, que se basan en el aislamiento de la señal del resto del circuito para no dañar los circuitos internos del PLC:

El módulo de entrada conecta las entradas físicas del PLC al resto del sistema. Cada terminal está eléctricamente aislada de los componentes electrónicos internos a través del uso de opto acopladores. De esta forma, podemos pasar el estatus de la entrada exterior (on/off) con un diodo emisor de luz y un fototransistor. Es una medida preventiva para efectos magnéticos y altos voltajes.

El módulo de salidas contiene interruptores (transistores o relés) activados por la CPU para conectar dos terminales y así permitir el paso de corriente a los circuitos externos.

1.2.4 Estructura externa

Los PLCs además se pueden dividir en grupos según las características de su estructura externa, las tres configuraciones más habituales son:

1.2.4.1 Estructura compacta

Este tipo de Controlador Lógico Programable se distingue por presentar en un solo bloque todos sus elementos, esto es, fuente de alimentación, CPU, memorias, entradas/salidas, etc.

Son los PLC de gama baja o micro-PLCs los que suelen tener una estructura compacta. Su potencia de proceso suele ser muy limitada dedicándose a controlar máquinas muy pequeñas o cuadros de mando.

1.2.4.2 Estructura semimodular

Se caracterizan por separar las E/S del resto del Controlador Lógico Programable, de tal forma que en un bloque compacto están reunidas las CPU, memoria de usuario o de programa y fuente de alimentación y separadamente las unidades de E/S .

1.2.4.3 Estructura modular

Su característica principal es la de que existe un módulo para cada uno de los diferentes elementos que componen el PLC como puede ser una fuente de alimentación, CPU, E/S, etc. La sujeción de los mismos se hace por riel DIN, placa perforada o sobre RACK, en donde van alojado el BUS externo de unión de los distintos módulos que lo componen.

Son los PLC de gama alta los que suelen tener una estructura modular, que permiten una gran flexibilidad en su constitución.

1.2.5 Modo de funcionamiento de un PLC

El proceso de funcionamiento de un autómata es el comportamiento que se quiere replicar. La importancia de este proceso es importante para comprender como funciona un PLC y así poder simular su modo de funcionamiento.

La CPU se ha previsto para que se ejecute cíclicamente una serie de tareas, dicha ejecución se denomina ciclo del autómata (figura 2), durante el cual, la CPU ejecuta la mayoría de las tareas siguientes:

- Lee las entradas
- Ejecuta el programa de usuario
- Procesa las peticiones de comunicación
- Ejecuta un autodiagnóstico
- Escribe las salidas

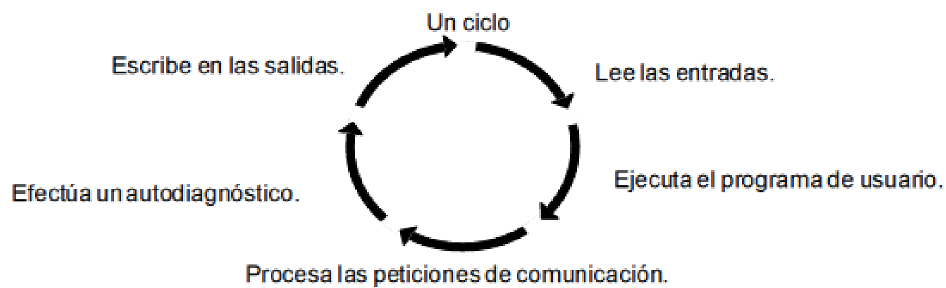


Figura 2: Ciclo de un PLC

Al principio de cada ciclo se leen los valores de las entradas digitales y se escriben en la imagen del proceso de las entradas. La CPU no actualiza las entradas analógicas como parte del ciclo normal, a menos que se haya habilitado.

Una vez guardados los valores de las entradas, se ejecuta el programa desde la primera operación hasta la última resolviendo los estados de salida y se guardan estos valores en la imagen del proceso.

A continuación se efectúa el procesamiento de peticiones de comunicación que la CPU haya recibido por el puerto de comunicación y se efectúa un autodiagnóstico del firmware de la CPU y la memoria del programa.

Una vez realizados estos procesos, se escriben los valores de la imagen del proceso de las salidas en las salidas digitales.

1.2.6 Lenguajes de programación

A la hora de programar un PLC se pueden usar diferentes tipos de lenguajes de programación. Todos ellos tienen elementos en común, pero también tienes sus características y ventajas individuales. Entre los lenguajes de programación más habituales se encuentran:

- Lista de instrucciones (AWL)
- Lenguaje de alto nivel (texto estructurado)
- Diagrama de contactos (KOP)
- Diagrama de Bloques funcionales (FUP)

1.2.6.1 Lenguaje AWL

El lenguaje AWL (Lista de instrucciones) permite crear programas de control introduciendo los nemónicos de las operaciones, como se puede observar en la figura 3. Por lo general, este lenguaje se adecúa especialmente para los programadores expertos ya familiarizados con los sistemas de automatización (PLCs) y con la programación lógica. Este lenguaje usa una pila lógica para trabajar.

```
NETWORK
LD     I0.0
LD     I0.1
LD     I2.0
A      I2.1
OLD
ALD
=      Q5.0
```

Figura 3: Ejemplo de programación en lenguaje AWL

El editor AWL también permite crear ciertos programas que, de otra forma, no se podrían programar con los lenguajes KOP ni FUP. Ello se debe a que AWL es el lenguaje nativo de la CPU, a diferencia de los editores gráficos en los que son aplicables ciertas restricciones para poder dibujar los diagramas correctamente y provienen de una traducción del lenguaje nativo.

1.2.6.2 Texto estructurado

En el lenguaje de texto estructurado las instrucciones son líneas de texto que utilizan palabras o símbolos reservados. Las operaciones se definen por los símbolos matemáticos habituales. También se dispone de funciones trigonométricas, logarítmicas y de manipulación de variables complejas. Sin embargo, lo que distingue realmente estos lenguajes avanzados de las listas de instrucciones son las tres características siguientes:

- Son lenguajes estructurados, donde es posible la programación por bloques con definición de variables locales o globales.
- Incluyen estructuras de cálculo repetitivo y condicional.
- Disponen de instrucciones de manipulación de cadenas de caracteres, muy útiles en aplicaciones de gestión, estadística, etc.

```
IF value  < 7 THEN
        WHILE value  < DO
                Value  := value  + 1;
        END_WHILE;
END_IF;
```

Figura 4: Ejemplo de programación en lenguaje de texto estructurado

1.2.6.3 Lenguaje KOP

El lenguaje KOP (diagrama de contactos) permite crear programas con componentes similares a los elementos de un esquema de circuitos. KOP es probablemente el lenguaje predilecto de numerosos programadores y encargados del mantenimiento de sistemas de automatización.

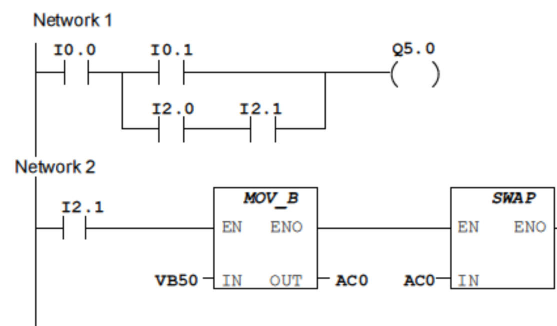


Figura 5: Ejemplo de programación en lenguaje KOP

Los programas KOP hacen que la CPU emule la circulación de corriente eléctrica desde una fuente de alimentación, a través de una serie de condiciones lógicas de entrada que, a su vez, habilitan condiciones lógicas de salida. El programa se ejecuta segmento por segmento, de izquierda a derecha y luego de arriba a abajo.

1.2.6.4 Lenguaje FUP

El lenguaje FUP (Diagrama de funciones) permite visualizar las operaciones en forma de cuadros lógicos similares a los circuitos de puertas lógicas. En FUP no existen contactos ni bobinas como en el editor KOP, pero sí hay operaciones equivalentes que se representan en forma de cuadros.

La lógica del programa se deriva de las conexiones entre dichas operaciones de cuadro. Ello significa que la salida de una operación se puede utilizar para habilitar otra operación para crear la lógica de control necesaria. Dichas conexiones permiten solucionar numerosos problemas lógicos.

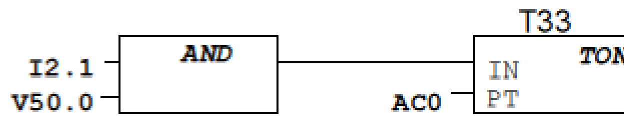


Figura 6: Ejemplo de programación en lenguaje FUP

Análisis de dispositivos a utilizar

2 Análisis de dispositivos a utilizar

En esta sección se elegirán los diferentes dispositivos para replicar el funcionamiento de un PLC. Por una lado, la elección del autómata para el desarrollo del proyecto, y, por otro lado, la elección del dispositivo de bajo coste que se usará para simular el autómata.

2.1 Autómatas programables

Se han elegido tres marcas que comercializan controladores para una comparativa entre sus dispositivos de gama más baja. Se eligen PLCs compactos debido a que lo que se busca es replicar las funcionalidades básicas.

Marca	Siemens	Omron	Schneider
Nombre	SIMATIC S7-200	PLC CP1L	Modicon M238
Entradas/salidas	14/10 ampliable	8/6 ampliable	10/10 ampliable
Velocidad procesamiento de bit	0.22us	0.55us	0.3us

Tabla 1: Comparativa de PLCs del mercado

2.1.1 Elección del PLC

Como se puede apreciar en la tabla 1 las principales características son muy similares en las tres marcas. Teniendo en cuenta su similitud y que en la Escuela Superior de Ingeniería y Tecnología se usan PLCs de la marca Siemens para la docencia, se va a proceder con el S7-200 como elección.

Siemens tiene en su catálogo los sistemas de automatización SIMATIC (conjunto de hardware y software coordinado de Siemens), entre los que destacan diferentes tipos de PLCs para realizar tareas dependiendo de su complejidad.

La familia de autómatas de Siemens se denomina S7 y se divide como se ha dicho anteriormente en grupos según el nivel de complejidad que se necesite al llevar a cabo un proyecto. Dada la naturaleza de este proyecto, en el cual se quiere conseguir la implementación de soluciones de automatización sencillas, la versión micro de estos controladores, el S7-200, es el ideal para automatizar este tipo de procesos.

2.1.2 Lenguaje de programación S7-200

Un autómata S7-200 se puede programar usando tres de los lenguajes mencionados anteriormente, estos lenguajes son:

- Lista de instrucciones (AWL)
- Diagrama de Bloques funcionales (FUP)
- Diagrama de contactos (KOP)

Debido a que los lenguajes de programación FUP y KOP son lenguajes gráficos, que aun que puedan ser más útiles e intuitivos a la hora de programar, no tienen unas características similares a un lenguaje de programación convencional.

El lenguaje de lista de instrucciones es el lenguaje nativo de la máquina, mientras que los lenguajes gráficos se crean a partir de él. Además, tiene más similitudes con los lenguajes tradicionales, en los que se programa a instrucción por instrucción.

Por ello, el lenguaje elegido para implementar la programación del S7-200 es el lenguaje AWL.

2.2 Dispositivos de bajo coste

Una vez se ha decido que el autómata a simular es un S7-200 de Siemens con una programación estructurada en lenguaje AWL. Para la elección de un dispositivo se tendrán que tener en cuenta una serie de características mínimas de hardware y software que el dispositivo debe cumplir:

- Pines de entrada/salida suficientes tanto analógicos como digitales para poder conectar las señales de actuadores y sensores.
- Que posea un microprocesador programable.
- Que posea un entorno de programación.
- Su precio de mercado debe ser reducido.

Teniendo estos requerimientos mínimos se ha hecho una comparativa de tres dispositivos de marcas diferentes que los cumplen. En la tabla 2 se especifican las características generales de cada uno:

	Pinguino 45k50	Raspberry B (pc funcional)	Arduino Uno
Procesador	PIC18F45K50	ARM1176JZF-S	ATmega328
RAM	2048B	512MB	32kB
Voltaje de trabajo	5v	5v	5v
Velocidad de la CPU	31kHz	400MHz	16MHz
E/S analógicas	5 compartidas con las digitales	-	6
E/S digitales / PWM	17/2	8	14/6
Precio	11,90 €	21,5€	20€ (10€ si no es marca Arduino)

Tabla 2: Comparativa de dispositivos de bajo coste del mercado

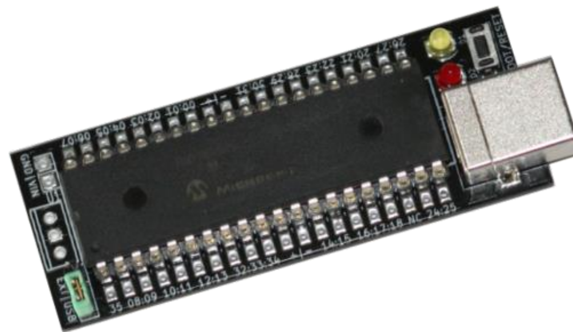


Figura 7: Placa Pingüino 45k50

En la figura 7 se puede ver la placa Pingüino 45k50. Es un dispositivo compacto, sencillo y barato, con un entorno de programación basado en C++. Tiene suficientes pines para soportar la implantación de un sistema de automatización. Tiene poca memoria y viene desmontado. Su uso no está muy extendido, ya que no es conocido.

Figura 8: Placa Raspberry

Las Raspberry, en la figura 8, se deben usar con sistemas operativos, tales como Linux para su uso y programación. Es más complejo de lo que se esperaría en un principio y no tiene la capacidad para soportar el número de entradas/salidas analógicas que se necesitan para la implementación de un PLC.

Figura 9: Placa Arduino Uno

La simplicidad de Arduino en cuanto a programación le convierte en la mejor opción. Su bajo coste así como sus características hace que éste sea una alternativa viable para desarrollar un proyecto de hardware.

2.2.1 Elección del dispositivo

Si se comparan los dispositivos anteriormente mencionados, se puede descartar la Raspberry ya que la mayoría de sus funcionalidades no son necesarias y tiene un coste superior a las otras dos alternativas.

La placa Pingüino y la Arduino tienen muchas similitudes siendo la Pingüino menos potente si no se usa un oscilador externo. Si se tiene en cuenta que la comunidad Arduino está muy extendida y el precio de las dos es muy similar, la elección final teniendo en cuenta una mayor eficiencia es un dispositivo Arduino.

Arduino es una plataforma electrónica de software abierto con un hardware y software fácil de usar. Dentro de la gama Arduino hay muchos dispositivos, se va a usar el Arduino Uno por ser el más simple y barato de todos los que ofrece la marca.

2.3 Introducción al entorno de programación Arduino

El software a utilizar para la implementación de las instrucciones que se quieren traducir de lenguaje AWL es el propio software de Arduino, el cual está basado en C++. El software Arduino es un software libre con un entorno de programación propio, en el cual una vez se escriba un programa, se compilará y se cargará dicho programa en la placa Arduino.

Su entorno de programación es simple y claro, el cual hará uso de los elementos propios de programación en C++, con algunas variaciones, para la creación de programas con el nivel de complejidad que se quiera realizar.

2.3.1 Funciones

Para programar estructuradamente en Arduino, como en cualquier otro lenguaje de programación, se hace uso de funciones. Una función es un bloque de código que tiene un nombre y un grupo de declaraciones que se ejecuta cuando se llama a la función, todas las funciones son de un tipo en función del valor que devuelvan a su finalización, pudiendo no devolver ninguno (tipo de función void).

Con el uso de funciones se pueden ejecutar tareas repetitivas y reducir el desorden de un programa. Para crear una función se usa la siguiente sintaxis:

```
tipo nombreDeFuncion(argumentos){
        instrucciones;
}
```

Una vez declarada la función y su comportamiento, se podrá llamar desde el programa principal.

2.3.2 Estructura del programa

Un programa Arduino consta de tres partes, una primera de declaraciones; una segunda, constituida por una función (setup), que sólo va a ejecutarse una vez; y una parte final, que será el programa en sí (loop).

En la primera parte, antes de cualquier función, se declararán las variables de forma global, para poder usarlas en el programa que se habrá de escribir, así como la inclusión de librerías, que se explicarán más adelante.

En la segunda parte, se ejecuta la función setup(), que sólo se ejecuta la primera vez que iniciamos la placa. En esta función se puede inicializar los pines de entrada y de salida. Además se pueden ejecutar todas las funciones e instrucciones que se quiera, pero recordando que esta función solo se ejecutará una vez.

```
void setup(){
        instrucciones;
}
```

La tercera parte del programa, es la función principal del programa, donde se escribirá todo el código. Esta función, loop(), es un bucle, de esta forma, una vez acabe de ejecutarse, en la última llave, volverá a ejecutarse desde el principio, creando un ciclo de programa infinito en el que se ejecutan continuamente las instrucciones dentro de la función loop().

```
void loop(){
        instrucciones;
}
```

Para obtener un código más claro y sencillo, en el entorno Arduino se puede hacer uso de las librerías de Arduino.

2.3.3 Librería Arduino

Una librería es una estructura de código que facilita el uso de operaciones complejas, de esta forma, se pueden declarar funciones en la librería que se podrán usar en la programación en el entorno Arduino sin necesidad de conocer el código interno de dicha función.

Las librerías normalmente incluyen los siguientes archivos:

- Un archivo .h (encabezado de C)
- Un archivo .cpp (código)
- Un archivo Keywords.txt

La librería de Arduino constará de 2 archivos mínimos necesarios para su correcto funcionamiento. Si se quiere hacer una librería de nombre “ejemplo” para su uso en la programación de Arduino, se deben crear dos archivos, “ejemplo.h” y “ejemplo.cpp”.

En el archivo .h se encuentran las cabeceras de todas las funciones que están definidas en el archivo .cpp, donde se encuentra todo el código de las funciones declaradas.

Ejemplo:

Si en "ejemplo.cpp" queremos tener una función que sume dos variables, una operación aritmética:

```
int funcion(int input, int &output) {
        output = input + output;
        return (output);
}
```

Se tendrá que declarar en "ejemplo.h" la declaración de la misma:

```
int funcion(int input, int &output);
```

Así, podremos hacer uso de ella en cualquier programa donde se incorpore la librería creada, para ello sería necesario incluirla en el programa.

Una vez que se tenga la librería con todas las funciones deseadas, para usarlo se ha de incluir en el programa de Arduino, para ello en la parte de declaraciones, escribimos lo siguiente de acuerdo al ejemplo:

```
#include <ejemplo.h>
```

Propuesta general de la solución realizada

3 Propuesta general de la solución realizada

3.1 Estructura general

Para abordar el problema de simular un autómata, el S7-200 con su lenguaje en AWL se debe seguir un procedimiento. Se quiere que una placa Arduino en última instancia sea capaz de procesar un código de entrada en lenguaje AWL y sea capaz de enviar y recibir señales del exterior.

En la figura 10 se observa la estructura del proceso a seguir para convertir un código fuente AWL en una imagen ejecutable para una placa Arduino Uno.

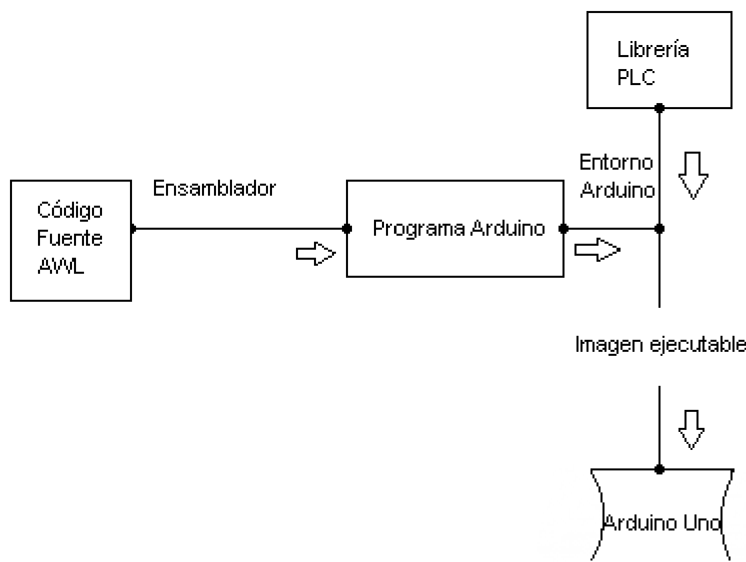


Figura 10: Flujo de programa de la solución realizada

El primer paso una vez se tiene el código fuente que se quiere ejecutar en la placa es ejecutar un ensamblador. El ensamblador es un compilador hecho en Python que escribe en un fichero de salida el programa equivalente en lenguaje Arduino a las instrucciones en AWL que se tenían en el archivo de lectura.

Una vez se tiene el programa en lenguaje Arduino, a través del uso de una librería, dentro del entorno de Arduino, que contiene las funciones equivalentes a las instrucciones en AWL, se crea la imagen ejecutable del programa y se carga en la placa Arduino Uno.

La librería creada en Arduino sirve para establecer la relación de funciones que simulan el comportamiento de sus instrucciones equivalentes en AWL.

3.2 Solución Hardware: Arduino más módulo de relés

Para el cometido de conectar la placa Arduino al exterior, normalmente con un voltaje distinto al de la placa, se debe usar algún elemento de protección. La placa Arduino Uno, con un voltaje de trabajo de unos 5v, debe ser adaptado para trabajar a voltajes mayores. Para proteger la placa se realizan dos circuitos, uno para las señales de entrada y otro para las de salidas.

Para las salidas, como protección para la placa Arduino se incorpora un módulo de relés, que se puede observar en la figura 11.

Figura 11: Módulo de 8 relés

.

3.3 Software: Repertorio de instrucciones AWL soportado

Dada la complejidad del PLC S7-200 respecto a la cantidad de instrucciones que se pueden usar en lenguaje AWL, se van a soportar las funcionalidades básicas del autómata. Así, el código fuente que se incorporé estará limitado por las instrucciones elegidas en AWL.

En cualquier programa de AWL la lógica de bits es esencial, por ello se han incorporado todas las instrucciones que trabajen con bits y con la pila lógica, pudiendo así, introducir y sacar bits de la pila y operar dentro de ella.

Para trabajar con señales analógicas, se han introducido operaciones de aritmética y comparación. De esta forma se puede operar como se desee con números enteros.

Además se han incorporado contadores y temporizadores así como instrucciones de control para un mayor control del programa. En la tabla 3 se muestran la lista de instrucciones escogidas:

Lenguaje AWL:	
Lógica de bits	
LD bit	ALD
LDN bit	OLD
A bit	LPS
AN bit	LRD
O bit	LPP
ON bit	LDS n
NOT	S bit,N
= bit	R bit,N
EU	
ED	
Comparación	**Aritmética**
LDW= IN1, IN2	+I IN1, OUT
LDW<> IN1, IN2	-I IN1, OUT
LDW< IN1, IN2	*I IN1, OUT
LDW> IN1, IN2	/I IN1, OUT
LDW<= IN1, IN2	
LDW>= IN1, IN2	
Temporizadores	**Contadores**
TON Txxx, PT	CTU Cxxx, Pv
TONR Txxx, PT	CTD Cxxx, Pv
TOFF Txxx. PT	CTD Cxxx, Pv
Mover	**Bucles**
MOVW IN,OUT	FOR INDX, INIT, FINAL
	NEXT

Tabla 3: Instrucciones soportadas por la librería Arduino

Se han introducido también una serie de marcas especial del S7-200 para un mayor control del programa, se pueden observar en la tabla 4.

Marca AWL	Explicación
SM0.0	Siempre On
SM0.1	Primer ciclo On
SM0.4	30 s Off / 30 s On
SM0.5	0.5 s Off / 0.5 s On
SM0.6	Ciclo Off /ciclo On
SMW22	Contiene el tiempo de ciclo

Tabla 4: Marcas especiales soportadas por la librería Arduino

Con estas instrucciones se cubren un amplio rango de posibilidades de automatización. No se han implementado instrucciones con otros tipos de datos como bytes o dobles palabras debido a que para soluciones sencillas no se necesita trabajar con tanta variedad de datos, de la misma forma se han omitido las instrucciones de conversión. El S7-200 posee un rango de instrucciones para operaciones complejas y de comunicación, las cuales no son relevantes en este proyecto.

Librería PLC

4 Librería PLC

Para que el programa Arduino reconozca las instrucciones equivalentes, se debe crear una librería que contenga tantas funciones como sean necesarias para replicar el funcionamiento de las instrucciones en AWL.

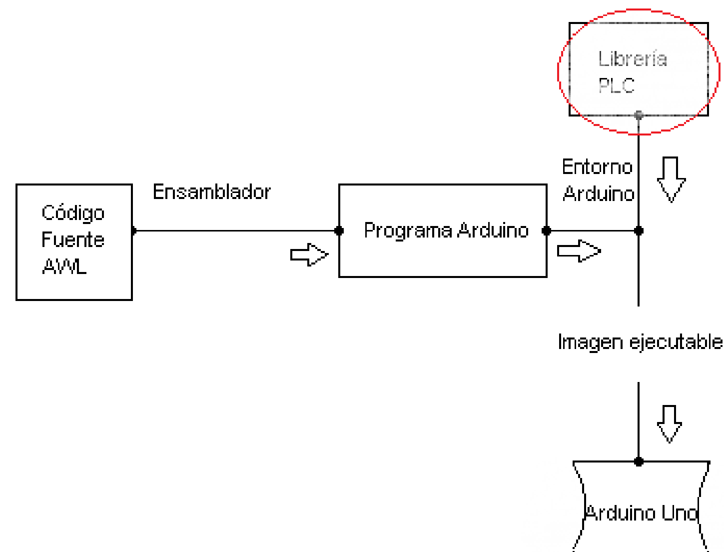


Figura 12: Detalle de la etapa de librería PLC

La programación puede atenderse siguiente la figura 12 en la que se muestran los distintosmódulos de operaciones que se tienen en lenguaje AWL.

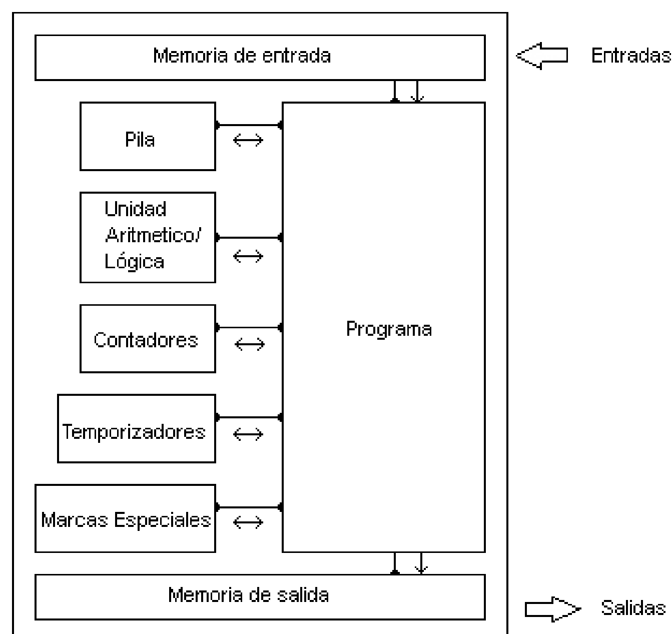


Figura 13: Bloques de programación

La creación de la librería PLC se basará en las instrucciones soportadas en AWL y en la máquina de pila para su funcionamiento.

4.1 Pila lógica

Las CPUs S7-200 utilizan una pila lógica para resolver la lógica de control. En AWL (lista de instrucciones) el usuario debe insertar operaciones para procesar ésta. Las instrucciones irán almacenando valores y operando con los diferentes niveles de la pila.

Bits de la pila lógica

S0	Pila 0	– Primer nivel (primer valor) de la pila
S1	Pila 1	– Segundo nivel de la pila
S2	Pila 2	– Tercer nivel de la pila
S3	Pila 3	– Cuarto nivel de la pila
S4	Pila 4	– Quinto nivel de la pila
S5	Pila 5	– Sexto nivel de la pila
S6	Pila 6	– Séptimo nivel de la pila
S7	Pila 7	– Octavo nivel de la pila
S8	Pila 8	– Noveno nivel de la pila

Figura 14: Bits de la pila lógica

En la figura 14 se ven los diferentes niveles de la pila, así cuando carguemos un nuevo valor se almacenará en el primer nivel dela pila, moviendo los demás valores de la pila un nivel, el último nivel de la pila se pierde.

Se tendrá que simular las funciones introducidas en lenguaje Arduino para replicar una máquina de pila.

4.2 Declaración de variables en la librería PLC

En la librería se van a declarar las variables necesarias para el uso de la pila y de las operaciones de bits y palabras. También se han de declarar los pines de entrada/salida de la placa Arduino.

De las variables que se van a usar en Arduino, tenemos 14 pines digitales en la placa, los dos primeros son de comunicación con la interfaz. Nos quedan 12 pines para las entradas y salidas, se han asignado 6 pines de entrada y 6 de salida, declarándolos en “PLC.h” con el número referenciado al pin, así I0.0 será X0, el cual esta referenciado al pin 2 de la placa.

```
const int X0 = 2;
…

const int Y0 = 8;
```

…

Dentro de los modos de direccionamientos, a parte de las entradas y salidas (Ix.x y Qx.x respectivamente) también hay variables de almacenamiento (Mx.x). Estas marcas tienen un rango de M0.0 a M31.7 dentro del PLC, habiendo en total 16 palabras MW de 16 bits cada una (2 bytes). Esto se traduce en Arduino declarando un vector en el que almacenaremos enteros. En el caso de que se quiera leer o escribir un bit, se accederá a ese bit especifico con una función.

```
int MW[16];
```

Para las entradas analógicas se declararán 6 variables Ainx, donde 'x' es el número de la entrada, que servirán para acceder al valor entre 0-1024 que proporciona el Arduino de una tensión de entrada de entre 0-5v.

```
int Ain0 = 0;
```

Además de las variables a usar durante el programa, se debe hablar de la pila del S7-200. Se tendrá que simular las funciones introducidas en lenguaje Arduino para replicar una máquina de pila, para ellos se hará los siguiente:

Dado que la pila en AWL tiene una longitud de 9 bits, la traducción al lenguaje Arduino será un vector con la misma longitud, en el que el nivel superior será el bit 0 del vector, a partir de ahora nivel superior de la pila.

En lenguaje Arduino se declara un vector de booleanos que será la pila lógica, siendo el primer nivel de la pila (top) la posición '0' de la pila.

```
bool pila [8];
int top = 0;
```

4.3 Unidad Aritmético/Lógica

A continuación se describirán la implementación de las instrucciones mencionadas en AWL con sus equivalentes en lenguaje Arduino, el código de las funciones se puede ver en el anexo 1 para un mayor detalle de lo que a continuación se describe.

4.3.1 Operaciones de lógica con bits

Las operaciones fundamentales con la pila son las de cargar un valor en el nivel superior de la pila y el de asignar a una variable el valor que se encuentra en dicho nivel superior.

AWL		Arduino	
LD bit	Guarda el valor de la variable 'bit' en el nivel superior de la pila	LD (int input) LD (bool input)	Guarda el valor booleano de una entrada digital o de una variable en el nivel superior de la pila
LDN bit	Guarda el valor de la variable 'bit' negada en el nivel superior de la pila	LDN (int input) LDN (bool input)	Guarda el valor booleano de una entrada digital negada o de una variable negada en el nivel superior de la pila
= bit	Asigna el valor del nivel superior de la pila a la variable 'bit'	Asignar(int output) Asignar(bool output)	Asigna el valor del nivel superior de la pila a un pin de salida o a una variable booleana

Tabla 5: Operaciones de cargar y asignar bits

En el lenguaje Arduino, el uso de una de las funciones depende de si se quiere acceder a un pin, la variable de entrada sería un entero, o si por el contrario se quiere acceder a un bit de memoria, la variable de entrada sería un dato booleano.

En el caso de que las variables sean marcas, para cargar un valor en la pila o asignarlo, debemos acceder a los bits individuales del vector MW[16], al cual pertenecen las marcas. Para dicho cometido se usan dos funciones:

extraerbit(int palabra, int N)

Esta función devuelve el valor booleano que se encuentre en el bit N de la variable palabra, que será una de las MW[x]. En este caso se requiere de la utilización de un LD(bool input) para que el valor del bit se cargue en la pila.

Para asignar un valor a un bit de una palabra, usamos la función que cumple este cometido, siendo similar a Asignar (bool output):

Asignarbit(int &palabra, int N)

Que asigna al bit N de la palabra MW[x] elegida, el valor del bit que esté en el nivel superior de la pila.

Las operaciones que realizan operaciones dentro de la pila lógica se muestran en la tabla 3.

AWL		Arduino	
A bit	Realiza la operación AND entre el valor del nivel superior de la pila y 'bit' y lo guarda en el nivel superior de la pila	A(int input) A(bool input)	Realiza la operación AND entre el nivel superior de la pila y una entrada digital o una variable booleana y lo guarda en el nivel superior de la pila
AN bit	Realiza la operación AND negada entre el valor del nivel superior de la pila y 'bit' y lo guarda en el nivel superior de la pila	AN(int input) AN(bool input)	Realiza la operación AND negada entre el nivel superior de la pila y una entrada digital o una variable booleana y lo guarda en el nivel superior de la pila
O bit	Realiza la operación OR entre el valor del nivel superior de la pila y 'bit' y lo guarda en el nivel superior de la pila	O(int input) O(bool input)	Realiza la operación OR entre el nivel superior de la pila y una entrada digital o una variable booleana y lo guarda en el nivel superior de la pila
ON bit	Realiza la operación OR negada entre el valor del nivel superior de la pila y 'bit' y lo guarda en el nivel superior de la pila	ON(int input) ON(bool input)	Realiza la operación OR negada entre el nivel superior de la pila y una entrada digital o una variable booleana y lo guarda en el nivel superior de la pila

Tabla 6: Operaciones And y Or

El uso de flancos con previa declaración de una variable que sirva para guardar el estado actual de la variable que queremos accionar se disponen de acuerdo a la tabla 5.

EU	Cuando se detecta un cambio de ‘0’ a ‘1’ en el nivel superior de la pila, éste se pone a ‘1’, de lo contrario se pone a ‘0’	EU (int &state)	Se guarda el estado del bit del nivel superior de la del ciclo anterior en ‘state’ y cuando se produce el flanco de subida, se pone el nivel superior de la pila a ‘1’
ED	Cuando se detecta un cambio de ‘1’ a ‘0’ en el nivel superior de la pila, éste se pone a ‘1’, de lo contrario se pone a ‘0’	ED (int &state)	Se guarda el estado del bit del nivel superior de la del ciclo anterior en ‘state’ y cuando se produce el flanco de bajada, se pone el nivel superior de la pila a ‘1’

Tabla 7: Operaciones con flancos

Otra instrucción fundamental en AWL es la capacidad de setear o resetear tanto salidas, como contadores y temporizadores. En el que se usa la misma instrucción para todos los casos, accediendo a cada variable según lo que se haya escrito en la instrucción. En lenguaje Arduino, se ha de llamar a una función distinta para interactuar con los diferentes tipos de datos.

<table>
<tr><td>S bit,N</td><td>Si el nivel superior de la pila está a ‘1’pone a ‘1’ la salida ‘bit’ y las ‘N’ siguientes</td><td rowspan="2">Dependiendo de los parámetros de entrada, se utilizarán diferentes funciones en Arduino para el mismo propósito en lenguaje AWL.</td></tr>
<tr><td>R bit,N</td><td>Si el nivel superior de la pila está a ‘1’pone a ‘0’ la salida ‘bit’ y las ‘N’ siguientes</td></tr>
</table>

Tabla 8: Operaciones de Set y Reset

Para el propósito de resetear o setear salidas se usan las siguientes funciones:

S (int output, int n)

R (int output, int n)

En ambos casos si se escribe una marca, en vez de las funciones estándar de la tabla 6, la función a utilizar sería una de las siguientes:

Sbit (int &palabra, int N, int n)

Rbit (int &palabra, int N, int n)

Donde la operación se realizaría en el bit N de la palabra elegida, reseteando o seteando tantos bits en función de la variable n.

Para resetear temporizadores y contadores, utilizamos las siguientes funciones:

RCTUD (bool &CT, int &cuenta)

En el caso de los contadores CTU y CTUD sólo se ha de poner a '0' la variable booleana que almacena el estado del Contador y su cuenta.

RCTD (bool &CT, int &cuenta, int pv)

Para los contadores CTD, se ha de poner a'0' el valor de la variable booleana y la cuenta volverá a su valor inicial que es PV.

RTON (bool &timer, unsigned long &tiempo)

La variable que almacena el valor del estado del temporizador se resetea, así como la variable que lleva la cuenta del tiempo transcurrido.

RTOFF (bool &timer, unsigned long &tiempo, unsigned long PT)

En los temporizadores TOFF el valor de la variable booleana vuelve a '0' y el tiempo vuelve a su valor inicial de PT.

RTONR (bool &timer, unsigned long &tiempo, int &estadoOff, unsigned long &tiempoOff, unsigned long &tiempoOn)

En los temporizadores TONR todas sus variables asociadas se ponen a '0'.

4.3.2 Operaciones lógicas de pila

En todas estas operaciones se procesan valores nuevos con los valores que ya se encuentran en la pila. También podemos procesar los valores que hay en el interior de la pila sin necesidad de acceder a valores exteriores.

<table>
<tr><td>NOT</td><td colspan="2">Niega el valor del nivel superior de la pila</td><td>NOT()</td><td>Niega el valor del nivel superior de la pila</td></tr>
<tr><td>ALD</td><td>ALD()</td><td colspan="3">Combina mediante AND el primer y segundo nivel de la pila, cargando el resultado en el nivel superior de la pila. Se pierde un nivel de la pila</td></tr>
<tr><td>OLD</td><td>OLD()</td><td colspan="3">Combina mediante OR el primer y segundo nivel de la pila, cargando el resultado en el nivel superior de la pila. Se pierde un nivel de la pila</td></tr>
<tr><td>LPS</td><td>LPS()</td><td colspan="3">Duplica el primer valor de la pila y lo desplaza dentro de la misma, el último valor se pierde</td></tr>
<tr><td>LRD</td><td>LRD()</td><td colspan="3">Copia el segundo valor de la pila y lo guarda en el nivel superior</td></tr>
<tr><td>LPP</td><td>LPP()</td><td colspan="3">Desplaza el primer valor de la pila fuera, resultando así en que el segundo valor se convierte en el primero</td></tr>
<tr><td>LDS n</td><td>LDS(int n)</td><td colspan="3">Copia el bit ‘n’ de la pila en el primer nivel de la pila y el último nivel de la pila se pierde al desplazar todos los bits</td></tr>
</table>

Tabla 9: Operaciones lógicas con la pila

4.3.3 Operaciones de comparación

Estas operaciones comparan dos valores asociados, que pueden ser números enteros o variables enteras, y escriben en el nivel superior de la pila el resultado de la comparación, siendo un ‘1’ si la comparación se cumple y un ‘0’ en caso contrario. La tabla 7 muestra las funciones.

AWL	Arduino	Explicación
LDW = IN1, IN2	LDWE(int input1, int input2)	Comparación de IN1 y IN2, el nivel superior de la pila se pone a '1' si la igualdad se cumple
LDW <> IN1, IN2	LDWNE(int input1, int input2)	Comparación de IN1 y IN2, el nivel superior de la pila se pone a '1' si la desigualdad se cumple
LDW < IN1, IN2	LDWLT(int input1, int input2)	Comparación de IN1 y IN2, el nivel superior de la pila se pone a '1' si IN1 < IN2
LDW > IN1, IN2	LDWGT(int input1, int input2)	Comparación de IN1 y IN2, el nivel superior de la pila se pone a '1' si IN1 < IN2
LDW ≤ IN1, IN2	LDWLTE(int input1, int input2)	Comparación de IN1 y IN2, el nivel superior de la pila se pone a '1' si IN1 ≤ IN2
LDW ≥ IN1, IN2	LDWGTE(int input1, int input2)	Comparación de IN1 y IN2, el nivel superior de la pila se pone a '1' si IN1 ≥ IN2

Tabla 10: Operaciones de comparación

4.3.4 Operaciones aritméticas

Las instrucciones aritméticas realizan la operación asignada si el nivel superior de la pila está a '1'. Estas instrucciones realizan operaciones con enteros, de manera que se deben utilizar números enteros o las palabras MW[x].

+ I IN1, OUT	sumaEN(int input, int &output)	Realiza la suma de input y output y guarda el valor en output - output = output + input
− I IN1, OUT	restaEN(int input, int &output)	Realiza la resta de input y output y guarda el valor en output - output = output − input
∗ I IN1, OUT	mulEN(int input, int &output)	Realiza la multiplicación de input y output y guarda el valor en output - output = output ∗ input
/ I IN1, OUT	divEN(int input, int &output)	Realiza la división de input y output y guarda el valor en output - output = output / input

Tabla 11: Operaciones aritméticas

4.4 Contadores

Contamos con tres tipos de contadores diferentes, de los cuales podremos usar hasta 256.

4.4.1 Contador ascendente

CTU Cxxx, Pv

Contador ascendente, el cual guarda el valor de la cuenta y su estado booleano en Cxxx, las entradas de habilitación son:

Pila [top + 1] = contar hacia arriba

Pila [top] = reset

El valor booleano del contador Cxxx pasará a valor '1' cuando la cuenta del contador Cxxx>Pv, cada vez que se lee un '1' en pila[top +1] la cuenta ascenderá en uno, si el primer nivel de la pila en algún momento se activa la cuenta se reseteará.

CTU (bool &CT, int &cuenta&, int pv)

En lenguaje Arduino se usan dos variables diferentes para guardar el estado del Contador y su cuenta, el comportamiento es el mismo que en lenguaje AWL.

4.4.2 Contador descendente

CTD Cxxx, Pv

Contador descendente que comienza su cuenta en Pv y se activa al llegar ésta a '0', sus entradas de habilitación son:

Pila [top + 1] = contar hacia abajo

Pila [top] = reset

Si el segundo nivel de la pila está a '1' el contador reduce su cuenta en 1, cuando el valor de la cuenta Cxxx < Pv, el estado del contador pasa a estar activado.

CTD (bool &CT, int &cuenta, int Pv)

El comportamiento es el mismo, usando la pila para su funcionamiento, guardando su valor de estado en una variable booleana y su valor de cuenta en otra variable.

4.4.3 Contador ascendente y descendente

CTUD Cxxx, Pv

Contador ascendente y descendente, guarda su valor en Cxxx y se activa una vez que su cuenta llegue al valor Pv. Sus entradas de habilitación son las siguientes:

Pila [top + 2] = contar hacia arriba

Pila [top + 1] = contar hacia abajo

Pila [top] = reset

Si el segundo o tercer nivel de la pila se activan, el contador Cxxx contará hacia abajo o arriba respectivamente empezando su cuenta en '0', si se activa el primer nivel de la pila, el contador volverá a su estado inicial.

CTUD (bool &CT, int &cuenta, int Pv)

Diferenciamos la variable booleana y de la que lleva la cuenta, cuando Cxxx > Pv, se activará el bit de estado del contador.

4.5 Temporizadores

En lenguaje AWL del S7-200 hay tres tipos de temporizadores, cada uno con una función específica:

4.5.1 Temporizador de retardo a la conexión

TON Txxx, PT (AWL)

Temporizador de retardo a la conexión, cuando la entrada de habilitación se activa (el nivel superior de la pila está activo) comienza la cuenta, una vez que el valor de Txxx > PT el bit Txxx se activa, en el momento que se desactive la entrada el temporizador vuelve a su estado inicial.

Dado que resulta imposible en Arduino usar una misma variable para guardar dos valores distintos, como sucede con Txxx, donde se almacena la cuenta y estado del temporizador, se usan dos variables, una para guardar el estado y otra para la cuenta.

TON(bool &timer, unsigned long &tiempo, unsigned long PT)

El comportamiento es el mismo que en AWL.

4.5.2 Temporizador de retardo a la desconexión

TOFF Txxx, PT (AWL)

Temporizador de retardo a la desconexión, cuando la entrada de habilitación se activa el bit Txxx también se activa, cuando la entrada se desactiva el temporizador comienza su cuenta, una vez que el valor de la cuenta Txxx > PT el bit Txxx se desactiva.

Como se ha visto anteriormente, se usan dos variables para guardar el estado del temporizador y la cuenta, el comportamiento resulta el mismo.

TOFF (bool &timer, unsigned long &tiempo, unsigned long PT)

4.5.3 Temporizador de retardo a la conexión con memoria

TONR Txxx, PT (AWL)

Temporizador de retardo a la conexión con memoria, en el momento que la entrada de habilitación se active (el nivel superior de la pila es ‘1’) comienza la cuenta, una vez que el valor de la Txxx > PT se activa el bit de salida Txxx. Si se desactiva la entrada, la cuenta se mantiene en el instante donde se encontraba hasta que se vuelva a reanudar.

En lenguaje Arduino, además de usar dos variables para el estado y la cuenta del temporizador, debido a la complejidad del mismo, se han de declarar otras 3 variables para guardar el tiempo cuando la entrada de habilitación no esté activa.

TONR (bool &timer, unsigned long &tiempo, unsigned long PT, int &estadoOff, unsigned long &tiempoOff, unsigned long &tiempoOn)

Para el uso de temporizadores, se deben declarar acorde a la tabla 9, donde se muestran las resoluciones de los distintos temporizadores, de esta forma su comportamiento y declaración es la misma que en lenguaje AWL.

TONR	1 ms	T0, T64
	10 ms	T1 – T4, T65 – T68
	100 ms	T5 – T31, T69 – T95
TON/TOFF	1 ms	T32, T96
	10 ms	T33 – T36, T97 – T100
	100 ms	T37 – T63, T101 – T255

Tabla 12: Resolución de los temporizadores

4.6 Marcas especiales

Se han simulado una serie de marcas especiales que se han creído convenientes por su utilidad en cualquier programa y su extendido uso. La tabla 10 muestra las marcas especiales y los valores que se almacenan en ellos.

Marca AWL	Explicación	Función de habilitación
SM0.0	Siempre On	-
SM0.1	Primer ciclo On	FSMuno()
SM0.4	30 s Off / 30 s On	FSMcuatro()
SM0.5	0.5 s Off / 0.5 s On	FSMcinco()
SM0.6	Ciclo Off /ciclo On	FSMseis()
SMW22	Contiene el tiempo de ciclo	watchdog()

Tabla 13: Lista de marcas especiales soportadas

Para simular el comportamiento de dichas marcas, se han creado funciones asociadas a cada una de ellas. Estas funciones intervienen en variables creadas en Arduino equivalentes a las marcas del S7-200. Estas variables están declaradas en el ".cpp" de la librería de Arduino.

bool SMcero = 1; → SM0.0

Siempre está a '1'.

bool SMuno = 0; → SM0.1

Sólo va a estar a '1' el primer ciclo del autómata, en este caso, el primer ciclo del Arduino. Este comportamiento se debe a la función FSMuno()

bool SMcuatro = 0; → SM0.4

Esta marca estará alternativamente 30 segundos encendida y apagada, su valor está asociado a la función FSMcuatro(), que varía su valor siguiendo el ciclo de 30 segundos.

bool SMcinco = 0; → SM0.5

Esta marca estará alternativamente 0.5 segundos encendida y apagada, su valor está asociado a la función FSMcinco(), que varía su valor siguiendo el ciclo de 0.5 segundos.

Unsigned int SMW22 = 0; → SMW22

Esta es la marca que mide el tiempo de ciclo del autómata, en este caso el tiempo de ciclo que tarda nuestro Arduino en recorrer todo el programa, esta marca está asociada a la función watchdog (), que medirá el tiempo transcurrido en milisegundos cada ciclo.

4.7 Operaciones adicionales

4.7.1 Operación de guardado de variable

Para poder guardar una variable entera en otro valor, se usa la instrucción siguiente, la cual copia el valor de IN en la variable OUT.

MOVW IN, OUT

En lenguaje Arduino se usa la función equivalente que tiene el mismo comportamiento.

MOVW (int input, int &output)

4.7.2 Instrucciones de control de flujo

FOR INDX, INIT, FINAL

La instrucción entrará en un bucle si el primer nivel de la pila está activado, los parámetros del bucle son los siguientes:

INDX: es la variable donde se almacena la cuenta del bucle

INIT: valor inicial del bucle

FINAL: valor final del bucle

La instrucción NEXT sirve para pasar a la siguiente iteración, el incremento es de uno en uno.

En lenguaje Arduino se ha creado la función FOR() que comprueba si el valor del primer de la pila está activo, si se cumple, se escribirá un bucle for en lenguaje Arduino gracias al compilador en Python, que se explicará más adelante.

Ensamblador

5 Ensamblador

5.1 Introducción

Una vez creada la librería Arduino, ya se podría simular el comportamiento de un PLC, usando las funciones que incluye dicha librería, pero se tendría que conocer como programar en lenguaje Arduino, y entender cómo trabajan las funciones equivalentes para poder hacer un programa. También habría que conocer los distintos tipos de declaraciones y cómo funcionan en el programa.

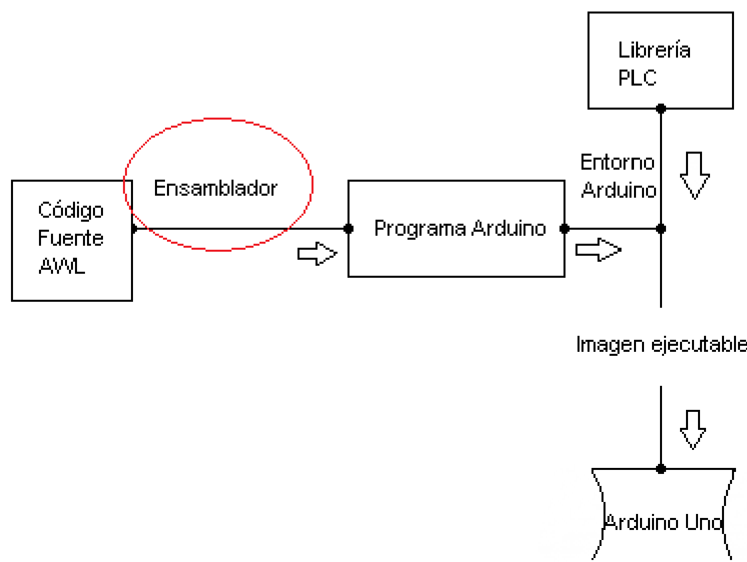


Figura 15: Detalle de la etapa de programación del ensamblador

Para solucionar este problema, se ha hecho un programa ensamblador con el lenguaje de programación Python, el cual permite que una persona desconocedora de Arduino, pero con conocimientos de programación de autómatas, pueda usar el dispositivo para la simulación, sin necesidad de conocer los fundamentos del lenguaje ni de la librería.

El archivo de lectura, donde se incorpore el programa en AWL, debe llamarse "ArduinoIn.txt", el archivo de salida donde está el código traducido, se llamará "ArduinoOut.txt".

De este modo, con la escritura de un archivo de texto con las instrucciones soportadas, ejecutando el ensamblador, se crearía un archivo de salida con el programa en lenguaje Arduino listo para introducirlo en la placa Arduino.

5.2 Programación en Python

El programa creado se basa en un procesamiento del fichero fuente en dos pasadas, en una primera pasada se declaran todas las variables necesarias y una segunda donde se escribe el código equivalente a las instrucciones AWL.

En el programa hay una serie de diccionarios que se describirán más adelante, un diccionario es una lista de funciones asociadas a cadenas de caracteres, para que cuando llamemos a uno de estos diccionarios, si la variable coincide con una de estas cadenas, se traduzca a la función que se desea.

El pseudocódigo siguiente muestra la estructura general del programa:

```
Programa principal
Crear (fichero_de_salida, escritura)
Abrir (fichero _de_entrada, lectura)
Write in fichero_de_salida (variables necesarias_programa_Arduino)
Primera lectura del fichero_de_entrada
     mnemónico, operandos ← Leer_línea()
     If mnemónico in diccionario (declaraciones)
          Case: define flancos ()
          Case: define contadores_CTU_CTD_CTUD ()
          Case: define temporizadores_TON_TOFF_TONR ()
Write in fichero_de_salida (funciones_inicialización_programa)
Segunda lectura del fichero_de_entrada
     Leer línea
     If nemónico in diccionario (instrucciones)
          Case: Instrucciones_sin_argumentos ()
          Case: Instrucciones_un_argumento ()
          Case: Instrucción_flancos ()
          Case: Instrucción_ Set_ Reset ()
          Case: Instrucción_LDS ()
          Case: Instrucción_Asignar ()
          Case: Instrucciones_comparación
          Case: Instrucciones_aritmética ()
          Case: Instr contadores_CTU_CTD_CTUD ()
          Case: Instr temporizadores_TON_TOFF_TONR ()
          Case: Instrucción_MOVW ()
          Case: Instrucciones_FOR_NEXT ()
Fin programa principal
```

En primera instancia, se crea y se abre un archivo donde se escribirá el código de salida. También se abre el archivo donde se encuentra el código que queremos transcribir.

5.2.1 Diccionario de declaración

Una vez tenemos los dos archivos abiertos, empezamos escribiendo en el fichero de salida las primeras líneas necesarias para el programa, como incluir la librería y la declaración de las variables externas usadas también en la librería, a las que necesitamos tener acceso desde el programa.

A continuación, en un bucle se leen las líneas, una por una hasta llegar al final del archivo de texto. En cada iteración del bucle, introducimos la línea de texto correspondiente en un vector. En primer lugar, quitamos los posibles comentarios separando el texto que haya entre las dobles barras (//) si lo hubiera.

Quitados los posibles comentarios, volvemos a partir el texto en dos, separándolo por el primer espacio, dejando así la instrucción en la componente ‘0’ del vector y el resto del operando en la componente ‘1’.

El siguiente paso es llamar a un diccionario de declaraciones con la componente ‘0’ del vector. En dicho diccionario, se han puesto todas las instrucciones que necesitan de declaración de variables, si la instrucción no está en el diccionario, el programa pasa a la siguiente iteración.

Las instrucciones que necesitan variables declaradas en lenguaje Arduino para funcionar correctamente son los flancos, los contadores y los temporizadores, cada uno de ellos con sus diferentes variables, como vimos anteriormente. En la declaración de flancos, simplemente cada vez que salta al diccionario escribe una variable flancox, donde x va incrementándose cada vez que se le llama, se tendrá una variable por cada flanco en el programa.

Respecto a los contadores y temporizadores, se separan los argumentos de la instrucción, se queda con el número asociado y se escriben las declaraciones necesarias según el tipo de contador o temporizador que sea. También se guardan todos sus valores significativos en otro diccionario específico para contadores y otro para temporizadores, que servirán para las cuestiones de reset.

5.2.2 Diccionario de instrucciones

Cuando se ha terminado el bucle de la declaración de variables, se asume que todas las variables necesarias están declaradas, a continuación se procede a escribir las funciones setup() y loop() necesarias para el programa Arduino, así como las funciones de inicialización de pines y de los procesos de las marcas especiales.

Lo que resta es el código de programa. Para ello, se utiliza otro bucle como el de la declaración de variables, que recorra de nuevo todo el archivo de texto de entrada, pero esta vez llamará a un diccionario que contiene todas las instrucciones posibles soportadas en AWL.

Las funciones a las que se llama se dividen en función de las similitudes entre instrucciones. Así, todas las instrucciones que en AWL no tienen argumentos están en la misma llamada, ya que sólo se requiere escribir el equivalente en lenguaje Arduino. Se exponen a continuación las diferentes posibilidades.

5.2.2.1 Sin Argumento y un argumento

Para las que tienen un argumento, se escribe en el archivo de salida la instrucción según la primera componente del vector, y se llama a otro diccionario con las diferentes variables que se pueden encontrar en dichas instrucciones. Hay dos diccionarios para este propósito, uno de argumentos de bit, y otro para argumentos de palabra, a los cuales se accederá en función de la instrucción.

A continuación se van a describir las demás llamadas a funciones debido a su complejidad dentro del código.

5.2.2.2 Flancos

Cuando se llama a esta función se escribe el código equivalente en Arduino, usando los flancos declarados anteriormente, empezando por el primero de nuevo.

5.2.2.3 Asignación

En la segunda componente del vector, que se utiliza para leer las líneas, estará guardado el argumento, el cual se guarda en una lista. Si la primera letra es una M, quiere decir que es necesario cambiar el estado de un bit de una palabra, por lo que se ha de escribir la instrucción adecuada.

Se usa la función 'asignarbit' vista previamente, que tiene dos argumentos de entrada, la palabra y el número de bit. Para saber los valores a introducir dada una marca Mx.y, se hace el módulo de la primera, si es '0' la y queda igual, pero si es '1' habría que sumarle 8 a la y para llegar al bit deseado. Esto es necesario, ya que las palabras son de 16 bits.

En cualquier otro caso, se llama al diccionario de las variables de bit en el que se escribe la equivalencia con respecto a AWL.

5.2.2.4 Set y Reset

Dentro de esta función están todas las posibles combinaciones que se pueden dar en AWL con sus respectivas equivalencias en Arduino. Dado que en AWL se usa la misma instrucción para todos los tipos de reset el código se dividirá en varias partes.

Se comprueba qué tipo de letra es la que lleva el primer argumento. Si es una 'Q' se llamará al diccionario de las variables de bit y se escribirá en el archivo de salida la función correspondiente.

En caso de ser una 'M', hay que escribir la función que setea o resetea bits de una palabra, por lo que el procedimiento es el mismo que cuando se asigna un valor a una marca.

Los otros dos posibles casos son: que se resetee un temporizador o contador. En este caso, se extraerán los datos de dicho temporizador o contador de los diccionarios creados en la declaración de variables, y se escribirá la función correspondiente en Arduino.

5.2.2.5 Instrucciones de comparación

Se comprueban los signos de comparación y se escribe lo correspondiente según el resultado. A continuación, se dividen los dos argumentos y se llama al diccionario de las variables de palabra para su escritura en el archivo de salida. En el caso de que se escriba un número en vez de una variable, el diccionario devolvería un error y se escribiría en el archivo de salida el número correspondiente.

5.2.2.6 Instrucciones de aritmética

Como en las instrucciones de comparación, se comprueba primero que tipo de instrucción es y se escribe en el archivo de salida el resultado. Seguidamente, se separan los argumentos y se vuelve a llamar al diccionario de variables de palabra.

5.2.2.7 Contadores

Para los contadores, como se vio anteriormente, en AWL se utiliza una sola variable para guardar el estado del contador y su cuenta. Ya se ha declarado previamente estas variables como Cvxxx para el valor de bit y Ccxxx para la cuenta. Se guarda el valor que acompaña al contador y se escriben los argumentos de entrada Cv y Cc junto con dicho valor, que será el número de identificador del contador. El otro argumento es el valor que se desea alcanzar, que se escribirá a continuación.

5.2.2.8 Temporizadores TON y TOFF

Se realizan de la misma forma que los contadores, salvo por una peculiaridad. Para que los temporizadores estén asignados como los temporizadores reservados para el PLC, se dividen según la resolución. Se añaden ceros al final en función del identificador del temporizador, para que la resolución sea la misma.

Hay que tener en cuenta que los temporizadores en lenguaje Arduino se miden todos en milisegundos, por lo que esta operación es necesaria para simular los diferentes temporizadores según su identificador.

5.2.2.9 Temporizadores TONR

Al llamar a esta función, se escriben los valores igual que los temporizadores anteriores, teniendo en cuenta que para los temporizadores TONR se usa otro rango de valores para las resoluciones.

También hay que tener en cuenta que los temporizadores TONR requieren de más variables de entrada, que se escribirán después del valor PT de acuerdo al valor del identificador del temporizador.

5.2.2.10 Instrucción MOVW

Escribe la instrucción equivalente y a continuación llama al diccionario de variables de palabra con los dos argumentos suministrados, escribiendo así la función en Lenguaje Arduino.

5.2.2.11 *Instrucciones FOR y NEXT*

Se introduce una condición dentro de Arduino con la función de la librería FOR(), que devuelve el estado del primer nivel de la pila. Dentro de la condición, se escribe un bucle 'for' directamente en Arduino con los valores que se leen de la instrucción en lenguaje AWL. Escribiendo todas las instrucciones siguientes dentro del bucle, el cual se cerrará cuando el compilador lea la instrucción NEXT. De esta forma, se cerrará el bucle creado en lenguaje Arduino con una llave '}'.

Prueba de funcionamiento

6 Prueba de funcionamiento

6.1 Estación 1: Medida de piezas

Para comprobar el correcto funcionamiento del programa elaborado se ha elegido realizar la automatización de la estación 1 de la planta de automatización del fabricante FESTO, utilizada para docencia, a partir de ahora planta FESTO.

Se podrá ver el resultado de escribir un código en AWL para esta estación y, a través del ensamblador y la ejecución del programa en lenguaje Arduino, vislumbrar el correcto funcionamiento de la misma

La estación 1 de la planta Festo (figura 3) es una estación de test, cuyo objetivo es descartar las piezas de tamaño no adecuado y llevar hasta la siguiente estación las piezas validas, de manera coordinada. Para controlar esta planta se utiliza un autómata S7-300, en nuestro caso se va a simular como si el autómata fuera un S7-200.

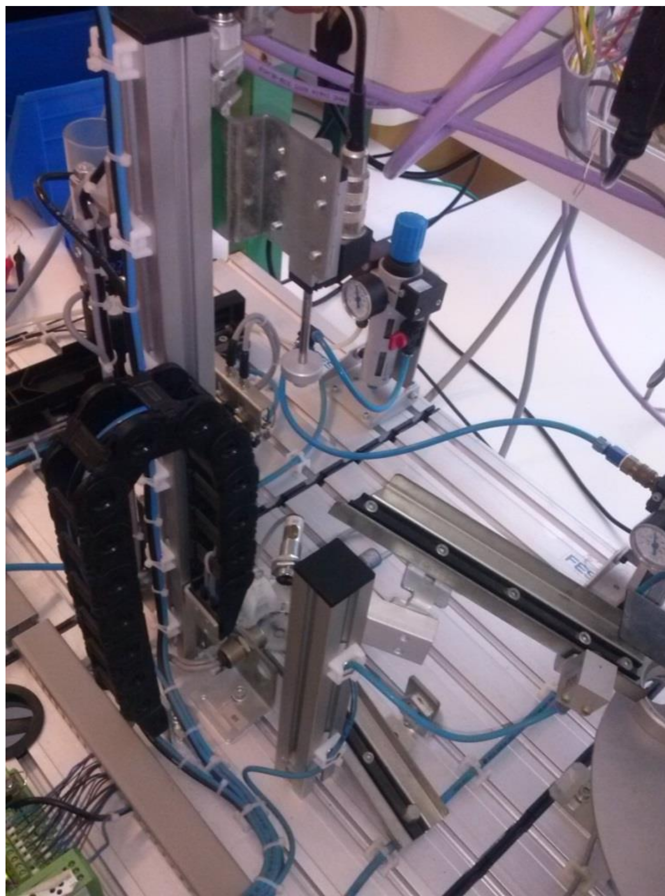


Figura 16: Estación 1 de la planta Festo

El funcionamiento general de la estación de test es el que se describe a continuación:

1. Hay una plataforma que puede estar situada en dos posiciones distintas (posición 1: abajo; posición 2: arriba). En cada una de estas posiciones hay una rampa con una determinada funcionalidad. En la posición 1 se encuentra la rampa 1, mientras que en la posición 2 encontramos la rampa 2.

2. En la posición 1 es donde llegan las piezas procedentes de la estación 0. Existe un sensor que detecta cuándo hay una pieza en la plataforma y otro que indica que la plataforma está abajo.

3. Cuando llega una pieza a la plataforma en la posición 1 se debe subir ésta hacia la posición2.

4. Una vez que la plataforma con la pieza llega arriba (posición 2), se debe bajar un sensor que mide la longitud de la pieza.

5. Esta medida se compara con el valor 'longitud normal de la pieza'. Si la pieza tiene el tamaño 'normal' debe ser enviada a la siguiente estación. Para ello es necesario activar un pistón que la empuja por la rampa 2, que comunica la estación 1 con la estación 2. Además, se usará una pequeña pestaña para retener la pieza en la rampa 2 e impedir así que pase directamente a la estación 2. La pestaña se retraerá permitiendo el paso de la pieza a la estación 2 una vez que ésta indique que está preparada para recibir una nueva pieza.

6. Una vez que se ha empujado la pieza de tamaño 'normal' hacia la rampa 2, la plataforma debe bajar a la posición 1 y permanecer a la espera de que le llegue una nueva pieza procedente de la estación 0.

7. Si la pieza cuya longitud se mide resulta ser más grande o más pequeña de lo 'normal', se considera que es una pieza defectuosa, por lo que debe ser enviada a la estación 2. En este caso, la plataforma debe bajar a la posición 1 con la pieza defectuosa.

8. Una vez que la plataforma con la pieza defectuosa llega a la posición 1 (abajo), se debe empujar ésta con ayuda de un pistón por la rampa 1, por la que se descartan las piezas que no tienen el tamaño adecuado. Una vez que se ha empujado la pieza defectuosa, la plataforma debe permanecer a la espera de que le llegue una nueva pieza procedente de la estación 0.

A continuación se muestra la tabla 14 y 15 donde se muestra la distribución de entradas y salidas de la placa Arduino, así como los sensores y actuadores con sus equivalencias.

Dirección S7-300	Equivalencia S7-200	Equivalencia en lenguaje Arduino	Pin físico de la placa Arduino	Sensor
E124.1	I0.0	X0	Pin digital 2	Detector de objeto en posición 1
E 124.3	I0.1	X1	Pin digital 3	Detector de final de carrera de posición 1
E 124.4	I0.2	X2	Pin digital 4	Detector de final de carrera de posición 2
E 124.5	I0.3	X3	Pin digital 5	Detector del estado del pistón)'1' extraído)
E 124.6	I0.4	X4	Pin digital 6	Detector del estado del sensor que mide la longitud de la pieza ('1' extraído)
PEW130	Ain0	Ain0	Pin analógico A0	Entrada analógica: resultado de la medida de la pieza

Tabla 14: Lista de sensores de la estación 1

Dirección S7-300	Equivalencia S7-200	Equivalencia en lenguaje Arduino	Pin físico de la placa Arduino	Sensor
A 124.0	Q0.0	Y0	Pin digital 8	Bajar la plataforma
A 124.1	Q0.1	Y1	Pin digital 9	Subir la plataforma
A 124.2	Q0.2	Y2	Pin digital 10	Extender pistón
A 124.3	Q0.3	Y3	Pin digital 11	Bajar/Subir sensor de medida
A 124.4	Q0.4	Y4	Pin digital 12	Activar/desactivar pistón de sujeción en rampa 2

Tabla 15: Lista de actuadores de la estación 1

6.2 Circuito de conexión

Dado que no se va a usar el PLC S7-300 para la automatización, si no la placa Arduino, se ha redirigido las señales de los sensores y actuadores a una caja de conexiones exterior a la estación. De esta forma, tendremos dos circuitos que se describirán a continuación.

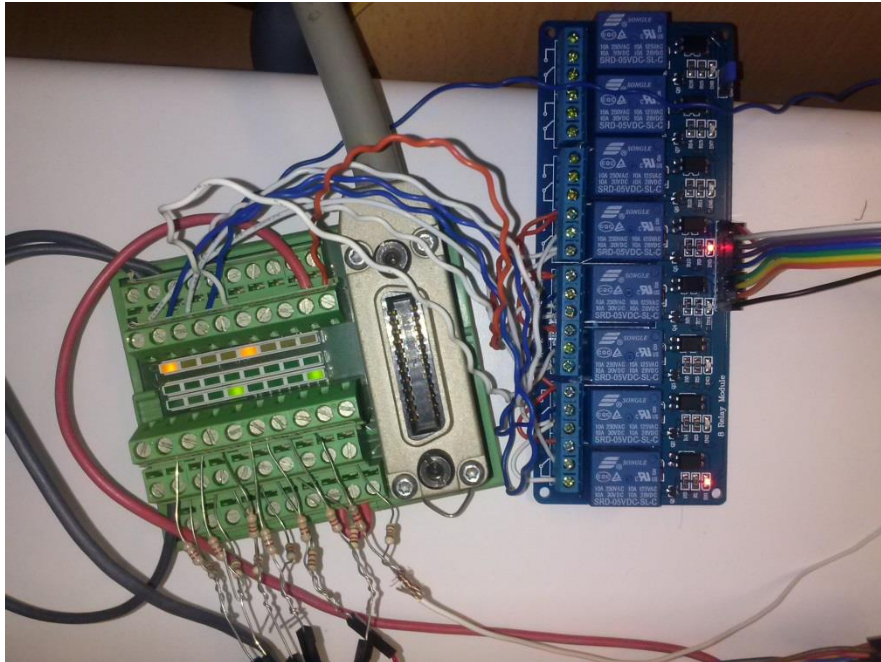


Figura 17: Caja de conexiones y módulo de relés

Figura 9: Caja de conexiones y módulo de relés

Se puede observar en la figura 4 la nueva caja de conexiones que está conectada pin a pin a la de la estación

Para los circuitos de entrada, debido a que la tensión de salida de los sensores trabaja con señales de 24v, se realiza un divisor de tensión con dos resistencias de 8.2kΩ y 1.2kΩ. De esta forma, la tensión que llega a la placa es de unos 3v si el sensor está activado y de 0v si no lo está.

La figura 5 muestra los circuitos de entrada de cada uno de los pines, incluido el de la señal analógica, cuyo voltaje variará dentro de estos márgenes.

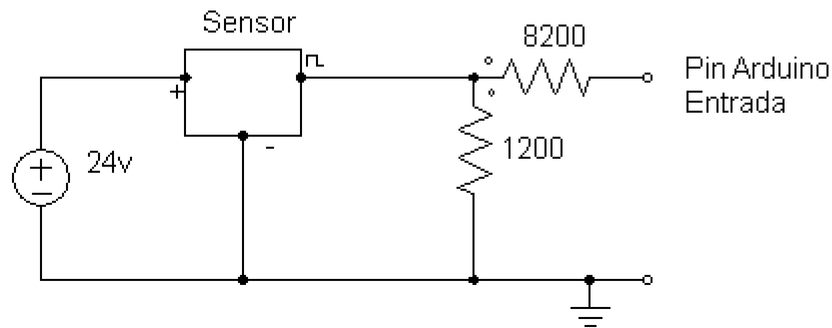


Figura 18: Circuito de entrada a la placa Arduino

Para los circuitos de salida (figura 18), los pines suministran un máximo de 5v y unos 140mA, características insuficientes para actuar sobre los actuadores. Para solucionar este problema y proteger a la placa de altas tensiones, se hace pasar cada una de las salidas por un circuito de relés, que aísla el circuito de entrada del de salida. La salida de los relés tiene un suministro de 24v para poder accionar los actuadores, a continuación se muestra el circuito de los pines de salida.

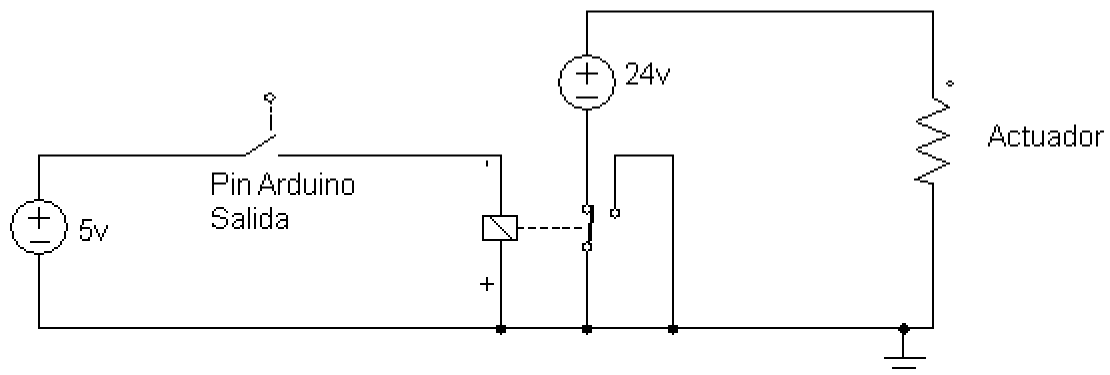


Figura 19: Circuito de salida de la placa Arduino

Una vez conectados todos los pines en entrada y de salida, ya se podría programar la planta para su correcto funcionamiento. Además, se ha hecho un circuito sencillo (figura 19) para simular la comunicación con la estación siguiente, que se simula mediante un botón, que si se pulsa enviamos una señal al pin físico de la placa número 7 de 5v.

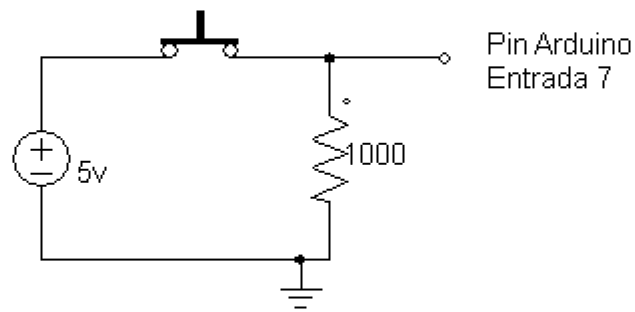


Figura 20: Circuito para la simulación de comunicación

Se puede ver el montaje de la placa Arduino junto con el circuito de control para la comunicación en la figura 20.

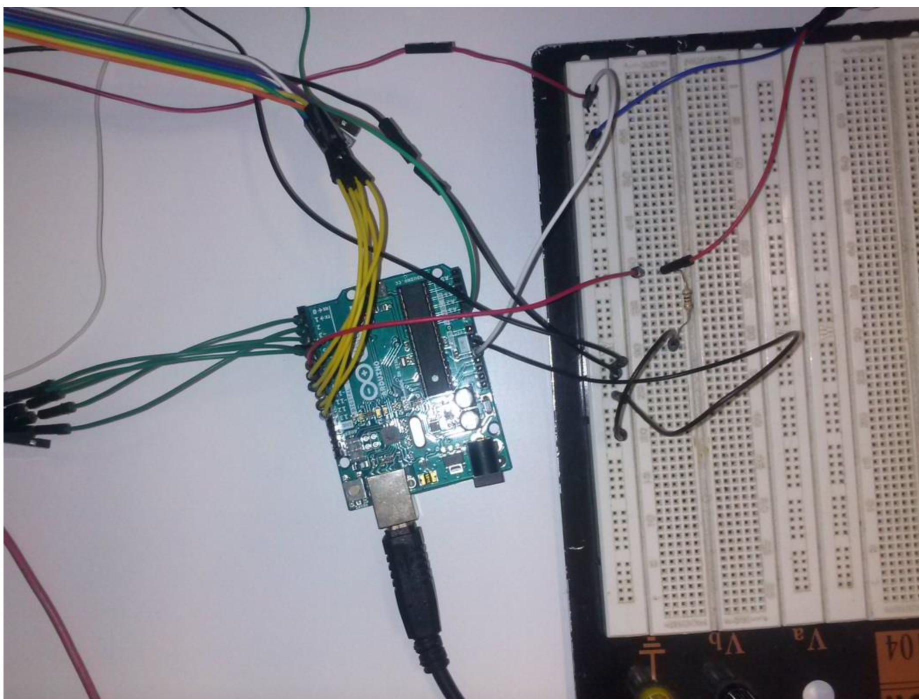


Figura 21: Conexiones placa Arduino y botón de simulación

6.3 Programación

Se va a proceder a mostrar una parte del programa realizado para la estación 1, para comprobar el resultado de la compilación y su correcto paso de instrucciones en AWL a lenguaje Arduino.

En un archivo de texto al cual nombraremos 'ArduinoIn.txt' (necesario para que el compilador lo lea) se escribirá el código en AWL utilizando las salidas y entradas según la conexión que se ha hecho. El código de programa mostrado es el principio del programa.

En él se hace uso de la mayoría de las instrucciones soportadas por el programa Arduino. Cabe destacar que las entradas a relé están invertidas, activando los relés con un nivel de 0v, y desactivándolos con un nivel de 5v. A continuación el código en AWL:

```
//Las operaciones de inicialización en el primer ciclo del autómata.

LD SM0.1 //En el primer ciclo del autómata
S Q0.0,1 //Se pone a 0 Bajar Plataforma
S Q0.2,3 // y se desactiva el Sensor de longitud, y los dos pistones

R M0.4,3 //Se pone a 1 la marca que indica...
LDN I0.1 //Si la mesa no esta bajada
ALD      // y es el primer ciclo del autómata
R Q0.0,1 //Se activa Bajar Plataforma
S Q0.1,1 // y se desactiva Subir Plataforma

//Cuando llega una pieza nueva a la estación

LD I0.1  //Si la plataforma está en la posición 1
LDN M1.0 // y la pieza no es mala
ALD
A I0.0   // y la plataforma esta bajada
S Q0.0,1 //Se desactiva Bajar Plataforma
R Q0.1,1 // y se activa Subir Plataforma
S M2.0,1 //El Sensor de longitud se puede bajar

//En caso de que la plataforma llegue abajo, se para

LD I0.1  //Cuando la plataforma llega abajo
EU       // con un flanco de subida
S Q0.1,1 //Se desactiva Subir Plataforma
R Q0.0,1 // y se activa Bajar Plataforma

//Una vez que la plataforma sube con la pieza, se activa el sensor

LD I0.2  //Si la plataforma se encuentra arriba
LDN I0.4 // y el Sensor de longitud no está extendido
ALD
A M2.0   // y se puede bajar dicho sensor
R Q0.3,1 //Se activa el Sensor de longitud
R M2.0,1 // y se resetea la marca para bajar el sensor
```

```
//Cuando se ha extendido el sensor, se mide la pieza

LD I0.4  //Si el sensor está extendido
A I0.2   // y la plataforma está arriba
TON T101,5 //contamos medio segundo res T101 = 100 ms
LD T101  //Cuando se cuenten los 0'5 s
EU       // en un flanco de subida
MOVW  Ain0,MW3 //Movemos el valor que está midiendo el sensor a la var
entera MW3
S Q0.3,1 // y subimos el Sensor de longitud
```

/*Las operaciones para comprobar si la pieza es pata, los valores están escogidos dentro del umbral en el que Arduino mide las piezas, estando en torno a 190 las piezas pequeñas, 207 las normales y 228 las grande*/

```
LDW< 195,MW3 //Si el valor medido es mayor que 195
LDW> 215, MW3// y es menor que 215
ALD
S M0.4,1 //La pieza es apta
```

A continuación se ejecuta el compilador, que creará un archivo de salida llamado 'ArduinoOut.txt', que contendrá el programa listo para ejecutar en la placa.

El archivo de salida se muestra a continuación, en el que aparecen la llamada a la librería, la declaración de las variables, y el código equivalente de la parte representativa del programa. El código completo se puede ver en el anexo 3.

Se incluye la librería y la declaración de las variables necesarias para soportar todas las funciones en caso de necesitarlas.

```
#include <PLC.h>

extern int MW[16];
extern unsigned int SMW22;
extern bool SMcero;
extern bool SMuno;
extern bool SMcuatro;
extern bool SMcinco;
extern int Ain0;
extern int Ain1;
extern int Ain2;
extern int Ain3;
extern int Ain4;
extern int Ain5;

//Se declaran las variables que se van a necesitar para las funciones


int flanco0= 0;
bool Tv101 = 0;
```

```
unsigned long Tc101= 0;
int flanco1= 0;
bool Tv102 = 0;
unsigned long Tc102= 0;
bool Tv104 = 0;
unsigned long Tc104= 0;
```

//Una vez terminada la declaración, se conrfiguran los pines de entrada/salida con la función setupPLC()

```
void setup(){

setupPLC();

}

void loop(){

//Se leen las entradas analógicas y se comprueban las marcas especiales en cada ciclo

procesos();
```

//Las operaciones de inicialización en el primer ciclo del autómata

```
LD(SMuno);
S(Y0,1);
S(Y2,3);
Rbit(MW[0],4,3);
LDN(X1);
ALD();
R(Y0,1);
S(Y1,1);

//Cuando llega una pieza nueva a la estación

LD(X1);
LDN(extraerbit(MW[0],8));
ALD();
A(X0);
S(Y0,1);
R(Y1,1);
Sbit(MW[1],0,1);
```

//En caso de que la plataforma llegue abajo, se para

```
LD(X1);
EU(flanco0);
S(Y1,1);
R(Y0,1);
```

// una vez que la plataforma con la pieza, se activa el sensor

```
LD(X2);
LDN(X4);
ALD();
A(extraerbit(MW[1],0));
```

```
R(Y3,1);
Rbit(MW[1],0,1);


//Cuando se ha extendido el sensor, se mide la pieza

LD(X4);
A(X2);
TON(Tv101,Tc101,500);
LD(Tv101);
EU(flanco1);
MOVW(Ain0,MW[3]);
S(Y3,1);


//Se comprueba que la pieza es apta midiéndola entre los dos umbrales

LDWLT(195,MW[3]);
LDWGT(215,MW[3]);
ALD();
Sbit(MW[0],4,1);
```

Se ejecuta este código en la estación 1, con el resultado de que la estación funciona correctamente y entrega las piezas aptas a la siguiente estación y desecha las piezas no aptas. El código al completo se encuentra en el anexo 3.

Conclusiones

7 Conclusiones

7.1 Conclusiones sobre el proyecto

Se ha hecho un proceso de selección estudiando previamente los diferentes dispositivos del mercado y sus características para optar finalmente por implementar el proyecto con un PLC S7-200 de Siemens, utilizando lenguaje AWL, sobre una placa Arduino Uno

Se ha simulado el comportamiento de un autómata mediante una placa Arduino. Se puede escribir un archivo de texto utilizando programación en AWL (lista de instrucciones), para posteriormente, ejecutando un compilador conseguir la traducción a un programa equivalente en lenguaje Arduino. Dicha traducción contendrá todas las variables y funciones para que el programa resultante realice el mismo cometido que un PLC.

Se ha construido una librería para Arduino que contiene todas las funciones necesarias para adaptar cada instrucción de AWL a lenguaje de Arduino y que el programa se comporte de la misma forma que un autómata. Dentro del cual se han llevado a cabo con las instrucciones más representativas y comunes a la hora de programar un autómata.

Se ha implementado un módulo ensamblador para conseguir una conversión de lenguaje AWL a lenguaje Arduino. Así, a partir de un fichero de entrada (el código en AWL) crea un fichero de salida con el programa listo para ejecutarse por el Arduino.

El ensamblador cumple su función, traduce un programa hecho en lenguaje AWL, siempre dentro de las limitaciones que se han expuesto en el proyecto, a un código resultante legible para un Arduino.

De esta forma, una persona externa no tiene por qué tener conocimientos sobre Arduino ni de su programación. Para usar el dispositivo de bajo coste, como es un Arduino, para tareas de automática, tan sólo debe saber cómo programar en AWL. Se ha hecho una programación estructurada a base de diccionarios, con los que se lee línea a línea el archivo de texto de entrada y va escribiendo en el de salida.

Se ha valida el sistema haciendo una prueba de funcionamiento en la estación 1 de la planta Festo. El programa hecho para dicha planta simula eficazmente el comportamiento de un autómata.

7.2 Líneas futuras

El amplio abanico de instrucciones y operaciones que posee el S7-200 deja la posibilidad de mejora abierta a la introducción de nuevas funciones en la librería, que puedan ser usadas de acuerdo a cómo trabajan en un autómata.

La mejora de la librería Arduino implicaría una necesaria modificación en el compilador. El programa está hecho de tal forma que su modificación para añadir nuevas instrucciones no sería complicada.

De esta forma, se podría aumentar el número de funciones soportadas haciendo un simulador mucho más completo.

Siguiendo el hilo de la programación, la posibilidad de dar nombres a las variables del programa para una mayor facilidad y comprensión del código.

La creación de un programa que englobe los dos ya hechos, para aportar simplicidad a la ejecución de los mismos, un entorno que traduzca e introduzca en la placa Arduino el código en AWL sin que el usuario tenga que interactuar con los códigos intermedios generados.

Por otra parte, puede que la complejidad de las conexiones a realizar sea un inconveniente. La creación de un elemento compacto que incorpore todos los elementos necesarios para simplificar la conexión entre placa Arduino y la estación en la que trabajar puede ser una buena línea de trabajo.

8 Conclusions

8.1 Conclusions

A selection process has been made to determinate which devices will be used in the implementation of the project. The final choice was a Siemens S7-200 PLC, using AWL language and an Arduino One board.

A PLC has been simulated using Arduino board. An assembler program reads a file where the AWL instructions are, and produces a translation of an equivalent program in Arduino language. This translation will define all the variables and functions needed to simulate the PLC’s behavior. Analyzing all the possible options and instructions a PLC can hold, like the S7-200, the project has been done with the more common and representative instruction to program a PLC.

An Arduino library has been created to develop this project. It contains all the necessary functions to adapt every AWL instruction to Arduino language. That way, every instruction that is written in the read file will have its equivalent in Arduino Language, to obtain this conversion, a compiler in Python language has been made. This compiler creates an output file with the program in Arduino language ready to use.

Regarding to the assembler programmed in Python language, it is done a structured programming with dictionaries, it reads line to line from the read file and writes in the output file. The assembler accomplishes its function, it translates a progam made in awl language, always inside the limitations exposed in the Project, which means, to a code that can be read in Arduino.

You can observe this results with the functionality test in station 1 in the Festo Plant. The progam made for this plant simulates effectively the behavior of a PLC.

8.2 Future lines

The wide range of instructions and operatiosn that the S7-200 has left the possibility of an improvement of the Arduino library, including new functions in the library. These functions will simulate other aspects of the PLC.

The improvement of the Arduino library would implicate a necessary modification of the compiler. Nevertheless, the use of dictionaries in the compiler would make improvements easy to make and add new code to the program.

Consequently, the number of functionalities and operation supported by the simulator would be much bigger.

Additionally, it should be considered the possibility of adding the functionality of giving names for variables to make the AWL code easier to read.

The creation of a joint program, which incorporates all the functionalities in only one program, would add simplicity to the user experience in order to use the Arduino PLC. This is, an environment that translates and introduces in the Arduino board the AWL code without having to manipulate the processes in between.

Furthermore, maybe the complexity of making the connections with the external devices would be an inconvenient. The creation of a compact device which incorporates all the necessary elements to simplify the connection between Arduino and the external signals could be a good future improvement.

Bibliografía

9 Bibliografía

[1]SIMATIC Sistemas de automatización S7-200, Manual de sistema:

Siemens, Referencia del manual: 6ES7298-8FA20-8DH0

[2]Página oficial de Arduino: https://www.arduino.cc/

[3]Página oficial de Python: https://www.python.org/

[4]Página con información referente a PLC's: http://www.plcopen.org/

[5]Página de Siemens: http: //www.siemens.com/entry/cc/en/

[6]Página de Schneider: http://www.schneider-electric.com/

[7]Página de Omron: http://industrial.omron.es/es/home

ANEXOS

10 Anexos

Índice Anexos

10.1 Librería Arduino 67

10.1.1 Archivo .h 67

10.1.2 Archivo.Cpp 69

10.2 Programa ensamblador en Python 85

10.3 Programa de prueba de funcionamiento 97

10.3.1 ArduinoIn.txt 97

10.3.2 ArduinoOut.txt 88

Librería Arduino

10.1 Librería Arduino

10.1.1 Archivo .h

El siguiente programa es el archivo “PLC.h” de declaración de funciones:

```
/*
PLC.h - Library
Creado por: Sergio Heras Aguilar
*/
```

En caso de que PLC_h no esté definido, se define y se incluye la librería con las funciones estándar de Arduino.

```
#ifndef PLC_h
#define PLC_h
#include "Arduino.h"
```

Se declaran las entradas y salidas y se les asigna un número constante, los pines de la placa.

```
const int X0 = 2;
const int X1 = 3;
const int X2 = 4;
const int X3 = 5;
const int X4 = 6;
const int X5 = 7;

const int Y0 = 8;
const int Y1 = 9;
const int Y2 = 10;
const int Y3 = 11;
const int Y4 = 12;
const int Y5 = 13;
```

Se introducen todas las funciones para describir su funcionamiento en el otro archivo.

```
        void setupPLC();
        void procesos();

        void LD (int input);
        void LD (bool input);
        void LDN (int input);
        void LDN (bool input);
        void A(int input);
        void A(bool input);
        void AN(int input);
        void AN(bool input);
        void O(int input);
        void O(bool input);
        void ON(int input);
        void ON(bool input);
        void NOT();
        void S(int output, int n);
        void R(int output, int n);
        void Sbit(int &palabra,int N, int n);
        void Rbit(int &palabra,int N, int n);
```

```
        void RCTUD(bool &CT, int &cuenta);
        void RCTD(bool &CT, int &cuenta, int pv);
        void RTON(bool &timer,unsigned long &tiempo);
        void RTOFF(bool &timer,unsigned long &tiempo,unsigned long PT);
        void  RTONR(bool  &timer,  unsigned  long  &tiempo,  int  &estadoOff,
unsigned long &tiempoOff, unsigned long &tiempoOn);

        void Asignar(int output);
        void Asignar(bool output);
        void EU(int &state);
        void ED(int &state);

        void sumaEN(int input, int &output);
        void suma(int input, int &output);
        void restaEN(int input, int &output);
        void resta(int input, int &output);
        void mulEN(int input, int &output);
        void mul(int input, int &output);
        void divEN(int input, int &output);
        void div(int input, int &output);

        void LDWE(int input1, int input2);
        void LDWNE(int input1, int input2);
        void LDWLT(int input1, int input2);
        void LDWGT(int input1, int input2);
        void LDWLTE(int input1, int input2);
        void LDWGTE(int input1, int input2);

        void ALD ();
        void OLD ();
        void LPS ();
        void LRD ();
        void LDS (int n);
        void LPP ();
        void watchdog();
        void FSMuno();
        void FSMcuatro();
        void FSMcinco();
        void MOVW(int input, int &output);
        void colocarPila();
        bool extraerbit(int palabra, int N);
        void asignarbit(int &palabra, int N);

        void TON(bool &timer, unsigned long &tiempo, unsigned long PT);
        void TOFF(bool &timer, unsigned long &tiempo, unsigned long PT);
        void TONR(bool &timer, unsigned long &tiempo, unsigned long PT, int
&estadoOff, unsigned long &tiempoOff, unsigned long &tiempoOn);
        void CTU(bool &CT, int&cuenta, int pv);
        void CTD(bool &CT, int &cuenta, int pv);
        void CTUD(bool &CTU, int &cuenta, int pv);

        bool FOR();

#endif
```

10.1.2 Archivo.Cpp

El siguiente programa es el archivo “PLC.cpp” donde se describen las funciones declaradas:

```
/*
PLC.cpp - Librería
Creado por: Sergio Heras Aguilar
*/
```

Se incluyen la librería estándar de Arduino y nuestro fichero de declaración de funciones.

```
#include "Arduino.h"
#include "PLC.h"
```

Se declaran todas las variables necesarias.

```
#define MASCARA 1

bool pila [8];
int top = 0;

int MW[16];

unsigned int SMW22 = 0; // tiempo de ciclo
bool SMcero = 1; //siempre a 1
bool SMuno = 0; // SM1 = setup
bool SMcuatro = 0; // 30 s On /30 s Off
bool SMcinco = 0;  //0.5 s On / 0.5 s Off

//Declaración de entradas analógicas

int Ain0 = 0;
int Ain1 = 0;
int Ain2 = 0;
int Ain3 = 0;
int Ain4 = 0;
int Ain5 = 0;
```

La primera función que es de obligada incorporación en el programa, inicializa los pines según sean de entrada y salida para un correcto funcionamiento de la placa y inicializa a ‘0’ todas las componentes del vector de enteros MW.

```
//función para inicializar los pines

void setupPLC(){

        for (int m = 0; m <= 16; m++){
                MW[m] = 0;
        }

        pinMode(X0, INPUT);
        pinMode(X1, INPUT);
        pinMode(X2, INPUT);
        pinMode(X3, INPUT);
        pinMode(X4, INPUT);
        pinMode(X5, INPUT);

        pinMode(Y0, OUTPUT);
        pinMode(Y1, OUTPUT);
```

```
        pinMode(Y2, OUTPUT);
        pinMode(Y3, OUTPUT);
        pinMode(Y4, OUTPUT);
        pinMode(Y5, OUTPUT);

        digitalWrite(Y0, LOW);
        digitalWrite(Y1, LOW);
        digitalWrite(Y2, LOW);
        digitalWrite(Y3, LOW);
        digitalWrite(Y4, LOW);
        digitalWrite(Y5, LOW);
}
```

La función procesos() ejecuta todas las funciones de las marcas especiales y asigna los valores captados por las entradas analógicas a sus respectivas variables.

```
//función que ejecuta los procesos deseados

void procesos(){
        watchdog();
        FSMuno();
        FSMcuatro();
        FSMcinco();
        Ain0 =analogRead(A0);
        Ain1 =analogRead(A1);
        Ain2 =analogRead(A2);
        Ain3 =analogRead(A3);
        Ain4 =analogRead(A4);
        Ain5 =analogRead(A5);
}
```

Las siguientes funciones operan con la pila lógica, hay dos funciones para cada operación en relación a que valor queremos introducir en la pila, el valor de un pin o el de una variable.

```
//Cargar valor en pila

void LD (int input){
        colocarPila();
        pila [top] = digitalRead(input);
}

void LD (bool input){
     colocarPila();
     pila[top] = input;
}

//Cargar valor negado en pila

void LDN (int input){
     colocarPila();
     pila [top] = digitalRead(input);
     if (pila[top] == 1) {
               pila[top] = 0;
     }
     else{
               pila[top] = 1;
     }
}
```

```
void LDN (bool input){
	colocarPila();
	pila [top] = input;
	if (pila[top] == 1) {
			pila[top] = 0;
	}
	else{
			pila[top] = 1;
	}
}

//operación AND lógica

void A(int input) {
	pila[top] = pila[top] & digitalRead(input);
}

void A(bool input) {
	pila[top] = pila[top] & input;
}

//operación AND lógica negada

void AN(int input) {
	pila[top] = pila[top] & digitalRead(input);
	if (pila[top] == 1) {
			pila[top] = 0;
	}
	else{
			pila[top] = 1;
	}
}

void AN(bool input) {
	pila[top] = pila[top] & input;
	if (pila[top] == 1) {
			pila[top] = 0;
	}
	else{
			pila[top] = 1;
	}
}

//operación OR lógica

void O(int input) {
	  pila[top] = pila[top] | digitalRead(input);
}

void O(bool input) {
	  pila[top] = pila[top] | input;
}
```

```
//operación OR lógica negada

void ON(int input) {
      pila[top] = pila[top] | digitalRead(input);
      if (pila[top] == 1) {
        pila[top] = 0;
      }
      else{
        pila[top] = 1;
      }
}

void ON(bool input) {
      pila[top] = pila[top] | input;
      if (pila[top] == 1) {
        pila[top] = 0;
      }
      else{
        pila[top] = 1;
      }
}

//operación NOT en el primer nivel de la pila

void NOT() {
        if (pila[top] == 1) {
                pila[top] = 0;
        }
        else{
        pila[top] = 1;
     }
}

//Operación de set en los pines de salida

void S(int output, int n) {
        if (pila[top] == 1) {
                int acu = 0;
                for ( int i = output; i < n + output; i++) {
                        digitalWrite(output + acu, HIGH);
                        acu++;
                }
        }
}

//Operación de reset en los pines de salida

void R(int output, int n) {
        if (pila[top] == 1) {
        int acu = 0;
        for ( int i = output; i < n + output; i++) {
                        digitalWrite(output + acu, LOW);
                        acu++;
                }
        }
}
```

```
//Operación de set para las marcas M

void Sbit(int &palabra,int N, int n){
        if(pila[top] == 1){
                int acu = 0;
                for( int i = N; i < N + n; i++){
                palabra ^= (-1 ^ palabra) & (1 << i);
                }
        }
}

//Operación de reset para las marcas M

void Rbit(int &palabra,int N, int n){
        if(pila[top] == 1){
                int acu = 0;
                for( int i = N; i < N + n; i++){
                palabra ^= (-0 ^ palabra) & (1 << i);
                }
        }
}
```

Estas funciones se usan para resetear cada tipo de contador o temporizador asignado previamente, dependiendo de su tipo.

```
//Reset de temporizadores y contadores

void RCTUD(bool &CT, int &cuenta){
        if(pila[top] == 1){
                CT = 0;
                cuenta = 0;
        }
}

void RCTD(bool &CT, int &cuenta, int pv){
        if(pila[top] == 1){
                CT = 0;
                cuenta = pv;
        }
}

void RTON(bool &timer,unsigned long &tiempo) {
        if(pila[top] == 1){
                timer = 0;
                tiempo = 0;
        }
}

void RTOFF(bool &timer,unsigned long &tiempo,unsigned long PT){
        if(pila[top] == 1){
                timer = 0;
                tiempo = PT;
        }
}
```

```
void RTONR(bool &timer, unsigned long &tiempo, int &estadoOff, unsigned
long &tiempoOff, unsigned long &tiempoOn){
	if(pila[top] == 1){
		timer = 0;
		tiempo = 0;
		estadoOff = 0;
		tiempoOff = 0;
		tiempoOn = 0;
	}
}

//Asignar un valor a un pin de salida

void Asignar(int output) {
	if (pila[top] == 1) {
		digitalWrite(output, HIGH);
	}
	else {
		digitalWrite(output, LOW);
	}
}

//Procesar el valor de una marca M

bool extraerbit(int palabra, int N){
	int tmp = 0;
	tmp = palabra >> (N);
	return(tmp & MASCARA);
}

//Asignar un valor a una marca M

void asignarbit(int &palabra, int N){
	palabra ^= (-pila[top] ^ palabra) & (1 << N);
}

//Asignar un valor a una variable

void Asignar(bool output) {
		output = pila[top];
}
```

Las funciones que operan con los niveles de pila, cada una de ellas de una forma diferente

```
//operaciones que trabajan con la pila lógica

void ALD () {
	pila[top] = pila[top] & pila[top + 1];
	for(int i = 1; i < 7; i++){
		pila[i] = pila[i +1];
	}
}
```

```
void OLD () {
        pila[top] = pila[top] | pila[top + 1];
        for(int i = 1; i < 7; i++){
                pila[i] = pila[i + 1];
        }
}

void LPS () {
        pila[top + 1] = pila[top];
        for(int i = 2; i < 8; i++){
                pila[i] = pila[i - 1];
        }
}

void LRD () {
        pila[top] = pila[top + 1];
}

void LDS (int n) {
        for( int i = 0; i < 8; i++){
                pila[i+1] = pila[i];
        }
        pila[top] = pila[n];
}

void LPP () {
        for( int i = 0; i < 8; i++){
                pila[i] = pila[i + 1];
        }
}
```

Las funciones de flanco tienen una variable declarada para guardar el valor del estado del ciclo anterior, de esta forma se puede saber en qué ciclo se produce el flanco.

```
//Flancos

void EU(int &state){
        if ( (pila[top] == 1) && (state == 0)  ){
                state =  1;
                pila[top] = 1;
      }
        else if( (pila[top] == 0) && (state == 1)){
                state = 0;
                 pila[top] = 0;
      }
         else if ( (pila[top] == 1) && (state == 1)  ){
                 state = 1;
                 pila[top] = 0;
      }
        else {
                 pila[top] = 0;
         }

}
```

```
void ED(int &state){
        if ( (pila[top] == 1) && (state == 0)  ){
                state =  1;
                pila[top] = 0;
       }
        else if( (pila[top] == 0) && (state == 1)){
                state = 0;
                pila[top] = 1;
       }
        else if ( (pila[top] == 0) && (state == 0)  ){
                state = 0;
                pila[top] = 0;
       }
        else {
                pila[top] = 0;
       }
}
```

Las funciones de aritmética realizan las operaciones asignadas según el primer nivel de la pila esté a '0' o '1'.

```
//Aritmetica

void sumaEN(int input, int &output) {
        if ( pila[top] == 1){
                output = input + output;
        }
        else{
        output = output;
        }
}


void restaEN(int input, int &output) {
        if ( pila[top] == 1){
        output = output - input;
        }
        else{
        output = output;
        }
}

void mulEN(int input, int &output) {
        if ( pila[top] == 1){
        output = output * input;
        }
        else{
        output = output;
        }
}

void divEN(int input, int &output) {
        if ( pila[top] == 1){
        output = output / input;
        }
        else{
        output = output;
        }
}
```

Las funciones de comparación comparan los valores enteros introducidos si el nivel superior de la pila está a '1'.

```
  //Comparaciones

void LDWE(int input1, int input2) {
        colocarPila ();
        if ( input1 == input2){
        pila[top] = 1;
        }
        else {
        pila[top] = 0;
     }
}

void LDWNE(int input1, int input2) {
        colocarPila ();
        if ( input1 != input2){
        pila[top] = 1;
        }
        else {
        pila[top] = 0;
        }
}

void LDWLT(int input1, int input2) {
        colocarPila ();
        if ( input1 < input2){
        pila[top] = 1;
        }
        else {
        pila[top] = 0;
        }
}

void LDWGT(int input1, int input2) {
        colocarPila ();
        if ( input1 > input2){
        pila[top] = 1;
        }
        else {
        pila[top] = 0;
        }
}

void LDWLTE(int input1, int input2) {
        colocarPila ();
        if ( input1 <= input2){
        pila[top] = 1;
        }
        else {
        pila[top] = 0;
        }
}
```

```
void LDWGTE(int input1, int input2) {
        colocarPila ();
        if ( input1 >= input2){
        pila[top] = 1;
        }
        else {
        pila[top] = 0;
     }
}
```

Funciones para los 3 tipos de temporizadores que se soportan, los cuales tienen dos variables para guardar su estado y su cuenta.

```
//Temporizadores

void TON(bool &timer,unsigned long &tiempo,unsigned long PT) {
        if (pila[top] == 0) {
                tiempo = 0;
                timer = 0;
        }
        else if (pila[top] == 1){
                if (tiempo == 0) {
                        tiempo = millis();
                        timer = 0;
                }
                else if (tiempo != 0){
                        if (millis() > (tiempo + PT)){
                                timer = 1;
                        }
                        else if (millis() < (tiempo + PT)) {
                                timer = 0;
                        }
                }
        }
}

void TOFF(bool &timer,unsigned long &tiempo,unsigned long PT) {
        if (pila[top] == 0) {
                if (tiempo == 0) {
                        tiempo = millis();
                        timer = 1;
                }
                else {
                        if (millis() > (tiempo + PT)) {
                                timer = 0;
                }
                else {
                        timer = 1;
                }
        }
        else if (pila[top] == 1){
                tiempo = 0;
                timer = 1;
        }
}
```

El temporizador TONR hace uso de otras tres variables adicionales para guardar la cuenta mientras el temporizador no está activo.

```
void  TONR(bool  &timer,  unsigned  long  &tiempo,  unsigned  long  PT,  int
&estadoOff, unsigned long &tiempoOff, unsigned long &tiempoOn) {
        if (pila[top] == 0) {
        if (tiempo >= PT ) {
                        timer = 1;
                }
        else if (estadoOff == 0 && tiempoOn != 0) {
                        tiempoOff = tiempo;
                        tiempo = 0;
                        estadoOff = 1;
                        timer = 0;
                }
                else{
                        timer = 0;
                }
        }
     else if(pila[top] == 1) {
                if(tiempoOn == 0) {
                        tiempoOn = millis();
                        timer= 0;
                }
                else if (tiempo >= PT ) {
                        timer = 1;
                }
                else if (estadoOff== 1) {
                tiempoOn = millis();
                estadoOff = 0;
                timer = 0;
                }
                else {
                tiempo = tiempoOff + millis() - tiempoOn;
                        timer = 0;
                        }
                }
        }
```

Los contadores leen los valores de los niveles de pila para aumentar o disminuir la variable de cuenta según el tipo de contador.

```
//Contadores

void CTU(bool &CT, int &cuenta, int pv){
        if (pila[top] == 1){
                cuenta = 0;
                CT = 0;
        }
        else if (cuenta >= pv){
                CT = 1;
     }
        else if( pila[top +1] == 1) {
                cuenta++;
                pila[top] = 0;
                if (cuenta >= pv){
                        CT = 1;
                }
        }
        else {
                CT = 0;
     }
}
```

```
void CTD(bool &CT, int &cuenta, int pv){
        if (pila[top] == 1){
                cuenta = pv;
                CT = 0;
        }
        else if( pila[top +1] == 1){
                cuenta--;
                pila[top] = 0;
                if (cuenta == 0){
                        CT = 1;
                }
        }
        else if (cuenta == 0){
                CT = 1;
     }
        else {
                CT = 0;
        }
}

void CTUD(bool &CT, int &cuenta, int pv){
        if (pila[top] == 1){
                cuenta = 0;
                CT = 0;
        }
        else if( pila[top +1] == 1) {
                cuenta++;
                CT = 0;
                if (cuenta >= pv){
                        CT = 1;
                }
        }
        else if (pila[top+2] == 1){
                cuenta--;
                CT = 0;
                if (cuenta >= pv){
                        CT = 1;
                }
        }
        else if (cuenta >= pv){
                CT = 1;
        }
        else {
                CT = 0;
        }
}
```

La función de watchdog cuenta el tiempo en milisegundos en el que se tarda en realizar un ciclo de programa e introduce dicho valor en la variable SMW22.

```
//SM y watchdog

void watchdog(){
        static int Tinit = 0;
        static int Tfin = 0;
        static int state = 0;
        if ( state == 0) {
                state = 1;
                Tinit = millis();
                SMW22 = Tinit -Tfin;
        }
        else if  ( state == 1){
                state = 0;
                Tfin = millis();
                SMW22= Tfin - Tinit;
     }
}
```

Las funciones de las marcas especiales se usan para asignarle el valor correspondiente en cada ciclo del autómata según el tipo de marca.

Los valores de las marcas SM0.4 y SM0.5 están asociados al tiempo en el que están on y off. Las funciones trabajan con variables estáticas ya que su valor de salida se incorpora a una variable ya creada.

```
void FSMcuatro(){
        static unsigned int Time = 0;
        static unsigned int internalstate = 0;
        static unsigned int state = 0;
        if (state == 0){
                if (internalstate == 0){
                        Time = millis();
                        SMcuatro= 0;
                        internalstate = 1;
        }
                if ((internalstate == 1) && ((millis() -Time) > 30000)){
                        internalstate = 0;
                        state = 1;
                }

        }
        if (state == 1){
                if (internalstate == 0){
                        Time = millis();
                        SMcuatro= 1;
                        internalstate = 1;
                }
                if ((internalstate == 1) && ((millis() -Time) > 30000)){
                        internalstate = 0;
                        state = 0;
        }
    }
}
```

```
void FSMcinco(){
	static unsigned int Time = 0;
	static unsigned int internalstate = 0;
	static unsigned int state = 0;
	if (state == 0){
		if (internalstate == 0){
			Time = millis();
			SMcinco= 0;
			internalstate = 1;
		}
		if ((internalstate == 1) && ((millis() -Time) > 500)){
			internalstate = 0;
			state = 1;
		}
	}
	if (state == 1){
		if (internalstate == 0){
			Time = millis();
			SMcinco = 1;
			internalstate = 1;
		}
		if ((internalstate == 1) && ((millis() -Time) > 500)){
			internalstate = 0;
			state = 0;
		}
	}
}
```

La función FSMuno() sirve para que la variable SMuno solo esté a ‘1’ el primer ciclo del programa.

```
void FSMuno(){
	static int state = 0;
	if (state == 0){
		SMuno = 1;
		state = 1;
	}
	else {
		SMuno = 0;
	}
}
```

Para mover el valor de una variable a otra utilizamos esta función.

```
//Mover variable

void MOVW(int input, int &output){
	if(pila[top] == 1){
		output = input;
	}
}
```

Esta función sirve para incorporar un nuevo valor a la pila, se copian todos los valores en el siguiente nivel de la pila, dejando el primer nivel libre para el nuevo valor.

```
//función para la pila

void colocarPila (){
        for(int i = 8; i > 0; i--){
                pila[i] = pila[i - 1];
        }
}
```

Función que sirve para comprobar con el ensamblador de Python si se puede hacer un bucle de control.

```
//FOR

bool FOR(){
        if(pila[top] == 1){
                return(1);
        }
        else{
        return(0);
        }
}
```

Programa ensamblador en Python

10.2 Programa ensamblador en Python

A continuación el programa en Python para compilar las instrucciones en AWL a lenguaje Arduino:

Se hace uso de diccionarios para los diferentes tipos de instrucciones, las definiciones son las siguientes, atendiendo a tipo de insntrucción:

```
#Instrucciones sin ningún argumento
def un():
    outFile.write(instr[0] + '();\n') #Se escribe en el fichero de salida
la misma instrucción

#Instrucciones con un sólo argumento
def dos():
    global argumento
    argumentos = []
    argumentos = linea[1]              #Se introducen el argumento en un
vector
    argumento = argumentos.split()  #Y si le separa del salto de linea \n
    argumentos  =  list(argumentos[0])#Nos  quedamos  con  la  primera
componente, la letra del argumento
    argumento = argumento[0]          #y volvemos a introducir el argumento
para su posterior procesado

    outFile.write(linea[0] + '(') #Escribimos la operacion correspondiente
    variableB[argumentos[0]]()  #y llamamos al diccionario de las variables
de bit
    outFile.write(');\n')

#instrucciones LDS, que requiere su definición por tener un argumento n
def LDS():
    n = []
    n = linea[1].split() #Separamos el salto de línea del numero
    outFile.write(linea[0] + '(' + n[0] + ');\n') #Escribimos la operación
de salida

#Instrucciones de flanco
def flanco():
    global flanco
    outFile.write(instr[0] + '(flanco' + str(flanco) + ');\n') # Escribimos
su operación
    flanco += 1 # y incrementamos el numero de flanco

#Instrucciones de asignación
def Asignar():
    global argumento
    argumentos = []
    argumentos = linea[1].split() #Separamos el salto de línea
    argumento = list(argumentos[0])#y cogemos la letra del argumento

    if argumento[0] == 'M':      #Si es una marca
        argumento.remove('M')   #Eliminamos la M de la lista
        argumento = ''.join(argumento) #y volvemos a formar un vector
        argumento = argumento.split('.') #Separamos los dos numeros X.Y
        X = int(argumento[0])//2 #Dividimos entre 2 el numero X para saber
que palabra es
        if  ((int(argumento[0]))  %  2)  ==  0  :  #Si  su  módulo  es  0,  es  un
numero par
```

```
            N =int(argumento[1]) # el numero de bit es el que indica Y
        else:
            N = 8 + int(argumento[1]) # Si no hay que sumarle 8
        outFile.write('asignarbit(MW['+ str(X) +'],' + str(N)+');\n')
    else:
        outFile.write('Asignar(') #Si no es una marca
        variableB[argumento[0]]() #Llamamos a la libreria de variables de
bit
        outFile.write(');\n')

#Instrucciones de Set y Reset
def SR():
    global argumento
    argumentos = []
    argumentos = linea[1].split(',') #Separamos los argumentos, la variable
del número
    argumento = argumentos[0] #Cogemos la variable para ver de que tipo es
    n = argumentos[1].split()

    if argumento[0] == 'Q': #Si es una Q
        outFile.write(linea[0] + '(') #Escribir en el archivo de salida lo
que corresponda
        variableB[argumento[0]]() #Llamamos al diccionario de variables de
bit
        outFile.write(');\n')

    elif argumento[0] == 'M': #Si es una marca, mismo procedimiento que con
las
        argumento = list(argumentos[0]) #instrucciones de asignar
        argumento.remove('M')
        argumento = ''.join(argumento)
        argumento = argumento.split('.') #MX.Y
        X = int(argumento[0])//2
        if ((int(argumento[0])) % 2) == 0 :
            N =int(argumento[1])
        else:
            N = 8 + int(argumento[1])

        if linea[0] == 'S': #Dependiendo de si es Set o Reset, escribimos
unas u otra
            outFile.write('Sbit(MW['+   str(X)   +'],'   +   str(N)+   ','+
n[0]+');\n')
        if linea[0] == 'R':
            outFile.write('Rbit(MW['+   str(X)   +'],'   +   str(N)+   ','+
n[0]+');\n')

    elif argumento[0] == 'C': #Si es un contador
        ct = list(argumento)
        ct.remove('C')      #Quitamos la C del identificador
        ct =  ''.join(ct)   #y unimos de nuevo el vector
        tipo,Cv,Cc,pv  =  contadores[ct]  #Con el identificador llamamos al
diccionario de contadores
        #y según que tipo de contador sea, escribimos lo correspondiente en
el archivo de salida
        if tipo == 'CTU':
            outFile.write('RCTUD('+Cv+','+Cc+');\n')
        elif tipo == 'CTUD':
            outFile.write('RCTUD('+Cv+','+Cc+');\n')
        elif tipo == 'CTD':
            outFile.write('RCTD('+Cv+','+Cc+','+pv+');\n')
```

```
    elif argumento[0] == 'T': #Si es un temporizador
        t = list(argumento)
        t.remove('T')   #Mismo procedimiento que con los contadores
        t =  ''.join(t)
        tipo  =  temporizadores[t]  #En  este  caso  solo  cogemos  el  tipo  de
temporizador
        #Una vez tengamos el tipo, cogemos las variables que nos interesan
del diccionario
        #y se las asignamos al archivo de salida
        if tipo[0] == 'TON':
            tipo,Tv,Tc,pv = temporizadores[t]
            outFile.write('RTON('+Tv+','+Tc+');\n')
        elif tipo[0] == 'TOFF':
            tipo,Tv,Tc,pv = temporizadores[t]
            outFile.write('RTOFF('+Tv+','+Tc+','+pv+');\n')
        elif tipo[0] == 'TONR':
            tipo,Tv,Tc,state,ton,toff,pv = temporizadores[t]

outFile.write('RTONR('+Tv+','+Tc+','+state+','+ton+','+toff+');\n')

#Instrucciones de comparación
def LDW():
    global argumento
    argumentos = []
    operando = list(linea[0]) #Convertimos la instruccion en una lista
    if len(operando) == 4: #si su longitud es 4, escribimos en el archivo
lo que
        if operando[3] == '=': # corresponda segun su tipo
            outFile.write('LDWE(')
        elif operando[3] == '<':
            outFile.write('LDWLT(')
        elif operando[3] == '>':
            outFile.write('LDWGT(')
    elif  len(operando)  ==  5:  #si  su  longitud  es  5,  realizamos  la  misma
operación
        if operando [3] == '<' and operando[4] == '>':
            outFile.write('LDWNE(')
        elif operando [3] == '<' and operando[4] == '=':
            outFile.write('LDWLTE(')
        elif operando [3] == '>' and operando[4] == '=':
            outFile.write('LDWGTW(')
    argumentos = linea[1].split(',') #Dividimos los dos argumentos
    argumento = argumentos[0] #Cogemos el primero
    try:
        variableW[argumento[0]]()  #Probamos  a  llamar  al  diccionario  de
palabras
    except KeyError: # Si no hay equivalncia, se supone que se ha escrito
un numero
        outFile.write(argumento) # que se escribe
    outFile.write(',')
    argumento = argumentos[1].split() #Separamos el salto de línea
    argumento = argumento[0] #Realizamos la misma operación
    try:
        variableW[argumento[0]]()
    except KeyError:
        outFile.write(argumento)
    outFile.write(');\n')
```

```
#Instrucciones de aritmética
def arit():
    global argumento
    argumentos = []
    operando = list(linea[0]) #Convertimos la instruccion en una lista
    if operando[0] == '+': #y escribimos la salida según su tipo
        outFile.write('sumaEN(')
    elif operando[0] == '-':
        outFile.write('restaEN(')
    elif operando[0] == '*':
        outFile.write('mulEn(')
    elif operando[0] == '/':
        outFile.write('divEn(')
    argumentos = linea[1].split(',') # Para los argumentos hacemos lo mismo
    argumento  =  argumentos[0]              #  que  con  las  operaciones  de
comparación
    try:
        variableW[argumento[0]]()
    except KeyError:
        outFile.write(argumento)
    outFile.write(',')
    argumento = argumentos[1].split()
    argumento = argumento[0]
    try:
        variableW[argumento[0]]()
    except KeyError:
        outFile.write(argumento)
    outFile.write(');\n')

#Instrucciones de contadores
def CT():
    argumentos = []
    outFile.write(linea[0]  + '(')          #Escribimos  el  tipo  de  contador
según el archivo de lectura
    argumentos  =  linea[1].split(',')       #Separamos  los  argumentos,  la
variable del numero de cuenta
    argumento  =  list(argumentos[0])       #Convertimos  la  variable  en  una
lista
    argumento.remove('C')                  #Borramos  la  C  para  quedarnos  con
el identificador
    argumento =  ''.join(argumento)      #Volvemos a unir el vector
    outFile.write('Cv'  +  argumento  +  ',Cc'  +  argumento)  #Escribimos  lo
correspondiente
    argumento = argumentos[1].split()   #Cogemos el numero de Pv
    outFile.write(',' + argumento[0] + ');\n') #y lo escribimos
```

```
#Intrucciones de temporización: TON y TOFF
def Tempo():
    argumentos = []
    outFile.write(linea[0] + '(')   #Escribimos el tipo de temporizador
    argumentos = linea[1].split(',') #Separamos los argumentos
    argumento = list(argumentos[0]) #Convertimos la variables en una lista
    argumento.remove('T')             #Borramos la que para quedarnos con el
identificador
    argumento =  ''.join(argumento) #Volvemos a unir el vector

    outFile.write('Tv' + argumento + ',Tc' + argumento) #Escribimos lo
correspondiente
    PT = int(argumento) #Usamos el identificador para calcular la
resolución
    if PT in (32,96): #En función del valor añadimos un numero de ceros al
final
        PT = argumentos[1].split()
        outFile.write(',' + PT[0] + ');\n')
    elif (33 <= PT <= 36) or (97 <= PT <= 100):
        PT = argumentos[1].split()
        outFile.write(',' + PT[0] + '0);\n')
    elif (37 <= PT <= 63) or (101 <= PT <= 255):
        PT = argumentos[1].split()
        outFile.write(',' + PT[0] + '00);\n')

#Instrucciones de temporización: TONR
def TONR():
    argumentos = [] #Mismo procedimiento que con los TON/TOFF
    outFile.write(linea[0] + '(')
    argumentos = linea[1].split(',')
    argumento = list(argumentos[0])
    argumento.remove('T')
    argumento =  ''.join(argumento)

    outFile.write('Tv' + argumento + ',Tc' + argumento)
    PT = int(argumento)
    if PT in (0,64):
        PT = argumentos[1].split()
        outFile.write(',' + PT[0] + ',')
    elif (1 <= PT <= 4) or (65 <= PT <= 68):
        PT = argumentos[1].split()
        outFile.write(',' + PT[0] + '0,')
    elif (5 <= PT <= 31) or (69 <= PT <= 95):
        PT = argumentos[1].split()
        outFile.write(',' + argumento[0] + '00,')

    outFile.write('TONRstate'+ argumento+',')
    outFile.write('TONRton'+ argumento+',')
    outFile.write('TONRtoff'+ argumento+');\n')
```

```
#Instrucciones de Mover
def MOVW():
    global argumento
    argumentos = []
    outFile.write('MOVW(') #Escribimos lo correspondiente en el archivo de
salida
    argumentos = linea[1].split(',') #Separamos los argumentos
    argumento = argumentos[0] #Nos quedamos con el primero
    try:
        variableW[argumento[0]]() #Llamamos al diccionario de variables de
palabra
    except KeyError: #Si no hay coincidencia suponemos que es un numero
        outFile.write(argumento)
    outFile.write(',')
    argumento = argumentos[1].split() #Separamos el salto de linea
    argumento = argumento[0] # Nos quedamos con la variable
    variableW[argumento[0]]() #Llamamos al diccionario de  palabras
    outFile.write(');\n')

#Instrucciones de control de flujo
def FOR():
    argumentos = []
    argumentos = linea[1].split(',') #Separamos los argumentos
    outFile.write('if(FOR() == 1){\n') #Escribimos lo correspondiente
    outFile.write(argumentos[0] + ' = ' + (argumentos[1])+';\n')
    outFile.write('for('+argumentos[0]+';'+argumentos[0]+'<=')   #Escribimos
un bucle for
    argumento = argumentos[2].split()
    outFile.write(argumento[0]+';'+argumentos[0]+'++){\n')

def NEXT():
    outFile.write('}}\n') #Cuando leamos NEXT, escribimos el fin del bucle
```

Dentro de la mayoría de funciones se llama a uno de los dos diccionarios de variables, uno para variables de bit y otro para variables de palabra. En el código que viene a continuación están definidas todas las posibles variables que pueden darse dentro del programa.

```
#variables booleanas

def XB():
    value = []
    value = linea[1].split() #Cogemos el argumento
    value = value[0].split('.') #Separamos el punto
    outFile.write('X' + value[1]) #Escribimos su equivalencia
def YB():
    value = []
    value = linea[1].split()
    value = value[0].split('.')
    outFile.write('Y' + value[1])

def MB():
    value = []
    value = argumento
    value = list(value)
    value.remove('M')
    value = ''.join(value)
    value = value.split('.') #MX.Y
    X = int(value[0])//2
    if ((int(value[0])) % 2) == 0 :
        N =int(value[1])
    else:
        N = 8 + int(value[1])
```

```
    outFile.write('extraerbit(MW['+ str(X) +'],' + str(N)+')')

def TB():
    value = []
    value = list(argumento) #Si queremos leer el estado de un temporizador
    value.remove('T')  #Quitamos la T, para quedarnos con el identificador
    value =  ''.join(value)
    outFile.write('Tv'+ value) #y escribimos lo correspondiente
def CB():
    value = []
    value = list(argumento)
    value.remove('C')
    value =  ''.join(value)
    outFile.write('Cv'+ value)

def SB():
    value = []              #Se escribe en el archivo de salida dependiendo
de lo
    value = list(argumento) #que se lea en el de entrada
    if value[4] == '0':
        outFile.write('SMcero')
    if value[4] == '1':
        outFile.write('SMuno')
    if value[4] == '4':
        outFile.write('SMcuatro')
    if value[4] == '5':
        outFile.write('SMcinco')

#Variables de palabra

def MW():
    value = []
    value = argumento.split('W')
    outFile.write('MW['+value[1]+']')

def TW():
    value = []
    value = list(argumento)
    value.remove('T')
    value =  ''.join(value)
    outFile.write('Tc'+ value)
def CW():
    value = []
    value = list(argumento)
    value.remove('C')
    value =  ''.join(value)
    outFile.write('Cc'+ value)

def SW():
    value = []
    value = list(argumento)
    outFile.write('SMW22')

def AW():
    value = []
    value = argumento
    outFile.write(str(value))
```

En la primera pasada el programa declara las variables necesarias para las funciones que las necesitan, que se encuentran en el diccionario de declaraciones, las siguientes líneas de código

muestran los casos posibles de variables que se han de declarar en caso de encontrar alguna instrucción de este tipo.

```
#Declaraciones de variables Arduino

def decFlanco():
    global flanco
    outFile.write('int flanco'+ str(flanco)+'= 0;\n')
    flanco += 1

def decCT():
    argumentos = []
    argumentos = linea[1].split(',') #Se separan los argumentos
    argumento = list(argumentos[0]) #Se convierte la variable en una lista
    argumento.remove('C') #Se borra la c
    argumento =  ''.join(argumento) #Se une el vector
    outFile.write('bool  Cv'  +  argumento  +  '  =  0;\n')  #Se  escrbie  lo
correspondiente
    outFile.write('int Cc' + argumento + '= 0;\n')

    pv  =  argumentos[1].split()  #Se  guarda  en  un  diccionario,  con  el
identificador para las operaciones de reset
    contadores[argumento] = (linea[0],'Cv'+argumento,'Cc'+argumento,pv[0])



def decT():
    argumentos = []
    argumentos = linea[1].split(',')
    argumento = list(argumentos[0])
    argumento.remove('T')
    argumento =  ''.join(argumento)
    outFile.write('bool Tv' + argumento + ' = 0;\n')
    outFile.write('unsigned long Tc' + argumento + '= 0;\n')

    pv = argumentos[1].split()
    temporizadores[argumento]                                              =
(linea[0],'Tv'+argumento,'Tc'+argumento,pv[0])


def decTONR():
    argumentos = []
    argumentos = linea[1].split(',')
    argumento = list(argumentos[0])
    argumento.remove('T')
    argumento =  ''.join(argumento)
    outFile.write('bool Tv' + argumento + ' = 0;\n')
    outFile.write('unsigned long Tc' + argumento + '= 0;\n')
    outFile.write('int TONRstate'+ argumento+'= 0;\n')
    outFile.write('unsigned long TONRton'+ argumento+'= 0;\n')
    outFile.write('unsigned long TONRtoff'+ argumento+'= 0;\n')

    pv = argumentos[1].split()
    temporizadores[argumento]                                              =
(linea[0],'Tv'+argumento,'Tc'+argumento,'TONRstate'+argumento,'TONRton'+arg
umento,'TONRtoff'+argumento,pv[0])
```

Todos los diccionarios declarados, con sus respectivas claves y valores para cuando llamemos a alguno de los diccionarios, salte a la clave si esta existiera.

```
#Diccionarios

operando = {'ALD':un,'OLD':un,'NOT':un,'LPS':un,'LRD':un,'LPP':un,
            'EU':flanco,'ED':flanco,
            'LD':dos,'LDN':dos,'A':dos,'AN':dos,'O':dos,'ON':dos,
            'S':SR, 'R':SR,
            'LDS':LDS,
            '=':Asignar,
            'LDW=':
LDW,'LDW<>':LDW,'LDW<':LDW,'LDW>':LDW,'LDW<=':LDW,'LDW>=':LDW,
            '+I':arit,'-I':arit,'*I':arit,'/I':arit,
            'CTU':CT, 'CTD':CT, 'CTUD':CT,
            'TON':Tempo,'TOFF':Tempo,
            'TONR':TONR,
            'MOVW':MOVW,
            'FOR' : FOR, 'NEXT':NEXT,
         }
variableB = {'I' : XB,
            'Q' : YB,
            'M' : MB,
            'T' : TB,
            'C' : CB,
            'S': SB,
            }

variableW = {'M' : MW,
            'T' : TW,
            'C' : CW,
            'S' : SW,
            'A':AW,
            }

declaracion = { 'EU':decFlanco, 'ED':decFlanco,
                'CTU':decCT, 'CTD':decCT, 'CTUD':decCT,
                'TON':decT, 'TOFF':decT,
                'TONR':decTONR,
                }
contadores = {}
temporizadores = {}
```

Aquí comienza el cuerpo del programa, que abre el archivo de lectura y crea uno de escritura donde se alojará el código saliente. El programa se realiza en dos pasadas del texto del archivo de lectura.

Una primera para las declaraciones de variables y una segunda para escribir las equivalencias del código en lenguaje Arduino.

```
#Programa principal

inFileName = 'ArduinoIn.txt'
outFileName = 'ArduinoOut.txt'
flanco = 0
i = 0
j = 0

with open(outFileName , 'w') as outFile: #Se crea archivo de salida
    with open(inFileName , 'r') as inFile: #Se abre archivo de lectura
        texto = []
        linea = []
        instr = []
        outFile.write('#include <PLC.h>\n\n')
        outFile.write('extern    int    MW[16];\n'+'extern    unsigned    int
SMW22;\n')
        outFile.write('extern bool SMcero;\n'+'extern bool SMuno;\n')
        outFile.write('extern bool SMcuatro;\n'+'extern bool SMcinco;\n')
        outFile.write('extern int Ain0;\n'+'extern int Ain1;\n')
        outFile.write('extern int Ain2;\n'+'extern int Ain3;\n')
        outFile.write('extern int Ain4;\n'+'extern int Ain5;\n')

        for line in inFile:      #Desde line 0 hasta que se acabe el texto
en f
            texto.append(line)   #va incorporando las lineas en el vector
texto
            linea = texto[j].split('//',1) #Se separan los comentarios del
código
            linea  =  linea[0].split('  ',  1)  #y  nuevamente  se  separa  las
instrucciones de los operandos
            j += 1

            instr = linea[0].split() #Se separa los saltos de linea si los
hubiera

            if len(instr) == 0: #Si hay una línea vacía
                continue        #se salta a la siguiente iteración
            try:
                declaracion[instr[0]]()  #Se  llama  al  diccionario  con  la
instrucción
            except   KeyError:         #En   caso   de   no   existir   ninguna
coincidencia, se ignora
                pass

        outFile.write('\nvoid  setup(){\n\nsetupTFG();\n\n}\n')   #Una   vez
acabada la primera pasada
        outFile.write('\nvoid loop(){\n\n')                        #se escriben
las funciones necesarias
        outFile.write('procesos();\n')
        inFile.seek(0) #Volvemos al inicio del texto
        flanco = 0     #Para usar los flancos declarados los ponemos a '0'

        for line in inFile:  #Realiza las mismas operaciones que la pasada
anterior
            texto.append(line)
            linea = texto[i].split('//', 1)
            linea = linea[0].split(' ',1)
            i += 1
```

```
            instr = linea[0].split() #Se separan los saltos de linea si los
hubiera

            if len(instr) == 0: #Si hay una línea vacía
                continue        #se salta a la siguiente iteración
            try:
                operando[instr[0]]() #Se llama al diccionario que incorpora
todas instrucciones posibles
            except KeyError:             #Si hay un error, se muestra por
pantalla
                print('Error en la instrucción: ' + line + '(línea:' +
str(i) + ')')

        outFile.write('\n}')

#Fin
```

Programa de prueba de funcionamiento

10.3 Programa de prueba de funcionamiento

10.3.1 ArduinoIn.txt

```
LD SM0.1 //En el primer ciclo del autómata
S Q0.0,1 //Se pone a 0 Bajar Plataforma
S Q0.2,3 // y se desactiva el Sensor de longitud, y los dos pistones
R M0.4,3 //Se pone a 1 la marca que indica...
LDN I0.1 //Si la mesa no esta bajada
ALD      // y es el primer ciclo del autómata
R Q0.0,1 //Se activa Bajar Plataforma
S Q0.1,1 // y se desactiva Subir Plataforma

LD I0.1  //Si la plataforma está en la posición 1
LDN M1.0 // y la pieza no es mala
ALD
A I0.0   // y la plataforma esta bajada
S Q0.0,1 //Se desactiva Bajar Plataforma
R Q0.1,1 // y se activa Subir Plataforma
S M2.0,1 //El Sensor de longitud se puede bajar

LD I0.1  //Cuando la plataforma llega abajo
EU       // con un flanco de subida
S Q0.1,1 //Se desactiva Subir Plataforma
R Q0.0,1 // y se activa Bajar Plataforma

LD I0.2  //Si la plataforma se encuentra arriba
LDN I0.4 // y el Sensor de longitud no está extendido
ALD
A M2.0   // y se puede bajar dicho sensor
R Q0.3,1 //Se activa el Sensor de longitud
R M2.0,1 // y se resetea la marca para bajar el sensor

LD I0.4  //Si el sensor está extendido
A I0.2   // y la plataforma está arriba
TON T101,5 //contamos medio segundo res T101 = 100 ms
LD T101  //Cuando se cuenten los 0'5 s
EU       // en un flanco de subida
MOVW  Ain0,MW3  //Movemos  el  valor  que  está  midiendo  el  sensor  a  la  var
entera MW3
S Q0.3,1 // y subimos el Sensor de longitud

LDW< 195,MW3 //Si el valor medido es mayor que 195
LDW> 215, MW3// y es menor que 215
ALD
S M0.4,1 //La pieza es apta

LD I0.2  //Si la plataforma está arriba
LDN M0.5 // y si la siguiente estación no tiene una pieza esperando
ALD
LDN I0.4 // y el Sensor de longitud no está extendido
```

```
ALD
A M0.4   // y la pieza es apta
TON T102,5//Contamos medio segundo
LD T102   // y cuando pasen los 0.5 s
R Q0.2,1  //Activamos el pistón que empuja la pieza
R Q0.4,1  // y el pistón que sujeta la pieza antes de caer a la siguiente
estación
S M0.5,1  // Activamos la marca de comunicación


LD I0.2  //Si la plataforma está arriba
A I0.3   // y el pistón está extendido
A M0.4  // y la pieza es apta
S Q0.2,1 //Retraemos el pistón
R Q0.0,1 // bajamos la plataforma
S Q0.1,1 // reseteamos Subir plataforma
R M0.4,1 // y reseteamos la marca que nos indica que la pieza es buena
R M1.0,1 // y resetamos la marca de pieza no apta


LD I0.2  //Si la plataforma está arriba
LDN I0.4 // y el Sensor de longitud esta retraído
ALD
LDN M0.4 // y la pieza no es buena
ALD
TON T104,20 //Contamos 2 segundos
LD T104  //Cuando pasen los 2 segundos
R Q0.0,1 //Bajamos la plataforma
S Q0.1,1 // y resteamos Subir plataforma
S M1.0,1 // e indicamos que la pieza mala


LD I0.1 //Si la plataforma esta abajo
A M1.0  // y la pieza es mala
R Q0.2,1// Activamos el pistón


LD I0.1 //Si la plataforma está abajo
A M1.0  // y la pieza es mala
A I0.3  // y el pistón está extraído
S Q0.2,1//Retraemos el pistón
R M1.0,1// Desactivamos la marca de pieza mala

LD I0.5 //Si apretamos el botón externo que indica que la siguiente esta
lista
R M0.5,1// reseteamos la marca de la comunicación
S Q0.4,1// y desactivamos el pistón de sujeción
```

10.3.2 ArduinoOut.txt

```
#include <TFG.h>

extern int MW[16];
extern unsigned int SMW22;
extern bool SMcero;
extern bool SMuno;
extern bool SMcuatro;
extern bool SMcinco;
extern int Ain0;
extern int Ain1;
extern int Ain2;
extern int Ain3;
extern int Ain4;
extern int Ain5;
int flanco0= 0;
bool Tv101 = 0;
unsigned long Tc101= 0;
int flanco1= 0;
bool Tv102 = 0;
unsigned long Tc102= 0;
bool Tv104 = 0;
unsigned long Tc104= 0;

void setup(){

setupTFG();

}

void loop(){

procesos();
LD(SMuno);
S(Y0,1);
S(Y2,3);
Rbit(MW[0],4,3);
LDN(X1);
ALD();
R(Y0,1);
S(Y1,1);

LD(X1);
LDN(extraerbit(MW[0],8));
ALD();
A(X0);
S(Y0,1);
R(Y1,1);
Sbit(MW[1],0,1);

LD(X1);
EU(flanco0);
S(Y1,1);
R(Y0,1);

LD(X2);
LDN(X4);
```

```
ALD();
A(extraerbit(MW[1],0));
R(Y3,1);
Rbit(MW[1],0,1);

LD(X4);
A(X2);
TON(Tv101,Tc101,500);
LD(Tv101);
EU(flanco1);
MOVW(Ain0,MW[3]);
S(Y3,1);

LDWLT(195,MW[3]);
LDWGT(215,MW[3]);
ALD();
Sbit(MW[0],4,1);

LD(X2);
LDN(extraerbit(MW[0],5));
ALD();
LDN(X4);
ALD();
A(extraerbit(MW[0],4));
TON(Tv102,Tc102,500);
LD(Tv102);
R(Y2,1);
R(Y4,1);
Sbit(MW[0],5,1);

LD(X2);
A(X3);
A(extraerbit(MW[0],4));
S(Y2,1);
R(Y0,1);
S(Y1,1);
Rbit(MW[0],4,1);
Rbit(MW[0],8,1);

LD(X2);
LDN(X4);
ALD();
LDN(extraerbit(MW[0],4));
ALD();
TON(Tv104,Tc104,2000);
LD(Tv104);
R(Y0,1);
S(Y1,1);
Sbit(MW[0],8,1);

LD(X1);
A(extraerbit(MW[0],8));
R(Y2,1);

LD(X1);
A(extraerbit(MW[0],8));
A(X3);
S(Y2,1);
Rbit(MW[0],8,1);
```

```
LD(X5);
Rbit(MW[0],5,1);
S(Y4,1);

}
```