



UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y
DISEÑO INDUSTRIAL

Grado en Ingeniería Electrónica y Automática Industrial

TRABAJO FIN DE GRADO

TÍTULO DEL TRABAJO

Autor

Cotutor (si lo hay): nombre y
apellidos

Departamento: departamento

Tutor: nombre y apellidos

Departamento: departamento

Ciudad, Mes año



UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y
DISEÑO INDUSTRIAL

Grado en Ingeniería Electrónica y Automática Industrial

TRABAJO FIN DE GRADO

TÍTULO DEL TRABAJO

Firma Autor

Firma Cotutor (si lo hay)

Firma Tutor

Título: título del trabajo

Autor: nombre del alumno

Tutor: nombre del tutor

Cotutor: nombre del cotutor

EL TRIBUNAL

Presidente:

Vocal:

Secretario:

Realizado el acto de defensa y lectura del Trabajo Fin de Grado el día de de ... en, en la Escuela Técnica Superior de Ingeniería y Diseño Industrial de la Universidad Politécnica de Madrid, acuerda otorgarle la CALIFICACIÓN de:

VOCAL

SECRETARIO

PRESIDENTE

Agradecimientos

Agradezco a

Resumen

Este proyecto se resume en.....

Palabras clave: palabraclave1, palabraclave2, palabraclave3.

Abstract

In this project...

Keywords: keyword1, keyword2, keyword3.

Índice general

Agradecimientos	v
Resumen	vii
Abstract	ix
1. Introducción	1
1.1. Motivación del proyecto	1
1.2. Objetivos	1
1.3. Materiales y software utilizados	1
1.4. Estructura del documento	3
2. Estado del arte	5
2.1. Lorem ipsum	5
3. Mecánica y soporte físico del proyecto	7
3.1. Visión general	7
3.2. Articulación uno. Giro en el eje Z	7
3.3. Articulaciones dos y tres. Posicionamiento en el plano sagital	7
3.4. Posicionamiento de sensores y actuadores	8
3.5. Estudio de la cadena cinemática completa	8
4. Electrónica involucrada en el proyecto	9
4.1. Procesamiento	9
4.2. Actuadores	9
4.3. Etapa de potencia	9
4.4. Sensores	9
5. Software involucrado en el proyecto	11
5.1. Librerías (directorio lib)	12
5.1.1. debug	12
5.1.2. rha_types	13
5.1.3. joint_handler	14
5.1.4. joint_rha	16
5.1.5. servo_rha	16
5.1.6. robot_rha	18
5.1.7. chuck_handler	18
5.2. SRC. Fichero de código principal	18
5.3. Interacción entre objetos, flujo de la información y de procedimientos	18

5.4. Test y verificación del software	18
5.5. Gestión de la complejidad y mantenibilidad:	19
6. Control	21
6.1. Control de velocidad para los servos G15 cube	21
7. Resultados y discusión	23
7.1. Resultados	23
7.2. Discusión	23
8. Gestión del proyecto	25
8.1. Ciclo de vida	25
8.2. Planificación	25
8.2.1. Planificación inicial	25
8.2.2. Planificación final	25
8.3. Presupuesto	25
8.3.1. Personal	26
8.3.2. Material	26
8.3.3. Resumen de costes	26
9. Conclusiones	27
9.1. Conclusión	27
9.2. Desarrollos futuros	27
Apéndice	28
A. Listado de piezas diseñadas	29
B. Montaje del prototipo	43
C. Reglas de codificación del Software	45
C.1. Aspectos generales	45
C.2. Ficheros de cabecera	45
C.2.1. Inclusión Múltiple	45
C.2.2. Orden de inclusión de ficheros	46
C.3. Ámbitos	46
C.3.1. Espacios de nombres	46
C.3.2. Variables Locales	47
C.4. Clases	47
C.4.1. Constructores y métodos de Inicialización	47
C.4.2. Estructuras o Clases	48
C.4.3. Control de Acceso	48
C.5. Tipos de datos	48
C.6. Nombres	48
C.6.1. Reglas generales	48
C.6.2. Nombre de los ficheros	48
C.6.3. Nombre de los directorios	49
C.6.4. Nombres para objetos	49

C.6.5. Nombres de variables	49
C.6.6. Nombres de atributos de clases	49
C.6.7. Nombres de miembros de estructuras	50
C.6.8. Nombres de funciones	50
C.6.9. Nombres de parámetros funciones	50
C.6.10. Espacios de nombres	50
C.6.11. Nombres de enumeraciones	50
C.6.12. Nombres de macros	51
C.7. Comentarios	51
C.7.1. Comentarios de ficheros	51
C.7.2. Comentarios de Clases	51
C.7.3. Comentarios de funciones	51
C.7.4. Comentarios y aclaraciones	52
C.7.5. TODO y notas	52
C.7.6. Código en desuso	52
C.8. Formato	52
C.8.1. Espacios y tabulaciones	52
C.8.2. Declaración y definición de funciones	53
C.8.3. Condicionales	53
C.8.4. Bucles	54
C.8.5. Valor de retorno de funciones y métodos	54
C.8.6. Formato para clases	54
C.8.7. Espacios de nombre	54
C.9. Espacios en blanco	54
C.9.1. Caso general	54
C.9.2. Bucles, condicionales y estructuras de control	55
C.9.3. Operadores	55
C.10. Espacio vertical	55

D. Documentación del software

57

Índice de figuras

3.1. Esquema de la cadena cinemática correspondiente a los GLOSARIO A GDL dos y tres	8
5.1. Estructuras de datos auxiliares	14
5.2. Esquema de ejecución de bucle de control de joint_handler	15
5.3. Lazo de control de la velocidad del servo. Cálculos realizados por el objeto <i>ServoRHA</i>	15
5.4. Atributos y métodos más relevantes del objeto <i>JointHandler</i>	16
5.5. Atributos y métodos más relevantes del objeto <i>JointRHA</i>	16
5.6. Lazo de control de la velocidad del servo. Cálculos realizados por el objeto <i>ServoRHA</i>	17
5.7. Atributos y métodos más relevantes del objeto <i>ServoRHA</i>	17
5.8. Análisis del código del proyecto	20

Índice de tablas

8.1. Costes del proyecto	26
A.1. Listado de piezas diseñadas de fabricación propia	30
A.2. Parámetros de las piezas para la estimación de peso	41

Capítulo 1

Introducción

En este capítulo no deben faltar los siguientes apartados:

1.1. Motivación del proyecto

1.2. Objetivos

El objetivo que este Trabajo de Fin de Grado persigue es el del diseño, construcción y control de brazo robótico. Este proyecto está enmarcado bajo el proyecto Robohealth, proyecto financiado por el Ministerio de Economía y Competitividad con el objetivo del diseño de sistemas robóticos y domóticos para entornos hospitalarios que mejoren el sistema de salud actual. (– **COMPLETAR** –<http://robohealth.es/>)

El brazo robótico llevará en su extremo una *tablet* que, mediante visión artificial será capaz de reconocer gestos del paciente para llevar a cabo diferentes acciones. El brazo tendrá que ubicar la *tablet* en todo momento a una distancia y posición respecto del paciente que posibilite su correcto funcionamiento.

El sistema estará en constante contacto con diferentes usuarios por lo que la seguridad de los mismos será prioritario, incluso desde la fase de diseño del prototipo.

Otra característica importante será el coste económico del prototipo, se busca desarrollar una solución de bajo coste que aumente la viabilidad de su implementación a gran escala.

Para alcanzar estos objetivos se probarán diferentes estructuras y materiales que maximicen los recursos estructurales del prototipo. Una vez se tenga una estructura básica se procederá a añadir los actuadores y sensores para posteriormente implementar el software que lo controle.

1.3. Materiales y software utilizados

Para la realización de las diferentes etapas del proyecto son necesarios los siguientes materiales. En este apartado se listan sin entrar en mucho detalle ya que se verán en capítulos posteriores.

Elementos para el montaje Mecánico:

- Barras de aluminio de sección cuadrada de 1/2"x1m (lado de la sección x longitud).
- Rodamientos 4x13?
- Rodamientos 3x13?
- Barras de acero cilíndricas de 4mm de diámetro
- Barras de acero cilíndricas de 3mm de diámetro
- Piezas impresas en impresora 3D. (Listado detallado en el Anexo A.)
- Piezas de metacrilato cortadas con láser. (Listado detallado en el Anexo A.)
- Tornillería: tornillos y tuercas
- Hilo de kevlar
- 2x - Poleas (LAS NEGRAS)

Se puede ver un análisis más detallado en el Anexo B, donde se detalla el montaje de las diferentes piezas que componen el brazo robótico.

Electrónica:

- 1x - Placa Arduino Uno
- 1x - Cytron G15 shield
- 3x - Servos G15 cube (Cytron)
- 2x - Potenciómetros

Software externo utilizado:

- Autodesk Inventor 2016: Utilizado para el diseño y ensamblaje de la parte física del proyecto.
- Atom con PlatformIO instalado: Utilizado para el desarrollo y verificación del software del proyecto.
- Lizard: Se trata de un software de análisis de la complejidad del código. Se compone de una serie de scripts en python que, al ser ejecutados devuelven un fichero con información referente a los ficheros de código sobre los que se invoca.
- Cloc: Herramienta para realizar el conteo de líneas en los ficheros de código. Permite diferenciar las líneas en blanco, comentarios, líneas de código, etc.
- cpplint: Análisis del cumplimiento de las reglas de codificación en el software. Es una herramienta desarrollada en python por Google para asegurar que los proyectos cumplen con sus reglas de codificación, que también se han seguido en este proyecto. Se pueden ver los aspectos más relevantes de las reglas de codificación en el Anexo C.
- doxygen:

1.4. Estructura del documento

A continuación y para facilitar la lectura del documento, se detalla el contenido de cada capítulo.

- En el capítulo 1 se realiza una introducción. – **COMPLETAR** –
- En el capítulo 2 se hace un repaso del actual estado del arte recogiendo algunas ideas que han inspirado el proyecto que aquí se describe.
- En el capítulo 3 se hace una descripción detallada del proceso de diseño de la parte mecánica del proyecto. Se resaltan algunos aspectos importantes así como consideraciones para su montaje.
- En el capítulo 4 se habla sobre la electrónica involucrada en el proyecto. En ella se describe y justifica la utilización de sensores y actuadores, así como la placa utilizada y la etapa de potencia.
- En el capítulo 5 se expone de forma extensiva el software desarrollado.
- El capítulo 6 expone de forma detallada los distintos aspectos de diseño y desarrollo del control del brazo.
- En el capítulo 7 se hace un repaso sobre los resultados generales a los que se ha llegado.
- En el capítulo 8 se hace un análisis de la gestión del proyecto.
- Finalmente, en el capítulo 9 se repasan las conclusiones y desarrollos futuros.

Como contenido adicional, al final del documento también se tienen los siguientes anexos:

- En el Anexo A se listan las piezas impresas que se requieren para montar el prototipo así como algunas características importantes.
- En el Anexo C se concretan las reglas de codificación más relevantes que se han aplicado en el desarrollo del código.
- En el Anexo D se adjunta la documentación del software generada con la herramienta *doxygen*

Además es interesante repasar los términos que se incluyen en el Glosario y que aparecerán referenciados a lo largo del documento.

Capítulo 2

Estado del arte

En este capítulo se hace un repaso de diferentes modelos de robots comerciales existentes en la actualidad relacionados con el campo asistencial robot-paciente. En el caso que nos ocupa no hay constancia de robots diseñados específicamente para la función que este proyecto propone, por lo que el estudio del estado del arte se ha centrado más en la situación y entorno que ocupará el robot más que en su utilización. Cualquiera de los robots mencionados a continuación podría adaptarse con poco trabajo para realizar la tarea de posicionar y orientar una *tablet* ante el usuario.

Las características comunes de estos dispositivos, entre otras son:

- Están diseñados para una interacción directa entre humano y robot. Concretamente estarán en contacto constante y directo con pacientes o personas con diversidad funcional. Esto deberá verse reflejado en su diseño y medidas de seguridad aplicadas.
- Son de dimensiones relativamente pequeñas, normalmente no muy superiores al alcance de las extremidades de un ser humano.
- Tienen interfaces sencillas y limpias para que usuarios poco experimentados puedan utilizarlos aprovechando al máximo las ventajas que ofrecen sin necesidad de pasar por una curva de aprendizaje que desmotive su uso.

2.1. Lorem ipsum

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas ornare erat nisl, a laoreet purus pellentesque id. Duis laoreet ipsum posuere est hendrerit, quis ornare nisi iaculis. Quisque imperdiet gravida egestas. Maecenas in mauris felis. Quisque quis imperdiet enim. Curabitur dignissim eget nisi lobortis placerat. Donec et magna rutrum, tempor magna a, consectetur tortor. Donec faucibus sodales sem, eu iaculis leo eleifend id. Nam semper lectus nisl, sed molestie erat pharetra quis. Quisque vestibulum metus elit, id interdum ligula dignissim a.

Praesent eu velit ac lectus tristique tristique vitae et tellus. Mauris dignissim feugiat orci, vitae luctus dolor finibus ut. Ut congue bibendum lectus, vitae congue ligula. Donec commodo, lacus ac iaculis scelerisque, nunc purus finibus diam, at lacinia sem justo non quam. Aenean tempor urna vitae quam pretium porta. Sed in lacinia ipsum. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per

inceptos himenaeos. Integer ut tristique est. Nam vitae interdum ligula, ac sodales dolor. Nulla mollis bibendum urna, sit amet interdum est aliquet at. Sed sagittis mi vel tellus posuere, eu rutrum arcu tristique.

Vestibulum aliquet orci pharetra justo auctor, pharetra viverra felis finibus. Ut ac gravida quam. Donec egestas turpis nisi, nec elementum orci feugiat at. In hac habitasse platea dictumst. Praesent mollis sem in felis feugiat, dapibus finibus metus scelerisque. Aliquam ultricies ante quis nibh laoreet, ac aliquam justo maximus. Etiam rhoncus pharetra imperdiet.

Nullam at libero quis augue tristique luctus eget placerat lorem. Donec pretium, dui scelerisque dapibus feugiat, ex lacus auctor ipsum, in ultricies odio justo in eros. Proin sodales velit non accumsan tempor. Mauris at consectetur est. Donec aliquam porttitor tortor, id malesuada nunc euismod vel. Ut id ullamcorper turpis, nec feugiat sapien. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi aliquam tempus tortor, et gravida lectus iaculis non. Interdum et malesuada fames ac ante ipsum primis in faucibus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Integer non maximus felis. Nullam ac tempor augue. Vestibulum in efficitur mauris. Sed in nulla ultrices, pharetra ligula et, blandit nunc. Quisque dictum magna eget diam maximus, ac pulvinar nisi tempor. Pellentesque quis feugiat elit.

Integer euismod in urna id placerat. Etiam urna elit, tempor et turpis venenatis, volutpat viverra lacus. In luctus arcu sit amet lectus rutrum, id ultricies mi pellen-tesque. Nulla bibendum, orci in elementum aliquam, mi purus sollicitudin orci, quis ornare nulla arcu placerat urna. Integer consequat, risus ac elementum pellen-tesque, nulla est lobortis justo, sed mattis nibh ligula nec velit. Integer sem mauris, luctus vitae venenatis a, tincidunt egestas purus. In et lectus semper, dapibus massa sed, ultrices nisi. Ut sit amet dolor porta, accumsan lectus ut, semper tellus. Praesent velit odio, facilisis quis sodales vel, molestie at risus. In sollicitudin mauris risus, ullamcorper ullamcorper ligula commodo sed. Ut libero tortor, rhoncus ut sagittis quis, fringilla nec nunc. Ut efficitur nisi id leo feugiat ultrices. Class aptent taci-ti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Sed at malesuada arcu.

Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridicu-lus mus. Sed consectetur, justo nec scelerisque accumsan, leo erat dictum odio, id feugiat nibh felis vel ipsum. Duis urna ante, commodo vitae neque varius, congue egestas turpis. Donec condimentum ullamcorper dapibus. Nulla sed sapien eu di-am commodo finibus. Nulla fringilla lectus vitae augue rutrum volutpat. Nulla in accumsan orci. Suspendisse eget diam massa.

Capítulo 3

Mecánica y soporte físico del proyecto

3.1. Visión general

3.2. Articulación uno. Giro en el eje Z

Junto con las articulaciones dos y tres, descritas en la Sección 3.3 están consideradas como los grados de libertad que gestionan la posición del extremo del robot en un espacio tridimensional. En adelante se las podrá denominar también "grados de libertad de posición".

Esta articulación está actuada por un *Servo G15 Cube* (descrito en la Sección 1.3. El movimiento de dicho servo se transmite a la articulación a través de un juego de ruedas que, solidarias a la parte superior (parte móvil) de la articulación y por rozamiento, transmiten el movimiento hasta la pista inferior (parte fija a la base del robot).

De esta forma aseguramos que el usuario, en cualquier momento podrá desplazar el robot superando el rozamiento de esta cadena de transmisión anulando, en caso de estar en proceso, el movimiento que pueda estar efectuando el *servo*.

3.3. Articulaciones dos y tres. Posicionamiento en el plano sagital

Estas dos articulaciones son las encargadas de posicionar el extremo en el plano sagital del robot.

Están formadas por dos mecanismos de cuatro barras acoplados en serie. Tienen la particularidad de que las barras son iguales dos a dos, de forma que las barras se mantienen siempre en paralelo. Esta particularidad asegura que el extremo se mantenga siempre perpendicular al plano del suelo, de forma que se desacopla la orientación del extremo de la posición del mismo.

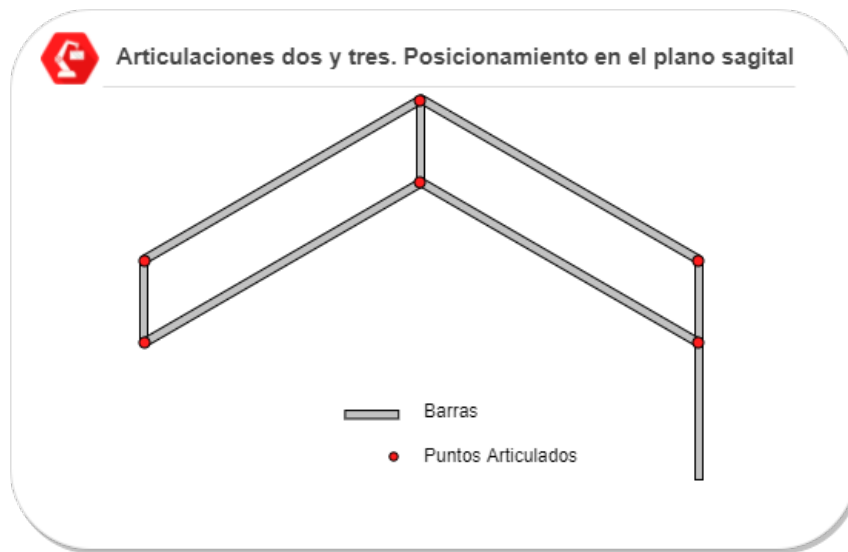


Figura 3.1: Esquema de la cadena cinemática correspondiente a los **GLOSARIO A GDL** dos y tres

3.4. Posicionamiento de sensores y actuadores

3.5. Estudio de la cadena cinemática completa

Capítulo 4

Electrónica involucrada en el proyecto

4.1. Procesamiento

4.2. Actuadores

Para los grados de libertad de posición del prototipo se ha optado por utilizar los *smart servos* G15 Cube de la marca Cytron.

4.3. Etapa de potencia

4.4. Sensores

Capítulo 5

Software involucrado en el proyecto

En este capítulo se hace una descripción detallada del *software* involucrado en este proyecto. Primeramente se hace un repaso de la estructura de directorios, que es característica y viene previamente marcada por el software utilizado. Una vez explicado se pasará a describir las diferentes librerías que se han desarrollado (sección 5.1) y del directorio de fuentes principal SRC (sección 5.2). Se deja para una sección posterior la interacción e integración de estos objetos así como una explicación más detallada del flujo de la información, funcionamiento del sistema como ejemplos de uso (sección 5.3). Finalmente se exponen los test realizados sobre el software (sección 5.4) así como la gestión de la complejidad (sección 5.5) del proyecto.

Para el desarrollo y test del software se ha utilizado el editor *Atom* ampliando su funcionalidad con el paquete *PlatformIO*, que expande las capacidades del editor base para permitir trabajar con diferentes placas – **COMPLETAR** –, entre ellas las de la gama de *Arduino*.

La elección de esta herramienta conlleva un formato en el árbol de directorios en los que se separa el código debido a la forma que tiene de compilar y linkar los diferentes ficheros. De esta manera y para mantener el orden los ficheros de código se separan en distintos directorios:

- lib: directorio donde se introducen, en carpetas, las librerías que se van a utilizar.
- src: directorio donde se introduce el fichero o ficheros de código principales.
- test: directorio donde se introducen los ficheros donde se codifican los test. Estos irán metidos dentro de directorios con el nombre de cada test.

Concretamente, en el caso de este proyecto, queda el siguiente árbol de directorios en el cual se han expandido hasta el nivel de ficheros en algunos casos de modo que sirvan de ejemplo:

```
— Sw
|— lib
|  |— debug
```

```

| | | + debug.cpp
| | | + debug.h
| |-- joint_handler
| | | + joint_handler.cpp
| | | + joint_handler.h
| |-- joint_rha
| |-- rha_types
| |-- robot_rha
| |-- servo_rha
| |-- chuck_handler
| |-- readme.txt
|-- src
| |-- main.cpp
| |-- utilities.cpp
| |-- utilities.h
|-- test
| |-- a_test_servo_rha
| | | + test_servo_rha.cpp
| |-- b_test_servo_realest_joint_rha
| |-- c_test_joint_handler_mock
|-- platformio.ini
|-- code_analysis
|-- makeAnalysis.sh

```

En los siguientes apartados se hará un análisis detallado del contenido de estos directorios. Se desarrollan primero los ficheros con información auxiliar y posteriormente en una jerarquía desde más afuera *<— ¿?¿W T F?¿?* detallando luego los componentes internos.

5.1. Librerías (directorio lib)

5.1.1. debug

Dentro de este directorio se encuentra el fichero `debug.h` donde se han definido macros para *debug* de los diferentes espacios. A continuación se muestra un ejemplo de como se codifican dichas macros para hacer un seguimiento de la clase *servo_rha* (Ver apartado 5.1.5).

```

#ifdef DEBUG_SERVO_RHA
#define DebugSerialSRHALn(a) { Serial.print("[DC] __ServoRHA::"); Serial.println(a); }
#define DebugSerialSRHALn2(a, b) { Serial.print("[DC] __ServoRHA::"); Serial.print(a); Serial.println(b); }
#define DebugSerialSRHALn4(a, b, c, d) { Serial.print("[DC] __ServoRHA::"); Serial.print(a); Serial.print(b); Serial.print(c); Serial.println(d); }
#else
#define DebugSerialSRHALn(a)
#define DebugSerialSRHALn2(a, b)
#define DebugSerialSRHALn4(a, b, c, d)
#endif

```

Como se puede apreciar estas macros utilizan la comunicación serial de *Arduino*, concretamente la función *print* y *println* de la librería *Serial* para mostrar los mensajes en un formato determinado. Además se definen de tal forma que se pueden activar o desactivar (en este mismo fichero *debug.h*) de forma que se enviarán o no los mensajes de *debug*. A través de este tipo de mensajes se puede hacer un seguimiento de la ejecución de los diferentes métodos o funciones de la librería afectada para ubicar fallos en los mismos.

Para activar la opción de *debug* bastará con descomentar la línea correspondiente:

```
// #define DEBUG.SERVO.RHA
// #define DEBUG.TEST.SERVO.RHA MOCK
#define DEBUG.TEST.SERVO.RHA.REAL
// #define DEBUG.CYTRON.G15.SERVO
// #define DEBUG.TEST.CYTRON.G15.SERVO
```

5.1.2. rha_types

Dentro de este directorio se encuentra el fichero *rha_types.h* donde se definen algunos tipos de datos y estructuras que se usan en el proyecto.

Estas estructuras de datos se listan a continuación, pudiendo ver sus respectivos diagramas de clases en la figura 5.1:

- *SpeedGoal* condensa en un solo objeto toda la información necesaria para codificar un objetivo de velocidad.
- *Regulator* encapsula el funcionamiento de un *Regulador PID* estandar. Guarda los valores de las constantes así como de la integral del error para luego, pasándole error, derivada del error e integral del error en un intervalo poder devolver la salida del regulador.
- *Timer* codifica un temporizador (en milisegundos) de forma que se le podrá preguntar al objeto si el tiempo ya ha pasado, bloqueando o no la ejecución del programa hasta el final del tiempo.
- *TimerMicroseconds* hereda las características del objeto *Timer* funcionando en microsegundos.

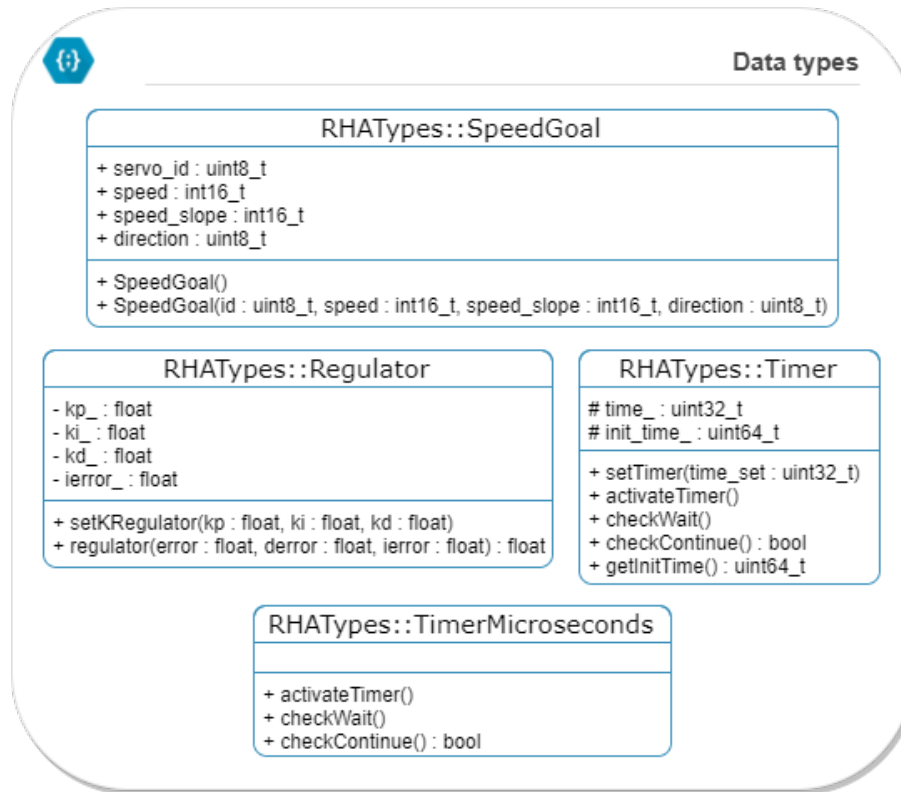


Figura 5.1: Estructuras de datos auxiliares

Se puede consultar información de más bajo nivel referente a estos tipos de datos en el Anexo D sección – **COMPLETAR** –.

5.1.3. joint_handler

La librería *joint_handler* se hace cargo de generar un objeto de la clase *JointHandler* que será en encargado de gestionar la sincronización de todas las articulaciones. Este objeto es propietario de las articulaciones e implementa un método que cíclicamente sincroniza el funcionamiento de todas las articulaciones. En el objeto *JointHandler* se implementa a su vez la comunicación con los *servos*, es decir, el encapsulamiento de los paquetes de datos con la información genérica de forma que la información que se envía cumpla con el protocolo de comunicación que comparten los *servos*. Además implementa las funciones que gestionan el envío y recepción de dichos paquetes de datos.

El bucle de control implementado se encarga de ir llamando una a una a todas las articulaciones para que hagan las siguientes operaciones (se puede ver representado el funcionamiento de dicho ciclo en la Figura 5.2):

1. Asegurar que cada articulación actualice la información propia, compuesta por la posición recibida de la realimentación así como toda la información proveniente del servo (datos de posición, velocidad, par soportado y dirección en que se aplican, voltaje y temperatura).
2. Llamar a cada articulación para que se realicen los cálculos correspondientes del *torque* que se enviará calculado a partir del error entre la velocidad real

y deseada. Este valor queda guardado en cada servo para ser posteriormente empaquetado.

3. Invocar a cada servo para que se adhieran al paquete de información que se va a enviar.
4. Se empaqueta la información a enviar a los diferentes *servos* en un mismo paquete, añadiendo posteriormente el correspondiente encabezado así como el comprobante de que el paquete se ha recibido correctamente (checksum). Una vez preparado el paquete se envía por el puerto serie.

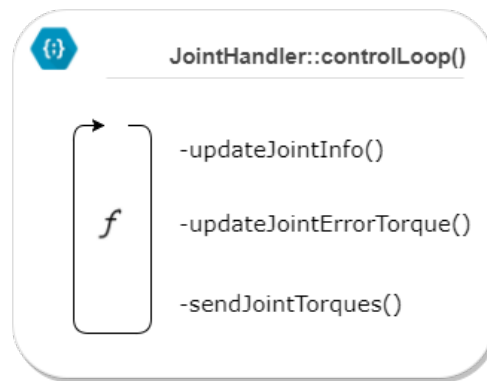


Figura 5.2: Esquema de ejecución de bucle de control de `joint_handler`

Como se puede intuir es en este objeto donde realmente se implementa el lazo de control de velocidad para todos los *servos* conectados al bus. Esta serie de operaciones descrita anteriormente constituye, de forma discretizada, el lazo de control representado en la Figura 5.3 para cada servo y asegura su correcto funcionamiento y sincronismo.

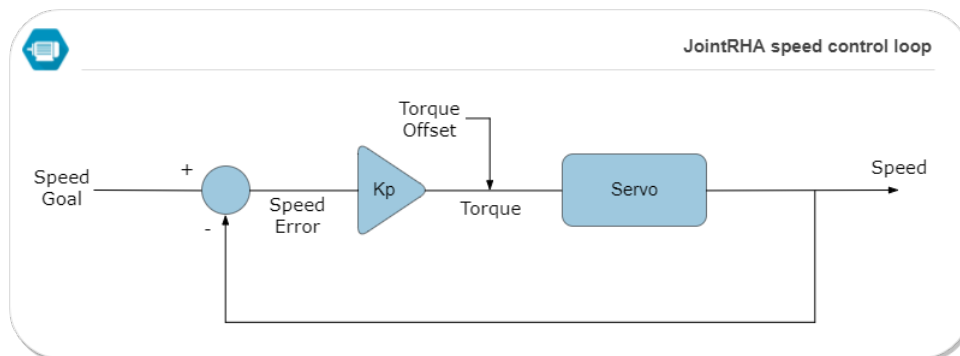


Figura 5.3: Lazo de control de la velocidad del servo. Cálculos realizados por el objeto `ServoRHA`.

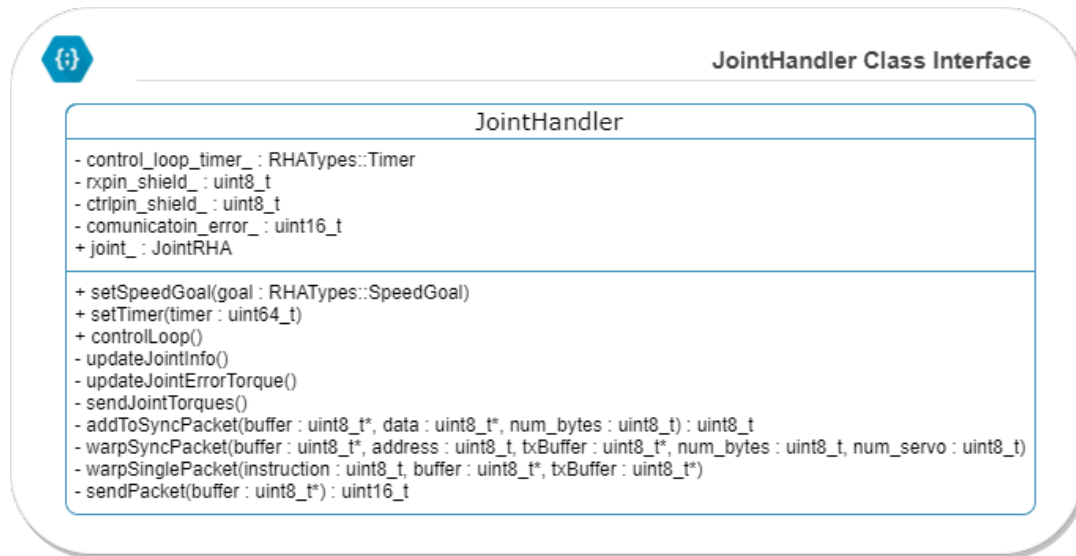


Figura 5.4: Atributos y métodos más relevantes del objeto *JointHandler*

Se puede consultar información de más bajo nivel referente al objeto *JointHandler* así como a sus atributos y métodos en el Anexo D sección – **COMPLETAR** –.

5.1.4. joint_rha

La librería *joint_rha* implementa un objeto de tipo *JointRHA* que aúna en un mismo objeto las lecturas y capacidades del objeto *ServoRHA* (explicado en la sección 5.1.5) junto con la realimentación de posición de la articulación en cuestión.

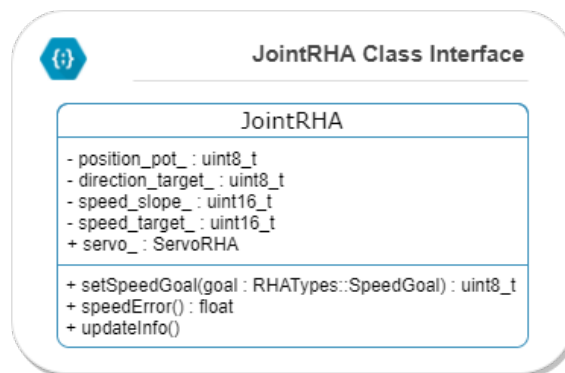


Figura 5.5: Atributos y métodos más relevantes del objeto *JointRHA*

Se puede consultar información de más bajo nivel referente al objeto *JointRHA* así como a sus atributos y métodos en el Anexo D sección – **COMPLETAR** –.

5.1.5. servo_rha

La librería *servo_rha* implementa un objeto del tipo *ServoRHA* que está encargado de gestionar toda la información referente al servo. Se encarga de encapsular la información específica a un servo en paquetes de datos a petición del objeto *JointHandler* que luego serán procesados por el mismo previo a ser enviados a través

del bus. Estos paquetes se forman a partir de la información contenida en el objeto *ServoRHA*, que además de formar los paquetes a enviar almacena la información referente al servo que se recibe de los mismos.

Gestiona además una parte importante referente al lazo de control de velocidad del servo vista en el apartado 5.1.3. El objeto *ServoRHA* contiene los datos del regulador utilizado así como el offset a aplicar. De esta forma, recibiendo el error realiza las operaciones para calcular y empaquetar el *torque* deseado. En la Figura 5.6 se representa, recuadrado, la parte correspondiente del lazo de control de velocidad (representado anteriormente en la Figura 5.3) que efectúa el objeto en cuestión.

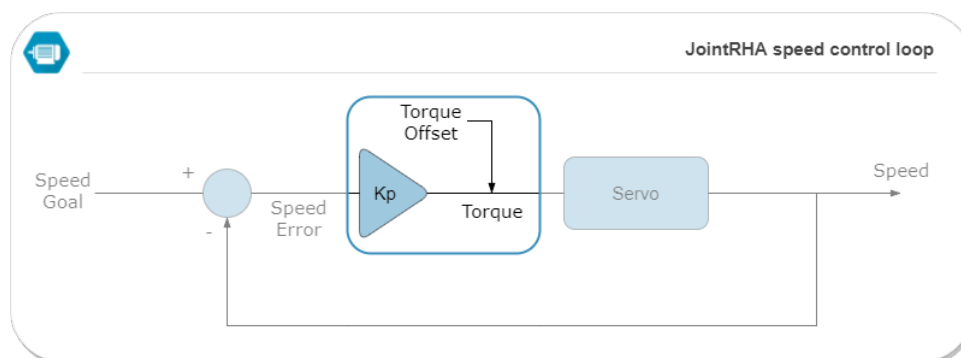


Figura 5.6: Lazo de control de la velocidad del servo. Cálculos realizados por el objeto *ServoRHA*.

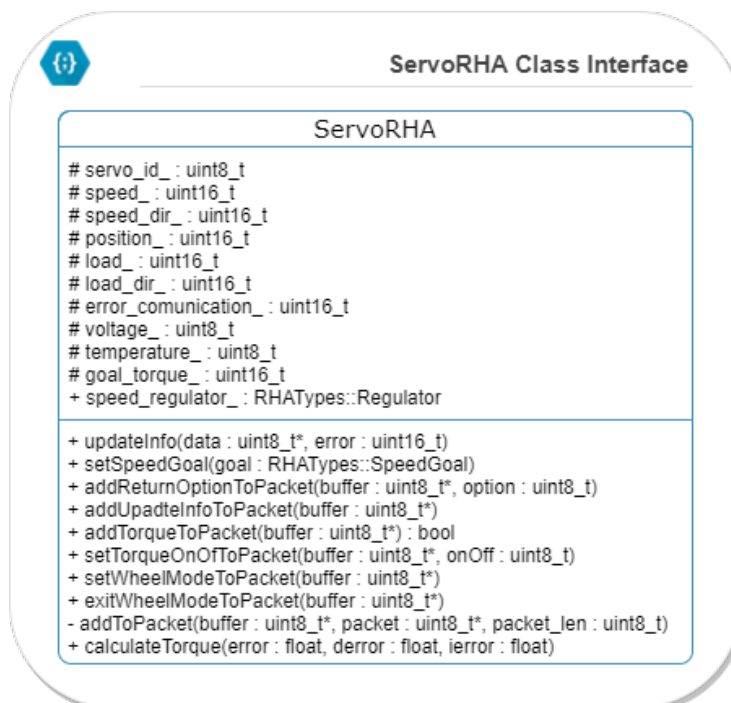


Figura 5.7: Atributos y métodos más relevantes del objeto *ServoRHA*

Se puede consultar información de más bajo nivel referente al objeto *ServoRHA* así como a sus atributos y métodos en el Anexo D sección – **COMPLETAR** –.

5.1.6. robot_rha

En la librería *robot_rha* se implementa el objeto de tipo *RobotRHA* encargado de coordinar el funcionamiento del robot. Implementa el ciclo de más alto nivel donde se actualizan los objetivos ya sean de posición o velocidad para llamar a las funciones correspondientes que lo traducen a valores articulares que luego se ejecutan.

Tiene diferentes modos de funcionamiento según de donde provengan los comandos a seguir:

- Control a través de un *Nunchuk* de la consola *Wii*. Se controla la velocidad del robot en sus diferentes ejes a través del joystick y los botones del mando.

5.1.7. chuck_handler

La librería *chuck_handler* codifica el objeto de tipo *ChuckHandler* que se encarga de implementar todo lo necesario para realizar lecturas del mando *Nunchuck* de la *Wii* y devolver comandos de velocidad con un periodo establecido. Se llamará al método correspondiente de forma continuada devolviendo este los valores de velocidad cuando se cumpla el tiempo mínimo establecido.

5.2. SRC. Fichero de código principal

5.3. Interacción entre objetos, flujo de la información y de procedimientos

5.4. Test y verificación del software

Como parte del proyecto se han desarrollado una serie de test para verificar el correcto funcionamiento de las diferentes librerías. El *testing* del *software* se ha utilizado como herramienta implémentándose solo en aquellos casos en que facilita la verificación y comprobación del correcto funcionamiento del código implementado. El *testing* del código no forma parte del núcleo del proyecto por lo que no se han establecido porcentajes mínimos de cobertura (cantidad de código que se está evaluando a través de las pruebas) ni objetivos mínimos. Los test son puramente funcionales desarrollándose los necesarios y no forzando el desarrollo de los distintos niveles de *test*: test unitarios, test de integración y test de sistemas. **ALGUNA CITA QUE HABLE DE LOS NIVELES DE TEST**

Para el desarrollo y ejecución de dichos test se ha utilizado la funcionalidad de test que viene integrada en *PlatformIO: PlatformIO Test*. Esta herramienta, permite definir una serie de test que pueden ser ejecutados en la propia placa. De esta forma se puede automatizar el proceso de test.

Una completa definición de test (desde test unitarios hasta test de integración) permite controlar de forma continuada el correcto funcionamiento del sistema frente a modificaciones en el código. De forma genérica estos test se dividen en:

- test_cytrong_g15_servo
- test_servo_mock

- test_servo_real

El el directorio SW/test se encuentran definidos los diferentes test que se realizan. Cada fichero de test, destinado a testear de forma parcial o completa una librería, tiene diferentes test definidos en formato de función para los diferentes métodos contenidos en la librería.

Para testear un método o grupo de métodos se define una función de test en la que se define la ejecución que se va a realizar, con las entradas predefinidas de forma que se puede comprobar como ciertas salidas o parámetros internos satisfacen las necesidades impuestas. Para la definición de estas condiciones así como de los test se sigue el formato propuesto desde *PlatformIO Test* y la API que adjuntan.

5.5. Gestión de la complejidad y mantenibilidad:

Para controlar el desarrollo del proyecto de forma paralela en todas sus partes asegurando así un control de la complejidad y mantenibilidad se han ido controlando diferentes métricas referentes al software del proyecto.

Además de dichas métricas se han ido haciendo revisiones periódicas del cumplimiento las reglas de codificación en el software (ver Anexo C). Para ello se ha utilizado un *script cplint* que automatiza la revisión del código.

Para obtener la información referente a la complejidad y desarrollo del software se han utilizado dos herramientas (*lizard* y *Cloc*) junto con una serie de *scripts* que se han desarrollado para automatizar la obtención y visualización de la información. Las métricas que se han obtenido y valorado son:

1. Número de líneas de código.
2. Número de líneas de comentarios.
3. Número de líneas de mensajes de *debug*.
4. Porcentaje de líneas de comentarios (media de todos los ficheros así como máximos y mínimos).
5. Porcentaje de líneas de mensajes de *debug* (media de todos los ficheros así como máximos y mínimos).
6. Número de ficheros.
7. Número de funciones.
8. Media de métodos por fichero.
9. Complejidad ciclomática (media entre todos los métodos así como valores extremos).

Las métricas de la uno a la cinco de la lista anterior permiten controlar un desarrollo paralelo y equilibrado entre el código así como la documentación del mismo. De igual forma, aunque menos importante, permiten ver el desarrollo paralelo de métodos de *debugging*. Este se considera menos importante ya que en su mayoría se

ha implementado, más que como metodología de desarrollo, cuando las pruebas del software lo requieren.

En la Figura 5.8 se puede una serie de gráficas donde se puede ver la relación entre el código, la documentación (comentarios) y el *debuging* (líneas de *debug*).

En la imagen se muestra el desarrollo temporal de dichos parámetros de forma que se puede apreciar un desarrollo paralelo tanto del código como de la documentación. Además, se puede apreciar como la relación entre documentación y el *debuging* con el total de líneas se mantiene relativamente constante gracias a las gráficas porcentuales. Esto concuerda con la metodología de desarrollo adoptada para el software del proyecto, y su control periódico a lo largo del tiempo ha permitido corregir desvíos en cualquiera de las partes.

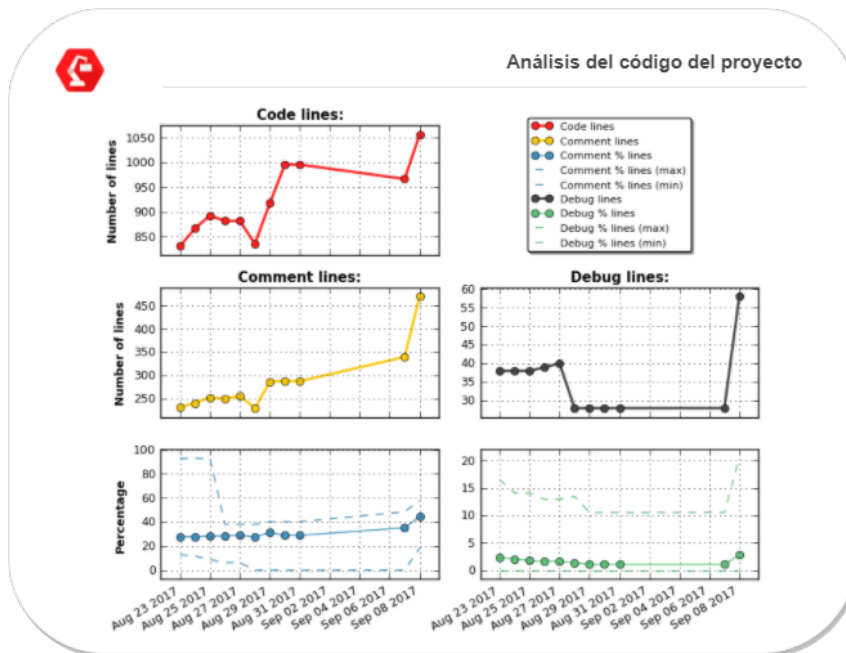


Figura 5.8: Análisis del código del proyecto

Capítulo 6

Control

En este capítulo se detalla el trabajo realizado dentro del ámbito de la ingeniería de control aplicada a este proyecto.

6.1. Control de velocidad para los servos G15 cube

Los *servos* de Cytron aún teniendo diferentes modos de control no poseen un control efectivo de la velocidad de rotación cuando se utilizan en el modo de giro continuo o *wheel mode*.

Capítulo 7

Resultados y discusión

En este capítulo...

7.1. Resultados

7.2. Discusión

Capítulo 8

Gestión del proyecto

En este capítulo se describe la gestión del proyecto: ciclo de vida, planificación, presupuesto, etc.

8.1. Ciclo de vida

Explicación de las fases del proyecto: definición, análisis, diseño, construcción, pruebas, implementación, validación, documentación. Ejemplo: diagrama de Pert.

8.2. Planificación

Se puede indicar mediante un diagrama de Gantt.

8.2.1. Planificación inicial

8.2.2. Planificación final

8.3. Presupuesto

Coste de los materiales en la Tabla 8.1:

– COMPLETAR –

Tabla 8.1: Costes del proyecto

Artículo	Coste Unitario ¹	Nº de unidades	Total
Arduino Uno	17.00 € ²	1	17.00 €
Cytron G15 Cube Servo	23.23 € ³	3	total
Cytron G15 Shield	6.64 € ⁴	1	6.64 €
Potenciómetro Serie TW	9.29 € ⁵	2	
Barras Aluminio sección cuadrada	5,626 € ⁶	5	28,13 €
Rodamiento 13x4	€		€
Rodamiento 10x3	€		€
Hilo Kevlar	23.23 € ⁷	1	23.23 €
GM Series Plastic Wheel	2.70 € ⁸	1	2.70 €

¹En los casos en que ha sido necesario se ha aplicado el cambio a Euros oficial propuesto por el Banco de España en el día en que se han consultado los precios: <https://www.bde.es/bde/es/>

²Precios consultados en la página oficial de Arduino a 07 de Septiembre 2017 (<https://store.arduino.cc/arduino-uno-rev3>).

³Precios consultados en la página oficial de Cytron a 07 de Septiembre 2017 (<http://www.cytron.com.my/p-g15>).

⁴Precio consultados en la página oficial de Cytron a 07 de Septiembre 2017 (<https://www.cytron.com.my/p-shield-g15>).

⁵Precio consultados en la página oficial de RS a 07 de Septiembre 2017 (<http://uk.rs-online.com/web/p/potentiometers/5028586/>)

⁶Precio consultados en la página oficial de RS a 07 de Septiembre 2017 (<http://es.rs-online.com/web/p/tubos-de-aluminio/3047894/>)

⁷Precio consultados en la página de compra a 07 de Septiembre 2017 (http://www.emmakites.com/index.php?main_page=product_info&cPath=336_365&products_id=1199)

⁸Precio consultados en la página oficial de Solarbotics a 11 de Septiembre 2017 (<https://solarbotics.com/product/gmpw/>)

8.3.1. Personal

8.3.2. Material

8.3.3. Resumen de costes

Capítulo 9

Conclusiones

Se presentan a continuación las conclusiones...

9.1. Conclusión

Una vez finalizado el proyecto...

9.2. Desarrollos futuros

Un posible desarrollo...

Apéndice A

Listado de piezas diseñadas

– COMPLETAR –

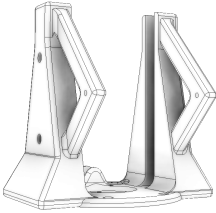


En este anexo se presenta una tabla listando las piezas que se han diseñado y fabricado para el proyecto. En la tabla A1 se puede ver una miniatura de la pieza en cuestión, la cantidad necesaria de cada tipo, una estimación del peso (material consumido en su fabricación) así como una breve descripción de la pieza y/o proceso de fabricación de la misma. A su vez llevan asociado una referencia alfanumérica que se corresponde con los ficheros en formato digital entregados, asignada de la siguiente forma:

$$RHA + \text{ubicación} + \text{número_de_pieza}$$

- *ubicación*: A\$ (articulación), B\$ (barra). La letra va acompañada de un número (sustituyendo al carácter \$) que variará dependiendo de donde se encuentre la pieza en el montaje. A1 - articulación uno, B2 - barra 2 y así sucesivamente.
- *número_de_pieza*: valor numérico que diferencia las piezas en la misma ubicación.

En caso de que la pieza se utilice en varias partes diferenciadas la referencia se tomará para la primera vez que aparece la pieza en orden ascendente (desde la barra 0 en adelante e igual desde la articulación 1).

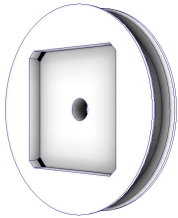
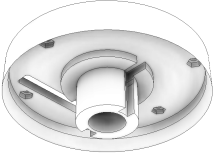
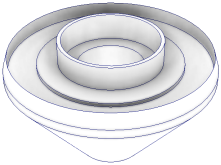
Tabla A.1: Listado de piezas diseñadas de fabricación propia

Num	Esquema Pieza	Referencia	Cantidad	Descripción	Peso Estimado ¹
1		RHAB1001	1	blah	193g
2		RHAB1002	1	blah	24g
3		RHAB1003	1	blah	24g

¹ El peso estimado se obtiene con el programa Cura – **COMPLETAR** –aplicando los parámetros de la tabla A.2. Este peso incluye el de los soportes necesarios para su impresión.

Continúa en la página siguiente

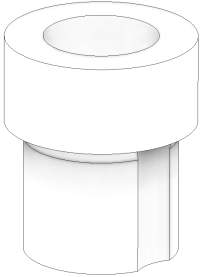
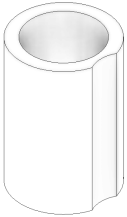
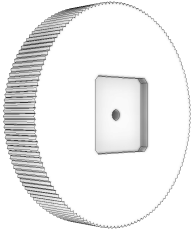
Tabla A.1 – *Continuación de la página anterior*

Num	Esquema Pieza	Referencia	Cantidad	Descripción	Peso Estimado ¹
4		RHAB1004	2	blah	3g
5		RHAA1005	1	PESO AL 90 %	54g
6		RHAA1006	1	blah	128g

¹ El peso estimado se obtiene con el programa Cura – **COMPLETAR** –aplicando los parámetros de la tabla A.2. Este peso incluye el de los soportes necesarios para su impresión.

Continúa en la página siguiente

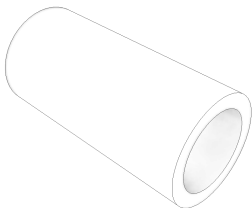
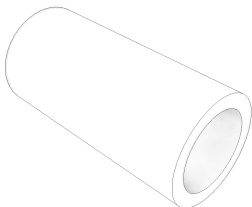
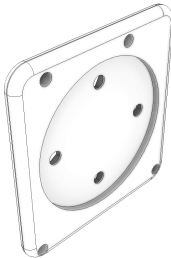
Tabla A.1 – *Continuación de la página anterior*

Num	Esquema Pieza	Referencia	Cantidad	Descripción	Peso Estimado ¹
7		RHAB0001	1	blah	blah
8		RHAB0002	1	blah	blah
10		RHAA1001	1	blah	18g

¹ El peso estimado se obtiene con el programa Cura – **COMPLETAR** –aplicando los parámetros de la tabla A.2. Este peso incluye el de los soportes necesarios para su impresión.

Continúa en la página siguiente

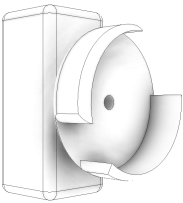
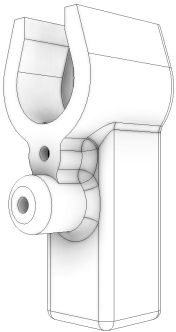
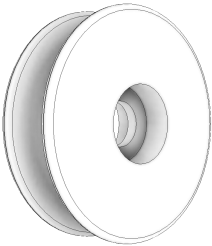
Tabla A.1 – *Continuación de la página anterior*

Num	Esquema Pieza	Referencia	Cantidad	Descripción	Peso Estimado ¹
19		RHAB1007	2	blah	1g
19		SEPARADORPOLEAS RHAB1008	2	blah	20g
9		RHAB1009	1	10g	

¹ El peso estimado se obtiene con el programa Cura – **COMPLETAR** –aplicando los parámetros de la tabla A.2. Este peso incluye el de los soportes necesarios para su impresión.

Continúa en la página siguiente

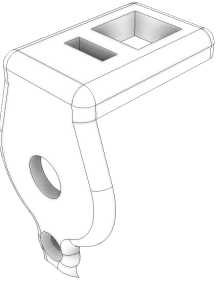
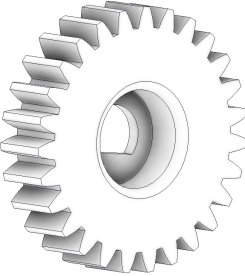
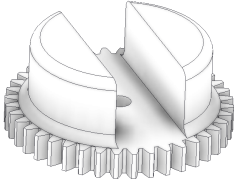
Tabla A.1 – *Continuación de la página anterior*

Num	Esquema Pieza	Referencia	Cantidad	Descripción	Peso Estimado ¹
11		RHAA2001	1	blah	blah
12		RHAA2002	1	blah	blah
13		RHAA2003	1	blah	8g

¹ El peso estimado se obtiene con el programa Cura – **COMPLETAR** –aplicando los parámetros de la tabla A.2. Este peso incluye el de los soportes necesarios para su impresión.

Continúa en la página siguiente

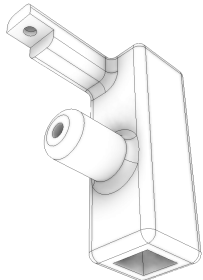
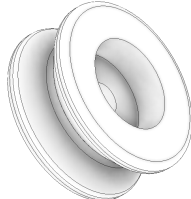

Tabla A.1 – *Continuación de la página anterior*

Num	Esquema Pieza	Referencia	Cantidad	Descripción	Peso Estimado ¹
14		RHAA2004	1	blah	blah
15		RHAA2005	1	blah	blah
16		RHAA2006	1	blah	blah

¹ El peso estimado se obtiene con el programa Cura – **COMPLETAR** –aplicando los parámetros de la tabla A.2. Este peso incluye el de los soportes necesarios para su impresión.

Continúa en la página siguiente

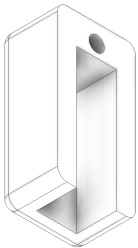
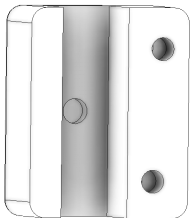
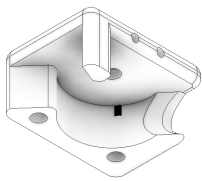
Tabla A.1 – *Continuación de la página anterior*

Num	Esquema Pieza	Referencia	Cantidad	Descripción	Peso Estimado ¹
17		RHAB2001	1	blah	blah
18		RHAB2002	3	blah	blah
19		RHAB2003	2	blah	6g

¹ El peso estimado se obtiene con el programa Cura – **COMPLETAR** –aplicando los parámetros de la tabla A.2. Este peso incluye el de los soportes necesarios para su impresión.

Continúa en la página siguiente

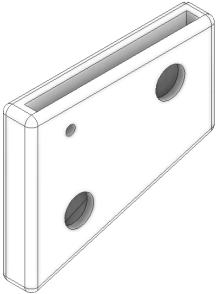
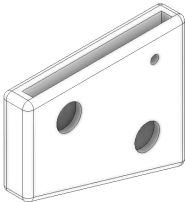
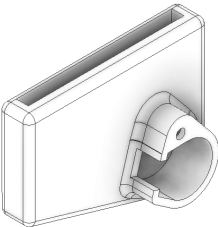
Tabla A.1 – *Continuación de la página anterior*

Num	Esquema Pieza	Referencia	Cantidad	Descripción	Peso Estimado ¹
20		RHAB2004	2	blah	blah
21		RHAB3001	1	blah	blah
22		RHAB3002	1	blah	blah

¹ El peso estimado se obtiene con el programa Cura – **COMPLETAR** –aplicando los parámetros de la tabla A.2. Este peso incluye el de los soportes necesarios para su impresión.

Continúa en la página siguiente

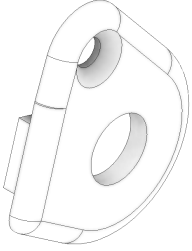

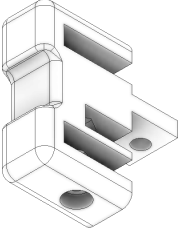
Tabla A.1 – *Continuación de la página anterior*

Num	Esquema Pieza	Referencia	Cantidad	Descripción	Peso Estimado ¹
23		RHAA3001	2	blah	39g
24		RHAA3002	1	blah	39g
25		RHAA3003	1	blah	44g

¹ El peso estimado se obtiene con el programa Cura – **COMPLETAR** –aplicando los parámetros de la tabla A.2. Este peso incluye el de los soportes necesarios para su impresión.

Continua en la página siguiente

Tabla A.1 – *Continuación de la página anterior*

Num	Esquema Pieza	Referencia	Cantidad	Descripción	Peso Estimado ¹
26		RHAA3004	1	1	2g
27		RHAA3005	2	blah	-
28		RHAA3006	4	blah	9g

¹ El peso estimado se obtiene con el programa Cura – **COMPLETAR** –aplicando los parámetros de la tabla A.2. Este peso incluye el de los soportes necesarios para su impresión.

Continúa en la página siguiente

Tabla A.1 – *Continuación de la página anterior*

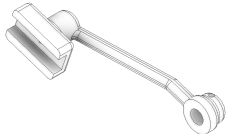
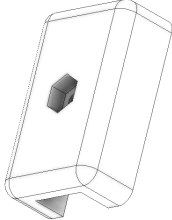
Num	Esquema Pieza	Referencia	Cantidad	Descripción	Peso Estimado ¹
29		RHAA3007	1	blah	24g
30		RHAA3008	1	blah	blah

Tabla A.2: Parámetros de las piezas para la estimación de peso

a	a	a	a
---	---	---	---

Apéndice B

Montaje del prototipo

Apéndice C

Reglas de codificación del Software

Las reglas de codificación aplicadas al software del proyecto se han obtenido, por utilizar una referencia, de las reglas aplicadas por Google en sus proyectos libres. Esta guía está ampliamente documentada e incluye su propia herramienta para comprobar su correcta aplicación – **COMPLETAR** – lo que facilita la revisión del código así como corrección de desviaciones de estilo.

En este anexo se traducen y resumen los aspectos más importantes de dicha guía. En algunos casos se han adaptado las reglas al caso concreto de este proyecto.

Establecer unas reglas de codificación, unificando un estilo en la notación y uso de la sintaxis, es interesante de cara a posibilitar una mayor facilidad de lectura en futuros desarrollos aumentando así la mantenibilidad del código.

Estas reglas aplican al código C++ del proyecto, no a los *scripts* auxiliares.

C.1. Aspectos generales

C.2. Ficheros de cabecera

En general todos los ficheros con extensión `.cpp` correspondientes a las librerías deberán ir acompañados del fichero de cabecera `.h` correspondiente.

Quedan exentos de cumplir esta regla los ficheros correspondientes a Test (unitarios, de integración, etc) así como ficheros que contengan únicamente una función `main()`.

C.2.1. Inclusión Múltiple

Para evitar problemas de inclusión múltiple todos los ficheros de cabecera con extensión `.h` deberán incluir guardas con el siguiente formato y escrito en mayúsculas: `<nombre_del_fichero>_<extensión>`.

Ejemplo:

```
#ifndef SERVO_RHA_H
#define SERVO_RHA_H
```

```
...
...
...
#endif
```

C.2.2. Orden de inclusión de ficheros

Para evitar problemas en las dependencias de las distintas librerías se incluirán las mismas dejando para el final las librerías propias del proyecto e incluyendo el resto de la más general a la más particular. **REVISAR EL CÓDIGO!**

Ejemplo orden al incluir cabeceras:

```
#include <stdint.h>      // lib estandar de c++
#include <Arduino.h>      // lib de Arduino
#include <SoftwareSerial.h> // lib para controlar el puerto serie. Basado
                          en Arduino

#include "debug.h"        // def y control de las funciones de debug
#include "rha_types.h"    // tipos de datos
#include "joint_rha.h"    // clase a incluir
```

¿DEF COMO ABREVIATURA?

Se deben incluir todos los ficheros que definan los símbolos utilizados en el fichero sobre el que se incluyen. Las declaraciones anticipadas de objetos no están permitidas salvo excepciones justificadas.

C.3. Ámbitos

C.3.1. Espacios de nombres

Como norma general las constantes, variables o funciones que no estén contenidas en ningún objeto se incluirán dentro de un espacio de nombres o *namespace* que haga referencia a la utilidad de las mismas.

No está permitido usar directivas del tipo `<using namespace ----;>`.

Los espacios de nombres se escriben con la primera letra de cada palabra, en caso de haber más de una, en mayúscula y sin separación de ningún tipo.

Ejemplo: Constantes referentes al test de comportamiento ante una entrada tipo rampa

```
namespace SlopeTest {
    #define SAMPLESLOPE 110
    #define SAMPLE_TEST_SLOPE 20
    #define SLOPE_SPEED 0.1
}
```

Prohibido el uso de:

```
using namespace StepTest;
```

¿DEF COMO ABREVIATURA?

C.3.2. Variables Locales

Las variables se definirán preferiblemente en el ámbito más bajo **¿SE DICE ASÍ?** en que se vayan a utilizar. Preferiblemente la inicialización de las mismas se hará junto a la declaración.

Se pueden dar excepciones, como pueden ser vectores sobre los que se iterará dentro de un bucle u otros casos similares. En estos casos se de declarará el objeto fuera del propio ámbito para evitar recursivas llamadas a constructor y destructor de los mismos.

Ejemplo:

```
// Siempre que la variable sobre la que se itera no se vaya a utilizar para
// posteriores operaciones:
for(int i = 0, i < 10; i++) {
}

//mejor que el caso siguiente, que adicionalmente incumple la regla preferente
// de inicializar la variable cuando se declara:
int i;
for(i = 0, i < 10; i++) {
}

//queda permitido declarar vectores u otros objetos similares antes si se va a
// iterar o trabajar sobre los mismos
int vector[5] = {1,2,3,4,5};
for(int i = 0, i < 10; i++) {
    Serial.print(vector[i]);
}
```

¿DEF COMO ABREVIATURA?

C.4. Clases

C.4.1. Constructores y métodos de Inicialización

Para todos los objetos debe haber constructores por defecto sin parámetros de entrada. Aunque se pueden añadir constructores que inicialicen los diferentes parámetros será obligatorio generar métodos que los inicialicen una vez construido el objeto. Arduino, aún estando basado en el lenguaje C++ no permite un uso completo de memoria dinámica. Es especialmente importante para declarar estos objetos como miembros de otros. **NO SE SI ESTÁ MUY CLARO :—**

Ejemplo:

```
//NO se permite:
- joint_rha.h -
ServoRHA servo_*;
- joint_rha.cpp -
servo_ = new ServoRHA(1, 10, 5);

//Se llama al constructor del objeto para luego inicializarlo:
- joint_rha.h -
ServoRHA servo_;
- joint_rha.cpp -
servo_.init(... params ...);
```

Para evitar funciones con muchos parámetros que reduzcan la legibilidad del código se permite generar diferentes inicializadores para los distintos parámetros. En la documentación del objeto deberá quedar bien claro que inicializadores deben invocarse para el correcto funcionamiento del mismo.

C.4.2. Estructuras o Clases

Por norma general las estructuras se utilizarán exclusivamente para objetos pasivos, objetos que contienen información. Todo lo demás se codificará dentro una clase.

En el caso de estructuras se permiten únicamente métodos para el manejo de los datos sin añadir ninguno tipo de comportamiento, están permitidos los constructores, destructores, métodos de reset, validación, etc. El acceso a los miembros de la estructura se hará directamente sobre los propios parámetros y no mediante métodos específicos. Los parámetros serán siempre públicos para ser consistente con este punto.

Para mayores funcionalidades se generará una clase.

C.4.3. Control de Acceso

Como norma general se declararan como privados todos los atributos de las clases exceptuando aquellos objetos que a su vez tengan, internamente, control de acceso definido (otras clases). De cara a generar Test con clases propias se permite la declaración de atributos como `protected`.

C.5. Tipos de datos

Los tipos de datos usados irán acordes con la librería `stdint.h`. Estos son del tipo `int16_t`, `uint32_t`, etc. Este tipo de datos garantiza el control del tamaño del dato declarado.

Se utilizarán los nombres `float` y `double` convencionales para declarar datos en coma flotante.

C.6. Nombres

C.6.1. Reglas generales

Los nombres deberán ser descriptivos. Por norma general no se utilizarán abreviaciones que no estén comúnmente aceptadas.

C.6.2. Nombre de los ficheros

Los nombres de los ficheros de código C++ se nombran en minúsculas separando, en caso de haber varias palabras, con un guión bajo. Los ficheros correspondientes a los test llevarán, precediendo al nombre la palabra "test".

Ejemplos:

```
joint_handler.h
joint_rha.cpp
test_servo_rha.cpp
```

C.6.3. Nombre de los directorios

Los ficheros de código irán contenidos en diferentes directorios para cada librería o conjunto de test. Estos directorios llevarán el nombre de la librería que contienen, en el mismo formato que la misma, en este caso sin extensión. Los test se ejecutan en el orden en que se ordenan los directorios. En este caso se añadirá un caracter para ordenar los mismos de manera adecuada.

Están exentos de esta regla los ficheros principales (que contienen la función `main()`, ó `setup()` y `loop()` en caso de ser ficheros con extensión `.ino`).

Ejemplos:

```
/lib/
  joint_handler/
  joint_rha/
/test/
  a_test_servo_rha/
  b_test_joint_rha/
```

C.6.4. Nombres para objetos

Los nombres llevarán mayúscula al comienzo así como al inicio de cada palabra, sin guion bajo como separación.

Ejemplo:

```
class ServoRHA{ ... };
class JointHandler{ ... };
struct SpeedGoal { ... } ;
```

C.6.5. Nombres de variables

Por norma general las variables se nombrarán en minúsculas, separando, cuando fuera necesario, las diferentes palabras mediante un guión bajo.

C.6.6. Nombres de atributos de clases

La norma para nombrar atributos de clases será igual que en el caso general acabando, en este caso, con un guión bajo.

Ejemplo:

```
class Regulator {
  float kp_, ki_, kd_;
  float ierror_[INTEGER_INTERVAL];
  uint8_t index_;
```

```
... } ;
```

C.6.7. Nombres de miembros de estructuras

Las variables miembro de estructuras serán nombradas de igual forma que en el caso general.

Ejemplo :

```
struct SpeedGoal {
    uint8_t servo_id;
    int16_t speed;
    int16_t speed_slope;
    uint8_t direction;
    ... } ;
```

C.6.8. Nombres de funciones

Las funciones comenzarán en minúscula marcando con mayúscula cada nueva palabra que aparezca. Los acrónimos irán en mayúscula. Esta regla afecta a métodos de clases a de igual manera a excepción de constructores y destructores.

Ejemplo :

```
class ServoRHA {
    ...
public:
    ServoRHA() { time_last_error_ = 0; time_last_ = 0; last_error_ = 0;
                error_ = 0; derror_ = 0; ierror_ = 0; }
    ServoRHA(uint8_t servo_id);
    void init(uint8_t servo_id);
    void addUpadteInfoToPacket(uint8_t *buffer);
    bool addTorqueToPacket(uint8_t *buffer);
    void setTorqueOnOfToPacket(uint8_t *buffer, uint8_t onOff);
    ... } ;
```

C.6.9. Nombres de parámetros funciones

Los parámetros de métodos y funciones se nombran siguiendo el caso general para nombrar variables.

C.6.10. Espacios de nombres

Como se ha visto en la sección C.3 los espacios de nombres se definen de manera equivalente a las clases.

C.6.11. Nombres de enumeraciones

En el caso de enumeraciones se seguirá la misma norma que para las clases y espacios de nombres. En este caso cabe la excepción de poder ser declaradas sin nombre.

C.6.12. Nombres de macros

Todo nombre precedido por una instrucción **#define** se nombrará en mayúsculas, separando las palabras, si las hubiera, mediante el uso del guión bajo. Esto aplica tanto a macros como constantes.

C.7. Comentarios

Es necesario el uso de comentarios para documentar el código y aumentar la legibilidad del mismo. En este caso se seguirá el estilo utilizado por *doxygen*, que será la herramienta utilizada para, posteriormente generar la documentación.

C.7.1. Comentarios de ficheros

Todos los ficheros deberán llevar comentarios en su cabecera. Estos comentarios tendrán el siguiente aspecto:

Ejemplo :

```
/**
 * @file
 * @brief Implements ServoRHA class. This object inherits from CytronG15Servo
 *        object to enhance its capabilities
 *
 * @Author: Enrique Heredia Aguado <enheragu>
 * @Date: 2017-Sep-08
 * @Project: RHA
 * @Filename: servo_rha.h
 * @Last modified by: quique
 * @Last modified time: 30-Sep-2017
 */
```

C.7.2. Comentarios de Clases

¡TDB!

C.7.3. Comentarios de funciones

Todas las funciones y métodos deberán llevar un comentario describiendo su funcionamiento así como los parámetros de entrada y salida. Estos comentarios tendrán el siguiente aspecto y se situarán encima de la definición de la función o método:

Ejemplo :

```
/** @brief Saves in buffer the package return level of servo (error information
    for each command sent)
 * @method ServoRHA::addReturnOptionToPacket
 * @param {uint8_t*} buffer array in which add the information
 * @param {uint8_t} option RETURN_PACKET_ALL -> servo returns packet for all
    commands sent; RETURN_PACKET_NONE -> servo never returns state packet;
    RETURN_PACKET_READ_INSTRUCTIONS -> servo answer packet state when a READ
    command is sent (to read position, temperature, etc)
 * @see addToPacket()
 */
```

C.7.4. Comentarios y aclaraciones

Cuando sea necesario hacer aclaraciones, a nivel de código se harán utilizando el estilo de comentario con doble barra `//`. Por lo general los nombres de variables y funciones deberán ser de por sí descriptivas por lo que este tipo de comentarios se reservan para partes del código especialmente enrevesadas.

Los comentarios, cuando vayan en línea con el código, se situarán a dos espacios del mismo, dejando un espacio entre el comentario en sí y la doble barra.

C.7.5. TODO y notas

En algunos casos se podrán dejar cosas para hacer en futuro (TODO) o notas aclaratorias (NOTE). En ambos casos se pondrá en mayúsculas y seguido de dos puntos. Quedando comentados mediante doble barra.

Ejemplo:

```
// TODO: complete CW and CCW selection
// NOTE: important the use of mascarees to obtain direction os movement
```

C.7.6. Código en desuso

En algunas situaciones hay fragmentos de código que ya no se utilizan o están temporalmente deshabilitados. Estos fragmentos serán comentados mediante barra y asterisco:

Ejemplo:

```
/* ...
... some code ...
... */
```

C.8. Formato

C.8.1. Espacios y tabulaciones

Por norma general se utilizará cuatro espacios como indentación para distintos ámbitos.

Ejemplo:

```
void ServoRHA::setWheelSpeedToPacket( ... ) {
    ...
    if ( ... ) {
        ...
    }
    ...
}
```


C.8.2. Declaración y definición de funciones

El valor de retorno así como los parámetros de una función deberán ir en la misma línea. En caso de no caber o para mayor claridad se pondrán a la misma altura que los anteriores.

Ejemplo:

```
void ServoRHA::setWheelSpeedToPacket(uint8_t *buffer, uint16_t speed, uint8_t
    direction) {
    ...
}

void ServoRHA::setWheelSpeedToPacket(uint8_t *buffer, uint16_t speed,
    uint8_t direction) {
    ...
}
```

C.8.3. Condicionales

Como norma general no se dejarán espacios entre los paréntesis. Si se dejará un espacio entre la sentencia `if` y el condicional, así como entre este último y la llave que abre el ámbito condicional.

Ejemplo:

```
//Forma correcta:
if (direction == CW) {
    speed = speed | 0x0400;
}

//Ejemplos incorrectos:
if(direction == CW) { // Falta un espacio tras la sentencia if
if (direction == CW){ // Falta un espacio entre el condicional y la llave
if(direction == CW){ // Combina los casos anteriores
```

En caso de que el condicional afecte solo a una sentencia esta se pondrá, como norma general, sin llaves y en la misma línea que el condicional. De igual forma se hará tras sentencias de tipo `else` o combinando `else if`. En caso de utilizar llaves se seguirá la norma que aplica a dicho caso.

Ejemplo:

```
if (speed1 < speed2-speed_margin) return ServoRHAConstants::LESS_THAN;
else if (speed1 > speed2+speed_margin) return ServoRHAConstants::GREATER_THAN;
else return ServoRHAConstants::EQUAL;
```

Cuando si afecta a diferentes líneas y hay sentencias de tipo `else`, estas irán en la misma línea de cierre de llave del condicional (siempre que no afecte a la legibilidad del código ya sea por presencia de comentarios u otras causas similares).

Ejemplo:

```
if (...) {
    ...
} else {
    ...
}
```

C.8.4. Bucles

El formato será equivalente al caso de los condicionales:

Ejemplo :

```
for (...) {  
    ...  
}  
for (...) oneLineStatement;  
while (condition) {  
    ...  
}
```

C.8.5. Valor de retorno de funciones y métodos

No es necesario utilizar paréntesis para rodear la expresión a retornar. Solo se utilizarán en los casos en que se utilizarían si se fuera a asignar dicha expresión a una variable.

C.8.6. Formato para clases

Las directivas **public**, **protected** y **private** irán indentados un espacio respecto a la definición de la clase. Por norma general irán precedidos por una línea en blanco (excepto cuando las preceda la definición de la propia clase).

Ejemplo :

```
class JointHandler {  
    private: // un espacio  
    ...  
    public:  
    ...
```

C.8.7. Espacios de nombre

Los espacios de nombres siguen la norma general para indentar diferentes ámbitos.

C.9. Espacios en blanco

Los espacios horizontales dependerán de cada caso. En ningún caso se finalizará una línea con un espacio en blanco.

C.9.1. Caso general

Ejemplo :

```
void JointHandler::setTimer(uint64_t timer) { // Un espacio entre el cierre  
    del parentesis y la apertura de llaves  
    class TimerMicroseconds : public Timer { // Espacio entre los dos puntos en  
        casos de herencia o inicializadores dentro de constructores. Se pone un  
        espacio a cada lado.  
    void checkWait() // No se deja espacio entre el nombre y los parentesis.  
        Tampoco entre parentesis vacios.
```

```
float getError() { return error_; } // Se deja espacio entre llaves e
    implementacion, a ambos lados.
```

C.9.2. Bucles, condicionales y estructuras de control

Ejemplo:

```
if (b) { // Espacio entre la sentencia if y la condicion, asi como esta misma
    con la apertura de llaves
} else { // Espacios al rededor de la sentencia else
}
switch (i) {
    case 1: // No se deja espacio antes de los dos puntos
        ...
    case 2: break; // Si se deja despues de los mismos
for (int i = 0 ; i < 5 ; i++) { // En caso de bucles for, ademas de los
    espacios al rededor de los parentesis se dejara un espacio tras cada punto
    y coma.
```

C.9.3. Operadores

Ejemplo:

```
// En general se deja un espacio al rededor de los distintos tipos de
    operadores
x = 0;
v = w * x + y / z;
v = w*x + y/z;
v = w * (x + z);

// No se separan operadores unarios de sus argumentos:
x = -5;
x++;
if (x && !y)
```

¿ESPACIOS VERTICALES Y HORIZONTALES DEBERIAN IR DENTRO DE LA SECCION DE FORMATO NO?

C.10. Espacio vertical

Por lo general se dejarian espacios verticales para una mayor claridad del código sin abusar de los mismos. Aunque separar diferentes partes puede ayudar demasiados espacios verticales pueden dificultar la lectura de código.

HABLAR DE COMO SE DEFINEN LAS VARIABLES, CUANDO SE PUEDEN PONER VARIAS EN LA MISMA LÍNEA Y DEMÁS

<https://google.github.io/styleguide/cppguide.html>

Apéndice D

Documentación del software
