



*Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingeniería de Sistemas Informáticos
Grado en Ingeniería de Computadores
Trabajo de Fin de Grado*

pArm – sistema informático empotrado para gobernar un brazo robótico de diseño abierto

Javier Alonso Silva
Mihai Octavian Stanescu
José Alejandro Moya Blanco

*Tutores: Norberto Cañas de Paz
Marina Pérez Giménez*

Madrid, 21 de octubre de 2020

Índice general

Índice de figuras	VI
Índice de cuadros	XV
Definiciones, siglas, y abreviatura	XIX
1. Motivación y objetivos	1
1.1. Estado del arte	1
1.1.1. Desarrollo de la robótica a lo largo de la historia	1
1.1.2. Los brazos robóticos	4
1.1.3. La actualidad	5
1.2. Motivaciones y objetivos del desarrollo del proyecto	11
1.3. Metodología	12
2. Explicación de la estructura del proyecto	13
2.1. Matemáticas	14
2.2. <i>Hardware</i>	16
2.3. <i>Software</i>	16
3. Especificación de requisitos	18
3.1. Introducción	18
3.1.1. Propósito	19
3.1.2. Alcance	19
3.1.3. Visión global	20
3.2. Descripción general	20
3.2.1. Perspectiva del producto	20

3.2.2. Interfaz del sistema	20
3.2.3. Interfaz de usuario	23
3.2.4. Memoria	23
3.2.5. Operaciones	24
3.2.6. Funciones del producto	24
3.2.7. Características del usuario	24
3.2.8. Restricciones	25
3.2.9. Supuestos y dependencias	25
3.2.10. Requisitos pospuestos	27
3.3. Requisitos específicos	28
3.3.1. Requisitos de la interfaz externa	28
3.3.2. Casos de uso	30
3.3.3. Requisitos funcionales	34
3.3.4. Restricciones del diseño	36
3.3.5. Atributos del sistema <i>software</i> y <i>hardware</i>	36
3.3.6. Requisitos no funcionales	36
4. Diagramas y diseño	37
4.1. Diagramas <i>software</i> de S1	38
4.2. Diagramas <i>software</i> de S2	44
5. Fundamentos matemáticos del proyecto	59
5.1. Cinemática directa	63
5.2. Cinemática inversa	64
5.3. Funciones jacobianas	73
5.4. Implementación final realizada	77
6. Hardware	78
6.1. Diseño 3D	78
6.2. Construcción del brazo	97
6.3. Configuración mecánica del brazo	116
6.4. Microcontrolador utilizado	117

6.5.	Desarrollo y componentes de la PCB	118
6.5.1.	Objetivos	118
6.5.2.	Componentes principales	119
6.5.3.	Diseño lógico y diagrama esquemático	120
6.5.4.	Conversión del diagrama esquemático a diagrama físico	134
6.5.5.	Conexionado de los componentes mediante pistas	143
6.5.6.	Verificaciones realizadas al diseño lógico y físico	149
6.5.7.	Construcción	151
6.5.8.	Verificaciones del prototipo construido	170
6.5.9.	Contratiempos ocurridos y soluciones planteadas	172
6.6.	Motores empleados (actuadores)	181
7.	<i>Software</i>	186
7.1.	S1	186
7.1.1.	Interfaz Gráfica de Usuario (GUI)	187
7.1.2.	Lógica de comunicaciones	194
7.1.3.	Protocolo de autenticación	198
7.1.4.	Pseudo-lenguaje de comunicación	199
7.1.5.	Logs	202
7.2.	S2	202
7.2.1.	Inicialización del sistema	205
7.2.2.	Control de los componentes	216
7.2.3.	Cálculo de movimientos/trayectorias	227
7.2.4.	Interpretación del pseudo-lenguaje	228
7.2.5.	<i>Heartbeat</i> y cifrado RSA	233
7.2.6.	Opciones de compilación	234
8.	Impresión 3D	237
8.1.	El proceso de impresión 3D	238
8.1.1.	El entorno de impresión 3D	240
8.1.2.	Los parámetros de configuración	243

9. Calidad y pruebas	247
9.1. Explicación de las pruebas	247
9.1.1. Pruebas en el diseño de las piezas 3D	247
9.1.2. Pruebas en la impresión 3D	250
9.1.3. Pruebas post-impresión	250
10. Demostración	252
11. Planificación, costes y tiempo empleado	254
11.1. Diagramas de Gantt	254
11.2. Sueldos propuestos y costes obtenidos	258
11.3. Coste de los materiales inicial - coste de los materiales final	259
11.4. Evolución del tiempo empleado, contratiempos y tiempo de desarrollo final . . .	260
11.4.1. Contratiempos de la impresión 3D	262
12. Conclusiones	268
12.1. Conclusiones técnicas	268
12.2. Conocimientos adquiridos y nuevas competencias	269
12.3. Reflexión final	270
13. Futuras mejoras	274
13.1. Impactos sociales y medioambientales	277
Bibliografía	279
A. Código fuente “<i>pArm configurator</i>”	284
A.1. Enlace a <i>Jupyter Notebook</i> para configurar el <i>pArm</i>	294
B. Enlaces útiles	295
C. Código fuente S2	296
C.1. <i>Header files</i>	296
C.2. <i>Source files</i>	339
D. Matriz pseudo-inversa cuando $J(\dot{q}) = 0$	393

E. Código fuente Sistema 1 – ordenador (S1)	394
F. Diagrama de Gantt al completo	447

Índice de figuras

1.1.	Flautista y tamborilero de Vaucanson [2].	1
1.2.	En 1774, “lady musician” por Jaquet-Droz [3].	2
1.3.	“The Turk”, creado por von Kempelen en 1769 [4].	2
1.4.	Barco a control remoto de Nicola Tesla, en 1898 [5]	3
1.5.	“Alpha”, el primer robot diseñado con fines militares y su posterior evolución, “Elektro”.	3
1.6.	Exposición mundial del 2005 en Japón [6].	4
1.7.	Grados de libertad de un brazo robótico y estructura del cuerpo humano [1]. .	4
1.8.	Vista exterior del Canadarm2 [7].	5
1.9.	Rover “Spirit”, desarrollado por la NASA y desplegado en 2004 [8].	6
1.10.	Modo de funcionamiento del sistema “ <i>rocker–bogie</i> ”, desarrollado por la NASA para sus rover [9].	6
1.11.	Rover “Opportunity”, desarrollado por la NASA y desplegado en 2004 [10]. .	7
1.12.	Lugares de aterrizaje de los rovers de la misión espacial a Marte [11].	7
1.13.	Robot Roomba en la estación de carga [12].	8
1.14.	Lo que ve un Tesla cuando está en conducción autónoma nivel 2 [13].	8
1.15.	Robot “ <i>Big-Dog</i> ” de Boston Dynamics [15].	9
1.16.	Robot “ <i>Atlas</i> ” de Boston Dynamics [15].	9
1.17.	Robot “KR-1000 Titan” de KUKA [16].	10
1.18.	Robot μ Arm de UFACTORY [17].	10
2.1.	Ejemplo de cadena cinemática [21].	15
2.2.	Diagrama del modelo cinemático [22].	15
3.1.	Diseño propuesto para la interfaz gráfica de usuario.	23

3.2. Diagrama de casos de uso	30
4.1. Diagrama de clases de S1.	38
4.2. Recorte del diagrama de clases de S1 el cual representa la lógica del sistema. .	39
4.3. Recorte del diagrama de clases de S1 el cual representa la interfaz gráfica de usuario.	40
4.4. Diagrama de estados del tratamiento de errores de S1.	41
4.5. Diagrama de estados del funcionamiento normal de S1.	42
4.6. Diagrama que representa el movimiento del brazo robótico tanto en S1 como en Sistema 2 – pArm (S2).	42
4.7. Diagrama de estados del encendido de S1.	43
4.8. Diagrama de estados del apagado de S1.	43
4.9. Diagrama de bloques de S2.	44
4.10. Diagrama de estados del método <code>main()</code> del <i>orchestrator</i>	46
4.11. Diagrama de estados del método <code>loop()</code> del <i>orchestrator</i>	47
4.12. Diagrama de estados del método <code>CheckMotorHealthStatus()</code> del <i>orchestrator</i> .	48
4.13. Diagrama de estados del método <code>CancelMovement()</code> del <i>orchestrator</i>	48
4.14. Diagrama de estados del método <code>CommunicateError()</code> del <i>orchestrator</i>	49
4.15. Diagrama de estados del método <code>DoHandShake()</code> del <i>orchestrator</i>	49
4.16. Diagrama de estados del método <code>ExecuteMovement()</code> del <i>orchestrator</i>	50
4.17. Diagrama de estados del método <code>ExecuteOrder()</code> del <i>orchestrator</i>	51
4.18. Diagrama de estados del método <code>moveArm()</code> del <i>orchestrator</i>	52
4.19. Diagrama de estados del método <code>uartInit()</code> del <i>UART</i>	52
4.20. Diagrama de estados del método <code>setBaudrate()</code> del <i>UART</i>	53
4.21. Diagrama de estados del método <code>sendMessage()</code> del <i>UART</i>	53
4.22. Diagrama de estados del método <code>receiveBitStream()</code> del <i>UART</i>	54
4.23. Diagrama de estados del método <code>init()</code> del <i>motorHandler</i>	54
4.24. Diagrama de estados del método <code>checkMotorStatus()</code> del <i>motorHandler</i>	55
4.25. Diagrama de estados del método <code>sendMovementToMotors()</code> del <i>motorHandler</i> .	56
4.26. Diagrama de estados del método <code>cancelMovement()</code> del <i>motorHandler</i>	56
4.27. Diagrama de estados del método <code>init()</code> del <i>movementComputer</i>	57
4.28. Diagrama de estados del método <code>computeXYZMovement()</code> del <i>movementComputer</i> .	57

4.29. Diagrama de estados del método <code>computeAngleMovement()</code> del <i>movementComputer</i>	58
5.1. Configuración geométrica del μ Arm.	60
5.2. Los distintos grados de libertad del μ Arm, representados por Z_i	60
5.3. Longitudes del brazo robótico [24].	60
5.4. Parte superior del <i>pArm</i> con el triángulo para obtener la cinemática inversa. .	68
5.5. Triángulo para la aplicación del teorema del coseno en la ecuación 5.12. . . .	69
5.6. Posición máxima en ‘ <i>y</i> ’, donde $x = 0$	70
5.7. Posición máxima en ‘ <i>x</i> ’, donde $y = 0$	70
5.8. Demostración de la correlación entre ‘ <i>x</i> ’ e ‘ <i>y</i> ’.	70
5.9. Triángulo resultante tras aplicar las modificaciones a los lados.	71
5.10. El triángulo colocado a modo de referencia sobre el <i>pArm</i>	71
5.11. Triángulo final orientativo junto con el ángulo ϕ respecto al plano del suelo. .	72
6.1. Concepto inicial del brazo robótico.	78
6.2. Logotipos de las herramientas utilizadas.	79
6.3. Flujo de trabajo del desarrollo y la impresión 3D.	79
6.4. Construcción de la forma general.	80
6.5. Agujeros para tornillos.	81
6.6. Agujeros centrales.	82
6.7. Tapa con torre central.	83
6.8. Desgastes y rebaje.	84
6.9. Torre con chaflanes.	85
6.10. Tapa con ranura para cables.	86
6.11. Proyección de la placa de control (naranja) sobre la base original del μ Arm (gris). .	87
6.12. Placa y motor dentro de la caja original del μ Arm.	88
6.13. Base y paredes tras realizar las modificaciones necesarias	88
6.14. Sistema de raíles de la placa.	89
6.15. Versión final de los carriles.	90
6.16. Pieza de sujeción del motor de la base.	90
6.17. Pieza de sujeción del motor (verde) dentro de la caja.	91

6.18. Tapa original (derecha) junto a la tapa modificada (izquierda).	91
6.19. Tapa superior transparente y base giratoria (verde).	92
6.20. Pieza de unión del motor y el eje.	93
6.21. Sistema completo.	93
6.22. Placa de sujeción de los motores laterales.	94
6.23. Base de giro con raíl.	95
6.24. La pieza sustituta del eje metálico (naranja).	95
6.25. Sistema completo del eje giratorio.	96
6.26. <i>End-effector</i> inicial (izquierda). <i>End-effector</i> modificado (derecha).	96
6.27. Brazo robótico completo tras las modificaciones.	97
6.28. Elementos mecánicos externos.	98
6.29. Base de la caja del brazo robótico.	99
6.30. Base y paredes de la caja del brazo robótico.	99
6.31. Caja completa del brazo robótico.	100
6.32. Sujeción del motor de la base.	100
6.33. Motor de la base ensamblado en su soporte.	101
6.34. Primera pieza del sistema de transmisión del movimiento.	101
6.35. Segunda pieza del sistema de transmisión del movimiento.	102
6.36. Tercera pieza del sistema de transmisión del movimiento.	102
6.37. Cadena de transmisión final con soporte para fin de carrera.	103
6.38. Interior de la caja.	103
6.39. Base rotatoria con pletinas.	104
6.40. Soporte para los motores laterales.	104
6.41. Se añade uno de los motores.	105
6.42. Segmento central del brazo robótico.	106
6.43. Pieza izquierda de la cadena de movimiento vertical.	106
6.44. Pieza central de la cadena de movimiento vertical.	107
6.45. Pieza derecha de la cadena de movimiento vertical.	107
6.46. Eje metálico interno.	108
6.47. Pieza que sirve para unir el segmento inferior con el superior.	109
6.48. Antebrazo montado en la cadena articulada principal.	110

6.49. Soporte del <i>end-effector</i>	110
6.50. Soporte del motor que actúa sobre el <i>end-effector</i>	111
6.51. Cadena articulada auxiliar derecha.	112
6.52. Triangulo de unión del las varillas de la cadena auxiliar izquierda.	113
6.53. Varilla inferior de la cadena auxiliar izquierda.	114
6.54. Varilla superior de la cadena auxiliar izquierda.	115
6.55. La parte superior el brazo robótico ensamblada en la base rotativa.	115
6.56. Brazo robotico completo.	116
6.57. Una estructura pantográfica que transmite el movimiento de un punto al siguiente [34].	117
6.58. Diagrama de conexionado del LM317 [35].	121
6.59. Diagrama esquemático del circuito de alimentación de los servomotores.	122
6.60. Diagrama de conexionado del regulador L7805 [36].	123
6.61. Diagrama de conexionado del regulador AZ1117H [37].	123
6.62. Diagrama esquemático de la etapa de alimentación del microcontrolador.	124
6.63. Diagrama esquemático del microcontrolador y sus periféricos.	125
6.64. Conexionado mínimo del microcontrolador [38].	126
6.65. Diagrama esquemático del conexionado del generador de señales.	126
6.66. <i>Pinout</i> del conector de la sonda de programación [38].	127
6.67. Conexión del pin \overline{MCLR}/V_{PP} [38].	127
6.68. Diagrama esquemático del puerto de programación.	128
6.69. Circuito lógico para los finales de carrera.	129
6.70. Diagrama esquemático del puerto TRIS.	130
6.71. Esquema del generador PWM [38].	131
6.72. Diagrama esquemático del puerto PWM.	132
6.73. Esquema del periférico UART [38].	132
6.74. Diagrama esquemático de los puertos UART.	133
6.75. Diagrama esquemático de los LEDs.	133
6.76. Diagrama esquemático completo.	134
6.77. Herramienta de asignación de huellas.	135
6.78. Ventana de asignación de huellas físicas.	136

6.79. Huella física de un condensador usado en la <i>Printed Circuit Board</i> (PCB).	137
6.80. Encapsulado elegido para el microcontrolador [38].	138
6.81. Huella física asignada al microcontrolador.	139
6.82. Herramienta de generado de listado de redes.	139
6.83. Archivo de listado de redes.	140
6.84. Acceso directo a la herramienta “PCBnew”.	140
6.85. Herramienta de importado de listado de redes.	141
6.86. Situación inicial del diseño nada mas importar los componentes físicos.	141
6.87. Distribución inicial de los componentes.	142
6.88. Ventana principal de “PCB Calculator”.	144
6.89. Cálculo del ancho de pistas de alimentación.	145
6.90. Cálculo del ancho de pistas de comunicación.	145
6.91. Cálculo inverso del ancho de pistas de comunicación.	146
6.92. Distribución final de los componentes físicos.	147
6.93. Diagrama físico final.	148
6.94. Representación 3D del diseño físico.	149
6.95. Herramienta de verificación de reglas de diseño.	151
6.96. Estructura de la placa de prototipado [39].	152
6.97. Tratado de las resinas mediante insolado [39].	153
6.98. Resultado tras el insolado [39].	153
6.99. Proceso de revelado [39].	154
6.100 Resultado tras revelado [39].	154
6.101 Resultado tras atacado [39].	154
6.102 Herramienta de trazado de placas.	155
6.103 Fotomáscara final de la PCB.	156
6.104 Proceso de impresión y recorte de las fotomáscaras.	156
6.105 Fotomáscara final.	157
6.106 Proceso de insolación de la PCB usando las fotomáscaras.	158
6.107 Placa de prototipado positiva insolada.	158
6.108 Revelador universal empleado.	159
6.109 Proceso de revelado de la PCB.	160

6.110 Proceso de atacado de la PCB.	161
6.111 Circuito impreso final tras el proceso de atacado.	161
6.112 Integrante del equipo verificando cortos.	162
6.113 Algunos de los cortos detectados.	162
6.114 Taladrado de la PCB.	163
6.115 PCB con orificios taladrados y margenes guillotinados.	163
6.116 Plano detallado del microcontrolador.	164
6.117 Aplicado de la pasta de soldadura.	164
6.118 Colocado del microcontrolador sobre la pasta de soldadura.	165
6.119 Diferentes etapas del proceso de horneado.	166
6.120 Imagen a contra luz de la PCB.	167
6.121 Comprobación de cortos y conductividad de las pistas del microcontrolador. .	167
6.122 Conexionado de las vías de la PCB.	168
6.123 Comienzo del proceso de soldadura.	168
6.124 Integrantes del equipo soldando componentes.	169
6.125 Proceso de soldadura en curso.	169
6.126 Integrantes del equipo soldando componentes.	170
6.127 Primer encendido de la PCB usando un código de prueba del PWM.	171
6.128 Prueba del funcionamiento de la UART.	171
6.129 Prueba del funcionamiento de los servomotores.	172
6.130 PCB tras proceso de atacado fallido.	173
6.131 Proceso de creación de la disolución de atacado.	173
6.132 Intentos de fabricación de la PCB.	174
6.133 Fractura en pista de cobre.	174
6.134 Parcheo de la pista usando hilo de grapinar.	175
6.135 Reconexión del nuevo módulo PWM al puerto.	175
6.136 Soldadura del hilo de reconexión con el nuevo pin.	176
6.137 Esquemático lógico del LM317.	177
6.138 Huella física del LM317.	177
6.139 Encapsulado físico del LM317.	177
6.140 Huella física del LM317 con pines reordenados.	178

6.141	Modificación del circuito de alimentación de los servomotores, marcada en azul.	179
6.142	Puentes entre pistas de los reguladores de tensión LM317.	180
6.143	Reflexión [40].	181
6.144	Motor de corriente continua	181
6.145	Motor paso a paso	182
6.146	Servomotor de corriente continua	182
6.147	Ejemplo genérico de control de posición mediante señal PWM [47].	183
6.148	Servomotor Parallax utilizado [48]	184
7.1.	Diseño propuesto para la interfaz gráfica de usuario.	188
7.2.	Diseño final para la interfaz gráfica de usuario.	189
7.3.	Fragmento del archivo XML que describe la apariencia de la GUI.	189
7.4.	Controles en modo coordenadas angulares.	191
7.5.	Controles en modo coordenadas cartesianas.	192
7.6.	Representaciones gráficas de la posición del brazo.	193
7.7.	Interfaz de usuario final en ejecución	194
7.8.	Vista esquemática del dsPIC33E [41].	209
7.9.	Periodo de la señal <i>Pulse–Width Modulation</i> (PWM) que ha de ser enviada al servomotor [48].	211
7.10.	Valores de <i>duty cycle</i> límites según el fabricante [48].	219
7.11.	Diagrama de secuencia para el intercambio de las claves RSA.	234
8.1.	Algunas de las piezas que no se imprimieron correctamente.	239
8.2.	Entorno de producción de Cura.	240
8.3.	Pieza del <i>pArm</i> casi completamente vertical.	241
8.4.	Pieza del <i>pArm</i> en otra posición para que sea más sencilla de imprimir.	242
8.5.	El “techo” de la pieza se marca de color rojo.	242
8.6.	Con el material de soporte la pieza es imprimible.	243
8.7.	Vista del cabezal de impresión junto con los extrusores de la Ultimaker 3 [59].	244
9.1.	Pieza de prueba para tornillos M4.	248
9.2.	Pieza de prueba para tornillos M3.	248
9.3.	Pieza de prueba para ejes de 4 mm de diámetro.	248

9.4. Distintas pruebas realizadas para obtener el tamaño buscado para métricas M3 y M4 así como los ejes de 4 mm de diámetro.	248
9.5. Modelos 3D del motor empleado junto a una pieza que se ensambla en su eje. .	249
9.6. Cantidad de dientes presentes en los motores empleados.	249
11.1. Diagrama de Gantt general.	254
11.2. Diagrama de Gantt del anteproyecto.	255
11.3. Diagrama de Gantt de la preparación de la convocatoria	256
11.4. Diagrama de Gantt del desarrollo del proyecto	257
11.5. Figura de prueba impresa en 3D.	261
11.6. Extrusores bloqueados y parcialmente dañados por una bola de plástico. . .	262
11.7. Una figura de prueba que necesita Acetato de polivinilo (PVA).	263
11.8. Cómo debería quedar una impresión con PVA según Ultimaker [60].	263
11.9. Comparación de la pieza de prueba con PVA frente a cómo deberían quedar según la web de Ultimaker.	263
11.10 <i>Nozzle</i> completamente obstruído con PVA.	264
11.11 Caja para guardar los plásticos de impresión y mantenerlos protegidos de la humedad.	265
11.12 Los extrusores bloqueados y dañados tras una colisión con una pieza. . . .	266
11.13 Figuras siendo correctamente impresas tras reparar los extrusores y usando material nuevo.	267
11.14 Laboratorio de fabricación de la escuela de telecomunicaciones prestado temporalmente para continuar con el proyecto.	267
13.1. Modo de funcionamiento del “ <i>filastruder</i> ” [62].	278

Índice de cuadros

3.1.	Requisitos del sistema S1.	21
3.2.	Posibles <i>chips</i> que se han planteado para el proyecto.	22
3.3.	Lista de motores propuestos para el sistema S2.	27
3.4.	Caso de uso 0001 - Encender brazo robótico (S2).	30
3.5.	Caso de uso 0002 - Apagar brazo robótico (S2).	31
3.6.	Caso de uso 0003 - Cerrar aplicación.	31
3.7.	Caso de uso 0004 - Cambiar modo de funcionamiento.	31
3.8.	Caso de uso 0005 - Control usando valores numéricos (S1).	32
3.9.	Caso de uso 0006 - Control describiendo trayectorias.	32
3.10.	Caso de uso 0007 - Control usando controles gráficos.	33
3.11.	Caso de uso 0008 - Control usando ratón.	33
3.12.	Caso de uso 0009 - Accionar el <i>end-effector</i>	33
3.13.	Caso de uso 0010 - Activar el modo <i>debug</i>	34
3.14.	Caso de uso 0011 - Desactivar el modo <i>debug</i>	34
5.1.	Longitudes y desviaciones del manipulador μ Arm.	60
5.2.	Tabla inicial de <i>Denavit–Hartenberg</i> para un manipulador basado en el μ Arm parametrizada.	62
5.3.	Tabla de <i>Denavit–Hartenberg</i> para un manipulador basado en el μ Arm parametrizada.	62
7.1.	Métodos de motor y descripciones.	222
7.2.	Métodos de planner y descripciones.	227
7.3.	Órdenes GCode interpretadas por S2.	231
7.4.	Opciones de compilación definidas para S2.	236

8.1. Configuración de la Ultimaker 3 para generar material de soporte en PVA. . .	245
11.1. Tabla completa de presupuestos.	259
11.2. Planificación de los meses de julio y agosto.	260

Definiciones, siglas, y abreviaturas

SDK *Software Development Kit*

ROS *Robot Operating System*

SW *software*

HW *hardware*

pArm *Printed – Arm*

USB *Universal Serial Bus*

ODS Objetivos de Desarrollo Sostenible

OS *Open-Source*

OH *Open-Hardware*

S1 Sistema 1 – ordenador

S2 Sistema 2 – *pArm*

GUI *Graphical User Interface*

GTK *GIMP Toolkit*

SoC *System On Chip*

PWM *Pulse–Width Modulation*

GPIO *General Purpose Input/Output*

UART *Universal Asynchronous Receiver–Transmitter*

RAM *Random Access Memory*

PLA Ácido Poliláctico

ABS Acrilonitrilo Butadieno Estireno

DSP *Digital Signal Processor*

PLL *Phase Loop Lock*

THT *Through–Hole Technology*

SMD *Surface–Mount Device*

PCB *Printed Circuit Board*

ALU *Arithmetic–Logic–Unit*

CPE Copoliéster

PVA Acetato de polivinilo

- *Software Development Kit* (SDK) – colección de herramientas de desarrollo *software* (SW).
- *hand–shake* – en informática, negociación entre pares para establecer de forma dinámica los parámetros de un canal de comunicaciones.
- *Robot Operating System* (ROS) – conjunto de librerías SW que ayudan a construir aplicaciones para robots.
- *Firmware* – SW programado que especifica el orden de ejecución del sistema.

- *Graphical User Interface* (GUI) – siglas que significan “Interfaz Gráfica de Usuario” (en castellano).
- *GIMP Toolkit* (GTK) – biblioteca de componentes gráficos multiplataforma para desarrollar interfaces gráficas de usuario.
- *System On Chip* (SoC) – tecnología de fabricación que integra todos o gran parte de los módulos, de un sistema en un circuito integrado.
- PWM – Señal cuadrada de periodo habitualmente constante, entre flancos de subida, en la que se modula el tiempo a nivel alto
- *General Purpose Input/Output* (GPIO) – pin genérico cuyo comportamiento puede ser controlado en tiempo de ejecución.
- *Universal Asynchronous Receiver–Transmitter* (UART) – estándar de comunicación dúplex.
- Dúplex – término que define a un sistema que es capaz de mantener una comunicación bidireccional, enviando y recibiendo mensajes de forma simultánea.
- Widget – la parte de una GUI (interfaz gráfica de usuario) que permite al usuario interconectar con la aplicación.
- *Random Access Memory* (RAM) – memoria volátil que permite operaciones de acceso aleatorio.
- *Deep-Sleep* – estado de un microcontrolador en el cual consume muy poca cantidad de energía.
- *bit* – unidad mínima de información de un computador digital.
- *Through-Hole Technology* (THT) – tecnología que utiliza agujeros pasantes que se practican en las placas de los circuitos impresos para el montaje de diferentes elementos electrónicos.
- *Surface-Mount Device* (SMD) – tecnología que utiliza componentes de montaje superficial para la inserción de diferentes elementos electrónicos en un circuito impreso.

Resumen

En este documento se va a tratar el desarrollo del *pArm*, un proyecto integral de ingeniería en el que se modela, diseña y construye un brazo robótico utilizando tecnología de impresión 3D como base. El objetivo principal es ofrecer una forma asequible y sencilla para que otra persona pueda replicar el proyecto y adentrarse en el mundo de la robótica por su cuenta.

Para ello, primero se eliciarán los requisitos que permitirán posteriormente modelar y diseñar el sistema de forma fiel. A su vez, se estudiarán las características del sistema *hardware* lo que permitirá desarrollar y construir una placa de control que será la encargada de gestionar los movimientos del brazo robótico.

Además, las fases de diseño anteriores simplifican el proceso de desarrollo del *software* que ejecutarán los sistemas y permitirán abordar el modelo matemático que rige la estructura pantográfica del brazo robótico atendiendo a las limitaciones tanto físicas como del sistema propuesto en sí.

Por otro lado, se modelarán y diseñarán nuevas piezas que permitirán construir el brazo robótico con otros tipos de componentes distintos a los del brazo original así como con la nueva placa de control.

Por último, se proponen futuras líneas de mejora que se consideran interesantes a la hora de completar el proyecto. Se incluyen además en los anexos el código fuente de las aplicaciones desarrolladas ya que se referencia directamente a lo largo del presente documento y para que quede constancia de la marca temporal del mismo.

Abstract

The *p*Arm development, an integral engineering project which models, designs and builds a robotic arm using 3D printing technology as a basis is going to be explained in the following document. The main objective is to offer an affordable and easy way for everyone to replicate this project so they can introduce themselves within robotics community.

Firstly, the requirements will be elicited which, will allow, in further steps of the development, the modeling and the design of the system. Concurrently, the hardware characteristics will be studied in order to allow the development and construction of the board that will handle the movements of the robotic arm.

In addition, the mentioned design steps simplify the software development process alongside the mathematical model, which is defined by both the physical structure itself and the proposed system.

On the other hand, new pieces will be designed in order to make the robotic manipulator compatible with new external componentes, compared to the original ones used in the μ Arm, and the new designed board.

Finally, new improvements that the team considers interesting to complete will be proposed. The annexes are included with the source code developed for both the applications, as they are directly referenced in the present document and for setting a timestamp of it.

Capítulo 1

Motivación y objetivos

1.1. Estado del arte

1.1.1. Desarrollo de la robótica a lo largo de la historia

El mundo de la robótica da acceso a resolver una gran variedad de problemas donde el ser humano estaba limitado físicamente: levantar cargas de gran peso, realizar tareas repetitivas durante tiempos prolongados, etc. Además, como bien se sabe, ha permitido el desarrollo de cadenas de producción en masa para poder desarrollar y crear los productos que usamos diariamente, desde el coche hasta el teléfono móvil.

Desde que se empezó a investigar en este campo, el desarrollo de los brazos robóticos ha sido exponencial: se empezó trabajando con pequeños autómatas hasta el desarrollo de la revolución industrial [1].

Los primeros modelos, como se puede ver en la figura 1.1, empezaron intentando hacer representaciones de las manos humanas. En particular, se crearon un flautista y un tamborilero los cuales eran capaces de tocar los respectivos instrumentos utilizando un complejo sistema de cables y engranajes para poder mover los “dedos” de los músicos.

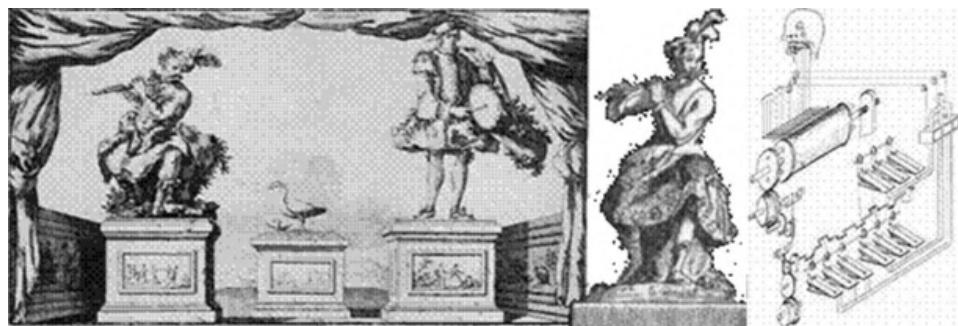


Figura 1.1: Flautista y tamborilero de Vaucanson [2].

Siguiendo con esta idea, se fue mejorando y desarrollando el modelo de imitación de las

articulaciones y los miembros de los humanos, llegando a construir estructuras más complejas y avanzadas, pensadas en su momento para poder tocar el clavicordio mediante un muñeco, como se muestra en la figura 1.2:

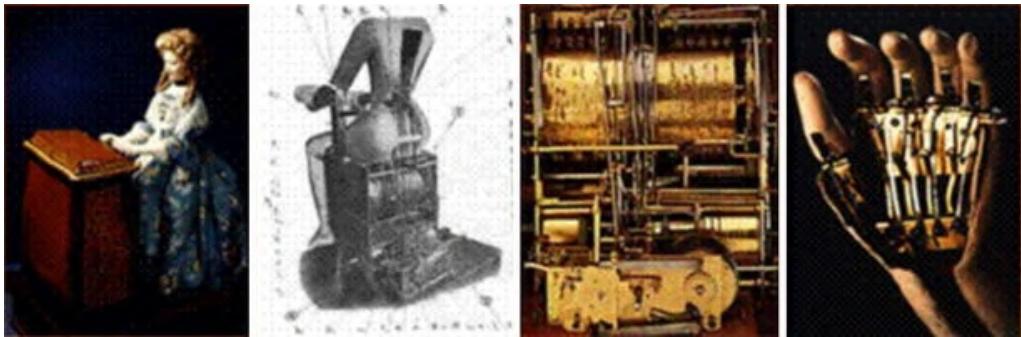


Figura 1.2: En 1774, “lady musician” por Jaquet-Droz [3].

Durante los años siguientes, el proceso se fue refinando hasta el punto de desarrollar un autómata el cual era capaz de jugar al ajedrez, llamado “The Turk” [4], construido en 1769. La estructura comprendía un conjunto de mecanismos los cuales eran controlados por un operador, encargado de realizar los movimientos del brazo izquierdo del autómata.

En la figura 1.3 se puede ver cómo está diseñado el sistema para mover un controlador pantográfico sobre el tablero de juego, controlado por el operador externo antes mencionado:

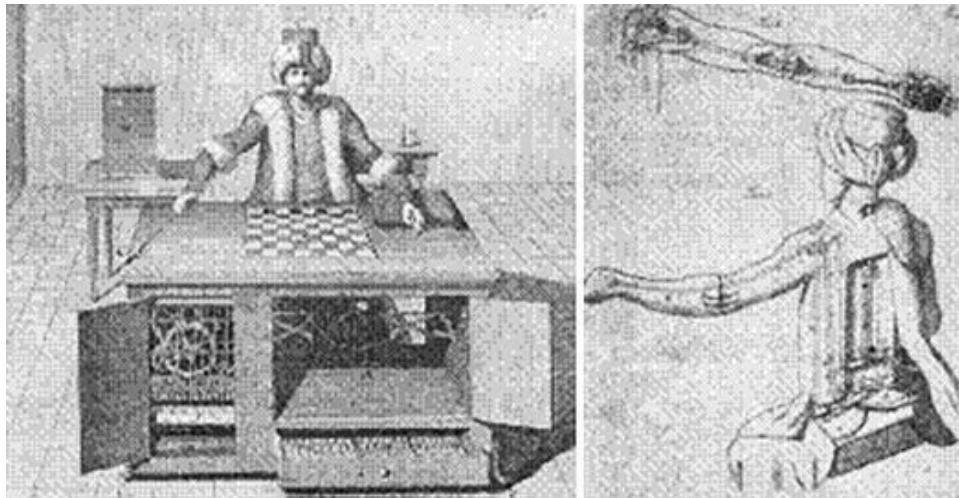


Figura 1.3: “The Turk”, creado por von Kempelen en 1769 [4].

Desde entonces, la robótica ha evolucionado y crecido de manera exponencial. Por una parte, debidas las distintas guerras que han habido en los últimos 200 años, se ha dado un gran impulso a la industria encargada de crear distintos dispositivos con fines de defensa y ataque. En particular, se potenciaron mucho los desarrollos de dispositivos por control remoto, destacando el diseño de NiKola Tesla en 1898 de un barco completamente automatizado, controlado por control remoto y sumergible, como se puede ver en la figura 1.4:

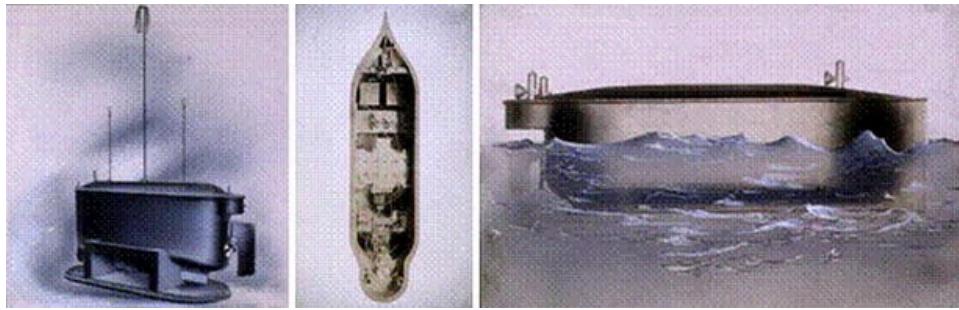


Figura 1.4: Barco a control remoto de Nicola Tesla, en 1898 [5]

Por otro lado, dada la cantidad de bajas de las Primera y Segunda Guerras Mundiales, se empezaron a desarrollar robots que permitieran sustituir a los militares en el campo de batalla, destacando en este campo el robot “Elektro”, creado por la compañía Westinghouse. Dicho robot supuso un gran éxito en la industria de los robots y armamentística, pudiendo moverse completamente, disparar armas, mover elementos faciales para “expresar emociones” e inclusive poder comunicarse.

En la figura 1.5, se puede ver a la izquierda la primera versión “Alpha” y, a la derecha, la versión mejorada “Elektro”:

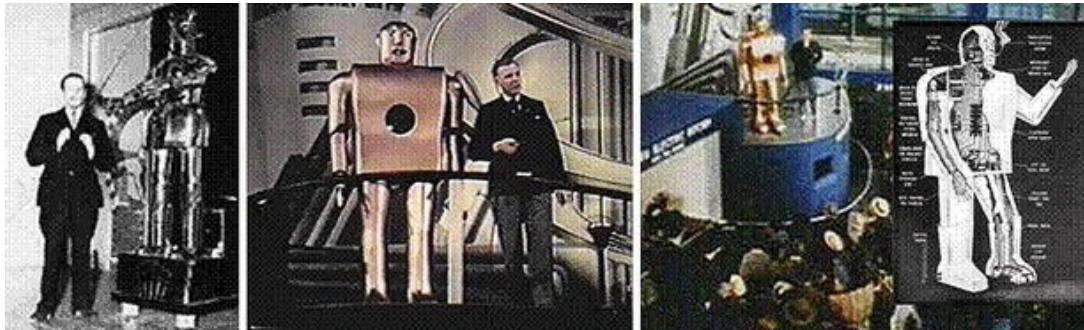


Figura 1.5: “Alpha”, el primer robot diseñado con fines militares y su posterior evolución, “Elektro”.

Toda esta evolución ha desembocado en la robótica moderna, en donde tenemos robots sofisticados y con distintos actuadores, pudiendo interactuar con muchísimos elementos de nuestro entorno y trabajar en distintas fases de producción de cadenas de montaje en serie. Además, se trabaja continuamente para que cada vez los robots puedan realizar más tareas de los humanos, mejorando cada vez más los “*end-effectors*” (controladores del final de los extremos del brazo). En la figura 1.6 se puede ver cómo robots medianamente antiguos (del 2005) ya podían realizar diversas actividades, como interactuar con las personas o tocar un instrumento.

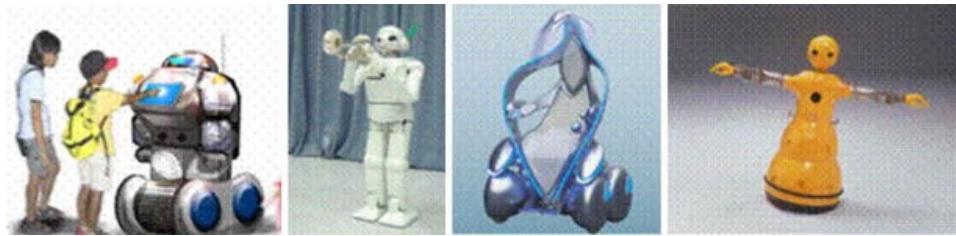


Figura 1.6: Exposición mundial del 2005 en Japón [6].

1.1.2. Los brazos robóticos

Con los avances actuales, el mundo de la robótica ha evolucionado a un nuevo nivel: con la inclusión de los transistores en lugar de las válvulas de vacío se han podido desarrollar circuitos integrados que manejan de manera mucho más sofisticada el control del brazo robótico.

En 1962, la empresa “Unimate” introdujo su primer brazo robótico de carácter industrial. Aproximadamente, se vendieron 8500 unidades. Este hito es importante en tanto a que se valoraron por primera vez los grados de libertad que debían de tener los brazos robóticos.

Estos planteamientos derivaron en distintos robots famosos que incluso siguen en activo hoy día. En 1969, Victor Scheinman, de la Universidad de Standford, desarrolló un brazo robótico que funcionaba alimentado por la electricidad y que se podía mover en los seis ejes, el cual se llamó “el brazo de Standford”. De forma paralela, Marvin Minsky, del MIT, desarrolló un brazo robótico para la investigación naval, para exploración submarina. En particular, el brazo tenía veinte grados de libertad ya que funcionaba mediante electricidad impulsando sistemas hidráulicos. Más tarde, Scheinman continuó desarrollando brazos robóticos, creando el “*Programmable Universal Machine for Assembly*”, más conocido como PUMA.

En la actualidad, los brazos robóticos se desarrollan y diseñan para seguir la estructura física del cuerpo humano (ver figura 1.7).

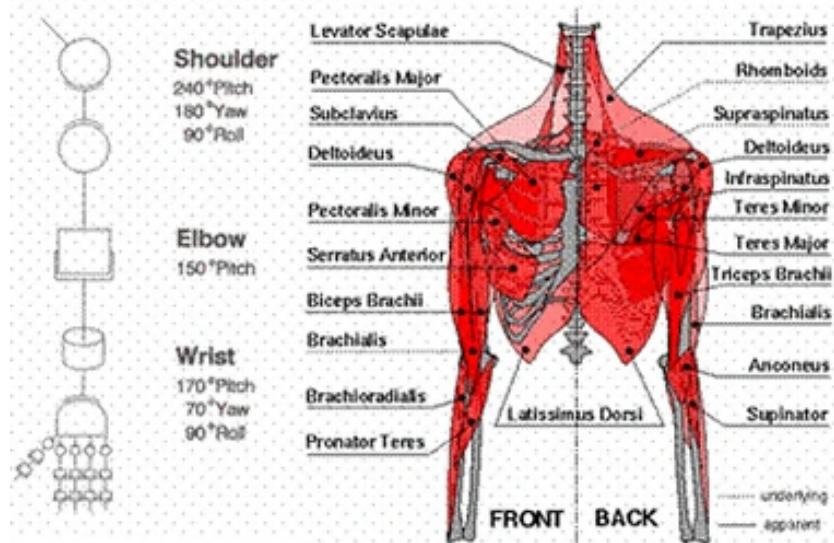


Figura 1.7: Grados de libertad de un brazo robótico y estructura del cuerpo humano [1].

De la estructura anterior, se deducen las siguientes partes:

- Articulación del hombro: dispone de tres grados de libertad que permiten subir y bajar, ir a la izquierda y derecha, y rotar sobre sí mismo.
- Articulación del codo: el codo permite extender, contraer y reorientar tanto la muñeca como la mano. Por lo general, se estima la extensión del codo en unos 150° .
- La muñeca: compone el último elemento del brazo robótico antes de llegar al “*end-effector*”. Es de los elementos más importantes debido a su gran capacidad de movimiento en las tres dimensiones. Sin esta articulación, el brazo robótico se asemejaría en funcionalidad a un robot pantográfico. Cada vez más, las articulaciones de la muñeca se vuelven complejas y sofisticadas. La muñeca humana, por ejemplo, puede moverse 45° desde el centro, pero se reduce mucho la capacidad de rotación de la misma. En la actualidad se está investigando cómo poder mejorar la relación de movimientos para permitir una mayor movilidad, pero las singularidades siguen siendo un gran problema. Por ejemplo, el robot quirúrgico da Vinci, pese a lo avanzado que pueda parecer, tiene problemas de bloqueo de las muñecas cuando se acerca a posiciones singulares.
- La mano: supone un “*end-effector*” diferenciado que define el propósito y la capacidad del brazo robótico. La mano es una herramienta capaz de realizar múltiples acciones muy variadas entre sí. Actualmente, se sigue investigando de forma activa sobre ello para intentar implementar controles sensoriales, de presión y de movimiento en los “*end-effector*” de los robots.

1.1.3. La actualidad

Durante los dos últimos decenios la robótica ha evolucionado de manera exponencial. Se ha trabajado de forma activa en mejorar ciertas condiciones industriales, espaciales y en el día a día de las personas. En el 2001 se puso en la ISS el brazo robótico “Canadarm2”, conocido oficialmente como “*Space Station Remote Manipulator System*” (SSRMS) (figura 1.8).



Figura 1.8: Vista exterior del Canadarm2 [7].



Figura 1.9: Rover “Spirit”, desarrollado por la NASA y desplegado en 2004 [8].

Además, en 2005, se desplegaron en Marte los rovers “Spirit” (figura 1.9) y “Opportunity” (figura 1.11).

El primero supuso un gran avance de la ingeniería, ya que crearon un robot teleoperado para enviarlo a un terreno muy hostil. Con las seis ruedas que tenía permitía una movilidad bastante elevada en terrenos muy desiguales, siendo todas ellas motrices e independientes entre sí y, en particular, las cuatro de los extremos direccionales (detalle en la figura 1.9). Además, la fisionomía de las mismas y su elevación permitía que el dispositivo se desplazara por distintos tipos de terreno de una manera óptima, utilizando un sistema de amortiguación conocido como “*rocker–bogie*”. Dicho sistema se caracteriza por no utilizar una suspensión hidráulica sino un diferencial en el centro del vehículo, garantizando así que el cuerpo del mismo siempre se encuentra con una inclinación igual a la mitad que presentan ambos (ver figura 1.10).

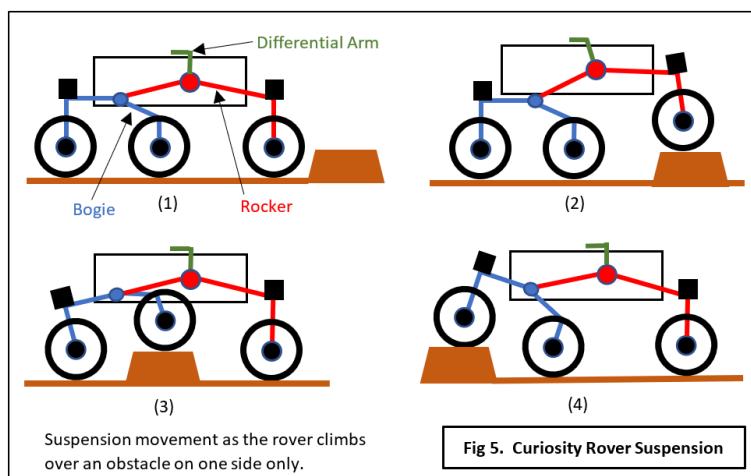


Figura 1.10: Modo de funcionamiento del sistema “*rocker–bogie*”, desarrollado por la NASA para sus rover [9].



Figura 1.11: Rover “Opportunity”, desarrollado por la NASA y desplegado en 2004 [10].

Por otra parte, se puede apreciar cómo el “Opportunity” tenía una estructura bastante similar al “Spirit” pero añadía alguna que otra mejora. La diferencia principal entre ambos era el lugar de aterrizaje (figura 1.12), ya que ambos servían para recorrer distintos puntos de la superficie marciana para recopilar datos y tomar muestras.

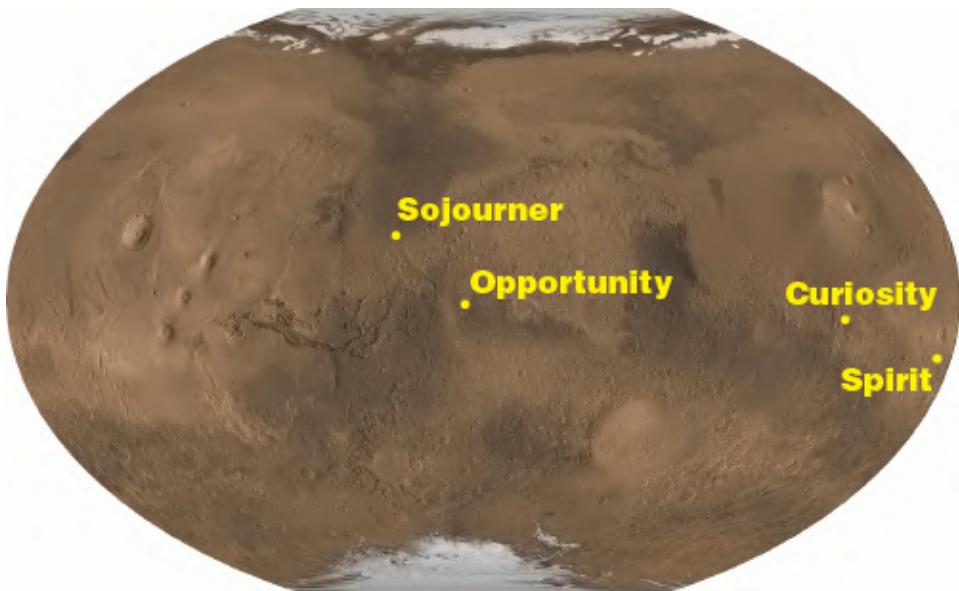


Figura 1.12: Lugares de aterrizaje de los rovers de la misión espacial a Marte [11].

La misión del “Spirit” duró 2623 soles frente a los 90 inicialmente planeados [8], y la misión del “Opportunity” duró 5352 soles frente a los también 90 inicialmente planteados [10]. Esto se traduce en 2695 días terrestres (6 años, 9 meses y 12 días) y 5498 días terrestres (15 años) respectivamente.

Por otra parte, los robots de aplicación doméstica también han ido creciendo cada vez más, naciendo en 2002 el popular Roomba (figura 1.13), de la empresa iRobot, o distintos brazos

articulados para, por ejemplo, ayudar a personas que carezcan de dichos miembros o asistir a personas mayores en sus hogares.



Figura 1.13: Robot Roomba en la estación de carga [12].

Además, la industria de los coches autónomos también ha crecido de forma exponencial, sobre todo con la llegada al mercado en 2003 de Tesla Motors y sus coches eléctricos que disponen del sistema “Autopilot” (figura 1.14), encontrándose actualmente en el nivel 2 de autonomía según la lista del SAE [13][14]. Los avances en esta industria permiten el control completo del vehículo, tomando decisiones en tiempo real sobre la suspensión, el giro de las ruedas, el control de tracción, etc. permitiendo además compartir información e ir mejorando el mundo de la robótica y de la industria del automóvil.



Figura 1.14: Lo que ve un Tesla cuando está en conducción autónoma nivel 2 [13].

Por otro lado, se ha avanzado mucho a nivel de robots militares y humanoides. Un ejemplo de ello es la empresa *Boston Dynamics*, la cual ha desarrollado múltiples robots con un grado

de libertad bastante elevado. Dichos robots se caracterizan por una gran estabilidad y la amplia variedad de movimientos que pueden realizar: andar, correr, saltar, subir escaleras, abrir puertas, etc. Actualmente, dos robots son los principales: “*Big-Dog*” (figura 1.15) y “*Atlas*” (figura 1.16).



Figura 1.15: Robot “*Big-Dog*” de Boston Dynamics [15].

El primero tiene un amplio uso militar: debido a su forma, es muy útil para llevar cargas pesadas durante largas distancias. Además, la configuración cuadrúpeda y los avanzados sistemas *software* y *hardware* del que dispone permite que el robot sea estable incluso en condiciones bastante complicadas, como puede ser un suelo helado.



Figura 1.16: Robot “*Atlas*” de Boston Dynamics [15].

“*Atlas*”, por otra parte, es un trabajo en progreso que permitirá, en un futuro, poder utilizarlo con fines militares y domésticos. Puede llevar objetos pesados con sus brazos y moverse con mucha agilidad, haciéndolo un robot muy polivalente según se quiera utilizar.

Finalmente, el desarrollo de los brazos articulados con múltiples finalidades también ha evolucionado mucho. Desde robots industriales tales como los desarrollados por la empresa KUKA, como el “KR-1000 Titan” (figura 1.17), o el brazo “M-2000”, de FANUC hasta brazos más pequeños con propósitos educacionales, como el μ Arm, de UFACTORY (figura 1.18).



Figura 1.17: Robot “KR-1000 Titan” de KUKA [16].



Figura 1.18: Robot μ Arm de UFACTORY [17].

Estos robots tienen múltiples propósitos: el primero, levantar y trasladar piezas muy grandes y pesadas con muchísima precisión. El segundo, disponer de un robot para poder aprender y utilizarlo para tareas como, por ejemplo, impresión 3D. Este último se desarrolló con la intención de ser accesible e intentar introducir en el mundo de la robótica a aquellos que

pudieran estar interesados, pero su alto coste impide el acceso a aquellos con una capacidad adquisitiva más baja.

1.2. Motivaciones y objetivos del desarrollo del proyecto

Durante el primer semestre del cuarto curso de Ingeniería de Computadores, hay dos asignaturas las cuales propiciaron el desarrollo de este proyecto: robótica y sistemas empotrados.

Con la primera, se vio la potencia de los brazos robóticos y se desarrolló un estudio sobre un manipulador del cual se ha hablado con anterioridad: el μ Arm [18]. Con la segunda asignatura, se vio cómo con sistemas de aplicación específica se podían desarrollar circuitos con suficiente potencia como para poder tomar el control de otros dispositivos más grandes y complejos aplicando la lógica estudiada a lo largo de los años.

Se tomaron en cuenta los conocimientos obtenidos de las asignaturas anteriores para empezar un desarrollo que uniera esos dos campos: diseñar un brazo robótico impreso en 3D el cual estuviera gobernado por un microcontrolador en una placa de control de propósito específico. Para ello, se parte de los diseños 3D provistos en la web de UFACTORY [19] para su posterior adaptación y reutilización. En lo referente a la placa de control, el brazo original utiliza una placa Arduino Mega [20], por lo que se decidió (para dar mayor peso a la parte de ingeniería e intentar reducir costes) diseñar e implementar al completo una placa con otro microcontrolador para gobernar dicho brazo robótico.

Principalmente, este trabajo se desarrolla bajo las dos perspectivas siguientes:

- Aplicar en un proyecto de ingeniería real las competencias y técnicas que se han ido aprendiendo a lo largo de los distintos cursos del Grado de Ingeniería de Computadores (61CI).
- Construir una alternativa asequible y accesible, tanto a niveles de *Open-Source* (OS) y *Open-Hardware* (OH), de un brazo robótico de manera que cualquier persona interesada en este ámbito de la ingeniería pueda introducirse y aprender, e incluso montar el brazo por sí mismo.

Para la primera perspectiva, la forma de afrontarla y desarrollarla está detallada en el punto siguiente (1.3). Para la perspectiva de desarrollo de un producto accesible y asequible, se partió desde el abaratamiento de costes: el brazo original μ Arm se encuentra disponible en venta por aproximadamente \$749. Dicho precio, pese a no ser especialmente elevado, impide a muchas personas el acceso a la robótica en un brazo que pretende ser educativo y útil. Por este motivo, se desarrolla este proyecto principalmente para resultar barato. Además, siguiendo con la política del brazo original, el proyecto se desarrolla bajo las premisas OS y OH, de manera que inclusive para aquellos que no puedan imprimir el brazo 3D se dispone de forma universal todos los diagramas, planos, esquemas, diseños y código fuente que se ha empleado para acabar desarrollando el brazo μ Arm.

1.3. Metodología

Dado que se pretende hacer un desarrollo de ingeniería completo, la metodología es un punto muy importante en este proyecto.

Primeramente, antes de hacer ningún tipo de desarrollo o implementación, se hizo un estudio del problema, y de lo que se pretendía obtener. Por una parte, se comprobó hasta qué punto podrían ser reutilizables los diseños provistos por UFACTORY en su página de GitHub. Esto permitió diseñar elementos nuevos, adaptar los recursos a lo que hay disponible, etc. Por otro lado, se estudió qué placa de control se quería utilizar. Debido a la familiaridad de los integrantes del equipo con los componentes de la familia “Microchip”, se plantearon distintas alternativas:

- Controladores de gama media de la familia PIC16F.
- Controladores de gama superior de la familia PIC32F.
- Procesadores digitales de la señal, de la familia dsPIC.

Se optó por utilizar los últimos mencionados, ya que disponen de un apoyo específico matricial y matemático para poder agilizar las operaciones realizadas, de forma que los cálculos necesarios se podrían realizar íntegramente en el microcontrolador.

Además, se estudió cómo se quería plantear la comunicación con el brazo: de forma completamente autónoma o mediante un equipo auxiliar. Para evitar la complejidad extra que habría surgido de desarrollar un sistema de control completamente autónomo del brazo por sí solo, se decidió conectarlo a un equipo auxiliar externo que lo gobierne, y que el *p*Arm no funcione si no es estando conectado.

Una vez se definieron estos puntos, se pasó al diseño lógico del sistema que deberán tener tanto S1 como S2, mediante especificación de requisitos, diagramas lógicos, diagramas físicos, diagramas de diseño, etc. Esta parte del proyecto es de las más importantes, ya que sustenta las ideas y las funcionalidades que habrán de estar presentes en el producto final. Mientras tanto, se han ido desarrollando pruebas y mecanismos de control para ir asegurando la correcta calidad del trabajo.

Finalmente, una vez completada esta parte de diseño, se pasa a la implementación real. Dada la situación del COVID-19, esta fase de implementación se ha retrasado sobremanera, impidiendo pues presentar el proyecto en el mes de julio, como estaba previsto, y teniendo que acotar los plazos de implementación a, posiblemente, un mes. En el momento de implementación, se creará la placa diseñada y se empezará la impresión de distintas piezas 3D, para comprobar su funcionamiento en conjunto e ir solucionando los posibles errores que aparezcan.

Durante este proceso, se ha ido desarrollando además de forma paralela la memoria que acompaña el proyecto, permitiendo ir actualizándola con los últimos cambios y mejoras que se han considerado de interés para aparecer descritas.

Capítulo 2

Explicación de la estructura del proyecto

El diseño y construcción de un brazo robótico es un proceso multidisciplinar en el que se deben emplear diversas áreas del conocimiento. Desde un primer momento, este proyecto se postuló como un proyecto completo de ingeniería, y es precisamente por eso que está dividido en varios bloques, los cuales desempeñan una función clave en el desarrollo correcto del mismo.

El proyecto está dividido en tres grandes bloques: modelo matemático, elementos *hardware* y elementos *software*. Cada una de estas partes se encuentra a su vez subdividida en diferentes partes o hitos. Sin embargo, no es necesario describirlos con tanta precisión por el momento para poder comprender la estructura completa del proyecto.

Cabe destacar que, desde un punto de vista de ingeniería, a cada uno de los grandes bloques anteriormente mencionados se le puede asociar a una función dentro del proyecto:

- El modelo matemático es la parte más teórica del proyecto y su función es la de aportar una base formal y lógica que permita predecir y controlar el comportamiento del brazo robótico. Este bloque se encuentra ubicado en el apartado 5 de la memoria.
- Los elementos *hardware* del proyecto constituyen la realidad física del brazo robótico y están estrechamente relacionados con la construcción del mismo, así como con el control de los actuadores y demás componentes físicos presentes en el brazo robótico. Este bloque se encuentra ubicado en el apartado 6 de la memoria.
- Los elementos *software* del proyecto constituyen el principal mecanismo para implementar el modelo matemático y la lógica de funcionamiento del sistema completo mediante la programación de los elementos *hardware* y de los sistemas que necesitan comunicarse con los mismos. Este bloque se encuentra ubicado en el apartado 7 de la memoria.

En cada uno de los bloques de desarrollo anteriores, ya sean *hardware* o *software* y requieran construcción física o implementación mediante programación, se contempla la realización de pruebas de funcionamiento así como las revisiones pertinentes.

Es importante remarcar que, debido a la complejidad del sistema, el mismo está dividido en dos subsistemas que aglutinan funcionalidades vitales para el correcto funcionamiento del manipulador robótico:

- S1: está formado por la interfaz de usuario que se ejecuta sobre un computador de propósito general; esta interfaz es gráfica y le permite controlar los movimientos del brazo robótico, así como visualizar el estado de los parámetros del mismo. Este subsistema es esencialmente un elemento software y está descrito en el apartado 6.1 de la memoria.
- S2: está formado por la estructura física del manipulador, los actuadores y la placa de circuito impreso de control. Este subsistema combina elementos *hardware* y *software*, así como conceptos del modelo matemático. Los elementos *software* se describen en el apartado 6.2 de la memoria, mientras que los elementos *hardware* se describen en el apartado 5 de la memoria.

A continuación, se describen de forma detallada todos los bloques descritos anteriormente y a su vez, se mencionan las principales subdivisiones de cada uno de ellos.

2.1. Matemáticas

Los modelos matemáticos aplicados a proyectos de manipuladores robóticos son usados principalmente para realizar cálculos relacionados con los aspectos cinemáticos y dinámicos de los mismos.

Los aspectos cinemáticos de un manipulador robótico describen cómo es el movimiento y las trayectorias del mismo sin tener en cuenta las fuerzas que lo afectan, mientras que los aspectos dinámicos describen cómo se ve afectado dicho movimiento en función de las fuerzas que actúan sobre él.

Ambos aspectos anteriormente mencionados deben de ser descritos mediante un modelo matemático que permita realizar cálculos sobre los movimientos del manipulador.

En este proyecto, se ha llevado a cabo únicamente el modelo cinemático, dado que debido a las características físicas del prototipo a construir, es decir, velocidades de desplazamiento, peso de las articulaciones o masa máxima de carga; se ha concluido que el modelo dinámico no aportaría demasiada información útil para llevar a cabo el control del manipulador. Cabe destacar que el modelo dinámico suele presentar una complejidad mucho mas elevada que el modelo cinemático en términos matemáticos y por ello se ha desecharido la posibilidad de llevarlo a cabo.

Desde un punto de vista técnico, el modelo cinemático de un manipulador robótico expresa cuál es la posición del extremo del mismo con respecto al tiempo y en función de la posición de las articulaciones del mismo. Normalmente, los brazos robóticos se pueden describir matemáticamente mediante el concepto de cadena cinemática:

Tal y como se puede apreciar en la figura 2.1, las articulaciones pueden rotar y permiten la movilidad de cada uno de los segmentos del manipulador. Dado que estas articulaciones rotan, su posición se expresa numéricamente mediante unidades angulares. El concepto de cadena cinemática hace referencia a que, dado que cada una de las articulaciones esta unida a la siguiente mediante un segmento, se genera una cadena de movimientos en la que la

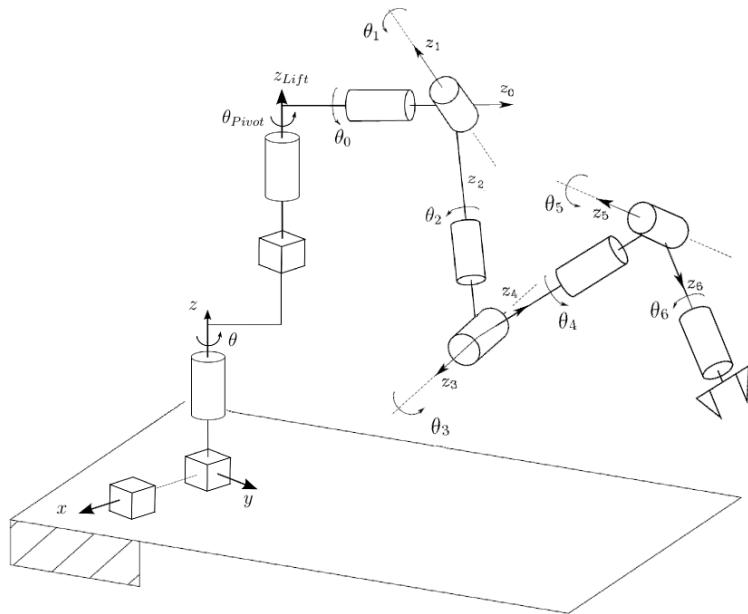


Figura 2.1: Ejemplo de cadena cinemática [21].

posición espacial de cada una de las articulaciones se ve afectada por la posición angular de las anteriores.

Aplicando este principio, el modelo cinemático expresa matemáticamente la posición cartesiana del extremo del robot en función de las coordenadas angulares de las articulaciones. Existen pues dos perspectivas del modelo cinemático:

- El modelo de cinemática directa expresa la posición espacial del extremo del manipulador en función de las coordenadas angulares de las articulaciones.
- El modelo de cinemática inversa expresa las coordenadas angulares de las articulaciones en función de las coordenadas cartesianas del extremo del manipulador.

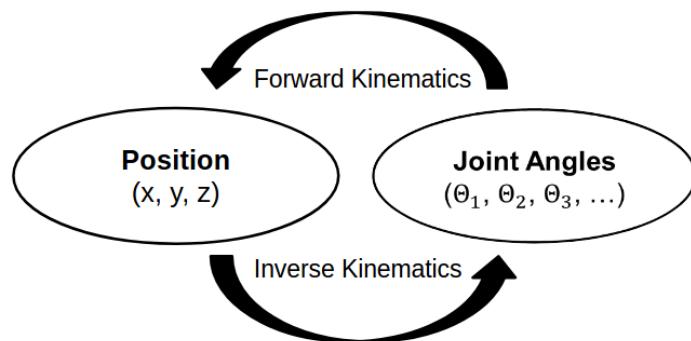


Figura 2.2: Diagrama del modelo cinemático [22].

En conclusión, el modelo matemático conforma la base teórica y formal que permite realizar el estudio de los movimientos del manipulador y es por ello que representa un bloque crucial dentro del proyecto.

2.2. *Hardware*

Los elementos *hardware* conforman la implementación física del manipulador y de todos los componentes empleados para controlarlo.

En términos generales, el *hardware* usado en el proyecto se descompone en diferentes elementos:

- Impresión 3D de la estructura física del manipulador.
- Motores empleados en el manipulador.
- Desarrollo de la placa de circuito impreso de control y microcontrolador empleado.
- Comunicaciones entre los diferentes subsistemas.

En primer lugar, la impresión 3D es la tecnología seleccionada para la fabricación de la estructura física del manipulador debido a su bajo coste y accesibilidad. Esta parte del proyecto se centra en llevar a cabo la fabricación y construcción de la estructura física del manipulador, así como su ensamblado y testeo. Este apartado se ubica en el apartado 5.1 de la memoria.

En segundo lugar, la elección de los motores que dotan de movilidad a la estructura es una decisión crucial y que depende principalmente de cuales sean las características físicas del manipulador, así como de las tareas que se quieran realizar con el mismo. Existen numerosas opciones en cuanto a motores, por ejemplo, motores DC, servomotores, motores paso a paso, etc. Este apartado se ubica en el apartado 5.5 de la memoria.

En tercer lugar, el desarrollo de la PCB de control y elección del microcontrolador representan la parte más importante dentro del bloque hardware del proyecto. El objetivo principal de esta parte del proyecto es llevar a cabo el diseño y construcción de una PCB personalizada, adaptada especialmente a los actuadores y microcontrolador usados para llevar a cabo el control del movimiento del manipulador. Se considera que esta PCB representa uno de los elementos hardware esenciales para el correcto desarrollo del proyecto. Este apartado se ubica en el apartado 5.3 de la memoria.

En último lugar, el diseño e implementación de los canales de comunicación y protocolos necesarios para comunicar los dos subsistemas principales requiere desarrollo hardware y software de forma equitativa, además, también representa uno de los elementos cruciales del proyecto. Este apartado se ubica en el apartado 5.4 de la memoria.

2.3. *Software*

Los elementos *software* del proyecto abordan los siguientes aspectos:

- Desarrollo de la aplicación de control del brazo robótico, implementada mediante una interfaz gráfica de usuario para garantizar su accesibilidad y facilidad de uso. Esta implementación se lleva a cabo en S1.
- Programación del microcontrolador e implementación del modelo matemático en la práctica con el objetivo de controlar los movimientos del brazo robótico. Esta implementación se lleva a cabo en S2.

En primer lugar, mediante el desarrollo de la aplicación de usuario se busca ofrecer una forma de controlar los movimientos del robot de forma fácil y accesible, para ello se ha desarrollado una interfaz de usuario que se ejecuta en un ordenador auxiliar. Desde esta aplicación el usuario puede controlar los movimientos del robot, además de monitorizar el estado del mismo. Las órdenes dadas por el usuario son enviadas al microcontrolador para su ejecución mediante los canales de comunicación mencionados anteriormente. Este desarrollo se ha llevado a cabo mediante el lenguaje de programación Python. Este apartado se ubica en el apartado 6.1 de la memoria.

En segundo lugar, la programación del microcontrolador representa una parte esencial del proyecto, ya que toda la lógica de funcionamiento y control de los actuadores del brazo robótico se lleva a cabo en el mismo. Es por ello que la labor principal del microcontrolador es orquestar el funcionamiento de los actuadores, así como de realizar el computo necesario para transformar las órdenes del usuario en movimientos consecuentes del brazo robótico. La programación del microcontrolador se ha llevado a cabo mediante el lenguaje C. Este apartado se ubica en el apartado 6.2 de la memoria.

Capítulo 3

Especificación de requisitos

3.1. Introducción

El μ Arm es un brazo robótico creado por la compañía UFACTORY¹ el cual se ha diseñado con propósito principalmente didáctico.

En la actualidad, se puede obtener uno a través de su página web o de proveedores externos, pero no está previsto fabricar más, por lo que en un tiempo estará fuera de existencias.

Debido a su propósito didáctico, todos los recursos sobre el manipulador son de código libre, por lo que resultan accesibles a cualquiera que los necesite. Entre otros, se encuentran²:

- *Firmware* del μ Arm Swift Pro.
- SDK de Python para el μ Arm Swift Pro.
- *Firmware* que maneja el controlador del brazo.
- ROS para el μ Arm Swift Pro.
- Distintos ejemplos para toda la gama de brazos robóticos.
- μ Arm *Creator Studio*.
- Visión esquemática de las conexiones de la placa Arduino.
- Modelos 3D del brazo robótico.
- Guías de usuario, desarrollador y especificaciones técnicas.

Aprovechando dichos recursos, se pretende desarrollar un brazo robótico basado en el μ Arm que esté impreso en 3D y sea controlado por un microcontrolador en conjunción con un ordenador cualquiera. Aprovechando los recursos provistos por UFACTORY, se busca que el

¹<https://www.ufactory.cc/#/en/uarmswift>

²todos los elementos descritos se encuentran disponibles tanto en GitHub como en la web de UFACTORY

brazo desarrollado sea más barato de construir (frente a los casi 800€ que cuesta el original) y que pueda ser desarrollado por cualquiera con acceso a Internet y a los recursos necesarios, a saber, una impresora en 3D y un SW de impresión en 3D.

3.1.1. Propósito

El propósito de este documento es establecer un punto de partida claro y conciso que permita empezar el desarrollo del brazo robótico sabiendo los puntos primordiales del mismo. A su vez, también pretende establecer ciertos puntos que se consideran importantes e incluso necesarios para poder continuar el desarrollo del sistema en un futuro, implementando nuevas funciones o arreglando errores que pudieran existir.

Este documento está dirigido a ingenieros que quieran llevar a cabo su propia implementación del brazo robótico o que quieran conocer la estructura en la que se basa el proyecto, así como las necesidades del mismo y las adiciones extraordinarias que se han incluido. A su vez, se pretende que sea accesible a cualquiera que pretenda iniciarse en el mundo de la robótica y que busque estudiar y aprender sobre el brazo robótico.

3.1.2. Alcance

El objetivo principal de este proyecto fin de grado es construir una brazo robótico similar al manipulador μ Arm, al cual se le ha asignado el nombre *Printed – Arm* (*pArm*).

Este brazo robótico debe ser capaz de moverse libremente dentro de su campo de movimiento, el cual está limitado por su estructura física. Además, el *pArm* debe ser capaz de coger, transportar y depositar objetos de poco peso y, en consecuencia, debe ser capaz de describir trayectorias previamente planificadas o calculadas en el momento.

Es importante destacar que, dado que el brazo robótico *pArm* no está sensorizado, este no será capaz de moverse de forma completamente autónoma ni de imitar movimientos realizados por el usuario.

Cabe destacar que el brazo robótico está controlado mediante un microcontrolador. Sin embargo, las instrucciones de movimiento y trayectorias no se computan, en principio, en el mismo sino en un ordenador auxiliar.

Debido a la estructura física, tamaño y materiales de fabricación, el *pArm* no es un brazo robótico pensado para la realización de tareas industriales ni para el transporte de cargas pesadas.

En relación a lo anteriormente mencionado, la aplicación principal del *pArm* es didáctica, dado que se busca construir un brazo robótico económico y sencillo que facilite la introducción de los usuarios a este tipo de tecnologías.

3.1.3. Visión global

En las siguientes páginas se pasa a explicar los distintos detalles del sistema que debe construirse, respetándose la siguiente estructura:

- Perspectiva del producto.
- Funciones del producto.
- Características del producto.
- Restricciones.
- Supuestos y dependencias.
- Requisitos propuestos.

3.2. Descripción general

3.2.1. Perspectiva del producto

El *pArm* se basa en el trabajo inicial del μ Arm, no utilizando directamente lo desarrollado por la empresa UFACTORY sino aprovechando el trabajo ya realizado y los recursos disponibles para estudiarlos.

Por otra parte, el *pArm* es dependiente de otro sistema que lo controle, ya que no se plantea como sistema autónomo. Por consiguiente, se proponen diversos métodos de conexión entre el brazo y dicho sistema. Por ejemplo, se puede utilizar el puerto serie *Universal Serial Bus* (USB) o bien comunicaciones inalámbricas, como *Bluetooth* y *WiFi*. Además, debido a su disponibilidad multiplataforma, se propone el uso de Python como alternativa de programación.

De ahora en adelante, se denominará “S1” al equipo que controla al *pArm*; y “S2” al brazo robótico en sí.

Para este proyecto, se ha de desarrollar el SW que se ejecutará en S1, así como el SW y el *hardware* (HW) que irán en S2. También habrá que adaptar la estructura mecánica para que esta pueda ser impresa en 3D.

3.2.2. Interfaz del sistema

En un principio, el sistema estará dividido en dos módulos:

S1

S1 consiste en un equipo el cual controlará el brazo robótico (S2). Para ello, tal y como se planteó anteriormente, se propone como lenguaje de programación Python, el cual soporta la ejecución con GUI.

En lo referente al sistema operativo, al ser una aplicación en Python la cual es multiplataforma, no se define ninguna restricción respecto al mismo.

Finalmente, se plantea la conexión con S2 utilizando el puerto serie USB, por lo que también será necesario que el equipo anfitrión S1 disponga de una conexión de ese estilo.

En resumen (ver la tabla 3.1):

Componente	Función	Restricciones
Sistema Operativo	Hospedar y ejecutar la aplicación Python que controlará el brazo robótico.	Debe poder ejecutar aplicaciones Python con GUI, por ejemplo, GTK.
<i>Conexión con S2</i>	Permitir la comunicación con el sistema S2 en modo dúplex.	Velocidad adaptable (<i>baud-rate</i>) y capacidad para gran ancho de banda.
Python	Control del sistema S2 y monitoreo del estado del mismo.	Versión Python $\geq 3.6.*$

Cuadro 3.1: Requisitos del sistema S1.

En principio, no será necesaria la conexión a Internet, pero tampoco se descarta el uso de la misma a la hora de poder recibir actualizaciones o en lo referente a futuras mejoras.

S2

Para el sistema S2 no se ha pensado en ningún microprocesador ni SoC en particular, pero se han contemplado algunos que cumplen con las características requeridas (ver la tabla 3.2).

Será necesario que el circuito escogido disponga de algún tipo de entrada de las propuestas para la comunicación con S1. Debido a la característica descrita en la tabla 3.1 sobre la interfaz de comunicación, no será estrictamente necesario que la velocidad sea adaptable (ya que se asume que se adaptará en S1); sin embargo, sí será requisito fundamental que la conexión sea dúplex y que soporte gran cantidad de datos con las menores pérdidas posibles.

Por otra parte, el microcontrolador deberá poder modular señales PWM para controlar los distintos motores de los que dispondrá el brazo. Sin embargo, en caso de que finalmente el *chip* escogido no disponga de dicha modulación, se podrán usar motores que cuenten con un *driver* que permitan controlarlos usando señales digitales y/o analógicas.

Teniendo en cuenta lo anterior, se plantea el uso de los siguientes dispositivos (ver tabla 3.2):

Placa	Ventajas	Desventajas	ID RS-Online y precio
ESP8266	SoC bastante barato (5 €) con conexión WiFi y modo de bajo consumo	Señal PWM generada por SW; poca cantidad de GPIO (6).	124-5505 – 19,29 €
ESP32	SoC con procesador de dos núcleos que permite comunicaciones WiFi y Bluetooth	No cuenta con GPIO pero permite la comunicación mediante el protocolo I ² C.	188-5441 – 25,29 €
PIC16F18326-I/P	Microcontrolador de 8 bits de baja potencia de consumo y bajo precio con capacidad de modular hasta dos señales PWM y con más memoria RAM que otros componentes de su familia. Finalmente, cuenta con bastantes salidas GPIO, suficientes como para añadir más componentes al sistema.	No está integrada en una placa (SoC) por lo que habría que hacer toda la lógica del diseño HW. No dispone de conexiones de red (aunque no son necesarias) y la capacidad de cómputo, en comparación con las otras propuestas, es menor.	124-1554 – 1,375 €
dsPIC33EP***GM604	Microcontrolador de 16 bits que cuenta con un procesador digital de señales, permitiendo realizar operaciones matriciales rápidamente. Además, cuenta con hasta 6 señales PWM y múltiples GPIO.	Al igual que el componente anterior, no está integrado en una placa SoC por lo que habría que diseñar toda la PCB que contenga el sistema.	825-1023 – 5,89 €

Cuadro 3.2: Posibles *chips* que se han planteado para el proyecto.

Finalmente, el equipo de desarrollo decide usar el dsPIC33EP512GM604-I ya que está disponible entre las distintas opciones que la universidad pone a su disposición y por tanto no hay que soportar el coste de este en el presupuesto del proyecto. Por otro lado, cumple con los requisitos técnicos que el equipo de desarrollo ha supuesto necesarios.

3.2.3. Interfaz de usuario

El usuario final del producto solamente interactuara de manera directa con S1. Para que esta interacción sea posible, se desarrollara un panel de control que permita al usuario definir movimientos que el robot deberá realizar. El panel de control se mostrará en una sola pantalla y permitirá al usuario, mediante una interfaz gráfica sencilla, mover de manera independiente cada uno de los motores del robot, o bien mediante el uso del ratón, describir trayectorias que el robot realizará en tiempo real replicando el movimiento del ratón.

Además, también se podrá desplazar el robot indicando la posición $\{x, y, z\}$ referente al *end-effector*. Se sugiere una interfaz que siga el siguiente diseño:

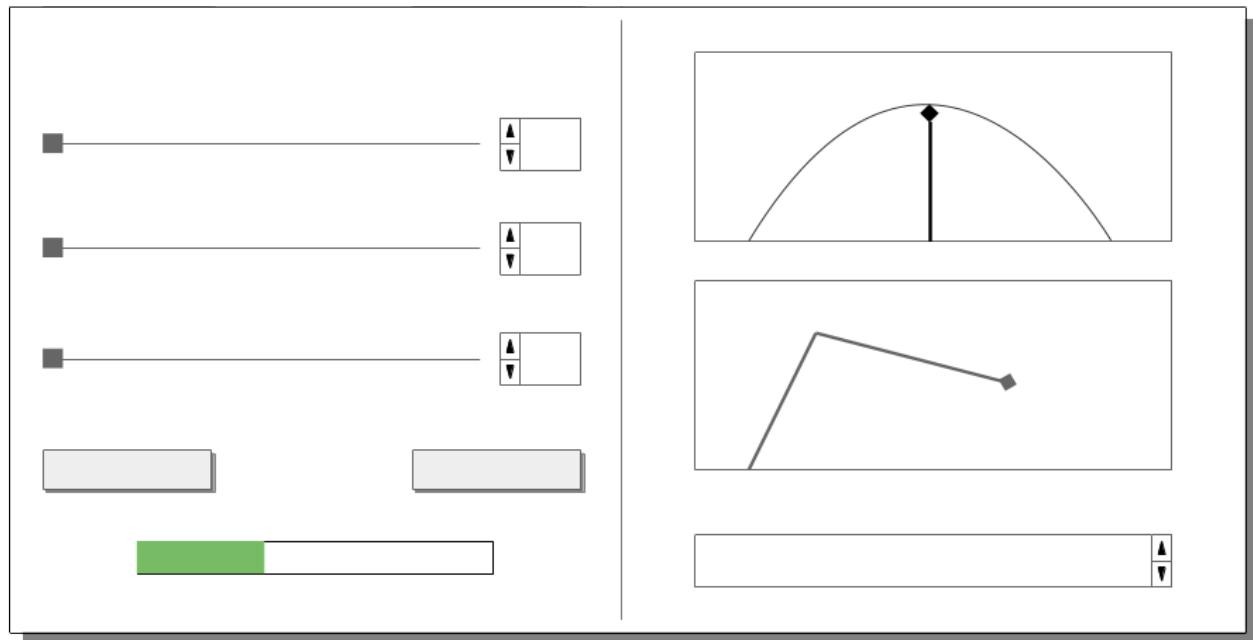


Figura 3.1: Diseño propuesto para la intefaz gráfica de usuario.

La especificación de los elementos de la interfaz anterior (figura 7.1) se hace en mayor profundidad en el punto 3.3.1 del documento.

3.2.4. Memoria

Por experiencia en proyectos relacionados, se estima que la memoria disponible para código y datos, en el microcontrolador seleccionado, ofrecen margen suficiente para este proyecto. Se detallan a continuación las características a nivel de memoria del microcontrolador:

- 512 KB de memoria *flash*, donde se alberga el programa principal.
- 48 KB de memoria RAM, donde se alojarán los datos temporales durante la ejecución, teniendo en especial consideración las matrices de las ecuaciones cinemáticas.

3.2.5. Operaciones

Los usuarios deberán desempeñar acciones tales que generen los movimientos deseados en el brazo. Estas acciones pueden implicar interactuar con los *widgets* presentes en el panel de control o bien efectuar movimientos con el ratón para que el robot los desempeñe directamente.

3.2.6. Funciones del producto

Las funcionalidades principales del brazo robótico han sido descritas de forma introductoria en apartados anteriores de este documento.

En general, existen dos funcionalidades principales que caracterizan tanto al *pArm* como al sistema de control del mismo:

- La funcionalidad principal del brazo robótico S2 es la de realizar movimientos dentro de su campo de movimiento y describir trayectorias previamente planificadas o calculadas en el momento. Mediante este movimiento, se pretende transportar objetos de poco peso. Además, para agilizar el funcionamiento y el procesado de las órdenes, será S2 el que gestione, compute y realice los movimientos que recibe por parte de S1, quedando este último para la interacción con el usuario y la gestión de S2.
- El sistema de control S1 ofrece la funcionalidad principal de planificar trayectorias y controlar el movimiento del brazo. Este sistema se muestra al usuario mediante una interfaz gráfica, la cual permite al usuario controlar el movimiento del brazo mediante la modificación de diversos parámetros.

3.2.7. Características del usuario

El sistema de control ejecutado en S1 ofrecerá una interfaz gráfica que permitirá al usuario interactuar con los parámetros del brazo robótico S2 y, por lo tanto, permitirá al mismo controlar el movimiento del robot así como la establecer la descripción de ciertas trayectorias.

Dado que el objetivo del proyecto es ofrecer un sistema didáctico, amigable y fácil de usar, no se imponen requerimientos específicos sobre el usuario en cuanto a conocimientos técnicos sobre programación, HW, electrónica o matemáticos.

El usuario debe estar familiarizado con la interacción y el uso básico de aplicaciones de escritorio para poder interactuar de forma correcta con el sistema de control del brazo.

A pesar de no ser completamente necesario, es recomendable que el usuario esté familiarizado con la estructura física del robot, los movimientos que este puede realizar y los parámetros que se usan para controlar al mismo, ya que de esta forma el control del brazo robótico será más eficaz y seguro.

3.2.8. Restricciones

Por estar ya disponible y ofrecer los periféricos necesarios para desarrollar este proyecto, se recomienda como alternativa inicial para el sistema S2 utilizar el dsPIC33EP512GM604.

En cualquier caso, como se ha mencionado anteriormente, es necesario que:

- Se provea de una interfaz para la comunicación que permita comunicarse con el sistema S1 de manera simultánea y con alta capacidad.
- El sistema ha de consumir la menor energía posible, entrando en el modo de *deep-sleep* cuando fuera posible.
- La estructura de S2 ha de ser imprimible en 3D, permitiendo así replicarlo.
- El sistema S1 ha de poder ejecutar aplicaciones Python según lo propuesto anteriormente, en particular, la versión de este superior a la 3.6. En otro caso, el sistema habrá de poder ejecutar la aplicación diseñada sin problemas e indiferentemente del sistema operativo.
- Todo lo realizado en el proyecto ha de ser OS y OH, permitiendo así que cualquiera pueda acceder y estudiar el proyecto.

3.2.9. Supuestos y dependencias

Indiferentemente de la placa que finalmente se use, el sistema ha de tener tres motores: uno para la base, otro para el primer segmento del brazo robótico y el último para el segundo segmento. Además, para controlar el *end-effector* hará falta una conexión con el extremo del brazo para poder añadir un pequeño motor que permita la rotación del mismo (ver el manual de desarrollador de UFACTORY para más información).

Para ello, en la tabla 3.3 se muestran distintas propuestas de motores que podrían ser viables para el proyecto. Intentando cubrir las necesidades, se tienen en cuenta para este proyecto:

- Motor paso a paso: dispositivo electromecánico que convierte una serie de pulsos eléctricos en desplazamientos angulares. Esto permite realizar movimientos muy precisos, los cuales pueden variar de 1,8° hasta 90°.
- Servomotor: dispositivos de accionamiento para el control de la velocidad, par motor y posición. En su interior suelen tener un decodificador el cual convierte el giro mecánico en pulsos digitales. Además, suelen disponer de un *driver* el cual permite comandar los distintos controles mencionados al principio.

Nombre	Tipo	Características	Código RS y precio
--------	------	-----------------	--------------------

Servomotor Parallax Inc.	Servomotor	<ul style="list-style-type: none"> ■ Voltaje entrada: 4 V a 6 V. ■ Conector de tres contactos. ■ PWM a 50 Hz. 	781-3058 – 16,01 €
Servomotor Faulhaber 9 W	Servomotor	<ul style="list-style-type: none"> ■ Par máximo: $9,5\text{ mNm}$. ■ Voltaje entrada: 6 V. ■ Potencia nominal: 9 W. ■ Conector MOLEX Microfit 3.0. 	184-6932 – 186,73 €
Motor paso a paso bobinado unipolar	Motor paso a paso	<ul style="list-style-type: none"> ■ Precisión de $1,8^\circ$. ■ Par de sujeción: 70 mNm. ■ Voltaje entrada: 6 V. ■ Conexión de 6 cables. 	440-420 – 30,29 €
Motor paso a paso híbrido	Motor paso a paso	<ul style="list-style-type: none"> ■ Precisión de $1,8^\circ$. ■ Par de sujeción: $1,26\text{ Nm}$. ■ Voltaje entrada: $2,5\text{ V}$. ■ Conexión de 4 cables. 	535-0439 – 108,69 €

Motor paso a paso híbrido	Motor paso a paso	<ul style="list-style-type: none"> ■ Precisión de $0,9^\circ$. ■ Par de sujeción: $0,44 \text{ Nm}$. ■ Voltaje entrada: $2,8 \text{ V}$. ■ Conexión de 4 cables. 	535-0401 – 66,72 €
------------------------------------	-------------------------	---	--------------------

Cuadro 3.3: Lista de motores propuestos para el sistema S2.

3.2.10. Requisitos pospuestos

En esta sección se describen algunos requisitos del sistema que se postergan a futuras implementaciones o versiones del proyecto.

En el comienzo del proyecto se plantearon algunas funcionalidades y requisitos que, finalmente, se han decidido postergar a futuras implementaciones del proyecto, principalmente debido a su complejidad. En la siguiente lista se presentan las mas relevantes, las cuales representan posibles mejoras futuras del *pArm*:

- Implementación del sistema de control y planificación de trayectorias en el microcontrolador del *pArm*, de esta forma se busca centralizar el computo en S2.
- Implementación de un sistema de descripción de trayectorias mediante imitación de movimientos realizados por el usuario, es decir, el usuario podría mover físicamente el *pArm* y memorizaría dicha trayectoria para posteriormente describirla.
- Construcción e implementación de diversos tipos de *end-effector* para el *pArm*, los cuales le dotarían de nuevas funcionalidades en cuanto a manejar objetos.
- Implementación de las estructura física del *pArm* utilizando materiales metálicos para mejorar su resistencia y estabilidad. Junto con esta mejora, se podrían utilizar nuevos rotores para dotar al *pArm* de una mayor capacidad de carga.

3.3. Requisitos específicos

3.3.1. Requisitos de la interfaz externa

Interfaz con el usuario

S1 dispondrá de una interfaz de usuario que deberá seguir el modelo propuesto en la figura 7.1. Dicha interfaz habrá de contar con los siguientes elementos:

- Dos alternativas gráficas de entrada que permitan establecer la posición final del *end-effector* bien mediante coordenadas articulares o bien mediante coordenadas angulares.
- Un actuador para poder escoger entre las alternativas mencionadas en el punto anterior.
- Un actuador para confirmar que se quiere mandar el movimiento al brazo robótico.
- Un actuador para detener un movimiento en ejecución del brazo.

Teniendo en cuenta el diseño propuesto en la figura 7.1, los componentes anteriores estarían representados por:

- Tres *sliders* los cuales establecerán los valores para los ángulos $\{\theta_1, \theta_2, \theta_3\}$, si se está trabajando en el modo de coordenadas angulares; o los valores de los puntos $\{x, y, z\}$, si se está trabajando en el modo de coordenadas cartesianas.
- Un botón desplegable con múltiples opciones que permitiría escoger entre los dos modos mencionados en el punto anterior.
- Un botón que permita confirmar los cambios en las coordenadas/ángulos antes de enviar definitivamente el movimiento al robot.
- Una barra de progreso la cual permite conocer una estimación de cuánto lleva el robot hecho del movimiento final previsto.
- Dos pequeñas ventanas que informan sobre la posición del brazo final una vez se han cambiado los valores de las coordenadas/ángulos. Dichas ventanas muestran una vista cenital del brazo, que indica cómo se mueve en el eje Y, y una vista de perfil del mismo, que indica cómo se mueve en el eje XZ.
- Una pequeña ventana que muestra *logs* relevantes respecto a la situación tanto de S1 como de S2.

Interfaz hardware

S2 está formado por el brazo robótico *pArm* y el microcontrolador que computa las instrucciones recibidas desde S1. Mediante dicho microcontrolador, S2 interactúa directamente con el HW. El microcontrolador realiza las labores de comunicación con S1, así como las labores de recepción y procesamiento de las instrucciones que controlan el movimiento del *pArm*.

Tras la recepción y procesamiento de las diferentes secuencias de bits, las cuales son instrucciones, el microcontrolador genera señales de salida mediante sus pines, las cuales controlan el movimiento de cada uno de los motores, así como del *end-effector*. Cabe destacar que, en el caso de utilizar motores que proporcionen información sobre su posición angular actual, el microcontrolador debe recibir dicha señal y procesarla, enviando dicha información a S1.

Dependiendo del tipo de motores que se utilicen finalmente, el microcontrolador debe ser capaz de generar señales analógicas PWM, así como señales digitales de control.

Interfaz de comunicaciones

Las comunicaciones que se realicen entre S1 y S2 están planteadas para utilizar UART como método de comunicación. Además, se mencionó como futura implementación poder hacer las comunicaciones entre ambos sistemas utilizando protocolos de red inalámbricos.

No se restringe la velocidad de transmisión (*baud-rate*), ya que se asume que S1 tendrá la posibilidad de adaptar su velocidad. Se escoge el USB como método para intercambiar la información debido a:

- Universalidad: los dispositivos cuentan con al menos una conexión USB.
- Energía: el USB provee 5 V al circuito que se conecta en el otro extremo. Además, la versión 2.0 del estándar, que es lo generalizado en microcontroladores, puede proveer hasta 500 mA al componente conectado.
- Simplicidad: no es necesario entender cómo se conectan los cables sino directamente conectar los extremos.

Para un correcto funcionamiento, la comunicación ha de ser bidireccional, en particular *full duplex*. De esta forma, se podrán recibir y enviar datos simultáneamente, pudiendo así conocer el estado del brazo robótico y actuar en consecuencia en caso de que se encuentre algún tipo de error o problema. Dado que la mayoría de placas base disponen de conexión UART las cuales se pueden utilizar para comunicación con el exterior y que los equipos informáticos habituales disponen de múltiples puertos USB, se pueden usar convertidores UART–USB económicos y eficaces para la comunicación entre ambos sistemas, quedando subsanado además la cuestión de la comunicación *full-duplex*.

3.3.2. Casos de uso

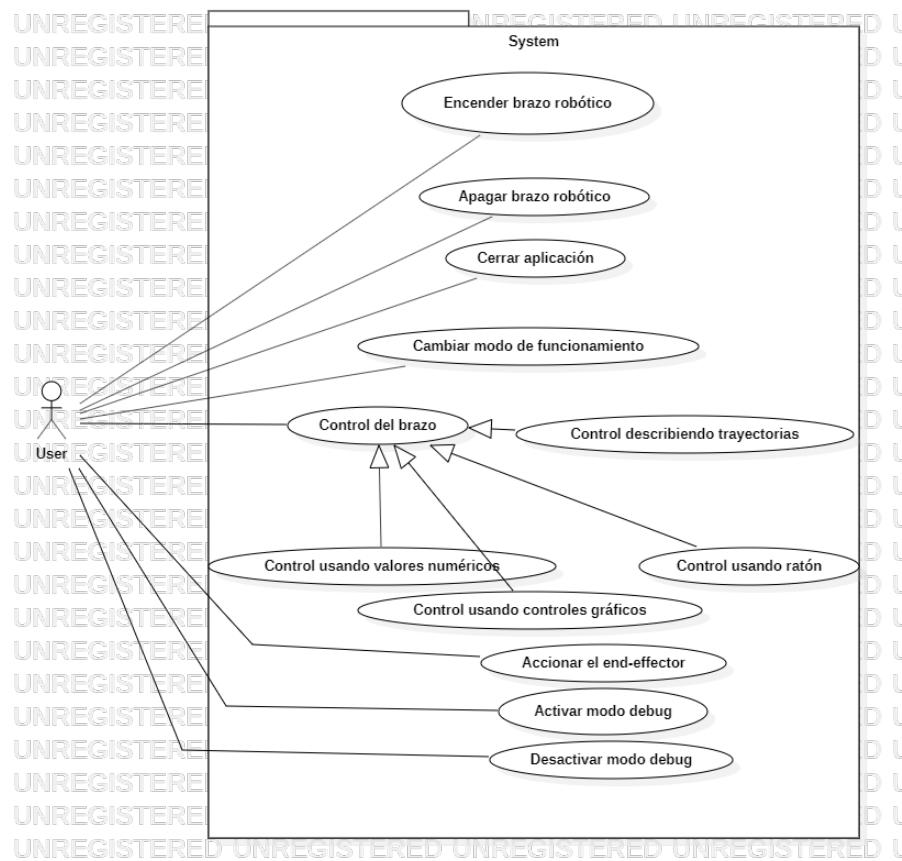


Figura 3.2: Diagrama de casos de uso

0001	Encender brazo robótico (S2)	
Descripción	El usuario deberá ser capaz de encender el sistema del brazo robótico de manera independiente de la aplicación de control.	
Secuencia Normal	Paso	Acción
	1	El usuario interactúa con el sistema para encenderlo.
	2	El sistema comprueba que los motores están correctamente conectados y que se mueven correctamente hasta el final de carrera.
Excepciones	Paso	Acción
	3	Si las comprobaciones no son satisfactorias el sistema activará un indicador luminoso y se informará del error a S1, si está conectado.
Importancia	1	
Comentarios	Sin comentarios	

Cuadro 3.4: Caso de uso 0001 - Encender brazo robótico (S2).

0002	Apagar brazo robótico(S2)	
Descripción	El usuario deberá ser capaz de apagar el brazo robótico desconectando la corriente del mismo.	
Secuencia Normal	Paso	Acción
	1	El usuario interactúa con el sistema para apagarlo.
Excepciones	Paso	Acción
		No existen
Importancia	1	
Comentarios	Sin comentarios	

Cuadro 3.5: Caso de uso 0002 - Apagar brazo robótico (S2).

0003	Cerrar aplicación	
Descripción	El usuario deberá ser capaz de cerrar la aplicación de control de manera independiente al brazo robótico.	
Secuencia Normal	Paso	Acción
	1	El usuario interactúa con la aplicación para cerrarla
Excepciones	2	Se comprueba que la aplicación se puede cerrar de manera segura. Esto implica asegurar que no hay ninguna comunicación en proceso antes de cerrar la aplicación así como que el brazo no se esté moviendo.
	3	Se realiza el cierre de la aplicación.
Importancia	1	
	Sin comentarios	

Cuadro 3.6: Caso de uso 0003 - Cerrar aplicación.

0004	Cambiar modo de funcionamiento (S1)	
Descripción	El usuario deberá ser capaz de seleccionar el modo de control del brazo robótico, pudiendo escoger entre control mediante ratón o control mediante parámetros.	
Secuencia Normal	Paso	Acción
	1	El usuario interactúa con la aplicación y selecciona el modo de control del robot.
Excepciones	2	El sistema cambia entre modo de control mediante ratón o modo de control mediante parámetros.
	Paso	Acción
Importancia		No existen
	1	
Comentarios	Sin comentarios	

Cuadro 3.7: Caso de uso 0004 - Cambiar modo de funcionamiento.

0005	Control usando valores numéricos (S1)	
Descripción	El usuario deberá ser capaz de cambiar el valor numérico de cada uno de los parámetros de control del brazo robótico	
Secuencia Normal	Paso	Acción
	1	El usuario interactúa con la aplicación y cambia el valor de los parámetros de control usando el teclado.
Excepciones	2	Se comprueba si el valor es correcto y se confirma el cambio del valor numérico.
	2.1	Se elimina el valor y se notifica al usuario sobre el error y se le pide que introduzca de nuevo el valor.
Importancia	1	
Comentarios	Sin comentarios	

Cuadro 3.8: Caso de uso 0005 - Control usando valores numéricos (S1).

0006	Control describiendo trayectorias	
Descripción	Se permitirá al usuario escoger una trayectoria predefinida que el brazo robótico deberá realizar.	
Secuencia Normal	Paso	Acción
	1	El usuario selecciona una trayectoria a realizar.
Excepciones	2	Se realiza dicha trayectoria
	Paso	Acción
Importancia	No existen	
	Comentarios	Esta característica no se implementa ya que se posterga para una futura versión.

Cuadro 3.9: Caso de uso 0006 - Control describiendo trayectorias.

0007	Control usando controles gráficos	
Descripción	La interfaz gráfica de la aplicación debe ofrecer control sobre los parámetros del brazo robótico mediante <i>sliders</i> .	
Secuencia Normal	Paso	Acción
	1	El usuario interactúa con la aplicación y mueve los <i>sliders</i> para variar los parámetros del brazo robótico.
Excepciones	Paso	Acción
	2	Se verifica si se puede realizar dicho movimiento y se ejecuta el cambio en los parámetros.
Importancia	1	
Comentarios	Sin comentarios	

Cuadro 3.10: Caso de uso 0007 - Control usando controles gráficos.

0008	Control usando ratón	
Descripción	Se permitirá al usuario controlar el brazo robótico de manera directa con el movimiento del ratón	
Secuencia Normal	Paso	Acción
	1	El usuario mueve el ratón realizando movimientos libres.
Excepciones	Paso	Acción
	2	Se comprueba que el movimiento no se sale de los márgenes permitidos
Importancia	1	
Comentarios	Esta característica no se implementa ya que se posterga para una futura versión.	

Cuadro 3.11: Caso de uso 0008 - Control usando ratón.

0009	Accionar el <i>end-effector</i>	
Descripción	Se permite al usuario abrir y cerrar la pinza	
Secuencia Normal	Paso	Acción
	1	El usuario interactúa con la aplicación para abrir y cerrar el <i>end-effector</i>
Excepciones	Paso	Acción
		No existe
Importancia	1	
Comentarios	Esta característica no se implementa ya que se posterga para una futura versión.	

Cuadro 3.12: Caso de uso 0009 - Accionar el *end-effector*.

0010	Activar modo <i>debug</i>	
Descripción	Se permite al usuario activar un modo tal que se pueda mandar al S2 el código de control del brazo robótico	
Secuencia Normal	Paso	Acción
	1	El usuario interactúa con S2 para ponerlo en modo debug.
	2	El sistema comprueba que el cambio de modo se puede hacer de manera segura. Es decir, no hay una comunicación específica del modo de funcionamiento actual en proceso antes de realizar el cambio.
Excepciones	3	El sistema cambia de modo.
	Paso	Acción
	2	El sistema detecta que el cambio de modo no se puede hacer de manera segura e impide que este se realice. Se informará del error a S1, si está conectado.
Importancia	1	
Comentarios	Esta característica no se implementa ya que se posterga para una futura versión.	

Cuadro 3.13: Caso de uso 0010 - Activar el modo *debug*.

0011	Desactivar modo <i>debug</i>	
Descripción	Se permite al usuario desactivar el modo debug tal que sea posible emplear el sistema de manera normal	
Secuencia Normal	Paso	Acción
	1	El usuario interactúa con S2 para desactivar el modo debug.
	2	El sistema comprueba que el cambio de modo se puede hacer de manera segura.
Excepciones	3	El sistema cambia de modo.
	Paso	Acción
	2	El sistema detecta que el cambio de modo no se puede hacer de manera segura e impide que este se realice. Se informara del error a S1, si esta conectado.
Importancia	1	
Comentarios	Esta característica no se implementa ya que se posterga para una futura versión.	

Cuadro 3.14: Caso de uso 0011 - Desactivar el modo *debug*.

3.3.3. Requisitos funcionales

– 001. Generar movimiento mediante el ángulo de cada uno de los motores

El sistema S2 cuenta con tres motores los cuales se encargan del movimiento de cada una de las partes del brazo. Se permitirá establecer individualmente cada ángulo $\{\theta_0, \theta_2, \theta_3\}$ y mover así el brazo a una posición final $\{x', y', z'\}$.

Este movimiento, siguiendo la maqueta definida en la figura 7.1, se realizará interactuando con *sliders*.

– 002. Generar movimiento mediante las coordenadas cartesianas del punto final

Se permitirá también el movimiento a un punto $\{x, y, z\}$ directamente, para lo que se obtendrán los ángulos $\{\theta_0, \theta_2, \theta_3\}$ que permiten alcanzar dicha posición.

Al igual que en el caso anterior, se podrá definir cada punto independientemente.

Este movimiento, siguiendo la maqueta definida en la figura 7.1, se realizará interactuando con *sliders*.

– 003. Selección del modo de funcionamiento del brazo

Dado que, como se ha mencionado en los puntos anteriores, hay dos maneras de hacer que el brazo se pueda mover, habrá de existir algún tipo de actuador en la interfaz de usuario que permita escoger entre dichos modos.

Esta acción, siguiendo la maqueta definida en la figura 7.1, se realizará interactuando con un botón.

– 004. Ejecución en un momento determinado

La interacción con los elementos comentados anteriormente no será efectiva hasta que el usuario indique que quiere que se realicen, permitiendo así confirmar que los datos introducidos son los correctos.

Esta acción, siguiendo la maqueta definida en la figura 7.1, se realizará interactuando con un botón.

– 005. Demostración del punto final del brazo

La interacción con los actuadores definidos anteriormente se verá reflejada en unas pequeñas ventanas que muestran cómo debería encontrarse el brazo tras realizar los movimientos indicados.

Esta demostración, siguiendo la maqueta definida en la figura 7.1, se mostrará mediante unos dibujos esquemáticos que representan el brazo visto de perfil y desde una vista cenital.

– 006. Otros requisitos

Se han considerado operaciones más avanzadas para el control del brazo (como trazar trayectorias o un control mediante el ratón en un plano 2D) las cuales no se reflejan en este documento ya que se ha postergado su desarrollo e implementación a una futura versión del sistema.

3.3.4. Restricciones del diseño

En esta sección se describen algunas limitaciones existentes debido a distintos motivos, principalmente al HW y estructura física del *pArm*.

En primer lugar, existe una limitación en cuanto a los materiales de fabricación de la estructura física del brazo, ya que se quiere construir mediante la combinación de dos materiales plásticos: Ácido Poliláctico (PLA) y Copoliéster (CPE). El primer material se utiliza para impresión en 3D y, dado que el *pArm* se quiere imprimir por piezas utilizando una impresora de este tipo, el PLA es un material adecuado. El CPE por su parte ofrece una alta resistencia a productos químicos y, lo que es más importante para este brazo robótico, una gran resistencia a temperaturas elevadas y a la fricción, lo que lo hace en un material ideal para diseñar e imprimir estructuras mecánicas [23].

Por otro lado, para simplificar los cálculos en el modelo dinámico, se ha optado por usar un manipulador robótico pantográfico. Este tipo de manipuladores tienen una estructura similar a un flexo y la principal ventaja es que los motores se encuentran muy cercanos a la base. De esta forma, el peso de los mismos no debe ser desplazado al realizar movimientos en las articulaciones del brazo.

3.3.5. Atributos del sistema *software* y *hardware*

Tanto para el SW como para el HW, se busca que ambos cumplan las siguientes premisas:

1. El sistema al completo ha de ser seguro, en el rango del brazo robótico. Esto es, no se permitirá a S2 realizar movimientos que puedan perjudicar la estructura del mismo de forma irremediable. A su vez, el sistema S2 deberá tener en cuenta posibles fallos en las órdenes de S1 y comprobar así que la secuencia de órdenes es correcta y no contiene posiciones inseguras.
2. Teniendo en cuenta lo desarrollado en el punto de “Descripción general” (3.2) y lo mencionado en la “Introducción” (3.1), es importante que el sistema sea mantenible. Esto se traduce en que, por una parte, se pueda actualizar para corregir problemas que se han encontrado una vez se ha desplegado el sistema; y que la sustitución de piezas o elementos del mismo resulte accesible y barato.
3. Finalmente, dado que el *pArm* está impreso en 3D, se busca que sea portable en lo referente a que pueda ser fácilmente transportado de un lugar a otro. Esto se traducirá en un bajo peso y que el área ocupada por el mismo sea también baja.

3.3.6. Requisitos no funcionales

Por motivos de tiempo, se dejan los requisitos no funcionales para una futura especificación.

Capítulo 4

Diagramas y diseño

Una parte importante de un proyecto integral de ingeniería es la elicitation de requisitos y la creación de diagramas que representen el sistema de manera abstracta en base a dichos requisitos.

El sistema de gobierno del *pArm* está compuesto de dos subsistemas al ser necesaria tanto una placa de control como un ordenador auxiliar desde el cual un operador humano pueda interactuar con el brazo. El SW del sistema de control del ordenador será representado mediante un diagrama de clases mientras que el SW que irá cargado en la placa de control será representado por un diagrama de bloques general y varios diagramas de estados que detallarán el comportamiento del sistema.

Para realizar dichos diagramas se ha empleado *Papyrus*, una herramienta de edición gráfica para realización de diagramas. Para modelizar los diagramas del sistema de control que irá en el ordenador auxiliar se ha empleado el estándar UML2 definido por la OMG. Por otro lado, para realizar los diagramas del SW que será cargado en la placa de control, se ha empleado el estándar SysML 1.4 ya que permite mejor representación del sistema empotrado.

En base a los anteriores requisitos el grupo de desarrollo ha generado diagramas para el software de S1 y S2. A continuación se procederá a detallar cada uno de los distintos diagramas de cada uno de los sistemas .

4.1. Diagramas software de S1

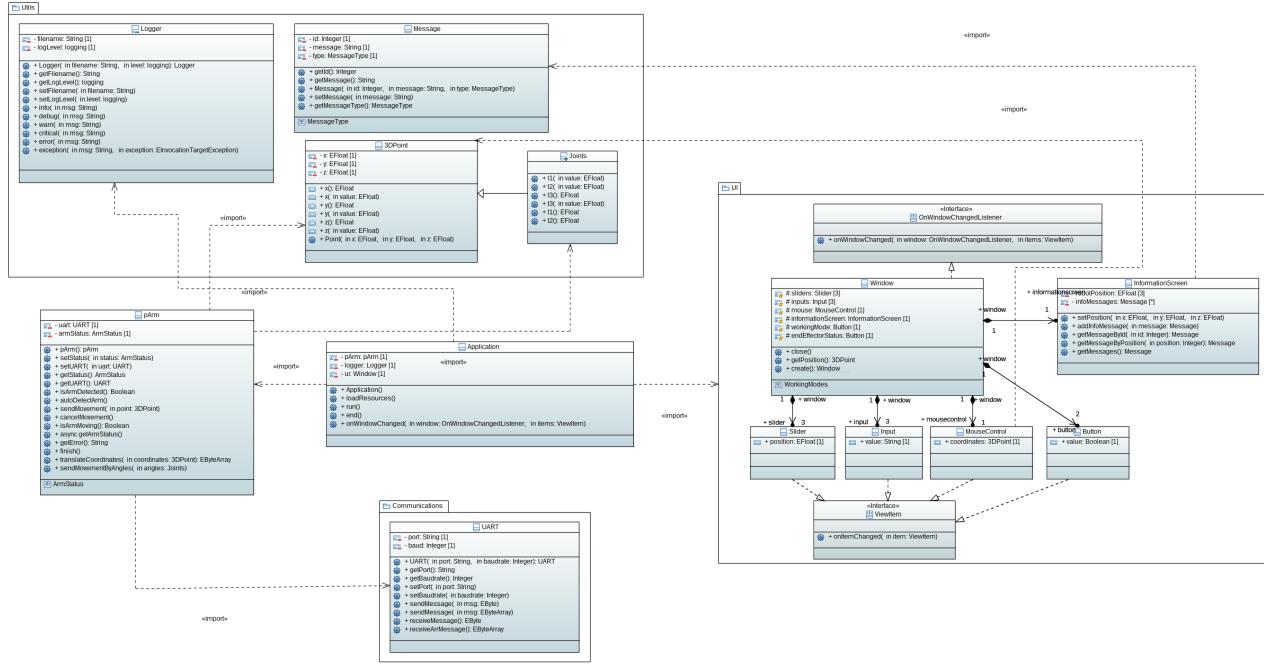


Figura 4.1: Diagrama de clases de S1.

En el diagrama 4.1 se observa el diagrama de clases completo de S1. Debido a su envergadura se procede a dividirlo en dos partes, a saber, la relacionada con la lógica del sistema y la relacionada con la GUI.

Para explicar la lógica del sistema S1 se hará una división por paquetes y posteriormente se procederá a explicar cada una de las clases que componen el paquete.

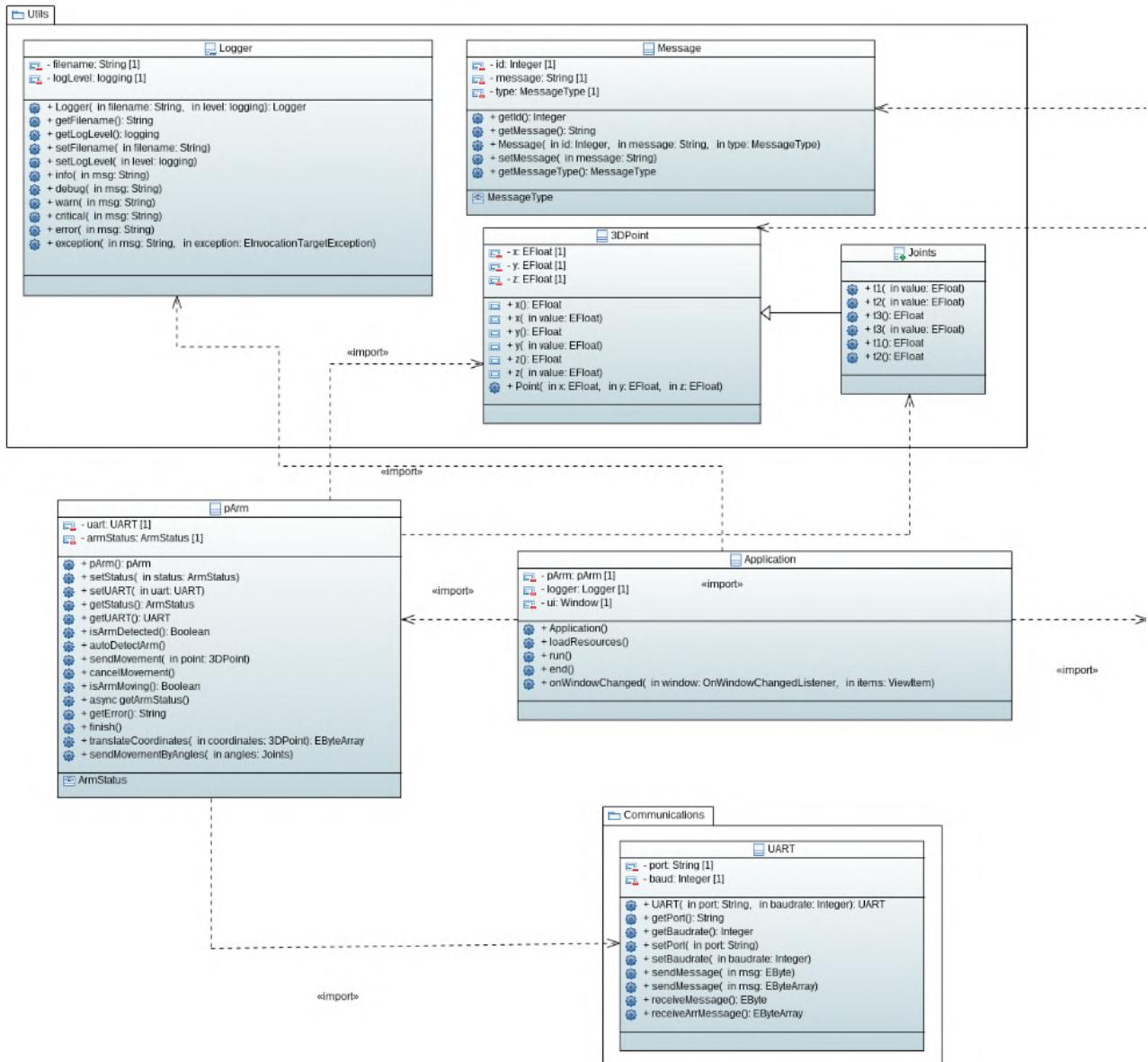


Figura 4.2: Recorte del diagrama de clases de S1 el cual representa la lógica del sistema.

- **Paquete Utils:** este paquete contiene clases cuyos objetos son instanciados con el objetivo de realizar labores genéricas que no están relacionadas de manera directa con la lógica o que no encajan en ningún otro paquete. Dentro de este paquete encontramos las siguientes clases:

- **Logger:** esta clase sirve para instanciar un objeto el cual genera archivos de registro del funcionamiento. Empleando ciertos métodos de esta clase, a lo largo del código, es posible guardar datos del sistema en un archivo, el cual es persistente en el tiempo. Posteriormente, se puede leer este archivo para poder hacer labores de *debugging* tanto en las etapas de diseño como en la etapa de producción y despliegue.
- **Message:** los objetos de esta clase sirven para dar una estructura general a los diferentes mensajes que se mostrarán en la GUI con el objetivo de mostrar la información de manera uniforme.

- **3DPoint**: los objetos de esta clase representan puntos en el espacio cartesiano y se emplean para poder aunar las coordenadas en un solo objeto contenedor. Con esto se consigue simplificar la comunicación de los datos dentro del sistema. Los métodos que contiene son *getters* y *setters* de las distintas coordenadas.
- **Joints**: hereda de 3DPoints y los métodos son *wrappers* de los *getters* y lo *setter* de esta.
- Clase **pArm**: contiene los métodos necesarios para realizar los movimientos del brazo, inicializar las comunicaciones y posteriormente gestionarlas. Es la clase principal de la lógica del sistema.
- Clase **Application**: esta clase inicializa la aplicación y los recursos necesarios para el funcionamiento de esta. En ella se instancian objetos de las clases **pArm**, **Logger** y **ui**.
- Paquete **Communications**: contiene a la clase **UART**. Sirve para inicializar los puertos UART y la tasa de transmisión. Además, facilita los métodos para escribir en el canal de transmisión.

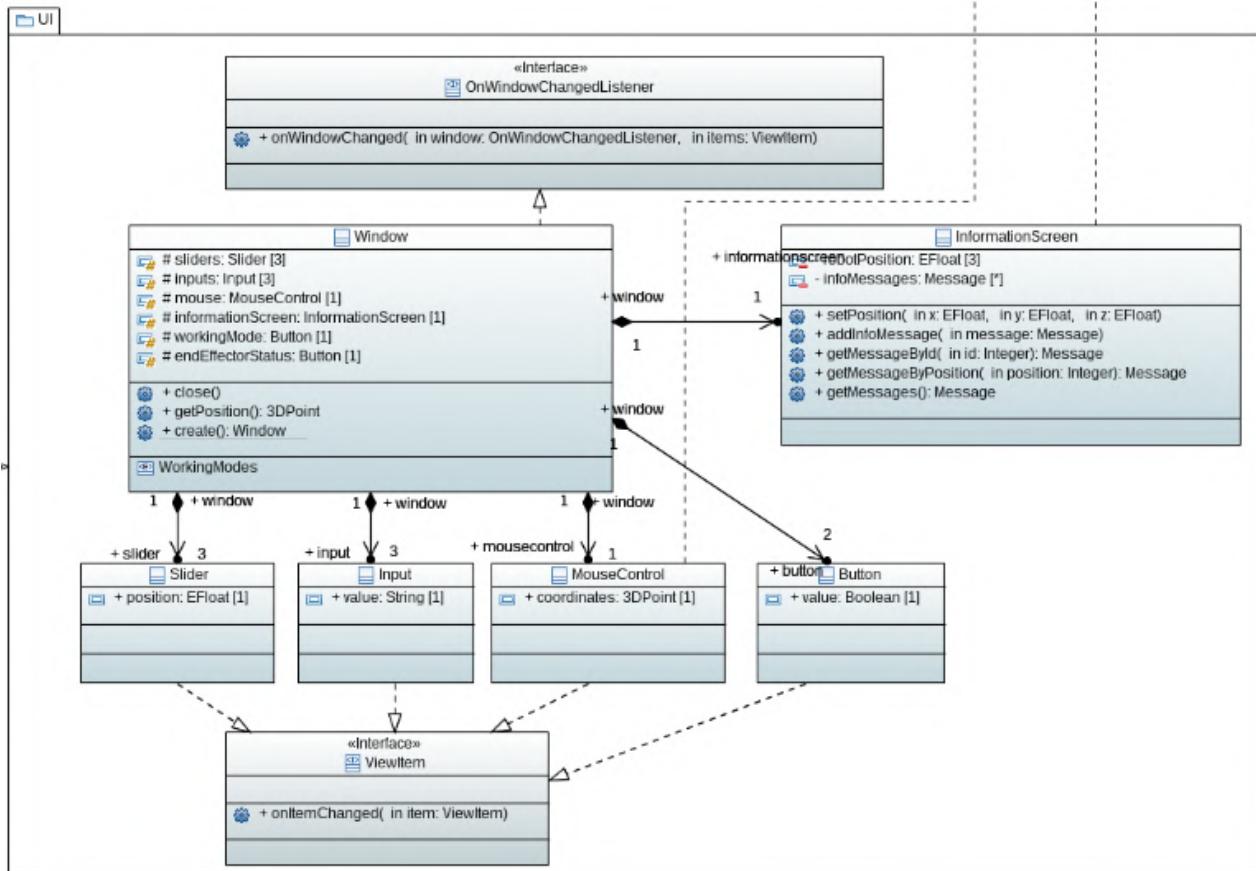


Figura 4.3: Recorte del diagrama de clases de S1 el cual representa la interfaz gráfica de usuario.

En este caso toda la GUI esta contenida dentro de un mismo paquete.

- Clase **Window**: esta clase representa la ventana principal de la aplicación donde se encuentran todos los elementos con el que le usuario puede interactuar.
- Clase **Slider**: *widget* de tipo *Slider* que aparece dentro de la ventana principal y sirve para definir valores de las coordenadas cartesianas y angulares de manera gráfica.
- Clase **Input**: *widget* de tipo *SpinBox* que parece dentro de la ventana principal y sirve para definir valores de las coordenadas cartesianas y angulares de manera gráfica directamente con el valor en concreto.
- Clase **MouseControl**: clase empleada para obtener las coordenadas del ratón.
- Clase **Button**: *widget* de tipo *Button* que se emplea para desencadenar acciones en el sistema.
- Clase **InformationScreen**: panel que contiene texto, el cual informa al operario de distintos datos relacionados con el brazo y la aplicación
- Interfaz **ViewItem**: función de *callback* para tener constancia de los cambios en los distintos *widgets*.
- Interfaz **OnWindowChangeListener**: función de *callback* para tener constancia de los cambios en la ventana de la interfaz.

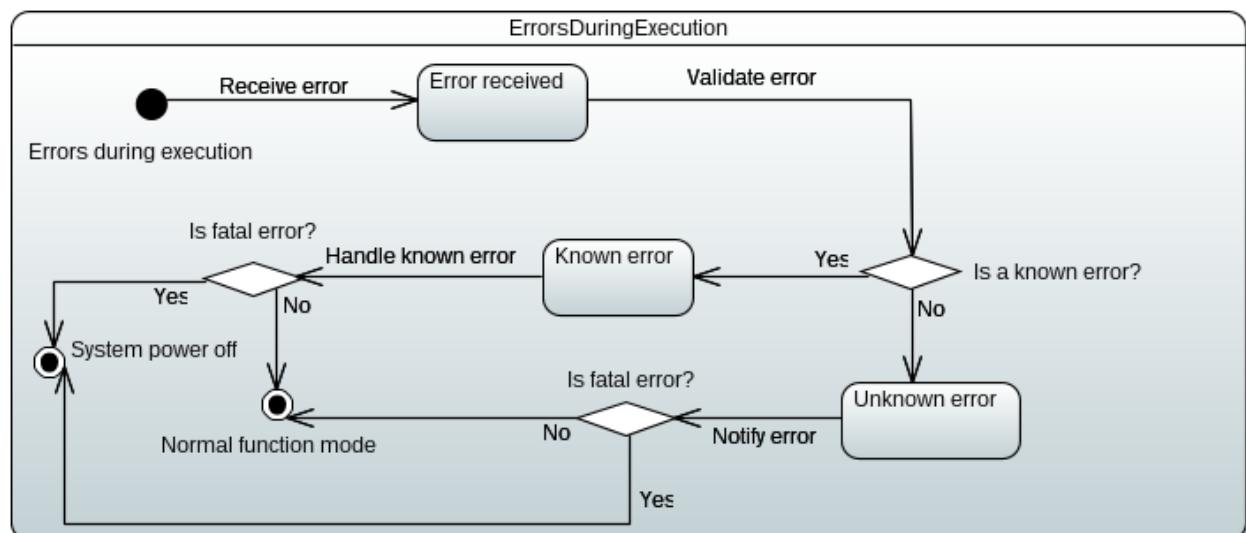


Figura 4.4: Diagrama de estados del tratamiento de errores de S1.

El diagrama 4.4 representa el tratamiento de errores en el S1. Se observa que, dependiendo de si los errores son fatales o no, el sistema se apagará o seguirá funcionando.

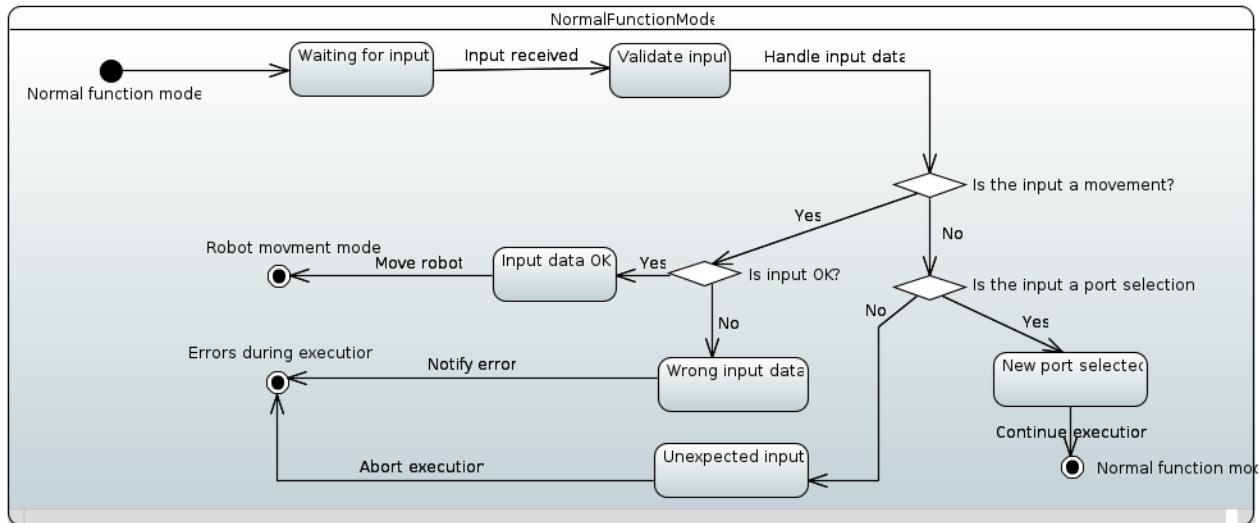


Figura 4.5: Diagrama de estados del funcionamiento normal de S1.

En este diagrama se observa que, durante el funcionamiento normal de la aplicación, se puede interactuar con esta o bien seleccionando un puerto o bien definiendo un movimiento que el brazo habrá de realizar. En el primer caso, la aplicación vuelve al funcionamiento normal de manera directa, mientras que en el segundo caso, si el movimiento es correcto se procede al modo de movimiento. Si por el contrario es un dato erróneo, se notifica del error.

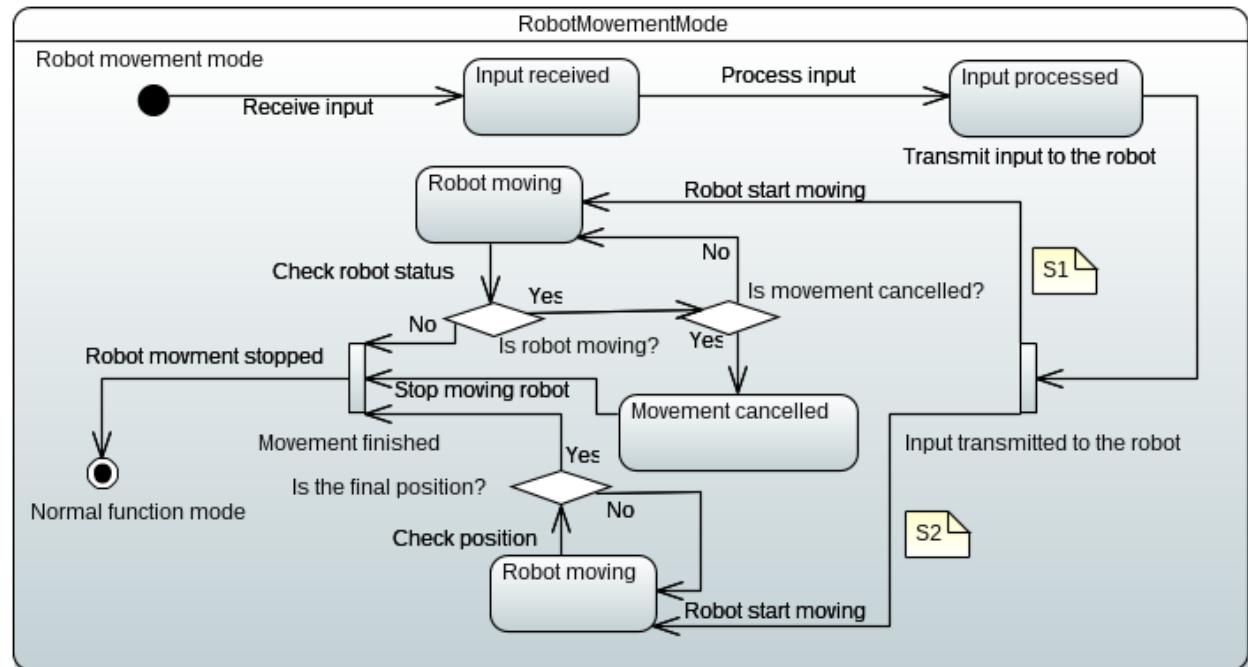


Figura 4.6: Diagrama que representa el movimiento del brazo robótico tanto en S1 como en S2.

Tras recibir una orden de movimiento esta es interpretada y los valores de la posición destino

son transmitidos a S2. En este momento, los dos sistemas empiezan a funcionar de manera concurrente. S1 se encargará de monitorizar si se recibe una orden de cancelar movimiento por parte del usuario y esperara a que S2 termine el movimiento. Por otro lado, S2 comprobará si su posición actual coincide con la posición destino y procederá a moverse en dirección a esta hasta que la alcance.

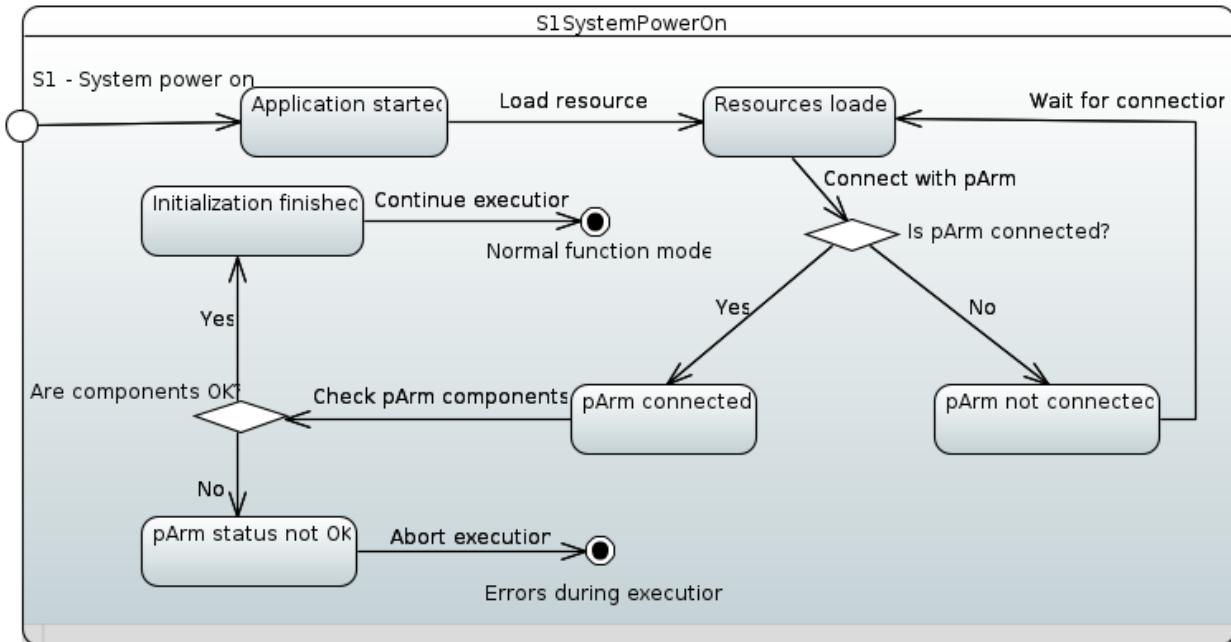


Figura 4.7: Diagrama de estados del encendido de S1.

El sistema S1 inicia la aplicación y carga los recursos necesarios para que esta pueda empezar a mostrarse por pantalla. Al conectarse con el brazo, S1 comprueba que S2 esté en un estado correcto y de ser así finaliza la inicialización del sistema y permite la interacción con S2.

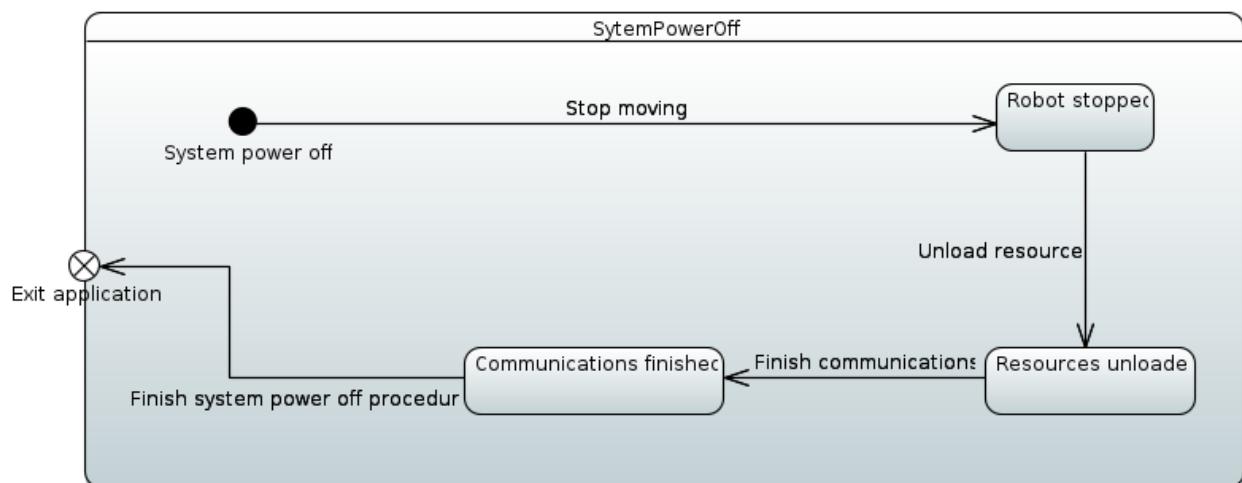


Figura 4.8: Diagrama de estados del apagado de S1.

Para apagar el sistema, este tiene que asegurarse que S2 no está realizando un movimiento. En caso contrario, S1 ordena a S2 que lo cancele. Tras S1 asegurarse que el movimiento ha parado, se cierra.

4.2. Diagramas *software* de S2

En el caso de los diagramas de S2 cabe destacar que los estados de error funcionan en cascada, es decir, dado que cada uno de los diagramas de estados representa una función del código, estas podrán tener errores en su ejecución y estos errores se irán propagando en cascada hacia niveles más altos de abstracción a lo largo de las diferentes funciones que han sido invocadas.

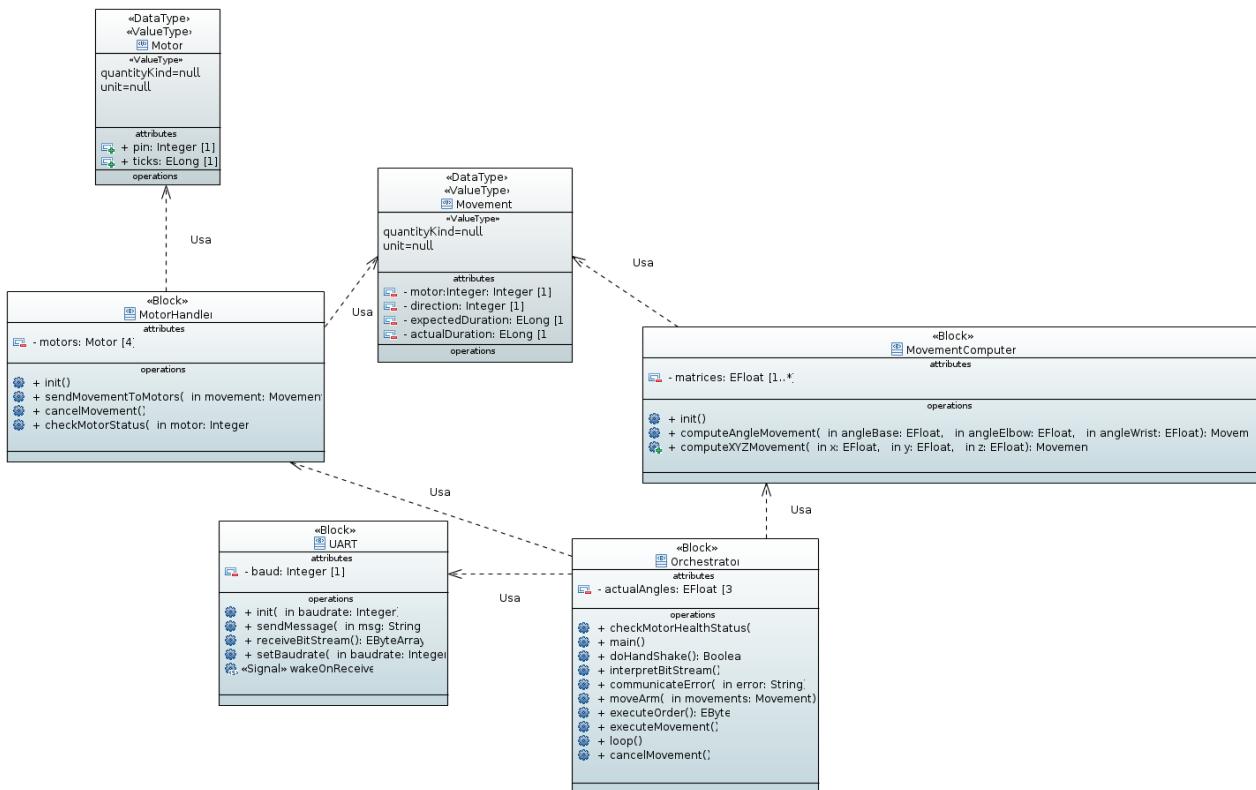


Figura 4.9: Diagrama de bloques de S2.

En el diagrama 4.9 se pueden observar los bloques que componen S2 además de dos tipos de datos los cuales han sido creados para facilitar el control de los motores del brazo.

A continuación se explican cada uno de los bloques:

- **MotorHandler:** este bloque es capaz de controlar los motores de manera directa empleando el tipo de dato “Movement” enviando la señal necesaria para realizar el movimiento requerido. Además, permite verificar el estado de los motores y cancelar los movimientos si esto fuese necesario.

- **UART:** este bloque es el encargado de la comunicación asíncrona entre S1 y S2. Controla la tasa de baudios de la comunicación y realiza la transmisión y la recepción de información hasta y desde S1. A través de este bloque se reciben las órdenes procedentes de S1 y se envían los errores y la posición del brazo a S1 desde S2.
- **Orchestrator:** encargado de coordinar los demás bloques. En él se encuentra la lógica principal de S2. Algunas de sus funciones más importantes son interpretar el flujo de bits que llega desde S1 para obtener una orden concreta; ordenar el movimiento del brazo empleando los demás bloques o hacer la sincronización inicial entre S1 y S2. Posteriormente se entrará en mayor detalle sobre el comportamiento de este bloque al analizar los diagramas de estados.
- **MovementComputer:** se encarga de computar el movimiento que se tendrá que comunicar a los motores. Para ello deberá obtener las posiciones deseadas gracias al bloque **UART** y al “**Orchestrator**”.

A continuación se explican las dos estructuras de datos que se aprecian en el diagrama 4.9

- **Motor:** este tipo de dato es empleado por “**MotorHandler**” para saber a qué pin debe mandar la señal “**PWM**” que gobierna los motores y durante cuántos *ticks* deberá estar activa dicha señal
- **Movement:** “**MovementComputer**” genera un vector de 3 posiciones de este tipo de dato, uno por cada motor de giro del brazo. El atributo **motor** guarda un entero que representa uno de los motores del brazo; **direction** sirve para conocer la dirección de giro de dicho motor; **expectedDuration** guarda la duración.

A continuación se explican los diagramas de estados de cada uno de los métodos que aparecen en el diagrama de bloques general.

En el caso del **Orchestrator** tenemos los siguientes diagramas:

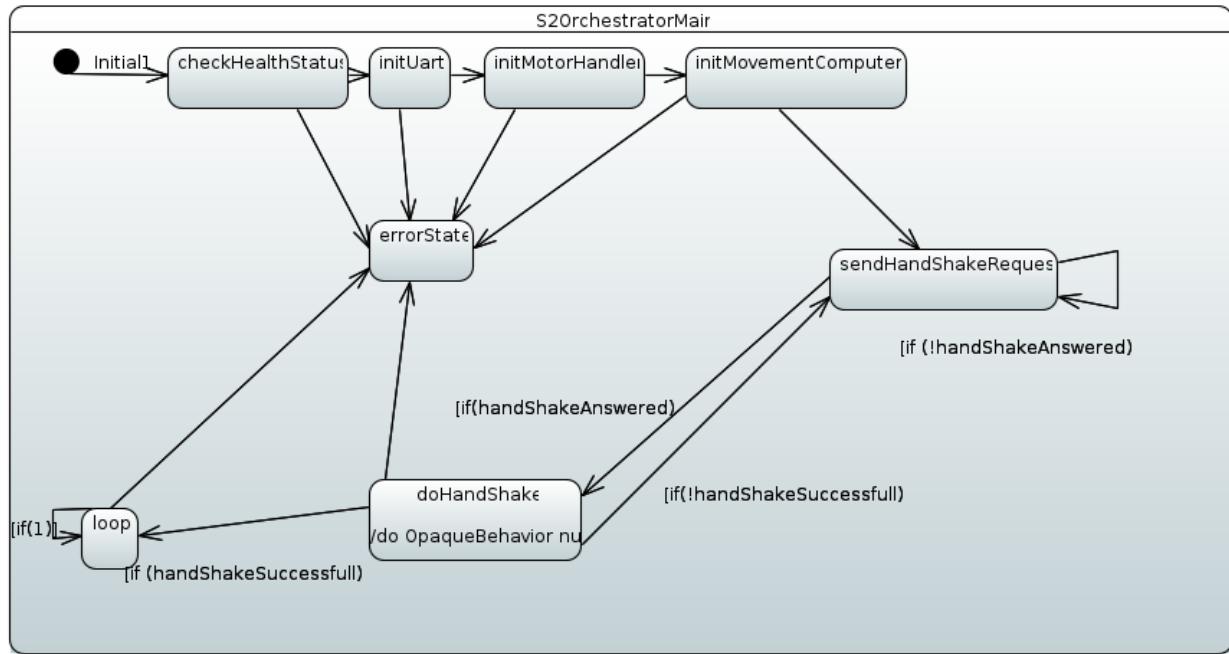


Figura 4.10: Diagrama de estados del método `main()` del *orchestrator*.

Este método solo se ejecutará una vez, en cuanto el sistema se ponga en marcha.

1. `checkHealthStatus`: se verifica la situación de los componentes del brazo robótico para confirmar que todos están en un estado adecuado para el funcionamiento.
2. `initUart`: se inicializa la UART definiendo una tasa de baudios concreto.
3. `initMotorHandler`: se inicializa el controlador de los motores.
4. `initMovementComputer`: se inicializa el computador de movimientos.
5. `sendHandshakeRequest` : se manda una petición de *handshake* para verificar si hay algún ordenador conectado. Si se detecta alguno se pasa al siguiente estado. Si no, se mantiene en ese estado mandando peticiones.
6. `doHandshake`: si en el estado anterior se detecta un ordenador se pasa a este estado. Se realiza una serie de intercambios de información para verificar que el ordenador conectado es adecuado para el control del brazo.
7. `loop`: se pasa al bucle de funcionamiento si el *handshake* ha sido correcto.
8. `errorState`: estado de error al que se llega si en alguno de los estados ocurre algún problema inesperado.

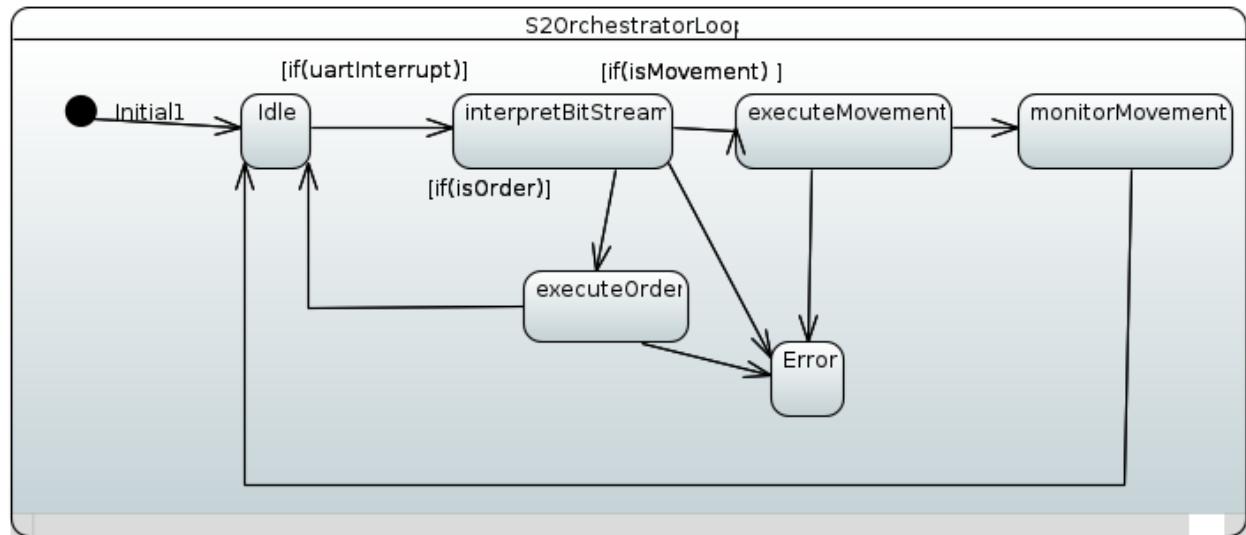


Figura 4.11: Diagrama de estados del método `loop()` del *orchestrator*.

Este método es el bucle principal del brazo robotico. Tras ejecutar `main()` el sistema entrará en este bucle y no saldrá hasta que se apaga.

1. **Idle**: el brazo se encuentra ocioso y a la espera de una orden desde S1.
2. **interpretBitStream**: tras una interrupción de la UART, S2 entiende que hay una orden o movimiento procedentes de S1 y se avanza a este estado. La trama de bits es interpretada para saber si es una orden o un movimiento.
3. **executeMovement**: si tras interpretar la trama de bits resulta que es un movimiento, el sistema avanza a este estado y se ponen en marcha los demás bloques para poder generar un movimiento en los motores en base a la posición recibida desde S1
4. **executeOrder**: si tras interpretar la trama de bits resulta que es una orden, el sistema avanza a este estado y se ponen en marcha los bloques necesarios para ejecutar dicha orden.
5. **monitorMovement** : tras empezar a ejecutar un movimiento, S2 empieza a monitorizarlo para poder determinar cuándo se ha terminado o, si es cancelado, actualizar la posición en la que se ha quedado el brazo.
6. **errorState**: estado de error al que se llega si en alguno de los estados ocurre algún problema inesperado.

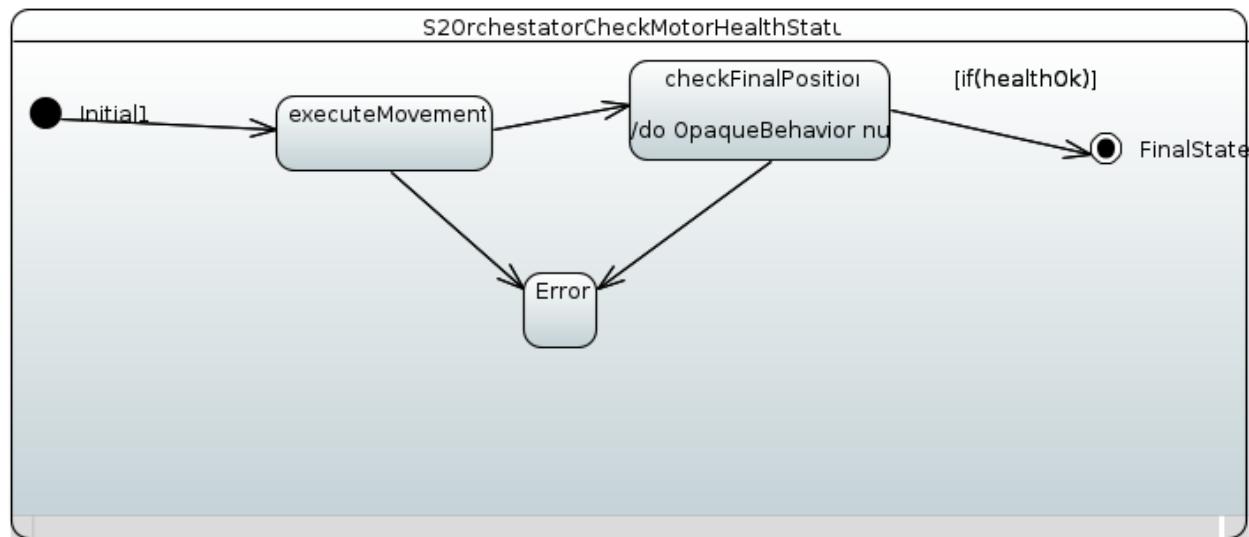


Figura 4.12: Diagrama de estados del método `CheckMotorHealthStatus()` del *orchestrator*.

Este método comprueba el estado de los motores para asegurar que estos tienen un funcionamiento correcto antes de recibir cualquier orden de movimiento.

1. **executeMovement:** se ejecuta un movimiento a una posición en la que todos los fines de carrera sean activados.
2. **checkFinalPosition:** se verifica que todos los fines de carrera han sido alcanzados pudiendo concluir que el brazo es capaz de mover todos sus motores.

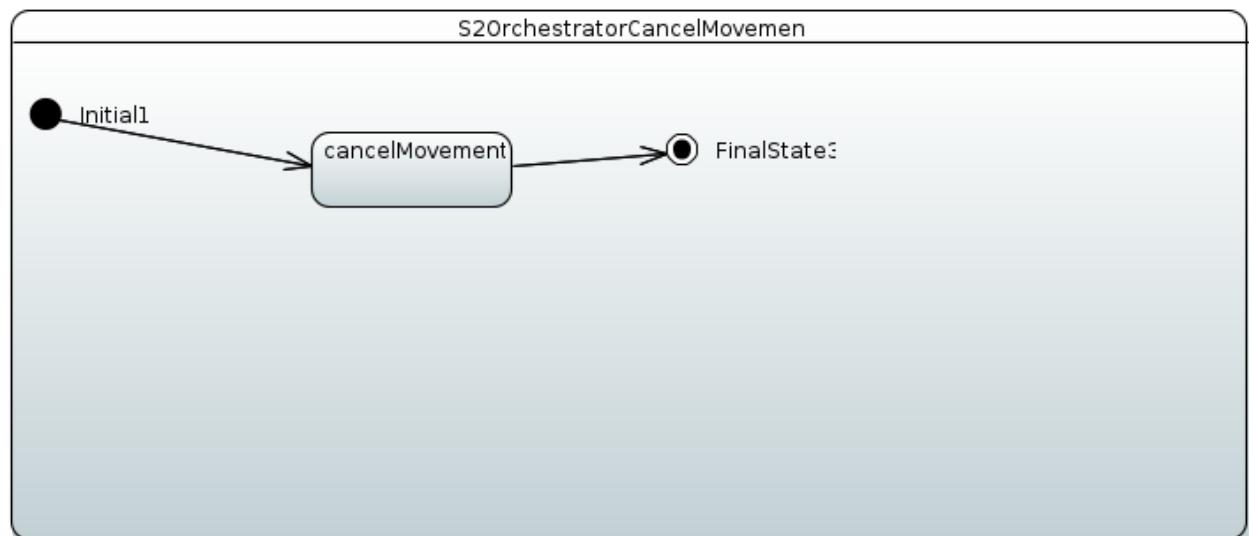


Figura 4.13: Diagrama de estados del método `CancelMovement()` del *orchestrator*.

Este método finaliza un movimiento que se este realizando.

- **cancelMovement:** se cancela el movimiento y se guarda la posición actual del brazo.

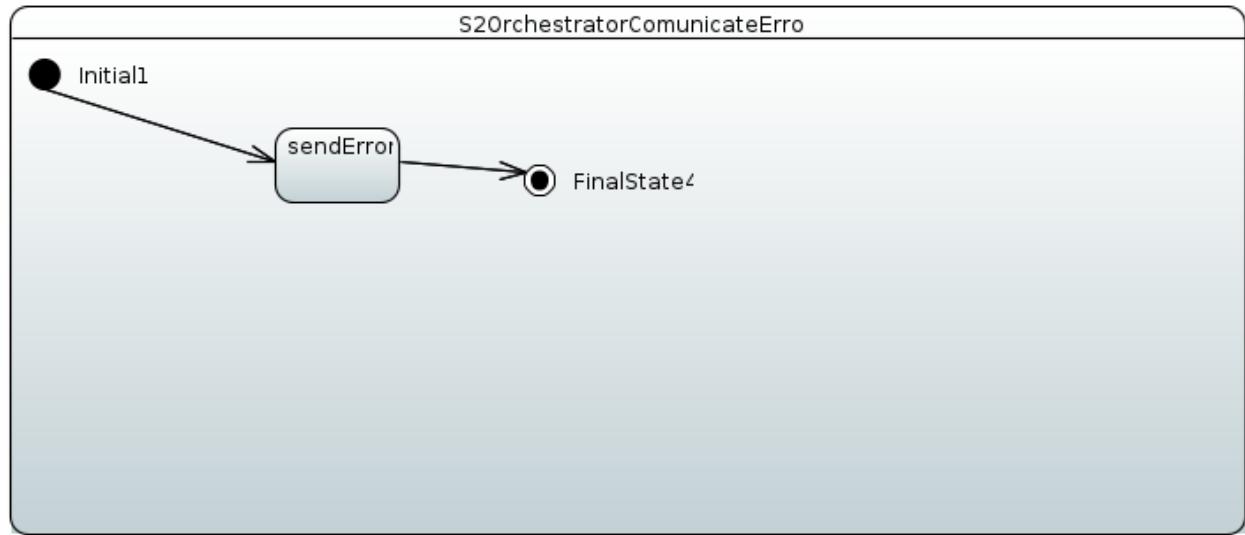


Figura 4.14: Diagrama de estados del método `ComunicateError()` del *orchestrator*.

Este método comunica un error a S1.

- **sendError:** se envía una trama de bits que representa un error ocurrido en S2.

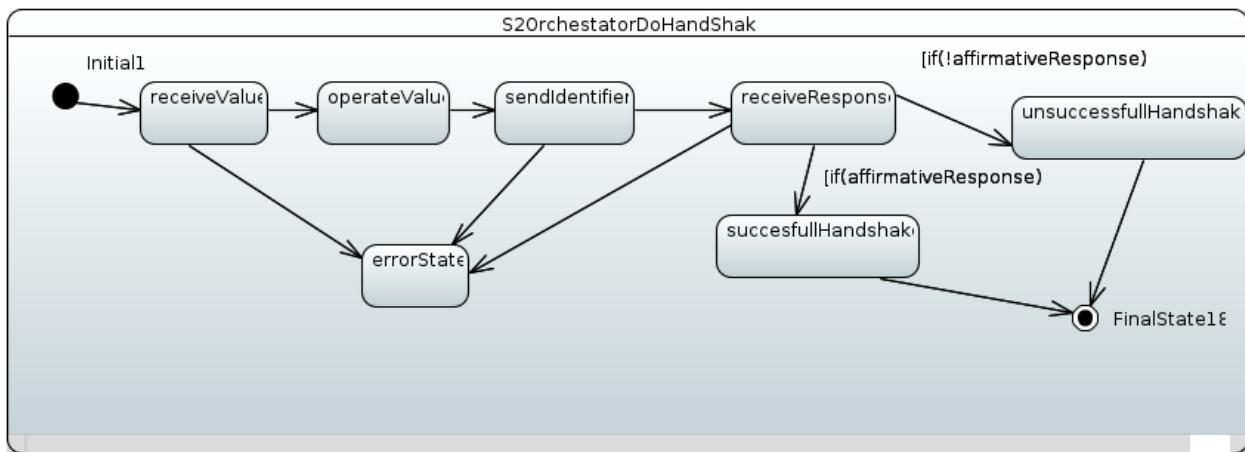


Figura 4.15: Diagrama de estados del método `DoHandShake()` del *orchestrator*.

Este método es el encargado de autenticar a los dispositivos entre sí y configurar un canal para su posterior comunicación

1. **receiveValue:** se realizan los procedimientos necesarios para recibir un valor desde S1 a través de la UART.

2. `operateValue`: se realiza una operación matemática con el valor recibido para generar de esta manera un identificador.
3. `sendIdentifier`: se envía dicho identificador de vuelta a S1.
4. `receiveResponse`: se recibe la respuesta de S1 para saber si el *handshake* ha sido realizado con éxito.
5. `successfulHandshake`: en caso de que en el estado `receiveResponse` se haya recibido una respuesta afirmativa se pasa a este estado que representa que los dispositivos han conseguido autenticarse entre sí.
6. `unsuccessfulHandshake`: en caso de que en el estado `receiveResponse` se haya recibido una respuesta negativa se pasa a este estado que representa que los dispositivos no han conseguido autenticarse entre sí.
7. `errorState`: estado de error al que se llega si en alguno de los estados ocurre algún problema inesperado.

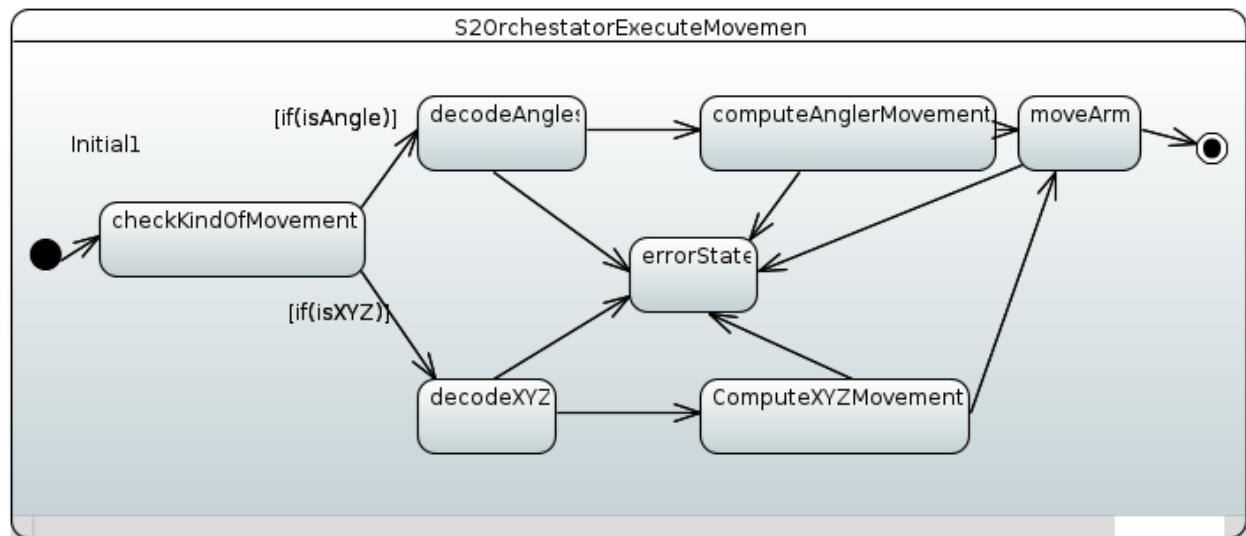


Figura 4.16: Diagrama de estados del método `ExecuteMovement()` del *orchestrator*.

Este método es el encargado de, una vez recibida la trama de bits que representa un movimiento desde S1, decidir si el movimiento ha sido representado como ángulos o posiciones cartesianas y posteriormente ordenar las operaciones necesarias para que se generen las señales PWM que moverán los motores.

1. `checkKindOfMovement`: se verifica si el movimiento ha sido transmitido como una posición cartesiana o como unos ángulos destino para los motores y se procede en consecuencia.
2. `decodeAngles`: en caso de que fueran ángulos, se transita a este estado. Se interpreta la trama de bits y se obtiene el valor numérico de los ángulos.

- 2.1. **computeAngleMovement**: se realizan comprobaciones para verificar que los ángulos están dentro de los límites del brazo y se procede a generar el vector de movimientos que se necesitan hacer para conseguir llegar desde la posición actual a la posición destino.
3. **decodeXYZ**: en caso de que fueran posiciones cartesianas se transita a este estado. Se interpreta la trama de bits y se obtienen las coordenadas en centímetros.
- 3.1. **computeXYZMovements**: para simplificar los cálculos matemáticos posteriores las posiciones cartesianas se convierten en ángulos. Se verifica si los ángulos están dentro de los límites del brazo y se procede a generar el *array* de movimientos que se necesitan hacer para conseguir llegar desde la posición actual a la posición destino.
4. **moveArm**: se envían los movimientos a los motores.
5. **errorState**: estado de error al que se llega si en alguno de los estados ocurre algún problema inesperado.

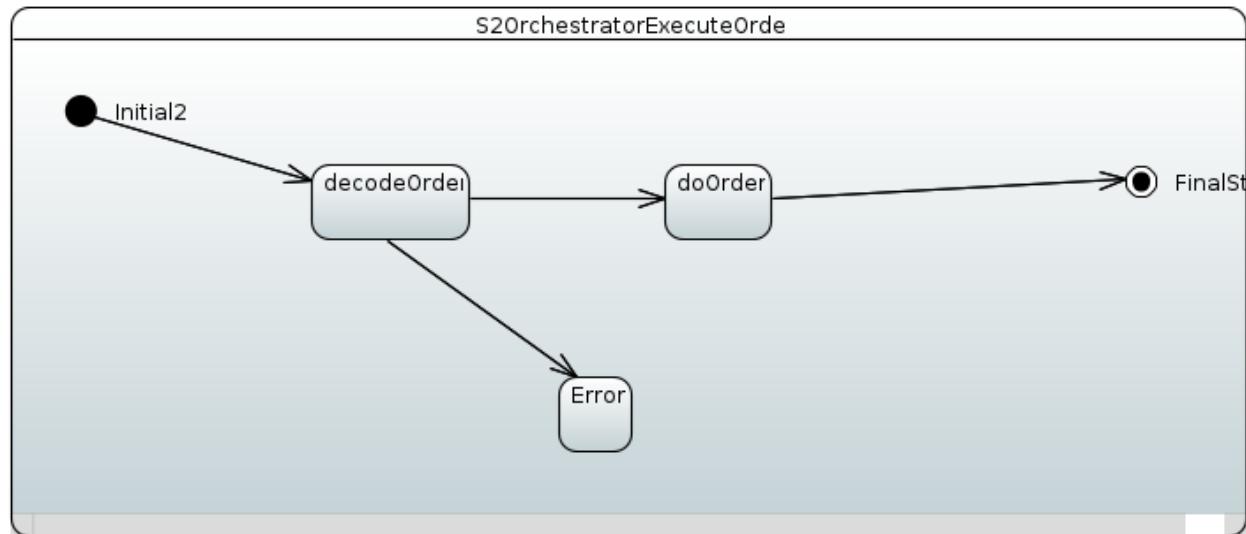


Figura 4.17: Diagrama de estados del método `ExecuteOrder()` del *orchestrator*.

Este método es el encargado de, una vez recibida la trama de bits que representa una orden distinta de realizar un movimiento desde S1, decodificar dicha orden y realizarla.

1. **decodeOrder**: se interpreta la trama de bits para obtener la orden proveniente desde S1.
2. **doOrder**: se ejecuta la orden obtenida en el estado anterior.
3. **Error**: estado de error al que se llega si en alguno de los estados ocurre algún problema inesperado.

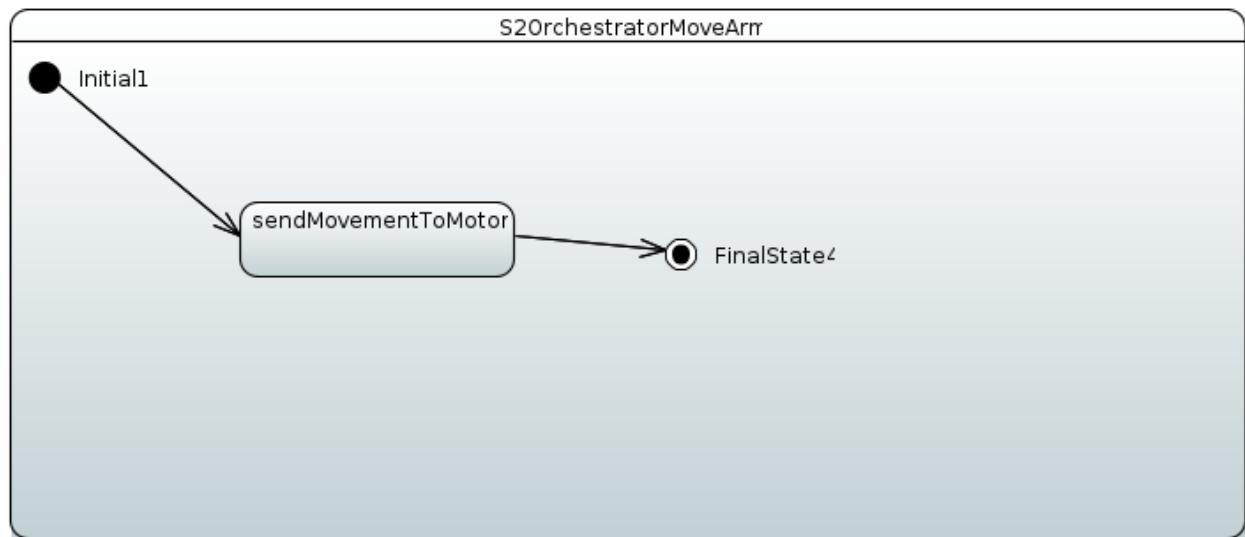


Figura 4.18: Diagrama de estados del método `moveArm()` del *orchestrator*.

Este método es el encargado de mandar los movimientos a los motores una vez estos se hayan computado.

- **sendMovementToMotors:** se mandan los movimientos necesarios a los motores.

En el caso del bloque UART tenemos los siguientes diagramas:

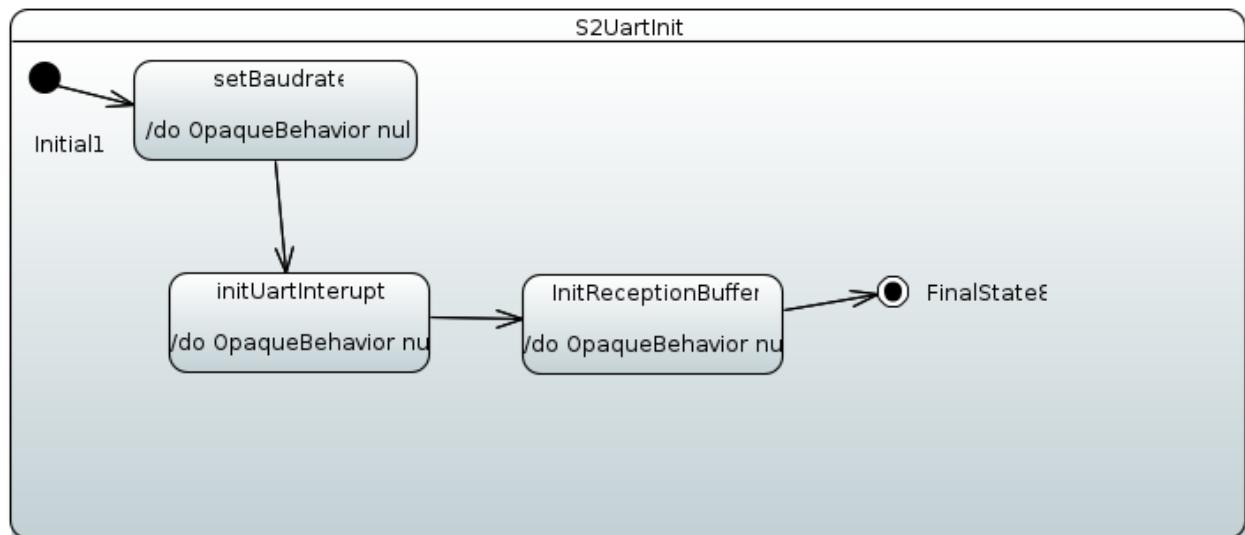


Figura 4.19: Diagrama de estados del método `uartInit()` del *UART*.

Se ejecuta este método al inicio de la comunicación a través de la UART para configurar la transmisión de datos.

1. **setBaudrate:** se establece una tasa de baudios para la transmisión asíncrona

2. `initUartInterrupt`: se configura el registro de interrupciones de tal manera que la UART sea capaz de generar una interrupción en el sistema con el objetivo de poder saber cuándo se ha recibido una nueva trama de bits.
3. `initReceptionBuffer`: se inicializa el *buffer* de recepción.

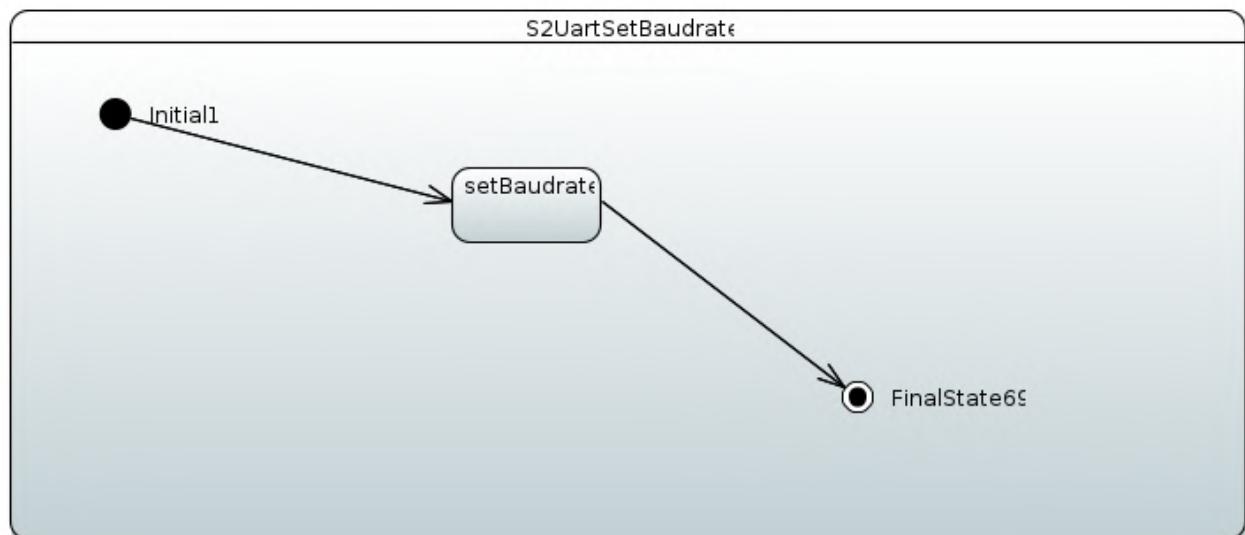


Figura 4.20: Diagrama de estados del método `setBaudrate()` del *UART*.

Se configuran los registros necesarios para obtener una tasa de baudios adecuados para la comunicación

- `setBaudrate`: se realiza la configuración que actualiza la tasa de baudios del sistema.

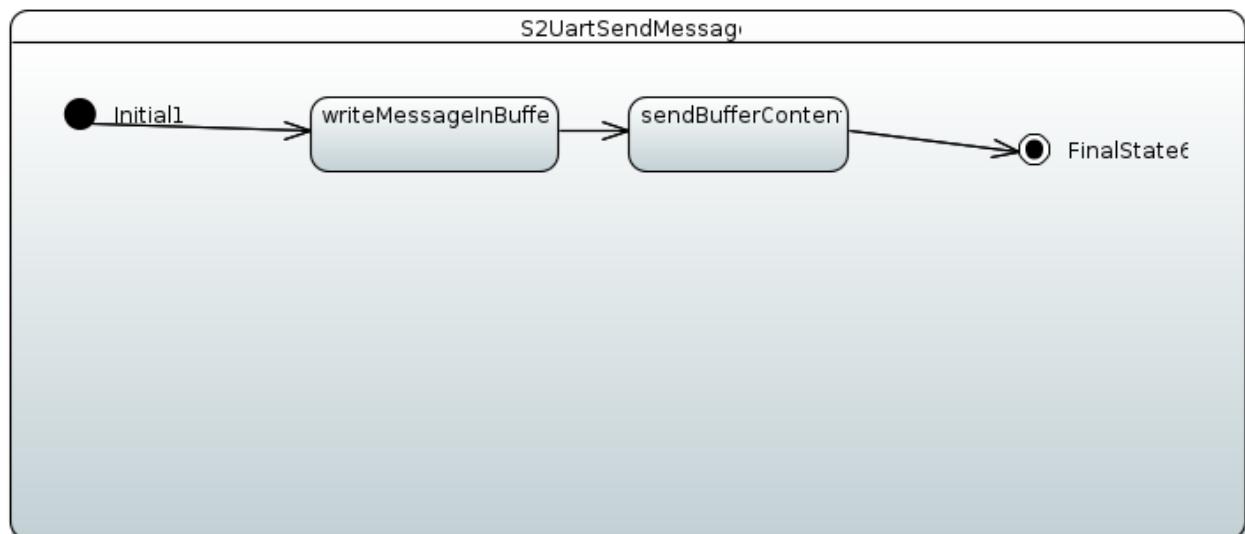


Figura 4.21: Diagrama de estados del método `sendMessage()` del *UART*.

Se escribe un mensaje en el *buffer* de envío y este es posteriormente enviado.

1. `writeMessageInBuffer`: se escribe el mensaje en el *buffer* de salida de la UART.
2. `sendBufferContent`: se envía el contenido del buffer a S1.

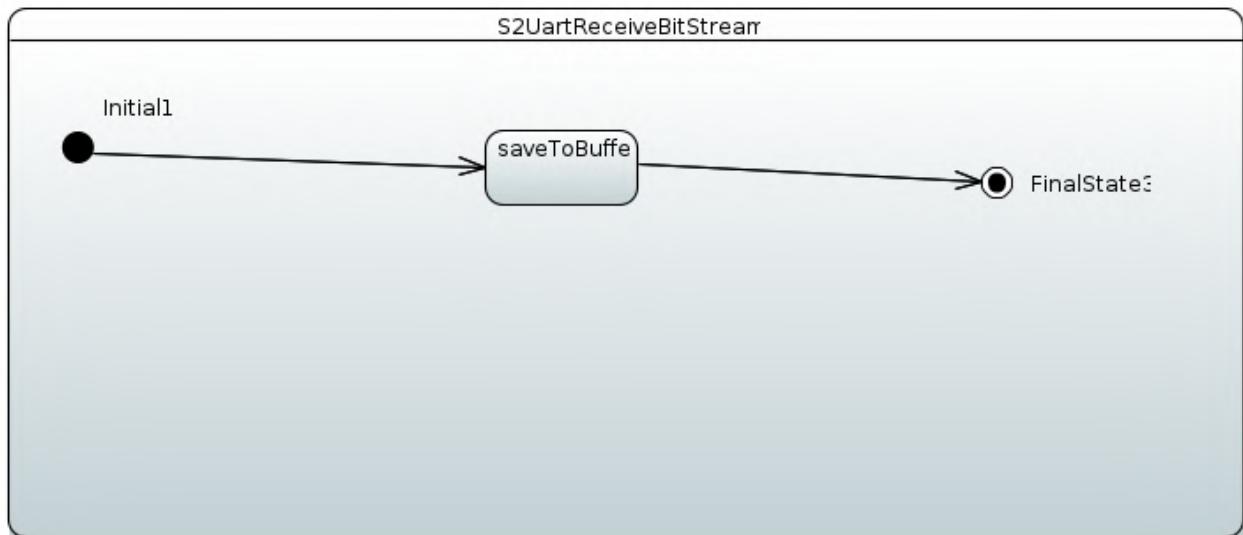


Figura 4.22: Diagrama de estados del método `receiveBitStream()` del *UART*.

Se guarda un mensaje en el *buffer* de recepción.

- `saveToBuffer`: se escribe el mensaje en el *buffer* de entrada de la UART.

En el caso del bloque `MotorHandler` tenemos los siguientes diagramas:

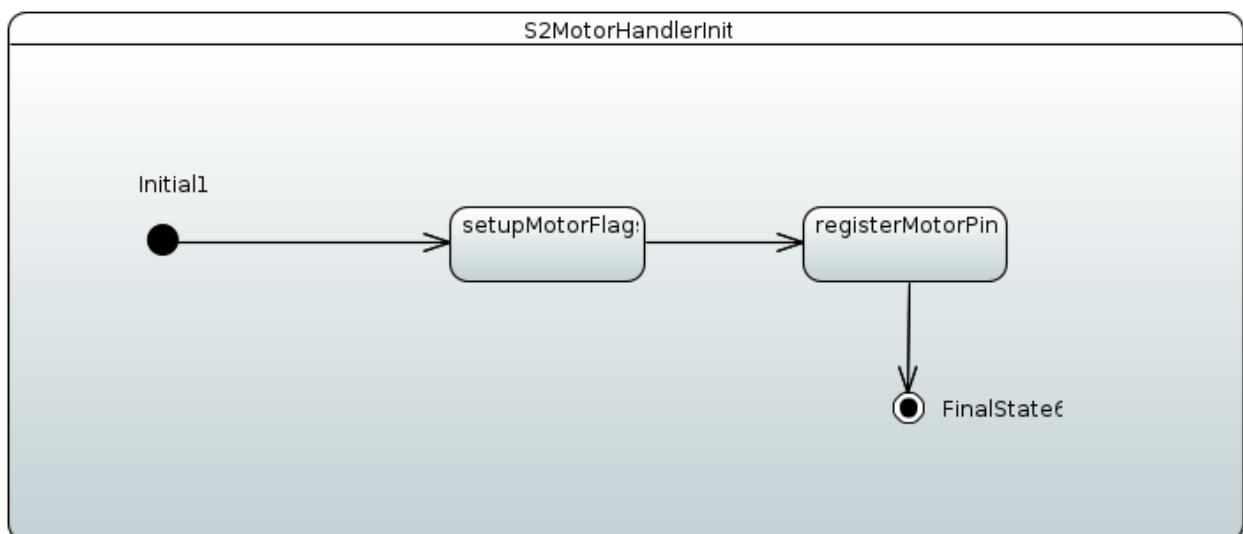


Figura 4.23: Diagrama de estados del método `init()` del *motorHandler*.

Se inicializan los *flags* y se registran los pines a los que están conectados los motores.

1. `setupMotorFlags`: se establecen los flags de los motores.
2. `registerMotorPin`: se registran los pines físicos a los que están conectados los motores con el objetivo de saber donde se deben enviar las señales PWM.

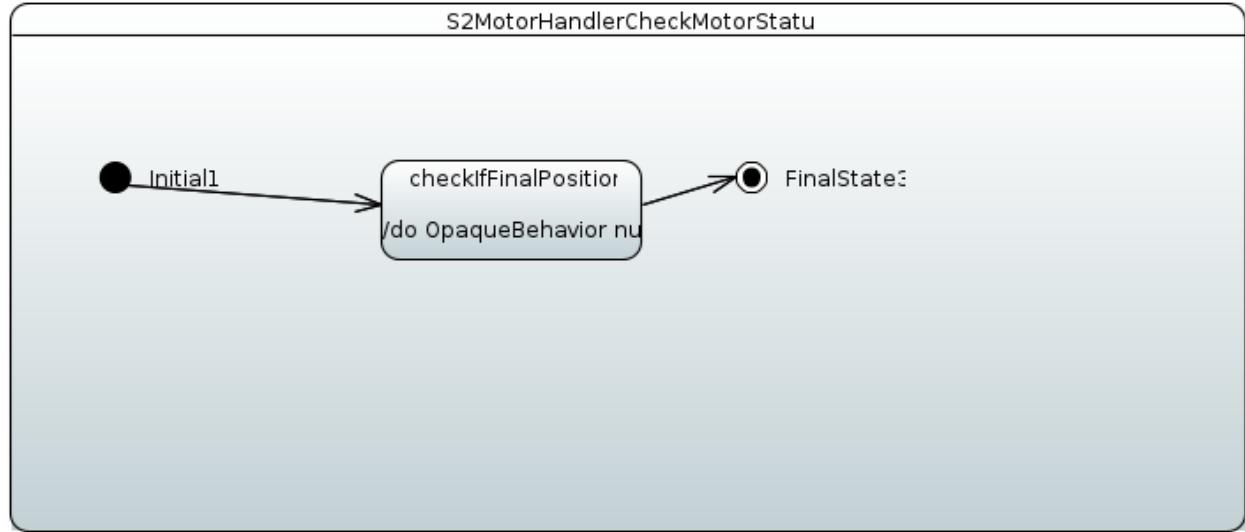


Figura 4.24: Diagrama de estados del método `checkMotorStatus()` del *motorHandler*.

Este método sirve para asegurar que los motores se encuentran en buenas condiciones de funcionamiento.

- `checkIfFinalPosition`: se envía a los motores a una posición en la que se sabe que debería estar en contacto con algún fin de carrera y, posteriormente, se verifica que dichos fines de carrera están activados. De esta manera se asegura que los motores pueden girar.

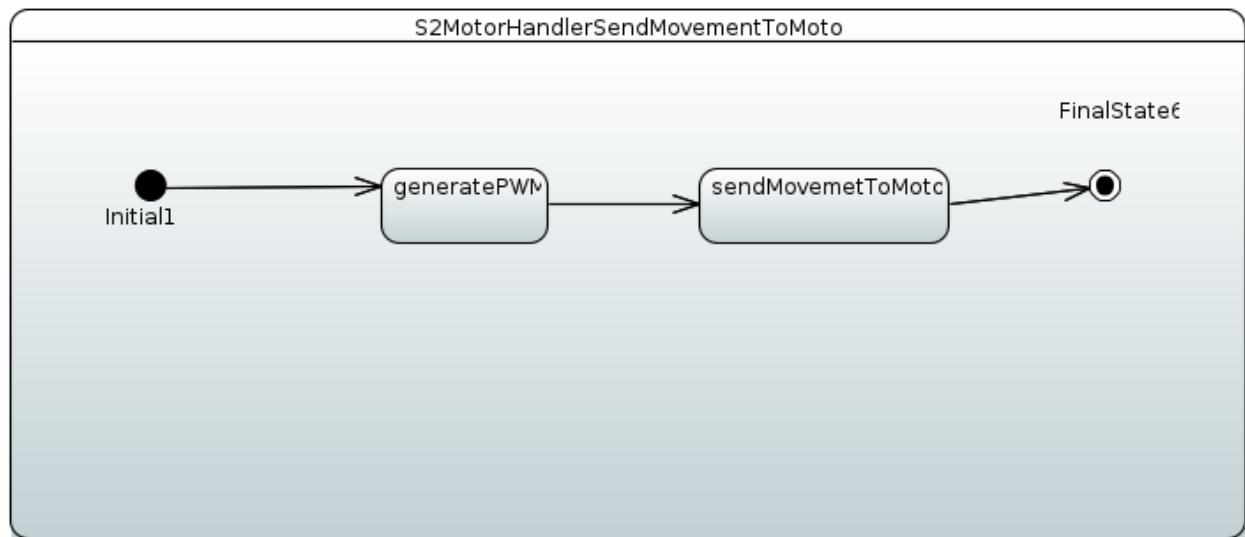


Figura 4.25: Diagrama de estados del método `sendMovementToMotors()` del *motorHandler*.

Se genera una señal PWM y se envía al motor correspondiente

1. `generatePWM`: en base al vector de movimientos que se obtiene a través del *orchestator* y del *movementComputer* se generan las señales PWM necesarias para poder realizarlos.
2. `sendMovementToMotor`: se envía la señal PWM al motor.

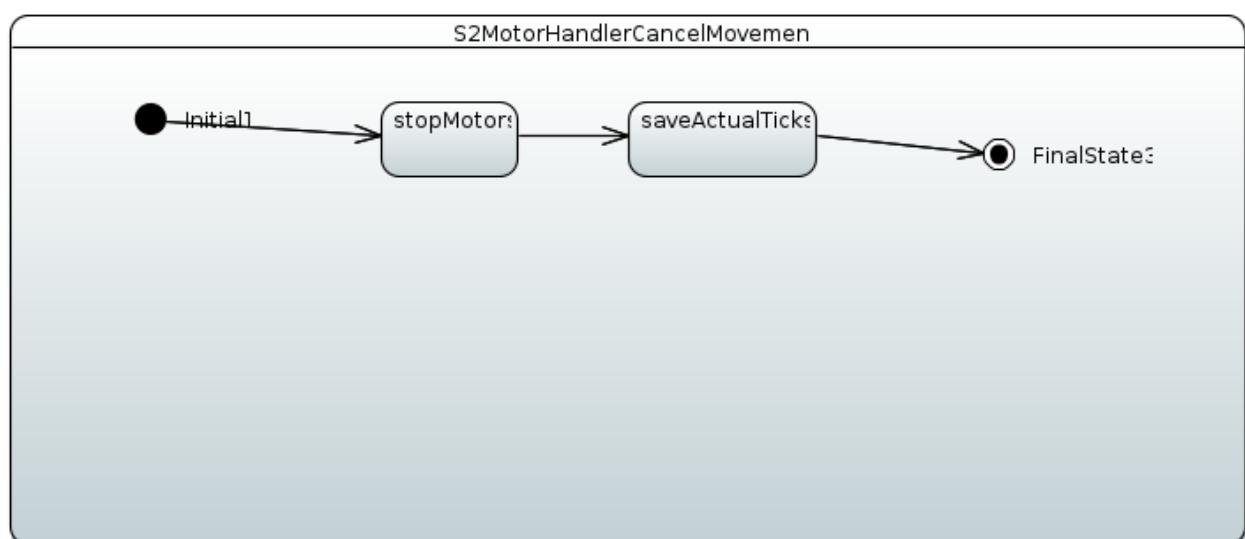


Figura 4.26: Diagrama de estados del método `cancelMovement()` del *motorHandler*.

Se cancela un movimiento que se esté ejecutando actualmente.

1. `stopMotor`: se para el motor.

2. `saveActualTicks`: se guardan los *ticks* actuales que el motor ha recorrido. Los *ticks* representan ciclos de instrucción durante los cuales el motor recibe una señal PWM específica. Se entra en detalle sobre este aspecto en apartados posteriores del documento.

En el caso del bloque `MovementComputer` tenemos los siguientes diagramas:

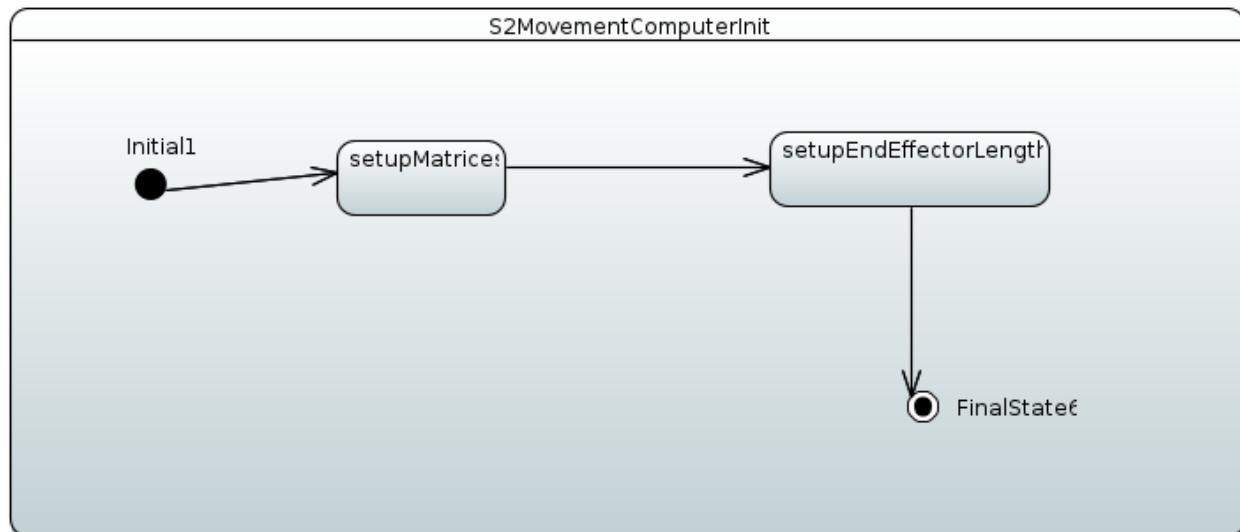


Figura 4.27: Diagrama de estados del método `init()` del *movementComputer*.

Se inicializan las matrices de la cinemática directa y se define la distancia del *end-effector* desde la base del brazo.

1. `setupMatrices`: se inicializan las matrices de la cinemática directa.
2. `setupEndEffectorLength`: se definen la distancia desde la base hasta el *end-effector*.

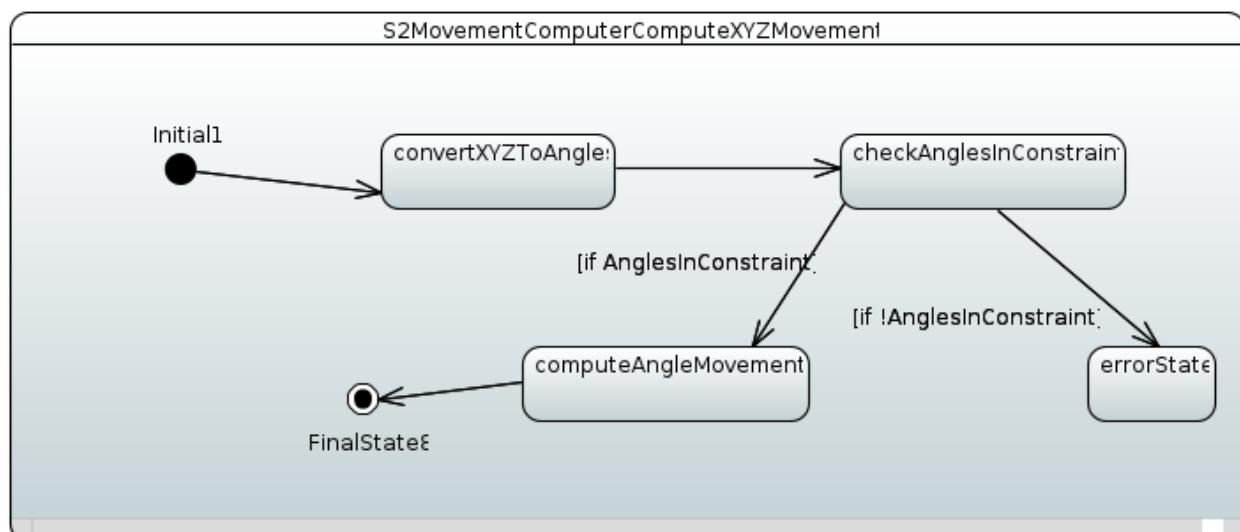


Figura 4.28: Diagrama de estados del método `computeXYZMovement()` del *movementComputer*.

1. `convertXYZToAngles`: se emplea la cinemática inversa para obtener la posición de los motores a partir de la posición del *end-effector*
2. `checkAnglesInConstraint`: Se verifica si los ángulos están dentro de las limitaciones de los motores.
3. `computeAngleMovement`: Se calcula el movimiento que se debe realizar para mover el *end-effector* desde la posición actual a la deseada.
4. `errorState`: Estado de error al que se llega si en alguno de los estados ocurre algún problema inesperado.

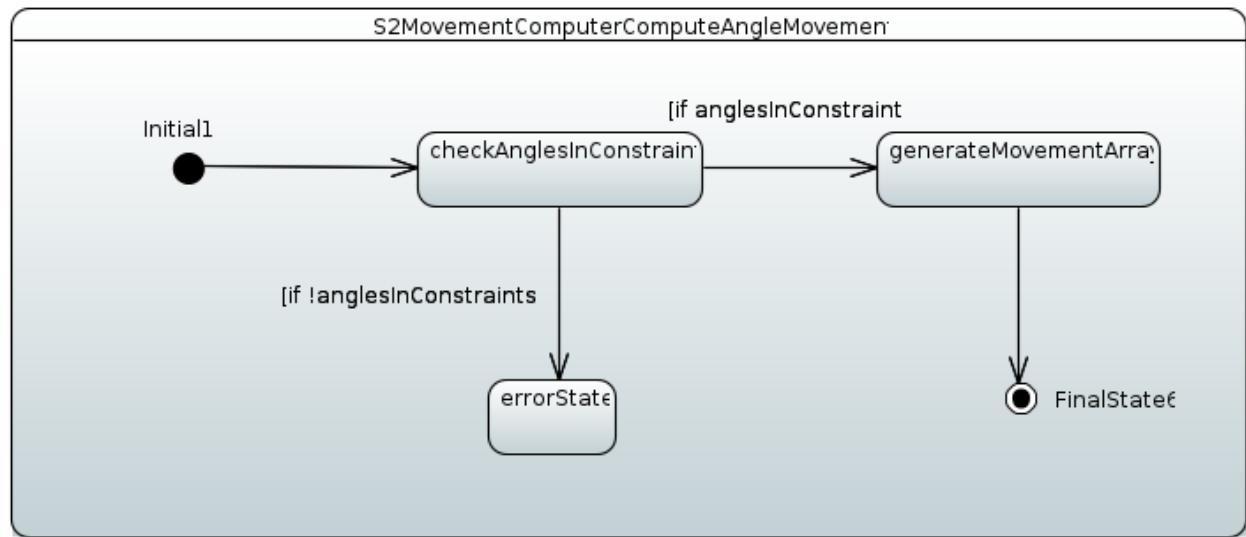


Figura 4.29: Diagrama de estados del método `computeAngleMovement()` del *movementComputer*.

1. `checkAnglesInConstraint`: se verifica si los ángulos están dentro de las limitaciones de los motores.
2. `generateMovementArray`: se generan los vectores de movimientos.
3. `errorState`: estado de error al que se llega si en alguno de los estados ocurre algún problema inesperado.

Capítulo 5

Fundamentos matemáticos del proyecto

En este proyecto el soporte matemático tiene una importancia destacada. Es por ello que se han tenido que superar retos como: el movimiento de los motores de forma coordinada para alcanzar diversas posiciones a lo largo del rango de movilidad del brazo.

La relación entre los ángulos de los ejes y el punto final del brazo no es trivial y es necesario un estudio previo de distintos factores para poder hacerlo correctamente. Por una parte, es necesario definir de la manera más precisa posible la configuración geométrica del brazo. Dicha configuración relaciona los distintos segmentos que conforman el manipulador según una convención de parámetros que trabaja sobre las posibles articulaciones que componen el brazo robótico y que se denotan por Z_i , donde i es el número de la articulación.

En particular, las relaciones a estudiar son:

- El ángulo presente entre dos articulaciones adyacentes $\widehat{Z_a Z_b}$ rad, el cual se denota por ' α_b '.
- La distancia presente entre dos articulaciones adyacentes $\overline{Z_a Z_b}$, representada por ' a_b '.
- El sentido de la rotación de una articulación, $\overrightarrow{X_a Y_a}$, denotada por ' θ_a '.

Estas relaciones permiten establecer la configuración geométrica del robot, fundamental para poder definir los movimientos posibles del mismo y generar tanto las matrices de la cinemática directa como obtener las ecuaciones de la cinemática inversa. Además, se puede obtener de la misma manera las matrices Jacobianas que permiten conseguir datos útiles como el trabajo, la velocidad o la potencia.

Para el μ Arm, se obtuvieron las siguientes configuraciones geométricas:

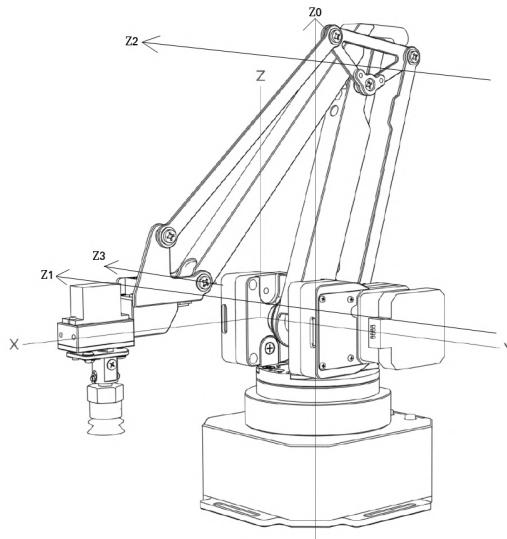
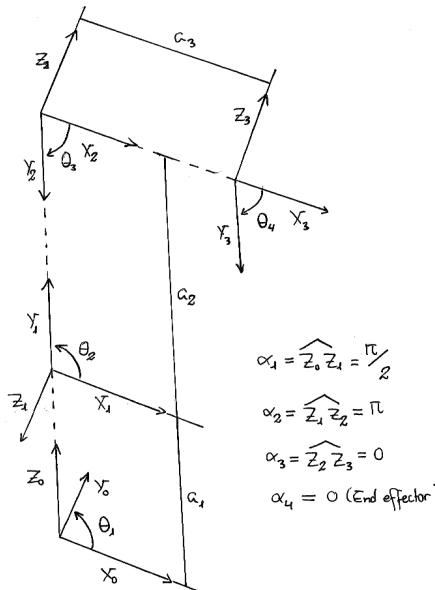
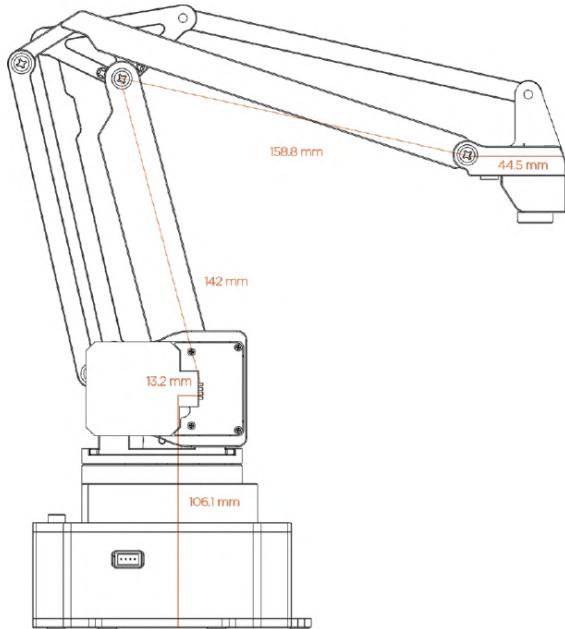


Figura 5.1: Configuración geométrica del μ Arm.

Con estos valores, ya se pueden obtener las distancias entre articulaciones así como las desviaciones entre las mismas, si las hay. En el caso particular del μ Arm, se obtiene unos datos como los siguientes (las medidas se han obtenido desde la guía del desarrollador de UFACTORY[24]):



i	a_i (mm.)	d_i (mm.)
1	13,2	106,1
2	142	0
3	158,8	0
4	44,5	0

Cuadro 5.1: Longitudes y desviaciones del manipulador μ Arm.

Figura 5.3: Longitudes del brazo robótico [24].

Esta información permite construir una tabla de *Denavit-Hartenberg* que recoge la informa-

ción del robot. Dicha tabla se conoce también como “parámetros de *Denavit–Hartenberg*”, que conforman cuatro variables que recogen, en una convención particular, la referencia de una cadena cinemática (objetos rígidos unidos a articulaciones que responden a una función matemática) o de un brazo robótico [25].

La convención de parámetros de *Denavit–Hartenberg* son los siguientes:

- d_i – distancia desde el origen de coordenadas de Z_{i-1} a la normal común¹ con Z_i .
- θ_i – el ángulo de rotación que forman el eje X_{i-1} y el eje X_i , tomando como eje de rotación Z_{i-1} . El sentido del ángulo viene definido además por x_i hasta y_i .
- a_i – la distancia de los ejes Z_i y Z_{i-1} , medida sobre su normal común ($\overline{Z_i Z_{i-1}}$).
- α_i – el ángulo, tomando como eje de rotación la normal común, desde Z_{i-1} hacia Z_i
 $(\widehat{Z_a Z_b})$.

Esta convención es especialmente interesante porque permite definir de forma precisa las relaciones entre las articulaciones, pudiendo conocer la rotación relativa entre dos de ellas y la traslación entre sus puntos. Además, aplicando las propiedades de las matrices, se puede obtener la relación absoluta entre las rotaciones y las traslaciones, lo que se traduce en conocer el punto exacto $\{x, y, z\}$ en el que se encuentra el *end-effector* cuando se giran las articulaciones $\{\theta_0, \theta_1, \dots, \theta_i\}$ rad respectivamente.

Esta relación se representa mediante una matriz, definida en la ecuación 5.1:

$${}^{i-1}T_i = \left[\begin{array}{ccc|c} \cos(\theta_i) & -\sin(\theta_i)\cos(\alpha_i) & \sin(\theta_i)\sin(\alpha_i) & a_i \cos(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i)\cos(\alpha_i) & -\cos(\theta_i)\sin(\alpha_i) & a_i \sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ \hline 0 & 0 & 0 & 1 \end{array} \right] = \left[\begin{array}{c|c} R' & T' \\ \hline 0 & 0 & 0 & 1 \end{array} \right] \quad (5.1)$$

donde R' representa la *rotación relativa* y T' la *traslación relativa*.

Otra ventaja de los parámetros de *Denavit–Hartenberg* es la posibilidad de definir elementos cinemáticos², como la velocidad ($W_{i,j}(k)$) y la aceleración ($H_{i,j}(k)$) de distintos cuerpos así como elementos dinámicos tales como la inercia (J), el momento lineal y angular (Γ) o las fuerzas y torques aplicados (Φ)².

Para un manipulador basado estructuralmente en el μ Arm, se obtienen unos parámetros de *Denavit–Hartenberg* como los mostrados en la tabla 5.2:

¹la normal común de dos articulaciones que no intersecan se define como la línea perpendicular a ambos ejes, que se usa para conocer la distancia entre ambos [26].

²si bien estos datos resultan muy útiles, para el proyecto no se consideran necesariamente relevantes, ya que están supeditados a la velocidad y a la masa (cuanto mayores sean, la aplicación de dichos elementos cinemáticos será mayor sobre los componentes del manipulador), y el brazo robótico no presenta ni una masa suficientemente elevada ni alcanza velocidades altas como para afectar en gran medida al comportamiento del mismo, pero sí se contempla realizar un estudio para completar este proyecto en una futura versión.

i	θ_i	d_i (mm.)	a_i (mm.)	α_i
1	θ_1	d_1	a_1	$\pi/2$
2	θ_2	0	a_2	π
3	θ_3	0	a_3	0
4	θ_4	0	a_4	0

Cuadro 5.2: Tabla inicial de *Denavit–Hartenberg* para un manipulador basado en el μ Arm parametrizada.

Además, también se vio que este tipo de estructuras pantográficas tienen su *end-effector* siempre paralelo al plano del suelo (equivalente a que el ángulo con el plano X es $\phi_e = \pi$), por lo que el cuarto elemento de los parámetros de *Denavit–Hartenberg* para un manipulador basado en el μ Arm se puede obviar para luego añadirlo como una traslación en el eje Z (T_Z), quedando como se muestra en la tabla 5.3:

i	θ_i	d_i (mm.)	a_i (mm.)	α_i
1	θ_1	d_1	a_1	$\pi/2$
2	θ_2	0	a_2	π
3	θ_3	0	a_3	0

Cuadro 5.3: Tabla de *Denavit–Hartenberg* para un manipulador basado en el μ Arm parametrizada.

Esta característica simplifica los cálculos del ángulo final ya que siempre es el mismo (π rad), quedando únicamente por calcular los valores de θ_2 y θ_3 .

Una vez obtenidos los datos correspondientes al robot, podemos definir múltiples relaciones entre los mismos:

- La cinemática directa, la cual permite saber el punto $\{x, y, z\}$ según unos ángulos $\{\theta_0, \theta_1, \theta_2\}$ de entrada.
- La cinemática inversa, que permite conocer qué ángulos $\{\theta_0, \theta_1, \theta_2\}$ posicionan el robot en un punto $\{x, y, z\}$.
- La matriz Jacobiana, donde se puede obtener un movimiento \vec{x} según qué velocidad haya en las articulaciones \vec{q} .
- La matriz Jacobiana inversa, la cual devuelve el valor de la velocidad en las articulaciones \vec{q} para generar un movimiento en el *end-effector* \vec{x} .

Dado que las cuatro relaciones anteriores son útiles para conocer y definir el comportamiento del robot, se estudiarán todas ellas para ver cómo se pueden utilizar en el manipulador.

5.1. Cinemática directa

La cinemática directa permite conocer la rotación relativa entre dos articulaciones junto con la traslación relativa entre las mismas, utilizando para ello la matriz definida en la ecuación 5.1.

Como se puede ver en dicha matriz, se relaciona una articulación i con el equivalente anterior, en este caso $i - 1$, obteniéndose así la rotación relativa R' y la traslación relativa T' . Si bien esta aproximación es sencilla, solo permite relacionar dos articulaciones entre sí y que estén en principio unidas por la normal común.

En este brazo se disponen de tres articulaciones, como se muestra en la tabla 5.3, por lo que interesa obtener la matriz 0T_3 , la cual relaciona directamente todas las articulaciones del brazo y permite obtener la rotación absoluta R y la traslación absoluta T .

La obtención de esta matriz es trivial y responde a la ecuación 5.2:

$${}^0T_3 = {}^0T_1 \cdot {}^1T_2 \cdot {}^2T_3 \quad (5.2)$$

Dada que la multiplicación de matrices ha de realizarse en cierto orden, los factores han de permanecer en la misma posición siempre. De esta manera, se obtienen las siguientes matrices intermedias (ecuaciones 5.3, 5.4, 5.5) y la matriz final (ecuación 5.6):

$${}^0T_1 = \begin{bmatrix} \cos(\theta_1) & 0 & \sin(\theta_1) & a_1 \cos(\theta_1) \\ \sin(\theta_1) & 0 & -\cos(\theta_1) & a_1 \sin(\theta_1) \\ 0 & 1 & 0 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.3)$$

$${}^1T_2 = \begin{bmatrix} \cos(\theta_2) & \sin(\theta_2) & 0 & a_2 \cos(\theta_2) \\ \sin(\theta_2) & -\cos(\theta_2) & 0 & a_2 \sin(\theta_2) \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.4)$$

$${}^2T_3 = \begin{bmatrix} \cos(\theta_3) & -\sin(\theta_3) & 0 & a_3 \cos(\theta_3) \\ \sin(\theta_3) & \cos(\theta_3) & 0 & a_3 \sin(\theta_3) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.5)$$

$${}^0T_3 = \begin{bmatrix} \cos(\theta_1) \cos(\theta_2 - \theta_3) & \sin(\theta_2 - \theta_3) \cos(\theta_1) & -\sin(\theta_1) \\ \sin(\theta_1) \cos(\theta_2 - \theta_3) & \sin(\theta_1) \sin(\theta_2 - \theta_3) & \cos(\theta_1) \\ \sin(\theta_2 - \theta_3) & -\cos(\theta_2 - \theta_3) & 0 \\ 0 & 0 & 0 \\ T_X + (a_1 + a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3)) \cos(\theta_1) & (a_1 + a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3)) \sin(\theta_1) & 1 \\ T_Z + a_2 \sin(\theta_2) + a_3 \sin(\theta_2 - \theta_3) + d_1 & 1 & 0 \end{bmatrix} \quad (5.6)$$

Como se puede apreciar en la matriz 5.6, se muestra un añadido a los valores de la traslación T : T_X y T_Z . Estas dos variables representan traslaciones tanto en el eje X y como en el eje Z , las cuales aparecen debido a que en el brazo se contemplan variaciones en la longitud de ciertos segmentos, que no se han tenido en cuenta a la hora de definir los parámetros de *Denavit–Hartenberg* y que, en el momento de la obtención de las matrices, no afectan directamente a los cálculos (se anulan con los senos y cosenos) pero han de aparecer en la “matriz final”, permitiendo así la obtención precisa de la posición del *end-effector*. En particular, la traslación T_Z añade la longitud del segmento a_4 que fue ignorado en los parámetros de *Denavit–Hartenberg*, como se mostró en la tabla 5.3.

Con estos valores se pueden obtener directamente las ecuaciones que relacionan los ángulos de entrada $\{\theta_1, \theta_2, \theta_3\}$ con el punto final $\{x, y, z\}$ en el cual se situará el brazo robótico (ecuación 5.7):

$$\left. \begin{array}{l} x = T_X + (a_1 + a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3)) \cos(\theta_1) \\ y = (a_1 + a_2 \cos(\theta_2) + a_3 \cos(\theta_2 - \theta_3)) \sin(\theta_1) \\ z = T_Z + a_2 \sin(\theta_2) + a_3 \sin(\theta_2 - \theta_3) + d_1 \end{array} \right\} \quad (5.7)$$

Si bien estas operaciones se pueden hacer manualmente, los cálculos simbólicos pueden resultar algo complejos y se han realizado utilizando una librería anteriormente desarrollada [18] (ver código fuente en el anexo A) y, además, se ha creado un *Jupyter Notebook* interactivo para poder realizar la configuración a medida e ir viendo los pasos que se han ido realizando [27]. Dicho cuaderno es accesible desde la URL especificada en el anexo A.1.

5.2. Cinemática inversa

La cinemática inversa se presenta como lo “más cercano” a nuestro mundo y a nuestra forma de actuar. El ser humano, como ser tridimensional, se mueve mediante coordenadas cartesianas formadas por puntos definidos en espacio conformado por los planos de los ejes XYZ , pero no se desenvuelve con la misma soltura con las coordenadas angulares. Cuando se realiza un giro en alguno de los brazos no se hace pensando: “*voy a mover el codo 15° a la derecha y el hombro 23° a la izquierda y así coloco la mano justo donde quiero*” sino que directamente se visualiza el movimiento que se pretende hacer, a dónde se quiere mover el brazo y se articulan los músculos para colocarlo en esa posición.

Por esto, cuando manipulamos un brazo robótico resulta más sencillo indicar a dónde se quiere que vaya el *end-effector* del brazo más que cuánto ha de rotar cada uno de los motores. De esta forma, el estudio de la cinemática inversa se convierte en una de las partes más importantes del modelo matemático de cualquier manipulador.

El problema surge en tanto que la cinemática inversa, a diferencia de la cinemática directa, no dispone de un método sistemático que permita obtener dicho modelo. Si bien en el punto anterior se vio cómo, a partir de una tabla de *Denavit–Hartenberg*, se calculaban las matrices que permiten obtener tanto las traslaciones como las rotaciones relativas y, al multiplicarlas, la traslación absoluta T y la rotación absoluta R , en la cinemática inversa no hay ningún modelo matemático que permita una aproximación directa genérica para cualquier manipulador.

A raíz de lo anterior, se plantean así dos maneras para poder obtener la relación entre coordenadas cartesianas y coordenadas articulares:

1. Mediante fuerza bruta. Como la obtención de la cinemática directa es siempre igual, según la precisión que se busque obtener a nivel de coordenadas cartesianas se puede plantear la opción de realizar un mapa de puntos: para un conjunto de coordenadas articulares $\{\theta_0^i, \theta_1^j, \dots, \theta_n^k\}$ se obtienen unas coordenadas cartesianas $\{x^{ij\dots k}, y^{ij\dots k}, z^{ij\dots k}\}$ (donde i, j, \dots, k representan unos ángulos en específico).

De esta manera, para el *pArm* en específico, se tienen $\{\theta_1, \theta_2, \theta_3\}$ y en total, suponiendo una precisión de un decimal considerando además un rango de giro de $[0, 180]^\circ$, se disponen de una combinación de 1800^3 posibles ángulos, lo que se traduce en un mapa de $5\,832\,000\,000$ ángulos que generan la misma cantidad de posiciones en *XYZ*. Si se quisieran usar dos decimales de precisión en ángulos (ya que hay motores capaces de ello), se tendrían pues $5,832 \cdot 10^{12}$ combinaciones de ángulos y puntos.

2. Mediante el cálculo numérico y el razonamiento matemático. Como no hay una ecuación genérica que permita el cálculo de la inversa, cualquier cálculo numérico ha de ser previamente razonado y estudiado. La aproximación a la cinemática inversa mediante este método es costosa y pueden haber situaciones en las que no resulte viable debido a la inversión en tiempo y coste: estudiar las distintas posiciones a las que puede llegar el manipulador, estudio de los puntos críticos del mismo así como plantear, si es necesario, soluciones para puntos con múltiples soluciones (aquellos a los que se puede llegar con combinaciones de los ángulos de entrada distintas).

A la hora de desarrollar la inversa, se ha de escoger entre alguna de las dos aproximaciones anteriores, teniendo en cuenta principalmente distintos criterios que pueden marcar la diferencia entre uno y otro:

- Por una parte, el rendimiento: el modelo matemático suele ser en general bastante eficiente en lo que a tiempo de cálculo se refiere, pero siempre va supeditado al manipulador que representa. Esto es, manipuladores con más grados de libertad implican en general un modelo matemático mucho más complejo, que según la complejidad o la cantidad de operaciones que lo definen puede no ser viable para el sistema en que se va a ejecutar.

Por otro lado, un mapa por su estructura y organización siempre permite el acceso a las claves y sus valores bajo un $\mathcal{O}(1)$, haciéndolos la mejor opción en términos de eficiencia si se busca una ejecución rápida.

- Por otra, la memoria: un mapa siempre requiere de mucha más memoria que una primitiva u otra estructura de datos. Principalmente se debe a su organización en memoria ya que, además de las claves y sus valores, se debe guardar un *hash* o un *set* (según esté implementada la librería) de todas y cada una de las claves para garantizar así que el tiempo de acceso sea $\mathcal{O}(1)$. Además, el mapa tendría que ir guardado o directamente en el espacio de código (y copiado a la RAM en tiempo de ejecución) o bien guardado en un fichero binario para su posterior carga en el sistema durante la ejecución, lo cual implica que sería necesario contar con ese espacio en el sistema de ficheros donde se guarde.

En cambio, el modelo matemático carece de este problema ya que se utilizan principalmente primitivas y operaciones matemáticas que se realizan directamente sobre un co-procesador, si existe, o sobre el procesador en sí. Aunque se puedan usar muchas primitivas, es difícil que alcancen en tamaño en memoria a un mapa.

- Además, hay que tener en cuenta el esfuerzo de la obtención. La aproximación por fuerza bruta requiere de bastante tiempo para la obtención del mapa al completo. Además, un cambio en la cantidad de decimales implicaría un recálculo casi completo del mapa con un aumento de tiempo exponencial, aunque se puede automatizar y que sea realizado por otro equipo.

Sin embargo, dado que el modelo matemático requiere de un razonamiento y estudio de tanto las características geométricas del manipulador como de las interacciones entre los elementos del mismo, el tiempo es en principio desconocido. Depende directamente de las aptitudes tanto matemáticas como técnicas del equipo trabajando en ello y, además, la verificación, comprobación y validación de los resultados obtenidos puede implicar tener que replantearlo y modelarlo de nuevo, necesitando así de más tiempo hasta que se consigan resultados conformes a los requisitos establecidos.

Para este proyecto se ha preferido hacer el modelo matemático ya que se plantearon las características del modelo por fuerza bruta pero fue descartado debido a una estimación de uso de memoria excesivo (no habría sido suficiente según la disponible en el dispositivo³). Además, dado que se cuenta con un procesador con gran capacidad de cómputo, las operaciones matemáticas se realizan a una gran velocidad y en particular las multiplicaciones, ya que se cuenta con un conjunto de instrucciones y con una *Arithmetic–Logic–Unit* (ALU) que permiten su realización a la misma velocidad que una suma con números de hasta 16 bits [28].

Para plantear la cinemática inversa del *pArm* se han de distinguir dos partes:

- La base (θ_0), que rota sobre el eje Y y cuyo movimiento no está supeditado al del resto de motores.
- El triángulo superior, conformado por $\{\theta_1, \theta_2\}$ donde ambos ángulos dependen de la posición final y están directamente relacionados.

Por otra parte, dada la configuración geométrica del robot, existen las siguientes premisas:

- x se encuentra comprendido en el rango $(0, A_{ML}]$, donde A_{ML} es “*Arm Maximum Length*” y viene definido por la ecuación 5.8:

$$A_{ML} = (\overline{A_L} + \overline{A_U}) \cdot \cos(\theta_{Max}^{LU}) + A_{EF_L} \quad (5.8)$$

donde cada uno de los elementos anteriores representan:

³teniendo en cuenta que habría sido necesario guardar tuplas de tres elementos por clave junto con tuplas de otros tres elementos para el valor, donde cada elemento sería de tipo `float` (lo que se traduce en 4B por elemento), habría supuesto un uso de aproximadamente: $(5,382 \cdot 10^9)^2 \cdot 4B = 1,158 \cdot 10^{20} B \approx 1,158 \cdot 10^{11} TB$, (suponiendo que las tuplas no usan espacio adicional) lo cual es inviable para el sistema.

$$\begin{cases} \overline{A_L} \equiv \text{"Arm Lower"} = 142 \text{ mm} \\ \overline{A_U} \equiv \text{"Arm Upper"} = 158,8 \text{ mm} \\ \theta_{Max}^{LU} \equiv \widehat{A_L A_{U Max}} = \frac{13\pi}{15} \text{ rad} \\ A_{EFL} \equiv \text{"Arm End-Effector Length"} = 44,5 \text{ mm} \end{cases}$$

- y por su parte se encuentra comprendido en el rango $[-A_{M_L}, A_{M_L}]$, donde A_{M_L} está definido en la ecuación anterior (ecuación 5.8).
- z pertenece al rango $[0, A_{M_H}]$, donde A_{M_H} es “Arm Maximum Height” y viene definido por la ecuación 5.9:

$$A_{M_H} = A_{B_H} + \overline{A_L} + \overline{A_U} \cdot \sin(\max_{A_L \parallel A_B} \theta_{A_L \parallel A_B}^{A_U}) \quad (5.9)$$

donde los elementos anteriores representan:

$$\begin{cases} A_{B_H} \equiv \text{"Arm Base Height"} = 106,1 \text{ mm} \\ \max_{A_L \parallel A_B} \theta_{A_L \parallel A_B}^{A_U} \equiv \text{"Ángulo máximo de } A_U \text{ cuando } A_L \parallel A_B" = \frac{\pi}{8} \text{ rad} \end{cases}$$

De esta forma, tenemos que:

$$\begin{aligned} x &\in (0, A_{M_L}] \\ y &\in [-A_{M_L}, A_{M_L}] \\ z &\in [0, A_{M_H}] \end{aligned}$$

Una vez definidas las características anteriores, se puede empezar a obtener los distintos ángulos. Por una parte, la obtención de θ_0 se puede realizar directamente como se muestra en la ecuación 5.10:

$$\theta_0 = \begin{cases} \arctan\left(\frac{y}{x}\right), & x \neq 0 \\ \frac{\pi}{2}, & x = 0, y \geq 0 \\ -\frac{\pi}{2}, & x = 0, y < 0 \end{cases} \quad (5.10)$$

La obtención de los dos ángulos restantes $\{\theta_1, \theta_2\}$ es más compleja y se han de realizar previamente ciertas modificaciones en los puntos de entrada.

La idea principal radica en plantear la estructura superior del brazo como un triángulo y, mediante operaciones y leyes trigonométricas, ir obteniendo distintos parámetros hasta finalmente conseguir los ángulos finales. A modo de guía, se muestra en la figura 5.4 una representación de cómo está el triángulo sobre el brazo robótico.

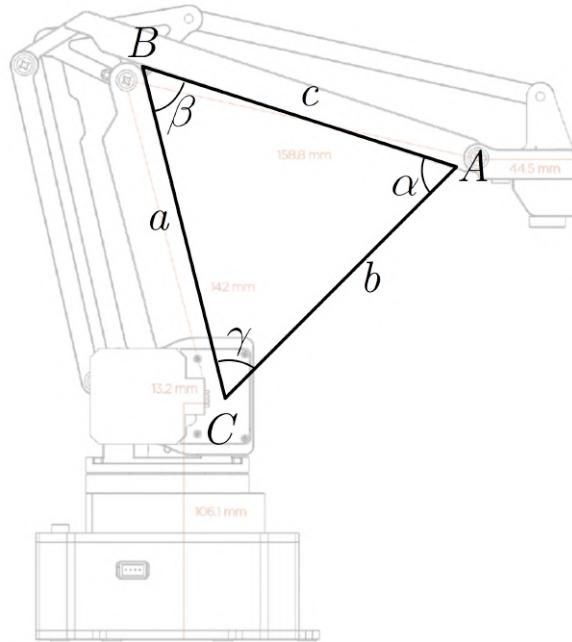


Figura 5.4: Parte superior del *pArm* con el triángulo para obtener la cinemática inversa.

La distribución en particular del triángulo permite aplicar el teorema del coseno (ecuación 5.11) y obtener así los lados o bien los ángulos.

$$c^2 = a^2 + b^2 - 2ab \cos(\gamma) \quad (5.11)$$

En particular, para el brazo robótico se conocen siempre el tamaño de los lados (según la figura 5.4) ‘a’ y ‘c’, pero el lado ‘b’ es variable. Según las medidas del brazo se tiene que:

$$\begin{cases} a = 142 \text{ mm} \\ c = 158,8 \text{ mm} \end{cases}$$

Como dicho lado ‘b’ es desconocido en principio y además los ángulos $\{\gamma, \beta, \alpha\}$ también lo son, no se podría aplicar el teorema del coseno ya que se necesita cumplir alguna de las siguientes condiciones:

- Se conocen dos lados y el ángulo entre ellos.
- Se conocen los tres lados.
- Se conocen dos lados y el ángulo opuesto a ellos.

Dado que se conoce siempre dos de los lados, se ha de estimar cuánto vale el lado restante ya que, teniendo los tres lados $\{a, b, c\}$, se puede obtener el ángulo ‘ γ ’ aplicando una forma

particular del teorema del coseno (ecuación 5.12):

$$\gamma = \arccos \left(\frac{-a^2 - b^2 + c^2}{-2ab} \right) \quad (5.12)$$

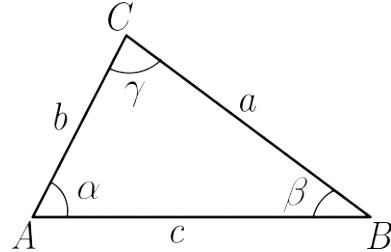


Figura 5.5: Triángulo para la aplicación del teorema del coseno en la ecuación 5.12.

Del triángulo mostrado en la figura 5.4 se conocen siempre las longitudes ‘ a ’ y ‘ c ’. La distancia ‘ b ’ sin embargo también se puede obtener. Para la cinemática inversa se tiene el punto (x, y, z) que referencia la posición final del *end-effector*. En la figura 5.4, el lado ‘ b ’ se encuentra siempre situado sobre el plano definido por los lados ‘ a ’ y ‘ c ’, por lo que de los puntos anteriores se sabe que el *end-effector* se sitúa en: $P_{ee} = (x', y, z')$, por lo que se puede definir un vector desde la base $\overrightarrow{x'yz'}$ que represente el lado ‘ b ’. Como se puede apreciar, no se usan los puntos (x, z) originales sino que previamente es necesario modificarlos:

- Para la coordenada ‘ x ’ se ha de reducir la longitud del *end-effector* ($\overline{A_{EF_L}}$) junto con la desviación de la base ($\overline{A_{B_D}}$) (ecuación 5.13):

$$x' = x - \overline{A_{EF_L}} - \overline{A_{B_D}} = x - 44,5 - 13,2 = x - 57,7 \quad (5.13)$$

- Para la coordenada ‘ z ’ es necesario quitar la altura de la base (A_{B_h}) además de añadir la altura del *end-effector* h_o para dejar un punto relativo al (0). Así, el punto z' se puede definir como (ecuación 5.14):

$$z' = z - A_{B_h} + h_o = z - 106,1 + 16,01 = z - 90,09 \quad (5.14)$$

Como el movimiento del brazo se realiza sobre un plano tridimensional, la posición y longitud desde la base hasta P_{ee} requiere tener en cuenta las tres coordenadas (x, y, z) . Por ejemplo, la posición $y = 200$ mm solo es alcanzable si el brazo está estirado al completo, pero en ese punto $x = 0$, al igual que la posición $x = 200$ mm solo es alcanzable si $y = 0$ (esto se muestra en la figura 5.8).

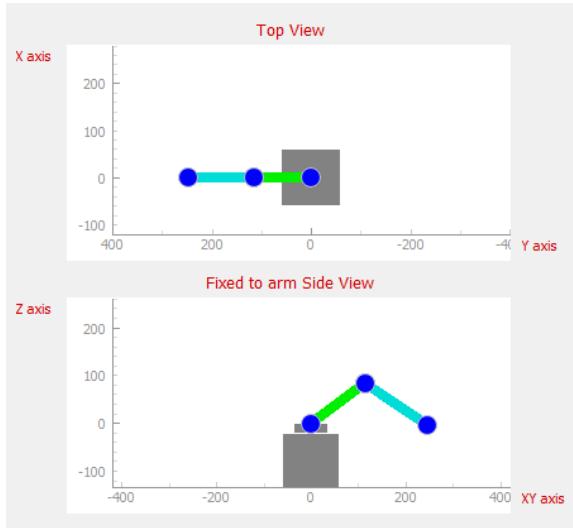


Figura 5.6: Posición máxima en ‘y’, donde $x = 0$.

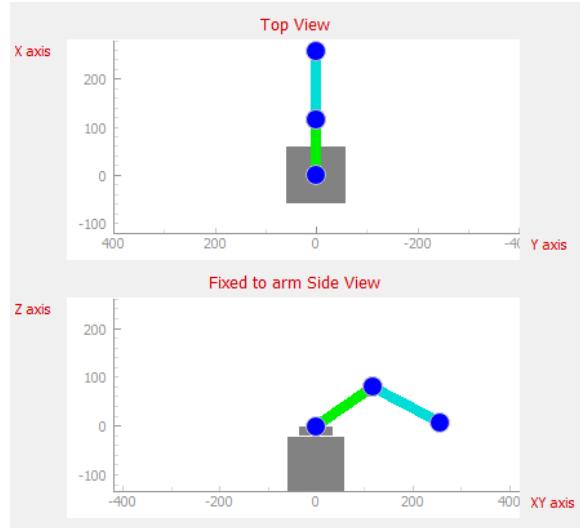


Figura 5.7: Posición máxima en ‘x’, donde $y = 0$.

Figura 5.8: Demostración de la correlación entre ‘x’ e ‘y’.

Con estos datos ya obtenidos se puede definir el lado ‘b’ como:

$$b = \left| \overrightarrow{x'y'z'} \right| = \sqrt{x'^2 + y'^2 + z'^2}$$

Cabe destacar que h_o varía según el *end-effector* que se encuentre acoplado al brazo robótico. Para el μ Arm, según la documentación oficial, dichas alturas varían y son[29]:

- 74,55 mm para el *end-effector* normal.
- 51,04 mm para el cabezal láser.
- 74,43 mm para el cabezal 3D.
- 74,43 mm para el cabezal con bolígrafo.
- 16,01 mm si no hay ningún *end-effector* conectado.

Para el p Arm, en esta primera versión del desarrollo, se establece $h_o = 16,01$ mm.

Con las modificaciones en los lados ya listas, se puede definir un triángulo que cumple que:

- Tiene dos lados fijos, ‘a’ y ‘c’, donde:

$$\begin{cases} a = \overline{A_L} = 142,07 \text{ mm} \\ c = \overline{A_U} = 158,8 \text{ mm} \end{cases}$$

- Tiene un lado variable ‘b’ definido por el vector $\overrightarrow{x'y'z'}$ y cuya longitud es: $b = \left| \overrightarrow{x'y'z'} \right| = \sqrt{x'^2 + y'^2 + z'^2}$.

Así, el triángulo resultante se define según las siguientes dimensiones y ángulos (figura 5.9):

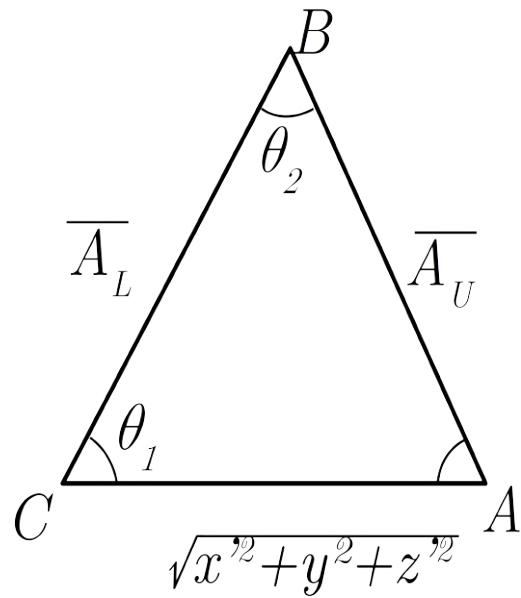


Figura 5.9: Triángulo resultante tras aplicar las modificaciones a los lados.

En particular, el triángulo anterior se puede colocar a modo de referencia sobre el brazo tal como se muestra en la imagen 5.10:

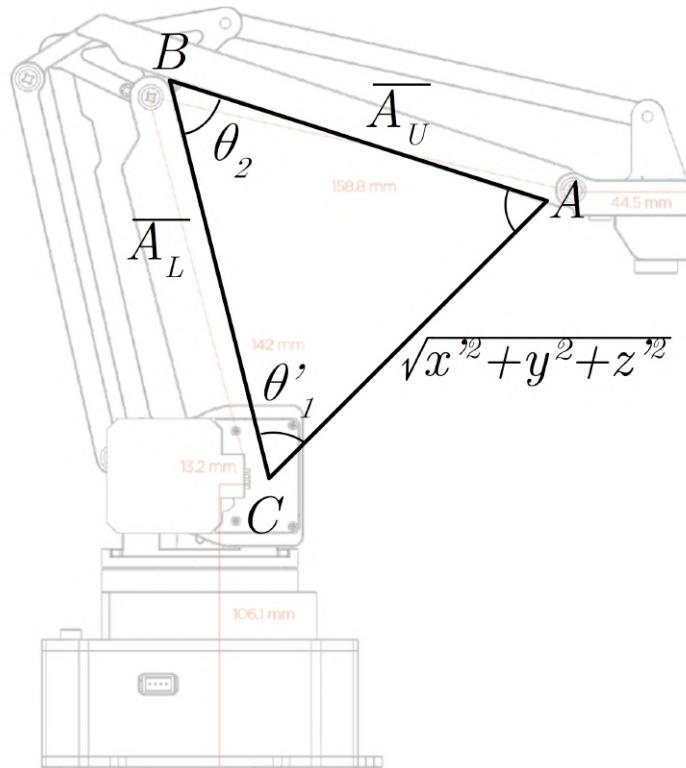


Figura 5.10: El triángulo colocado a modo de referencia sobre el *pArm*.

Para el triángulo mostrado en la figura 5.9 se aplica dos veces el teorema del coseno (ecuación 5.12) para obtener los valores de θ'_1 (ver ecuación 5.15) y θ_2 (ver ecuación 5.16):

$$\theta'_1 = \arccos \left(\frac{-\overline{A_L}^2 - (x'^2 + y^2 + z'^2) + \overline{A_U}^2}{-2\overline{A_L}\sqrt{x'^2 + y^2 + z'^2}} \right) \quad (5.15)$$

$$\theta_2 = \arccos \left(\frac{-\overline{A_L}^2 - \overline{A_U}^2 + x'^2 + y^2 + z'^2}{-2\overline{A_L} \cdot \overline{A_U}} \right) \quad (5.16)$$

Como se puede apreciar en las ecuaciones anteriores, todavía no se tienen los valores finales de los ángulos sino una primera aproximación a ellos. Esto es debido a que el ángulo θ_1 que aparece en el triángulo de la figura 5.9 no empieza desde un plano paralelo paralelo al plano del suelo, en este caso, el plano XY (tal y como se puede ver en la figura 5.10).

Para poder obtener los ángulos reales se ha de añadir el ángulo ' ϕ ' que relaciona el triángulo 5.10 con el plano del suelo, tal y como se puede ver en la figura 5.11:

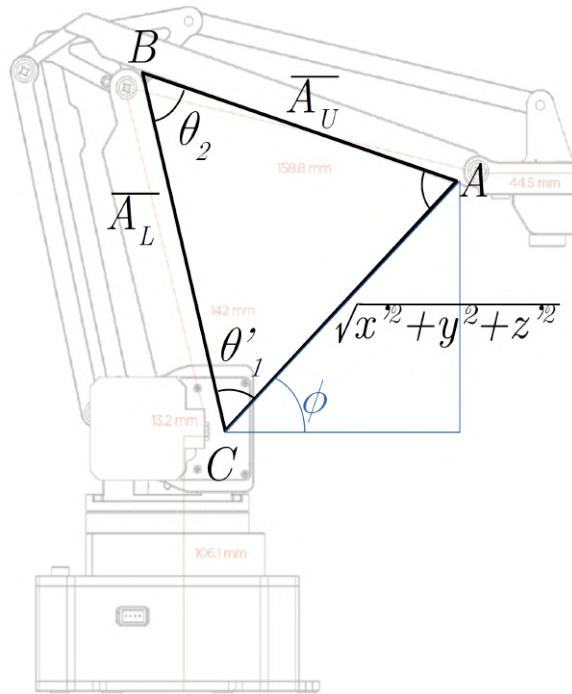


Figura 5.11: Triángulo final orientativo junto con el ángulo ϕ respecto al plano del suelo.

La obtención de dicho ángulo se muestra en la ecuación 5.17

$$\phi = \arctan \left(\frac{z'}{\sqrt{x'^2 + y^2}} \right) \text{ rad} \quad (5.17)$$

y θ_1 se obtiene mediante la ecuación 5.18⁴:

$$\theta_1 = \theta'_1 - \phi \quad (5.18)$$

Finalmente, por seguridad, se puede comprobar que en efecto los distintos ángulos obtenidos están comprendidos dentro del rango de movimiento de cada una de las articulaciones [29]:

$$\begin{aligned}\theta_0 &\in \left[\frac{\pi}{18}, \frac{151}{180}\pi \right] \\ \theta_1 &\in \left[0, \frac{113}{150}\pi \right] \\ \theta_2 &\in \left[0, \frac{1199}{1800}\pi \right]\end{aligned}$$

5.3. Funciones jacobianas

Previo a comenzar este apartado, se quiere destacar que dicho apartado ha sido introducido con la única intención de plantear un análisis estrictamente cinemático (no dinámico) del movimiento del robot.

Las matrices Jacobianas son una herramienta que permite definir la relación dinámica entre dos representaciones diferentes de un sistema. Para un manipulador de n grados de libertad (con $n > 1$) se puede definir la posición del mismo de dos formas posibles:

1. Mediante la posición y orientación del *end-effector*, denominado por x .
2. Mediante el conjunto de los ángulos de las articulaciones, denominado por q .

El modo de funcionamiento de las matrices Jacobianas se puede definir como el efecto que se produce en el *end-effector* ‘ x ’ tras un movimiento de las articulaciones ‘ q ’, entendiendo así la Jacobiana como la matriz transformada de la velocidad.

Formalmente, la matriz Jacobiana se define como un conjunto de ecuaciones diferenciales parciales (denotado en la ecuación 5.19):

$$J = \frac{\partial x}{\partial q} \quad (5.19)$$

la cual puede ser expresada como:

$$\dot{x} = J \cdot \dot{q} \quad (5.20)$$

⁴el ángulo ϕ se suma o se resta según la orientación del triángulo ya que la arcotangente varía en signo según la posición de los puntos x' y z' , tal como se muestra en la documentación al desarrollador[29].

donde \dot{x} y \dot{q} representan las derivadas de x, q respecto al tiempo.

En la ecuación 5.20 se expresa que la velocidad del *end-effector* es igual al producto de la Jacobiana J multiplicada por la velocidad de las articulaciones. ¿Para qué resulta útil tener estos datos? La expresión 5.20 permite el trabajar con trayectorias en un espacio diferente al que se dispone normalmente[30]. Esto permite el control del *end-effector* mediante la generación de señales de control (en términos de fuerza) a aplicar en (x, y, z) . Las matrices Jacobianas pues permiten un cálculo directo de las señales de control en un espacio controlado, como son los torques de los motores/articulaciones, dada otra señal que no controlamos, como la fuerza a aplicar en el *end-effector*.

Anteriormente se ha visto que la Jacobiana representa la relación entre velocidades parciales del *end-effector* y las articulaciones, pero se ha hablado de trabajar con las fuerzas de cada uno de ellos. Para el *pArm*, se pueden definir las siguientes premisas:

$$\begin{aligned} x &= [x, y, z]^T \\ q &= [\theta_0, \theta_1, \theta_2]^T \end{aligned}$$

Como se conoce la velocidad, se puede definir el trabajo (W) como la fuerza que hay que aplicar durante una distancia, definido por la ecuación 5.21. Por otra parte, la potencia (P) se define como la cantidad de trabajo efectuado por unidad de tiempo[31], definido por la ecuación 5.22.

$$W = \int F^T \cdot v \, dt \quad (5.21)$$

$$P = \frac{W}{\Delta t} \quad (5.22)$$

Atendiendo a lo comentado anteriormente, se puede afirmar que es equivalente representar el movimiento del brazo articulado en base al movimiento de sus articulaciones a representarlo en base a la velocidad del *end-effector*.

Construyendo la matriz Jacobiana

Como se mostró anteriormente, la velocidad del *end-effector* se puede expresar como el producto de la matriz Jacobiana por la velocidad de las articulaciones (ecuación 5.20). Para dicha ecuación se tienen los siguientes datos:

$$\dot{x} = J \cdot \dot{q} \left\{ \begin{array}{l} \dot{x} = (X_e, Y_e, Z_e, \phi_e) \\ \dot{q} = (\theta_0, \dots, \theta_n, d_1, \dots, d_n) \\ J = \text{matriz Jacobiana} \end{array} \right.$$

La obtención de la matriz Jacobiana $J(\dot{q})$ se ha de realizar obteniendo las submatrices Jacobianas que relacionan la velocidad lineal ‘ v ’ y la velocidad angular ‘ ω ’. La matriz Jacobiana que relaciona la velocidad lineal se define como (ecuación 5.23):

$$J_v(\dot{q}) = \begin{bmatrix} \frac{\partial X_e}{\partial \theta_0} & \frac{\partial X_e}{\partial \theta_1} & \dots & \frac{\partial X_e}{\partial \theta_n} & \frac{\partial X_e}{\partial d_1} & \dots & \frac{\partial X_e}{\partial d_n} \\ \frac{\partial Y_e}{\partial \theta_0} & \frac{\partial Y_e}{\partial \theta_1} & \dots & \frac{\partial Y_e}{\partial \theta_n} & \frac{\partial Y_e}{\partial d_1} & \dots & \frac{\partial Y_e}{\partial d_n} \\ \frac{\partial Z_e}{\partial \theta_0} & \frac{\partial Z_e}{\partial \theta_1} & \dots & \frac{\partial Z_e}{\partial \theta_n} & \frac{\partial Z_e}{\partial d_1} & \dots & \frac{\partial Z_e}{\partial d_n} \end{bmatrix} \quad (5.23)$$

y la matriz que relaciona la velocidad angular se define como (ecuación 5.24):

$$J_\omega(\dot{q}) = \begin{bmatrix} \frac{\partial \phi_X}{\partial \theta_0} & \frac{\partial \phi_X}{\partial \theta_1} & \dots & \frac{\partial \phi_X}{\partial \theta_n} & \frac{\partial \phi_X}{\partial d_1} & \dots & \frac{\partial \phi_X}{\partial d_n} \\ \frac{\partial \phi_Y}{\partial \theta_0} & \frac{\partial \phi_Y}{\partial \theta_1} & \dots & \frac{\partial \phi_Y}{\partial \theta_n} & \frac{\partial \phi_Y}{\partial d_1} & \dots & \frac{\partial \phi_Y}{\partial d_n} \\ \frac{\partial \phi_Z}{\partial \theta_0} & \frac{\partial \phi_Z}{\partial \theta_1} & \dots & \frac{\partial \phi_Z}{\partial \theta_n} & \frac{\partial \phi_Z}{\partial d_1} & \dots & \frac{\partial \phi_Z}{\partial d_n} \end{bmatrix} \quad (5.24)$$

De esta manera, la matriz Jacobiana ‘ J ’ se puede definir como (ecuación 5.25)⁵:

$$J_{ee}(\dot{q}) = \begin{bmatrix} J_v(\dot{q}) \\ J_\omega(\dot{q}) \end{bmatrix} = \begin{bmatrix} - (a_1 + a_2 \cos(\theta_1) + a_3 \cos(\theta_1 - \theta_2)) \sin(\theta_0) & (-a_2 \sin(\theta_1) - a_3 \sin(\theta_1 - \theta_2)) \cos(\theta_0) & a_3 \sin(\theta_1 - \theta_2) \cos(\theta_0) \\ (a_1 + a_2 \cos(\theta_1) + a_3 \cos(\theta_1 - \theta_2)) \cos(\theta_0) & (-a_2 \sin(\theta_1) - a_3 \sin(\theta_1 - \theta_2)) \sin(\theta_0) & a_3 \sin(\theta_0) \sin(\theta_1 - \theta_2) \\ 0 & a_2 \cos(\theta_1) + a_3 \cos(\theta_1 - \theta_2) & -a_3 \cos(\theta_1 - \theta_2) \\ 0 & 1 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad (5.25)$$

Una de las utilidades de la matriz Jacobiana es la obtención de los puntos críticos, es decir, aquellos en los que el determinante de dicha matriz se hace cero. Los puntos críticos resultan de especial interés ya que definen posiciones en el manipulador que o bien son inalcanzables o bien someten a la estructura física del mismo a una gran tensión, pudiendo resultar dañado en el proceso o de llegar a una “posición de no retorno”, donde los motores que componen el brazo puede que no dispongan de suficiente fuerza para moverse a otra posición.

⁵el cálculo de las matrices Jacobianas puede resultar complejo de realizar sobre todo a nivel simbólico, por lo que se deja en el anexo A.1 un enlace a un *Jupyter Notebook* que agiliza y guía durante el proceso de obtención de estas matrices. El código fuente para su obtención no obstante se encuentra disponible en el anexo A.2.

Para la matriz Jacobiana anterior (ecuación 5.25), se obtiene el siguiente determinante (ecuación 5.26):

$$|J_{ee}(\dot{q})| = -a_2 a_3 (a_1 + a_2 \cos(\theta_1) + a_3 \cos(\theta_1 - \theta_2)) \sin(\theta_2) \quad (5.26)$$

Analíticamente se puede observar que los puntos críticos del *pArm* se dan para los valores de $\theta_2 = 0$ y $\theta_2 = \pi$, punto en el que el brazo está o bien completamente recogido o bien completamente estirado. La cuestión radica en que, viendo la configuración geométrica del brazo robótico, el ángulo de π rad se vuelve inalcanzable ya que los valores máximos del ángulo θ_2 son [29]:

$$\theta_2 \in \left[0, \frac{1199}{1800}\pi\right]$$

Por el contrario, la posición de 0 rad se habrá de tener en cuenta para evitar que el brazo esté expuesto a un nivel elevado de tensión durante tiempo prolongado. Entre los dos segmentos superiores del brazo robótico se situará un fin de carrera a efectos de evitar dicha tensión además de regular y calibrar los motores.

Al igual que en el caso de la cinemática directa, se puede obtener una matriz Jacobiana inversa que permite, dada la velocidad lineal del *end-effector* \dot{x} , obtener qué par han de generar las articulaciones \dot{q} para obtener dicha fuerza. La Jacobiana inversa depende directamente de que el determinante sea distinto de cero ya que, en otro caso, implicará que la matriz es una matriz singular y que por consiguiente no es invertible [32].

Para el caso anterior existe una “pseudo–inversa” de Moore–Penrose [33] la cual permite la obtención de una matriz inversa aún cuando su determinante es cero. Dicha pseudo–inversa se denota J^+ y se define por (ecuación 5.27):

$$J^+ = J^T (J \cdot J^T)^{-1} \quad (5.27)$$

Además, se cumple que si la inversa de la matriz Jacobiana existe entonces su pseudo–inversa también existirá, y será igual a la matriz inversa:

$$J^+ = J^{-1} \iff \exists J^{-1}$$

Como los puntos críticos son $\theta_2 = 0$ y $\theta_2 = \pi$ se puede obtener un valor de la inversa que será igual a la pseudo–inversa, donde ambas dependen del parámetro θ_2 para existir. El valor que se obtiene de la inversa es el siguiente (ecuación 5.28)⁶:

⁶el cálculo simbólico de tanto la inversa como de la pseudo–inversa puede resultar algo complejo por lo que se ha dejado en el anexo A.1 un *Jupyter Notebook* para realizar las operaciones de forma interactiva y guiada. No obstante, el código fuente se encuentra disponible en el anexo A.2.

$$J^{-1} = J^+ = \begin{bmatrix} \frac{\sin(\theta_0)}{a_2 \cos(\theta_1) + a_3 \cos(\theta_1 - \theta_2) + a_1} & \frac{\cos(\theta_0)}{a_2 \cos(\theta_1) + a_3 \cos(\theta_1 - \theta_2) + a_1} & 0 \\ -\frac{\cos(\theta_0) \cos(\theta_1 - \theta_2)}{a_2 \sin(\theta_2)} & -\frac{\sin(\theta_0) \cos(\theta_1 - \theta_2)}{a_2 \sin(\theta_2)} & -\frac{\sin(\theta_1 - \theta_2)}{a_2 \sin(\theta_2)} \\ -\frac{(a_2 \cos(\theta_1) + a_3 \cos(\theta_1 - \theta_2)) \cos(\theta_0)}{a_2 a_3 \sin(\theta_2)} & -\frac{(a_2 \cos(\theta_1) + a_3 \cos(\theta_1 - \theta_2)) \sin(\theta_0)}{a_2 a_3 \sin(\theta_2)} & -\frac{a_2 \sin(\theta_1) + a_3 \sin(\theta_1 - \theta_2)}{a_2 a_3 \sin(\theta_2)} \end{bmatrix} \quad (5.28)$$

No ha sido necesario emplear la matriz pseudo-inversa en este proyecto, pero se muestra en el anexo D por si en un futuro interesa recurrir a ella. Esta matriz ha sido calculada empleando un *Jupyter Notebook* que se encuentra en el anexo A.1.

5.4. Implementación final realizada

Una vez concluido el estudio sobre el fundamento matemático del proyecto, se decidió qué usar de lo visto anteriormente.

Por una parte, se vio cómo el uso de la fuerza bruta generando un mapa de ángulos y puntos resultaba inviable teniendo en cuenta el espacio disponible en el microcontrolador así como el tiempo necesario para calcularlo.

Con respecto a las funciones Jacobianas, si bien su estudio permite crear muchas relaciones entre velocidades y fuerzas, dado que la masa del robot es pequeña y la velocidad es constante, el uso de dichas funciones para el control del mismo no añade mucha más información de la que ya se dispone. Además, en favor de lo anterior, en el código fuente original del μ Arm tampoco contempla las funciones Jacobianas a la hora de manejar ni los puntos ni la velocidad [29], por lo que se puede asumir que su uso no es necesario.

Por esto mismo, el control del brazo se realizará utilizando la cinemática directa para obtener el punto $\{x, y, z\}$ del *end-effector* cuando se aplican unos ángulos de entrada $\{\theta_0, \theta_1, \theta_2\}$; y la cinemática inversa para obtener la relación entre un punto de entrada $\{x, y, z\}$ y los ángulos que lo generan $\{\theta_0, \theta_1, \theta_2\}$.

Teniendo en cuenta las características mencionadas sobre el microcontrolador a utilizar, el tiempo de ambas operaciones es bastante pequeño (unos 15 μ s la cinemática directa y 100 μ s para la cinemática inversa, según una estimación con el simulador.), por lo que su uso no añade un desfase suficientemente grande como para considerarse notorio.

Capítulo 6

Hardware

6.1. Diseño 3D

Aprovechando la licencia original GPL 3.0 del μ Arm, se ha recuperado el modelo 3D proporcionado por UFACTORY como punto de partida. Se han tenido que realizar diversas modificaciones de distintas piezas para adaptarlas a los materiales que se van a usar, los motores que se emplearán y el tamaño de la nueva placa, entre otros.

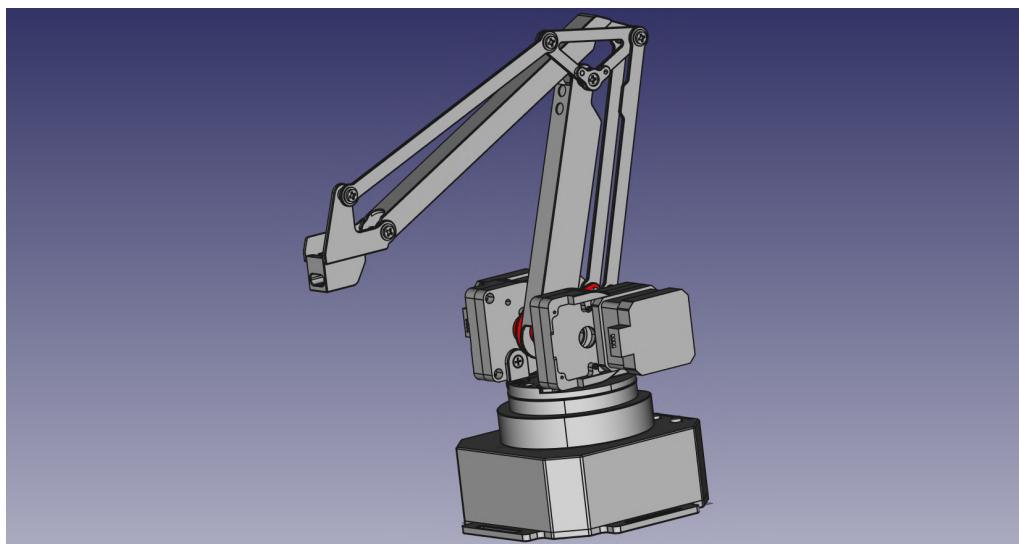


Figura 6.1: Concepto inicial del brazo robótico.

Las herramientas que han sido empleadas para visualizar y modificar el modelo y posteriormente imprimir las piezas han sido respectivamente FreeCAD y Ultimake Cura.



Figura 6.2: Logotipos de las herramientas utilizadas.

El flujo de trabajo que se ha seguido desde el modelo 3D hasta la impresión de una pieza ha sido el mostrado en la figura 6.3:



Figura 6.3: Flujo de trabajo del desarrollo y la impresión 3D.

Antes de proceder a explicar cada una de las nuevas piezas que se han diseñado, se tomará una de ellas como ejemplo para explicar el proceso de diseño detallado.

Inicialmente se parte de un bloque que represente la forma general de la pieza, para posteriormente añadir detalles. En el caso de la pieza que se usa como ejemplo, tenemos un cuadrado de 120 mm de ancho por 120 mm de alto y 3 mm de grosor.

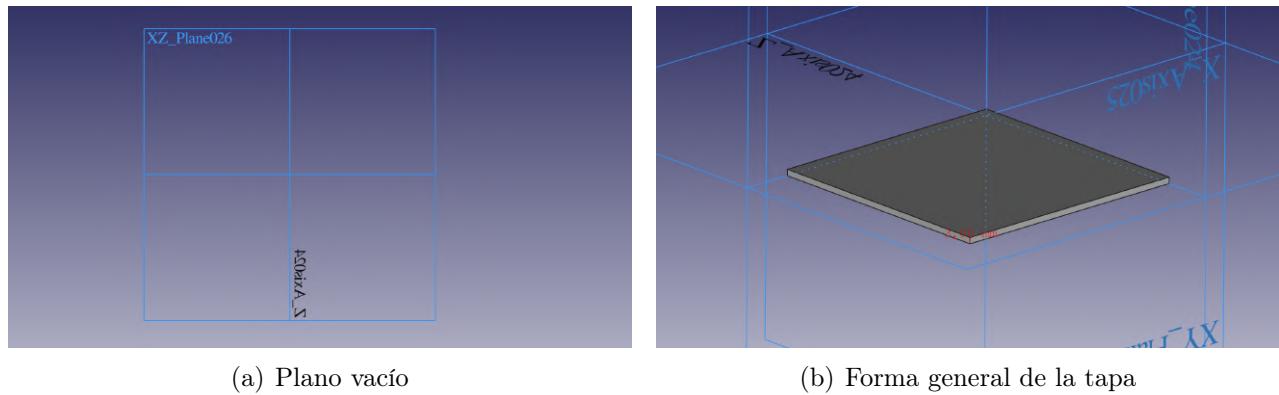


Figura 6.4: Construcción de la forma general.

Tras crear la forma general, se añaden los agujeros en las esquinas para poder atornillar la tapa. Los agujeros tienen un radio de 1,9 mm y se distancian de los laterales 3,9 mm para hacerlos coincidir con los agujeros de las paredes.

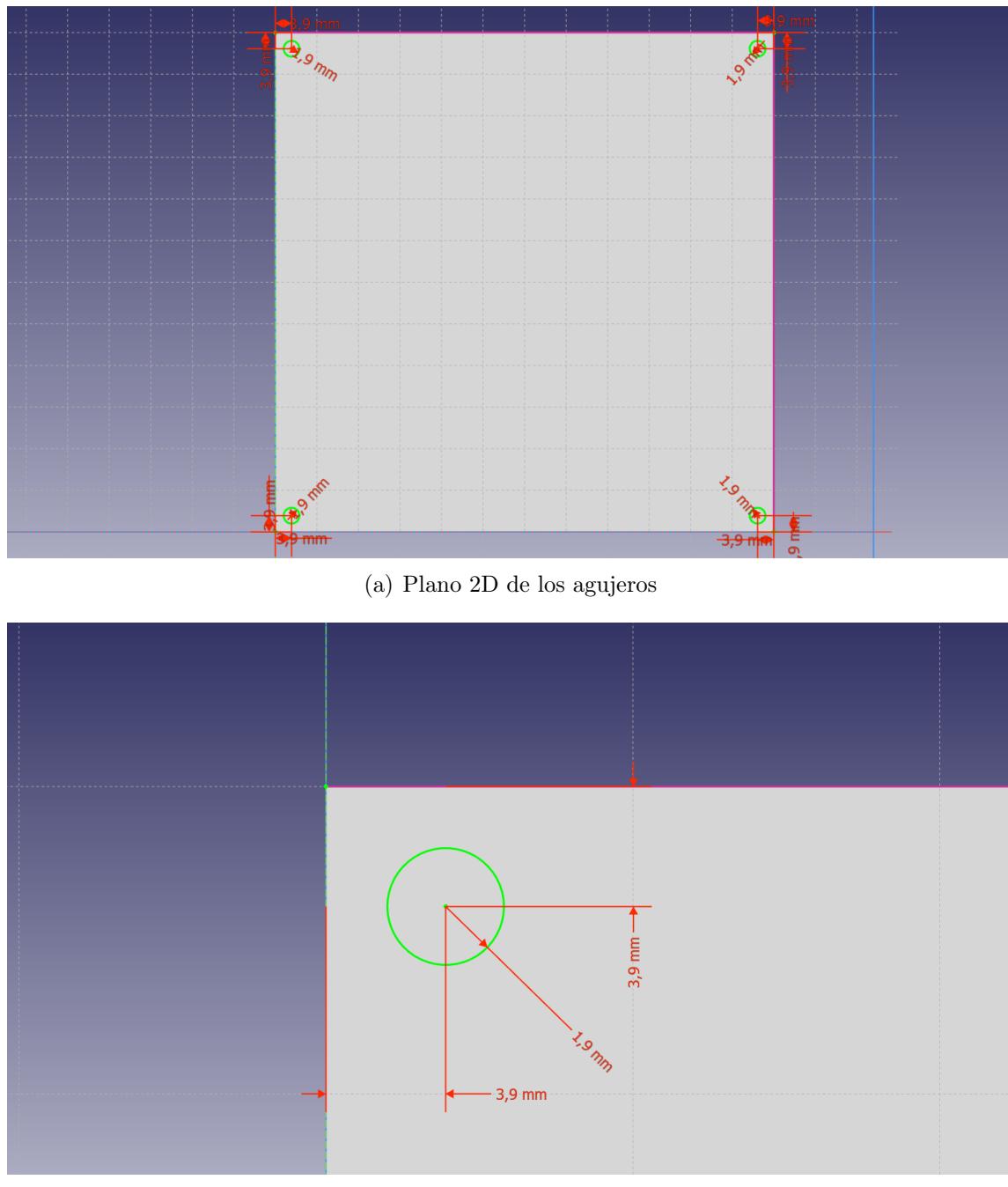
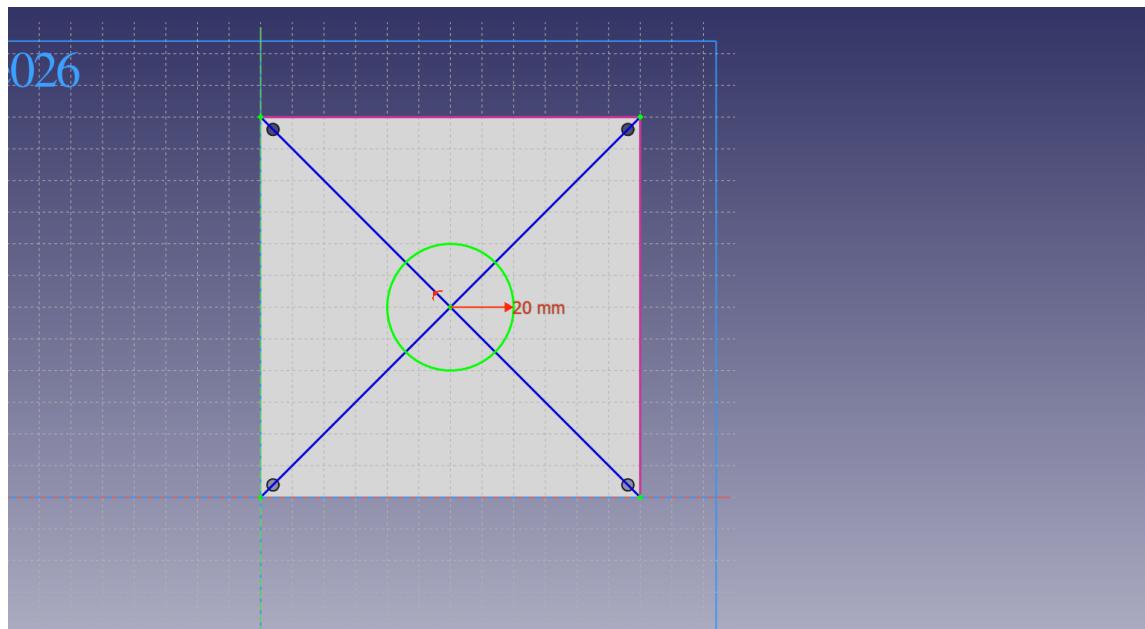
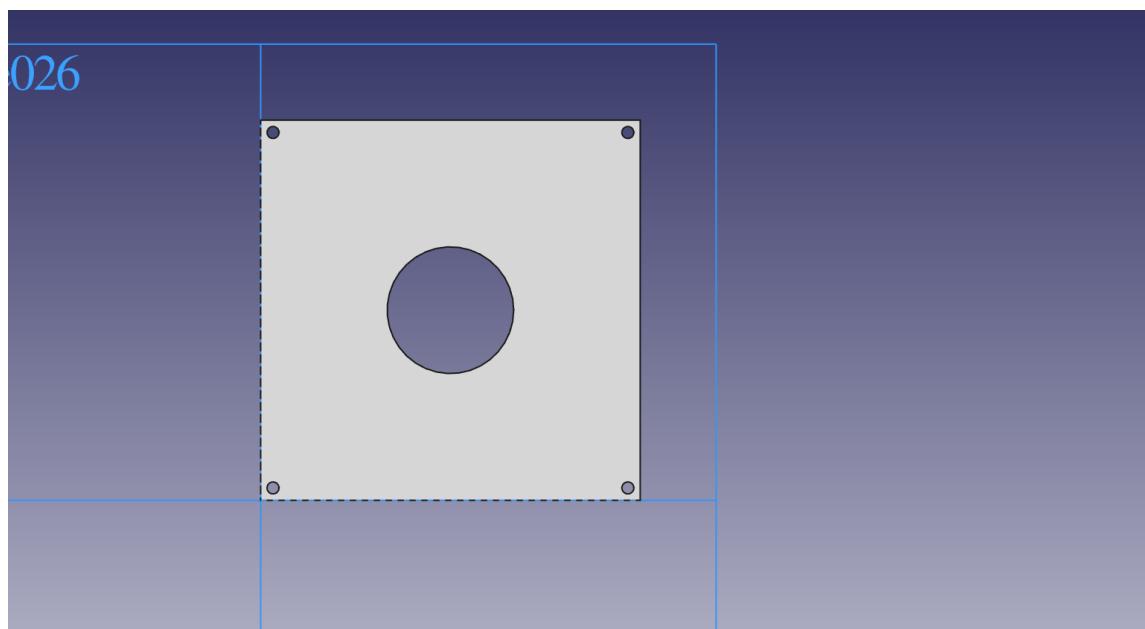


Figura 6.5: Agujeros para tornillos.

Tras realizar los agujeros de los exteriores de la pieza, se hace un agujero central y se extruye una torre centrada sobre dicho agujero.

(a) *Sketch* del agujero central

(b) Agujero central sin torre

Figura 6.6: Agujeros centrales.

Tras definir el agujero, se extruye la torre:

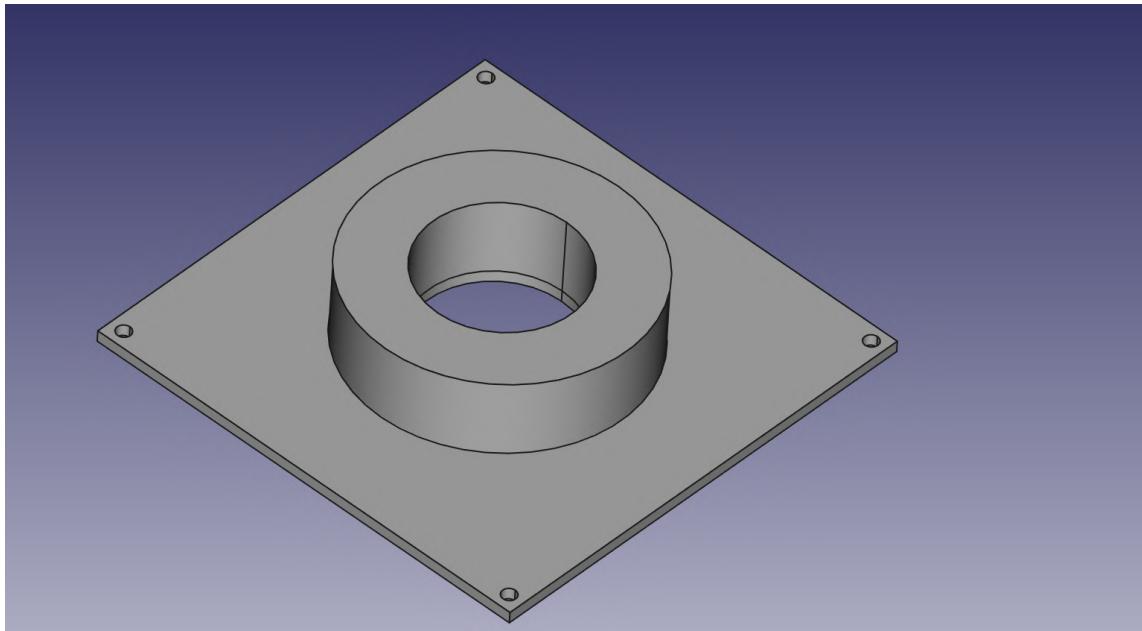
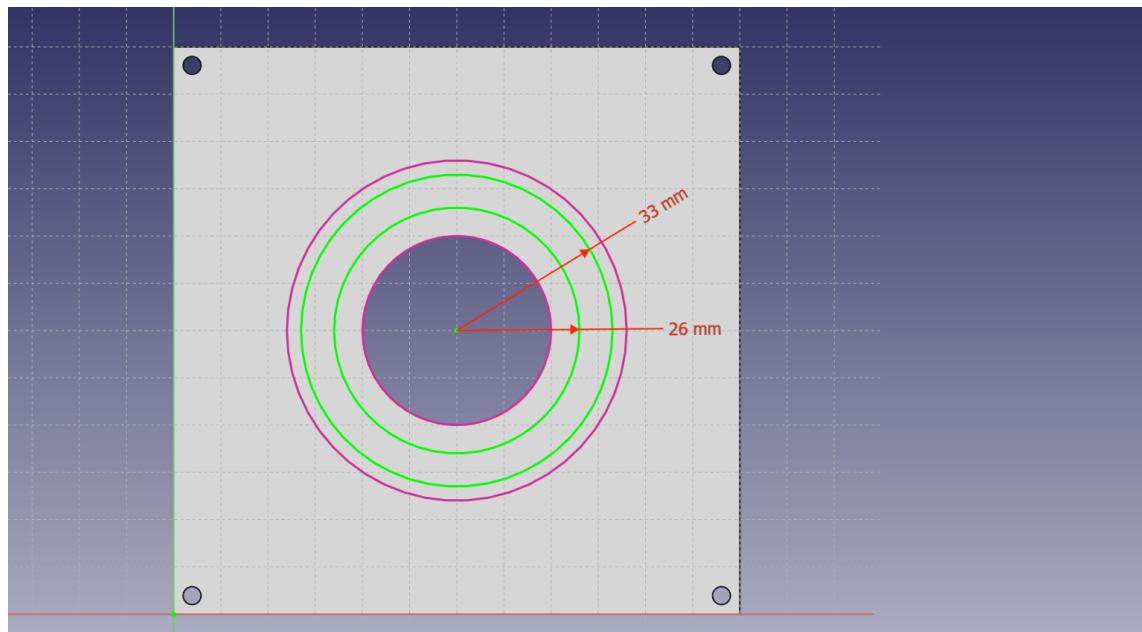
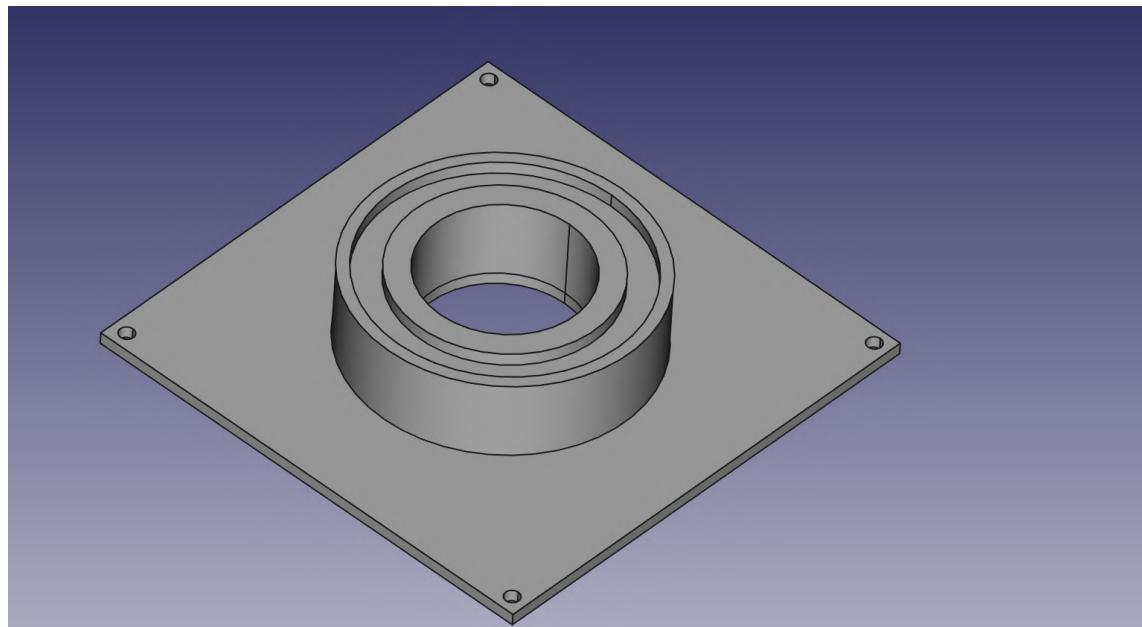


Figura 6.7: Tapa con torre central.

A continuación, se procede a eliminar material de la torre con el objetivo de disminuir la superficie de contacto con el disco rotativo y, por tanto, eliminar parte del rozamiento.

(a) *Sketch* del desgaste.

(b) Torre central tras el rebaje.

Figura 6.8: Desgastes y rebaje.

Para disminuir aún más el rozamiento debido a posibles bordes imperfectos que queden en el disco, se realiza un chaflán en el diámetro interior y exterior.

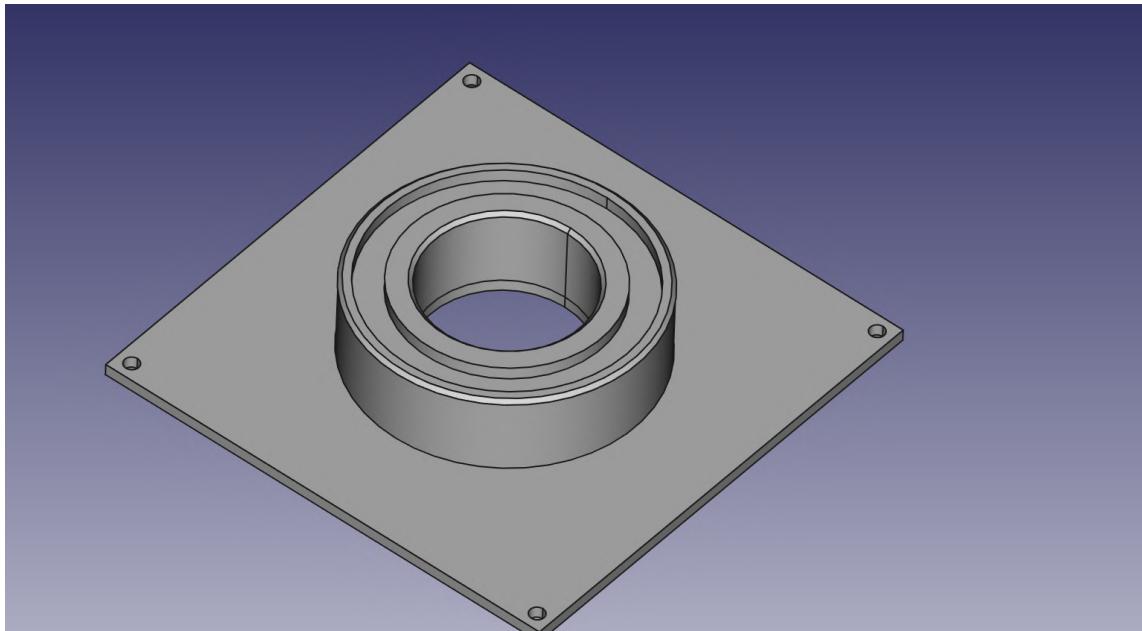


Figura 6.9: Torre con chaflanes.

Finalmente, se añade una ranura para que sea posible llevar los cables de los motores exteriores a la placa de control que se haya en la caja.

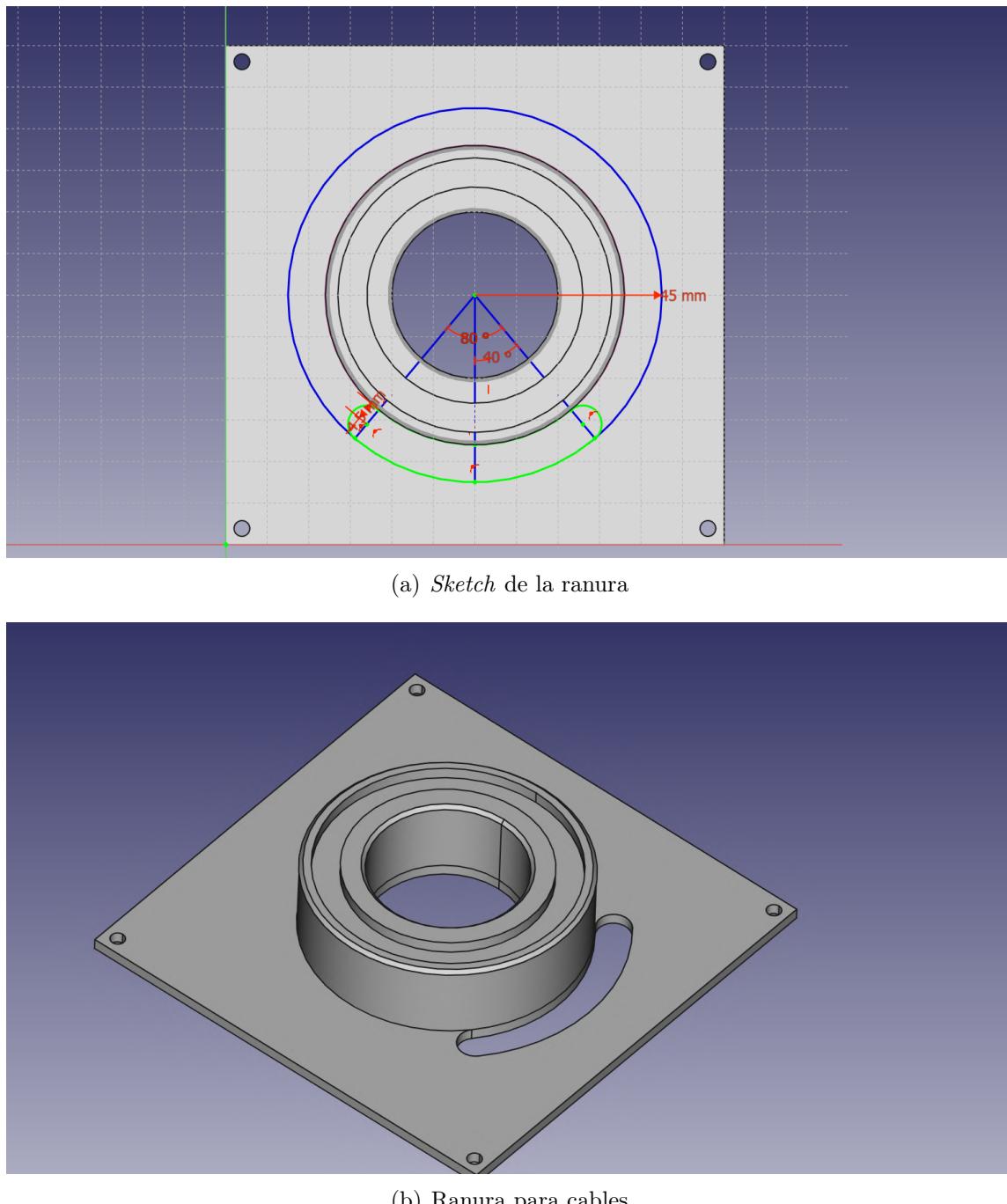


Figura 6.10: Tapa con ranura para cables.

Con esto concluimos la explicación del proceso de creación de piezas y pasamos a explicar cada una de las piezas por separado, detallando los motivos por los cuales es necesario remodelar algunas de ellas y los inconvenientes y contratiempos que han surgido durante el modelado y la impresión de estas.

En primer lugar, se explicará la caja que alberga la placa de control y uno de los motores.

La placa de control del brazo robótico no es la misma que en el caso del μ Arm de UFACTORY.

Además, los motores que se han empleado en este proyecto son servomotores con carcasa y sistema de sujeción distintos a los motores paso a paso del μ Arm. Debido a estos dos factores se ha tenido que diseñar nuevas partes para la base del brazo robótico.

Más en concreto, la necesidad de rediseñar esta parte es debida a que la base original era demasiado pequeña en superficie para permitir introducir en ella la placa de control. Además, el servomotor no podría haber cabido junto con la placa ya que la altura era insuficiente. Por otro lado, los sistemas de sujeción presentes en la caja existente no podían ser empleados para la placa de control desarrollada en este proyecto.

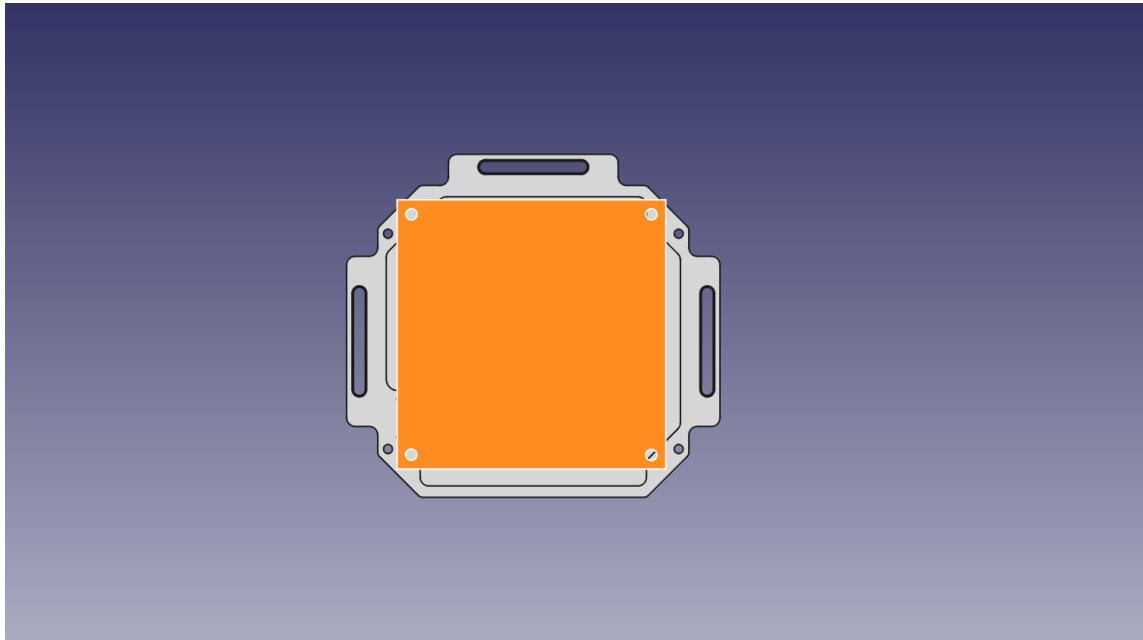


Figura 6.11: Proyección de la placa de control (naranja) sobre la base original del μ Arm (gris).

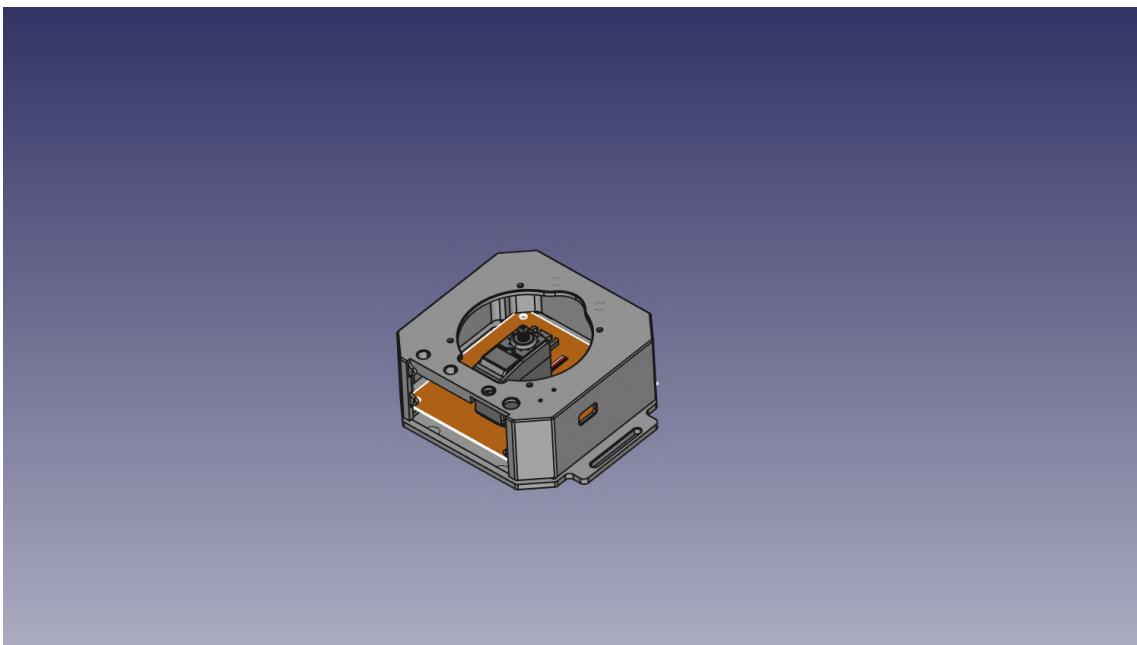


Figura 6.12: Placa y motor dentro de la caja original del μ Arm.

Como se observa en la figura 6.12 el motor sobresale por encima de las paredes y no hay ninguna manera de sujetarlo a estas o a la base.

Para solucionar los anteriores problemas se diseña una nueva base en la que se pueda encajar la placa, además de unas paredes lo suficientemente altas para poder introducir el motor junto con esta.

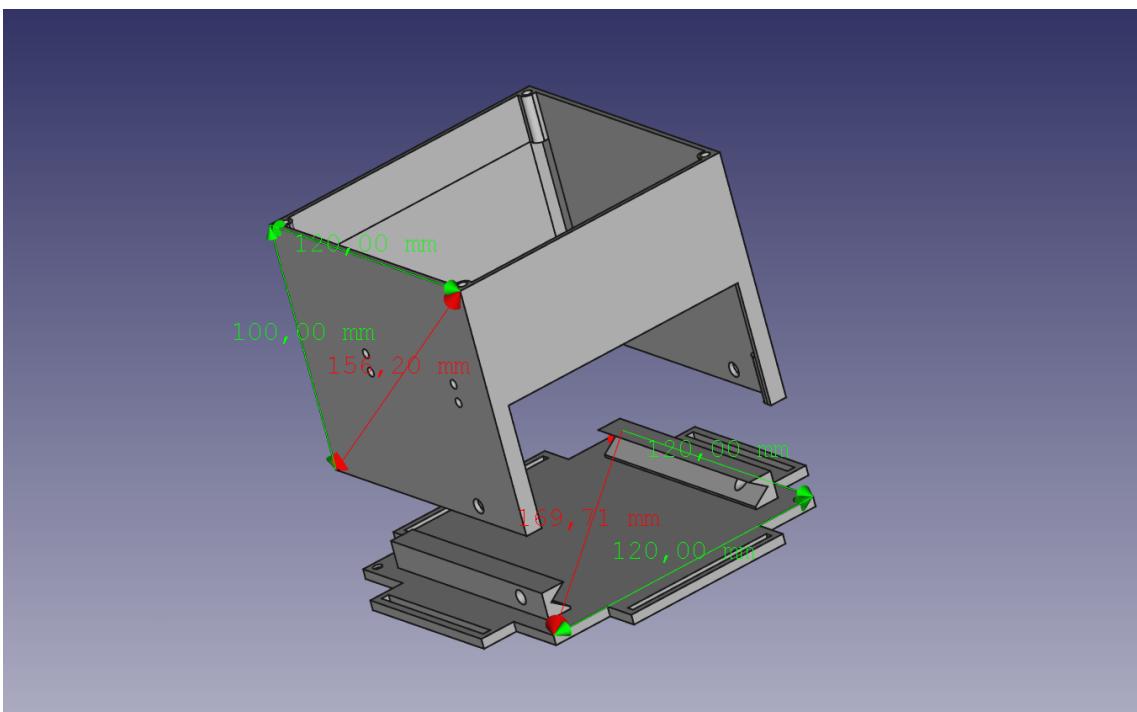


Figura 6.13: Base y paredes tras realizar las modificaciones necesarias

Inicialmente las paredes de la caja se habían diseñado con un grosor de 2 mm. Tras una serie de pruebas se concluyó que dicho grosor no era suficiente para proporcionar resistencia y estabilidad suficiente, por tanto, se optó por uno de 3 mm en la versión final.

En la base se observan los raíles que servirán para introducir y retirar la placa de la estructura.

Se ha optado por un sistema de raíles en vez de una sujeción fija ya que, de esta manera, se consigue una mayor versatilidad a la hora de extraer e introducir la placa en labores de depuración y construcción.

Además, debido al escaso espacio dentro de la caja, se hace prácticamente imposible la introducción de las herramientas necesarias para atornillar la placa a la estructura. Gracias al sistema de raíles se evita desmontar ciertas piezas al intentar introducir o extraer la placa.

Después de que la placa sea insertada en estos carriles, se asegura su posición mediante los agujeros laterales que pueden observarse en la figura 6.13.

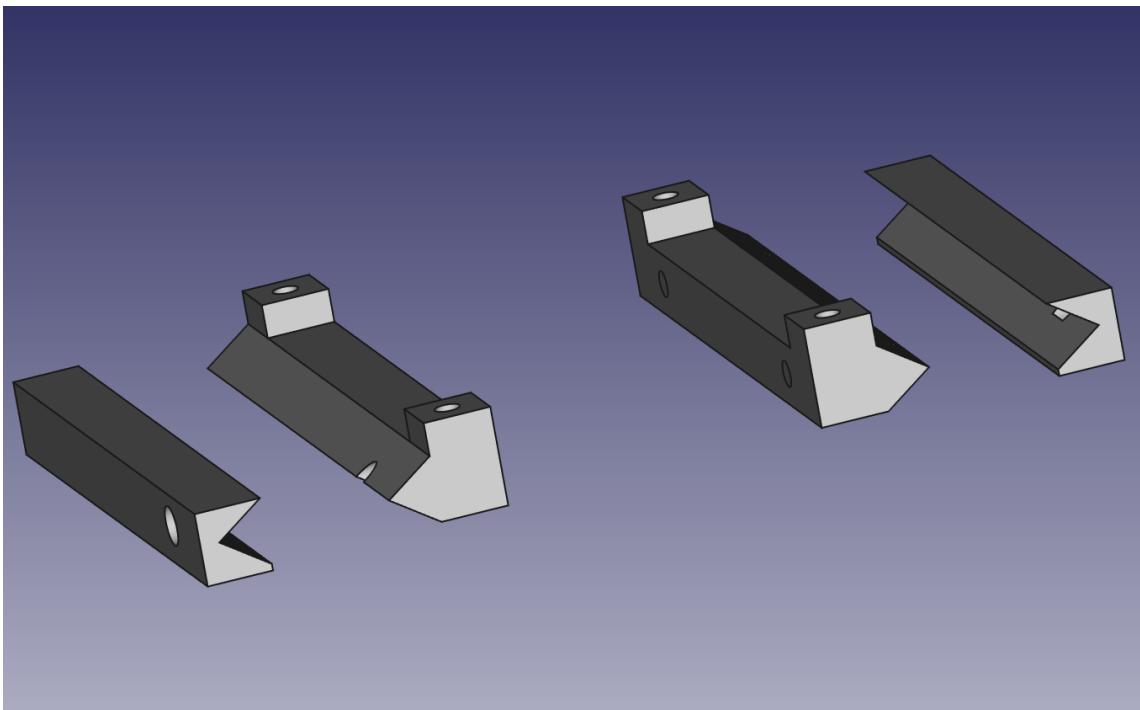


Figura 6.14: Sistema de raíles de la placa.

En la figura 6.14 se observan las piezas que se añadirán a la placa de control para poder deslizarla sobre los carriles. En el exterior de la imagen aparecen los carriles presentes en la base de la caja, donde se puede ver que, al estar la placa completamente introducida en los carriles, los agujeros del carril y del raíl se posicionan de tal manera que se puede introducir un pasador que asegura la posición de la placa.

Debido a que tanto los raíles como los carriles se fabricaron con unas medidas teóricas, extraídas del diseño físico inicial de la placa, se comprobó que el ensamblaje físico no era posible.

Para solucionar este problema se desplazaron los agujeros por los que se unía el raíl a la placa. Esto se observa en la figura 6.15:

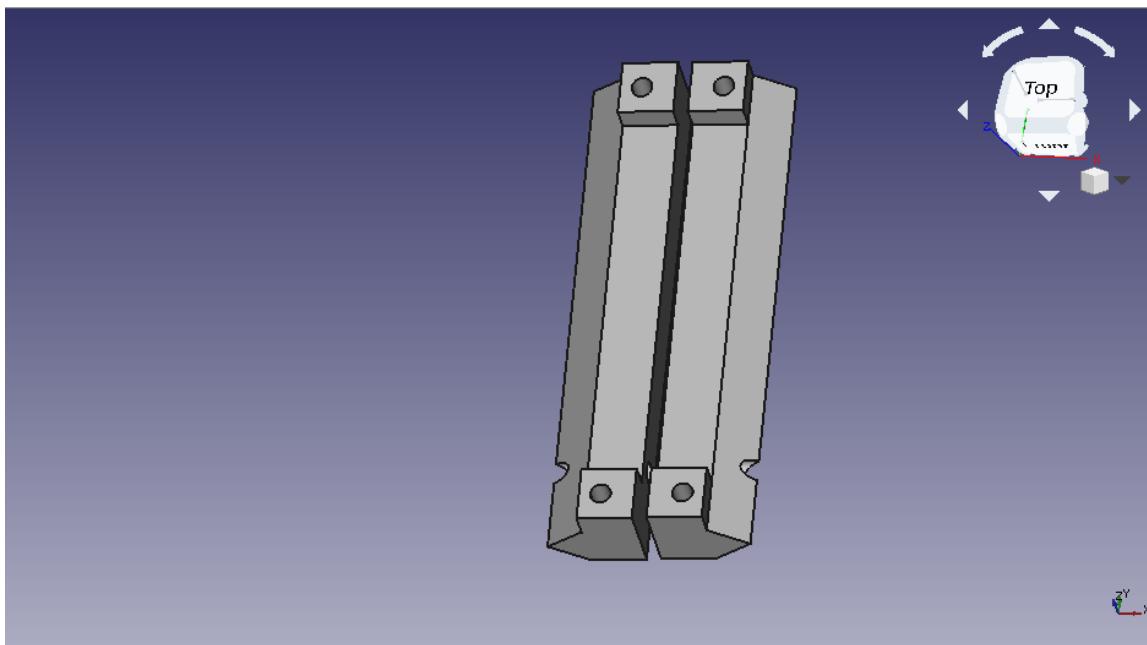


Figura 6.15: Versión final de los carriles.

Por otro lado, para poder sujetar el motor que moverá el brazo alrededor del eje vertical, se ha tenido que diseñar la siguiente pieza:

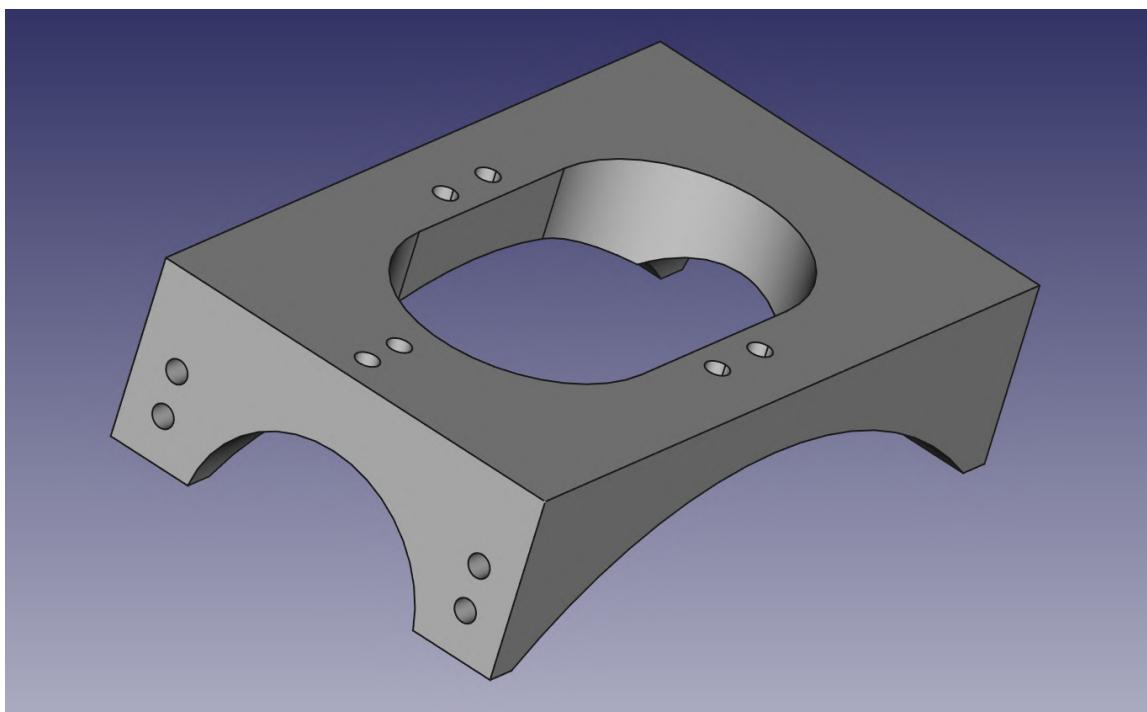


Figura 6.16: Pieza de sujeción del motor de la base.

Esta pieza se atornilla a las paredes de la caja y empleando las solapas del motor, este se atornilla en el centro de la pieza como se puede ver en la figura 6.17.

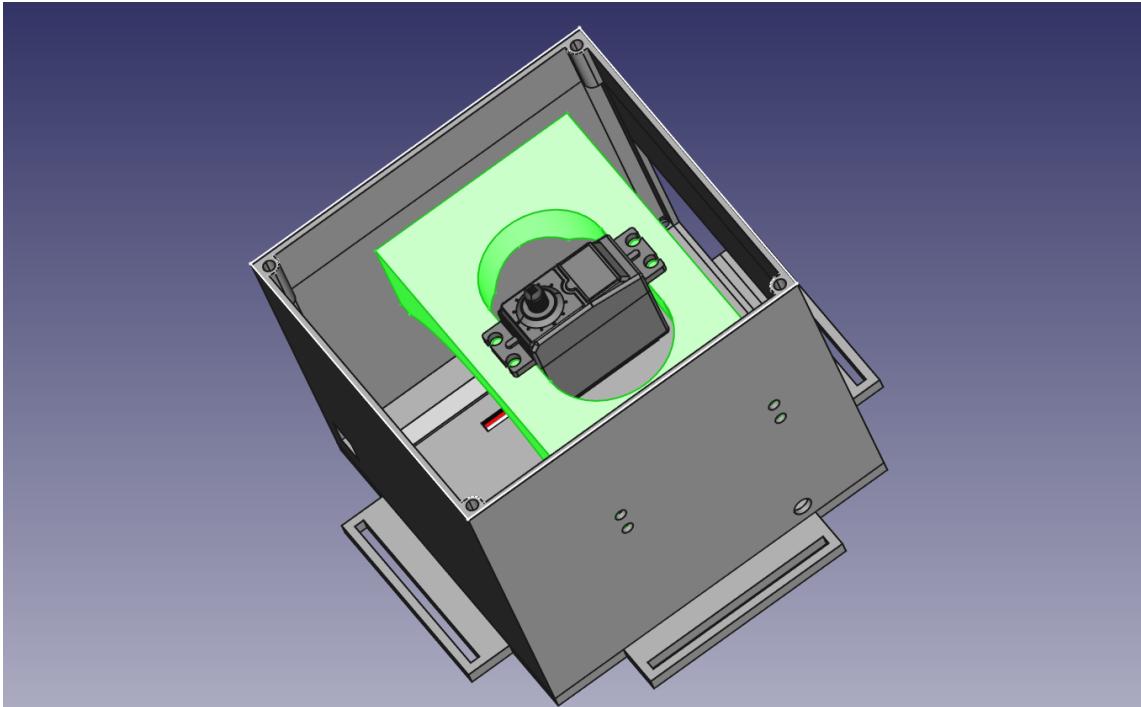


Figura 6.17: Pieza de sujeción del motor (verde) dentro de la caja.

Dado que el tamaño de la base y de las paredes ha cambiado, la tapa superior debe también ser modificada para adaptar su tamaño y su sistema de sujeción a los nuevos diseños

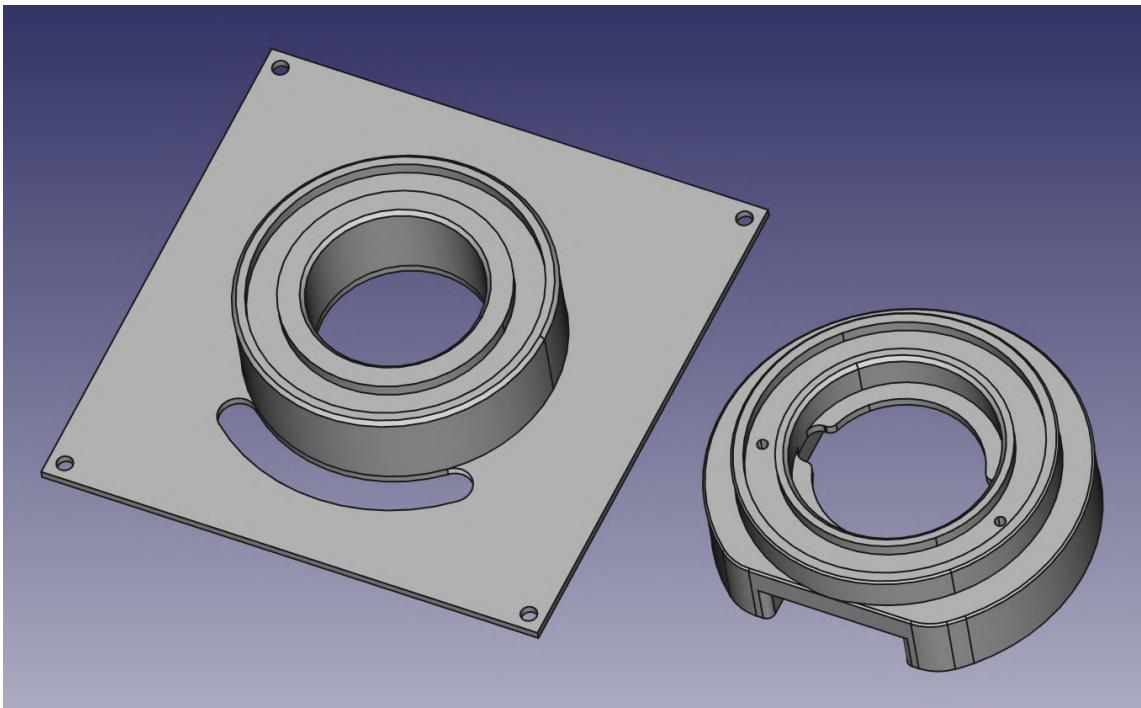


Figura 6.18: Tapa original (derecha) junto a la tapa modificada (izquierda).

En la figura 6.18 se observa que en la parte superior las diferencias entre ambas piezas son

mínimas respetándose los diámetros del disco exterior e interior sobre los que descansará la base giratoria.

En la figura 6.19 se puede observar cómo la base giratoria descansa sobre los anillos superiores y se inserta dentro de la tapa.

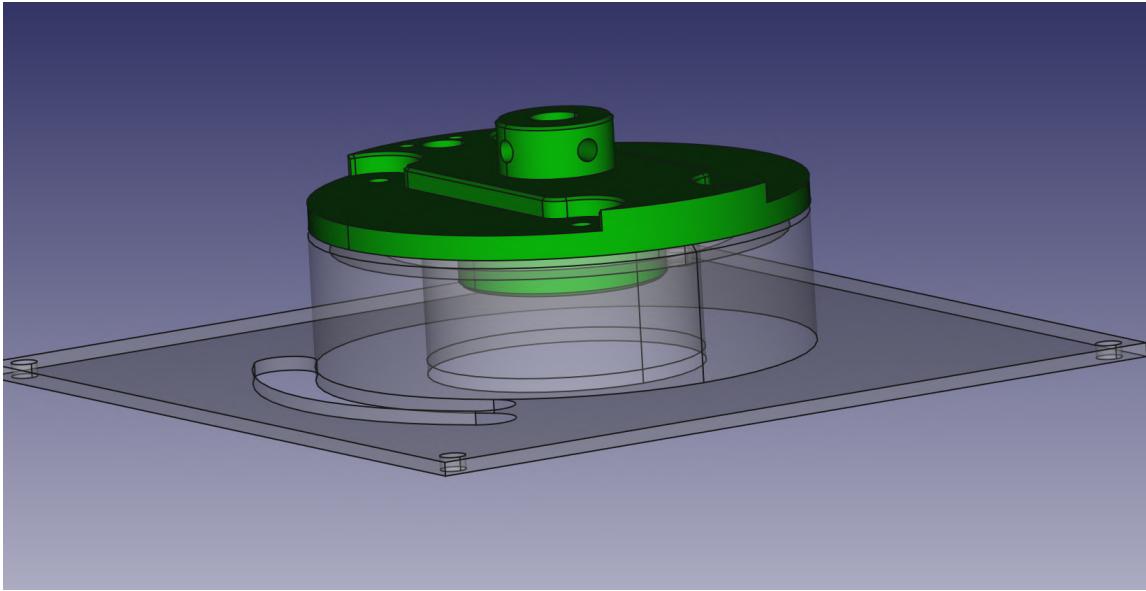


Figura 6.19: Tapa superior transparente y base giratoria (verde).

De esta manera se consigue que el peso de la parte móvil del brazo descansen sobre la tapa y no sobre el eje de rotación. Por otro lado, el hecho de que la base giratoria protruya hacia el interior de la tapa permite que la pieza esté más cerca del motor permitiendo acortar el eje.

A en la figura 6.20 se muestra la pieza que conecta el motor a la base giratoria a través de un eje. Esta pieza tiene en su base un desgaste tal que permite que el engranaje del motor pueda ser introducido dentro, mientras que en la parte de arriba tiene unos agujeros que permiten la sujeción de un eje metálico el cual llega hasta la base giratoria y transmite el movimiento del motor a esta.

La pieza plana que protruye hacia un lateral sirve para que el cilindro pueda alcanzar un fin de carrera que indique cuándo el motor ha llegado a una posición extrema.

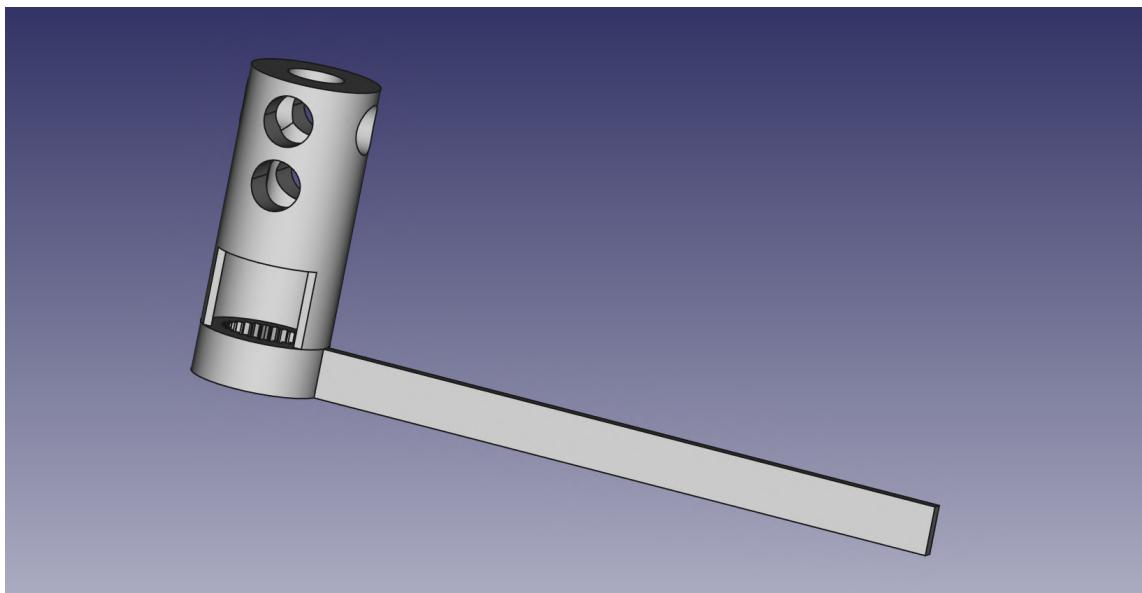


Figura 6.20: Pieza de unión del motor y el eje.

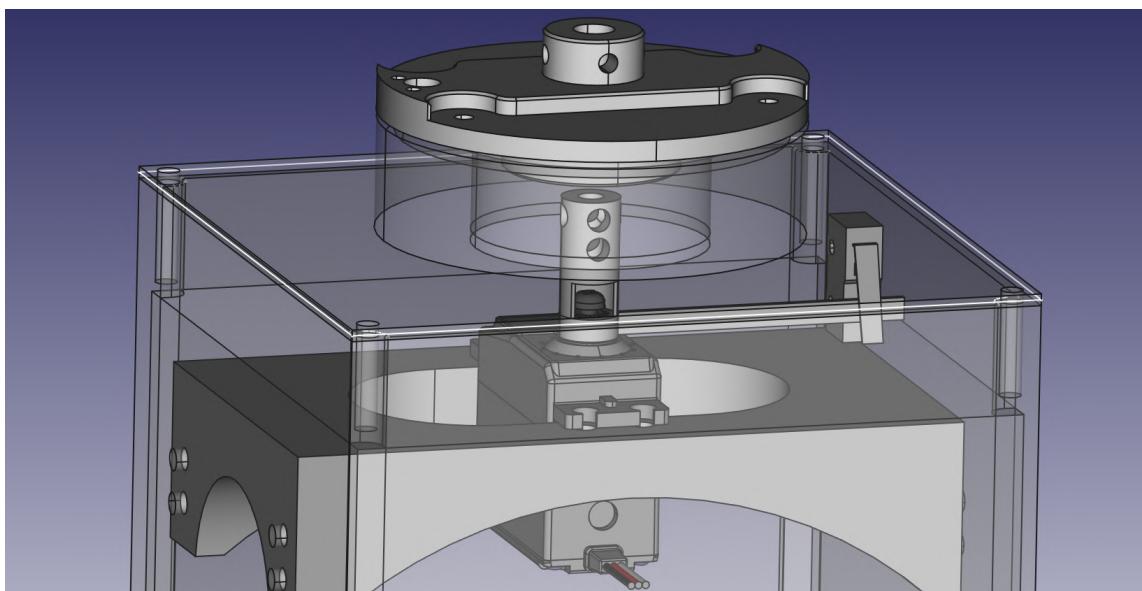


Figura 6.21: Sistema completo.

En la figura 6.21 se observa la construcción completa de la base. En esta imagen se dejan las paredes y la tapa superior con cierta transparencia para poder ver el interior de la caja. Entre el cilindro montado encima del motor y la base giratoria habrá un eje metálico que comunicará el movimiento.

Por otro lado, para que sea posible ubicar los motores que se encargaran del movimiento vertical del brazo se han de crear nuevas piezas en las que puedan ser atornillados. Esta pieza puede observarse en la figura 6.22:

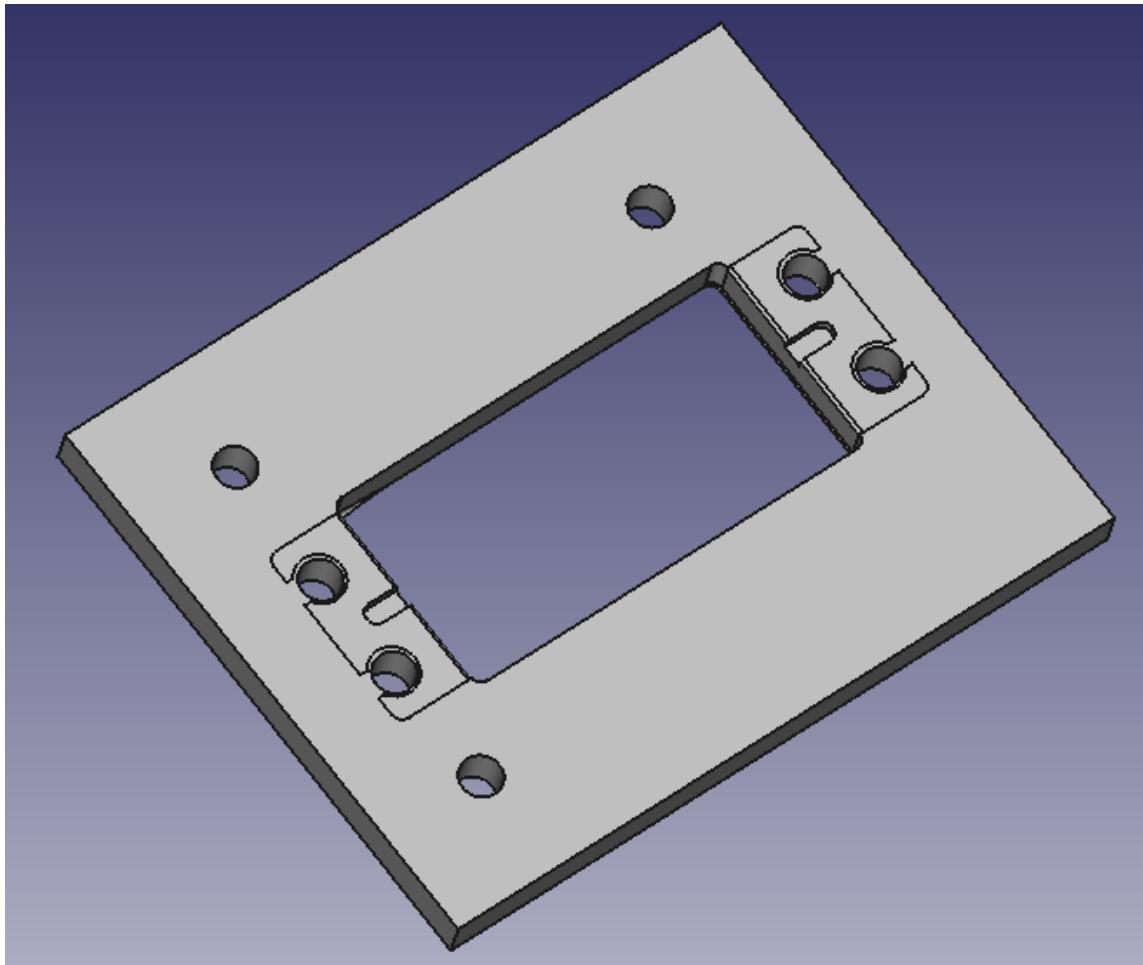


Figura 6.22: Placa de sujeción de los motores laterales.

En fases más avanzadas del proyecto, el equipo de desarrollo observó, tras realizar pruebas, que el eje metálico no podía ser asegurado en el centro del cilindro (más detalles en el apartado 9.1.3). También concluyó que en determinadas configuraciones geométricas del brazo, este podía ejercer demasiada fuerza sobre el eje de giro horizontal.

Por ello se tuvieron que realizar los siguientes cambios:

- En el disco rotativo se añadió un carril que encajará en la ranura de la tapa superior. El cometido de este carril es evitar movimientos en el plano horizontal, dar mejor soporte ante el peso y ayudar a guiar el movimiento.

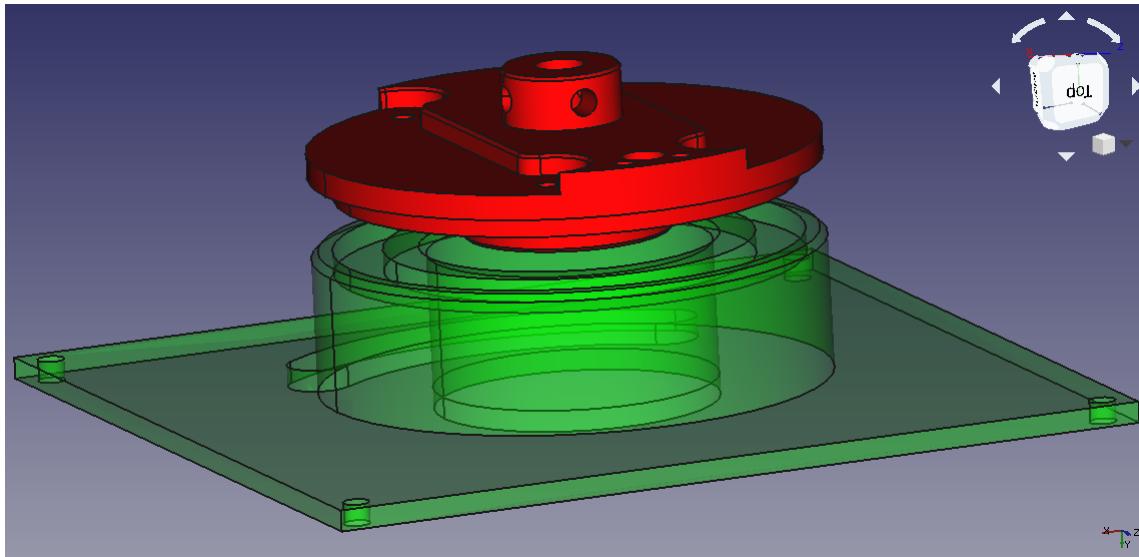


Figura 6.23: Base de giro con raíl.

- Para evitar emplear un eje metálico, el equipo de desarrollo diseña una pieza que pueda ser acoplada a la base giratoria (figura 6.24):

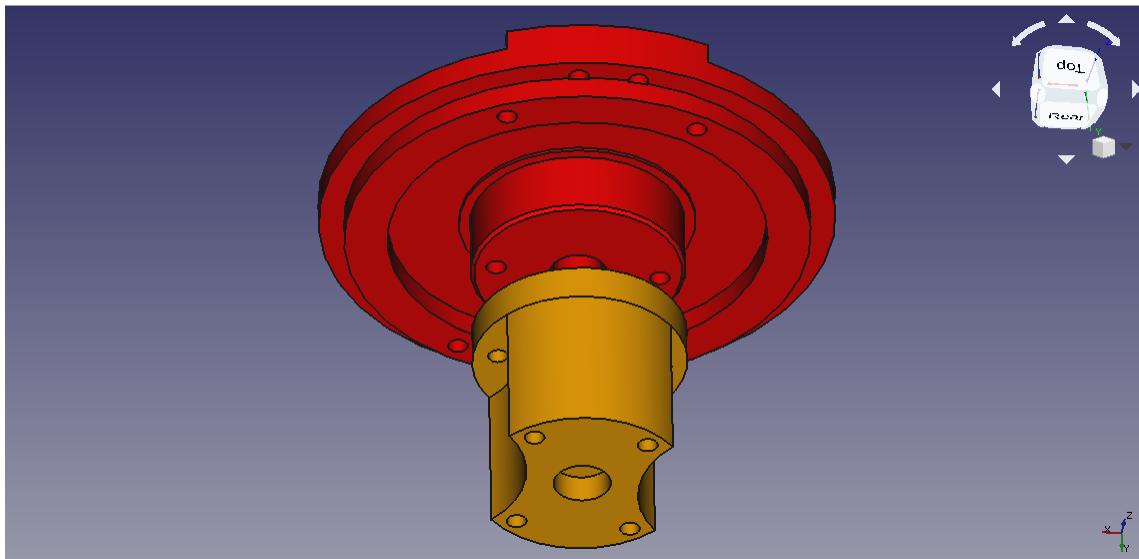


Figura 6.24: La pieza sustituta del eje metálico (naranja).

Según se observa en la figura 6.24 la base giratoria y la parte que servirá para sustituir al eje metálico, serán unidas mediante dos tornillos. También se puede observar el carril de la base rotatoria en una perspectiva ascendente.

- En la imagen 6.25 se puede observar la ultima pieza del sistema. Su cometido es servir como engranaje sobre el eje del motor y transmitir el movimiento a las piezas superiores. Será unida a la pieza superior mediante 4 tornillos.

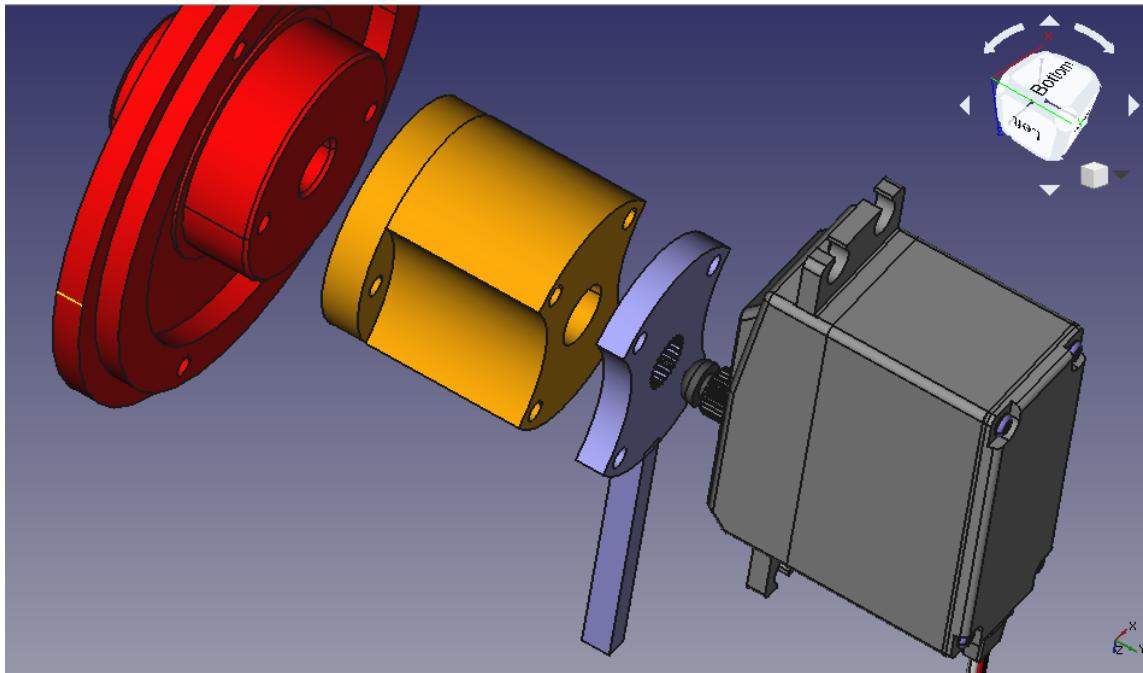


Figura 6.25: Sistema completo del eje giratorio.

Otra pieza que tuvo que ser modificada es la sujeción del *end-effector*:

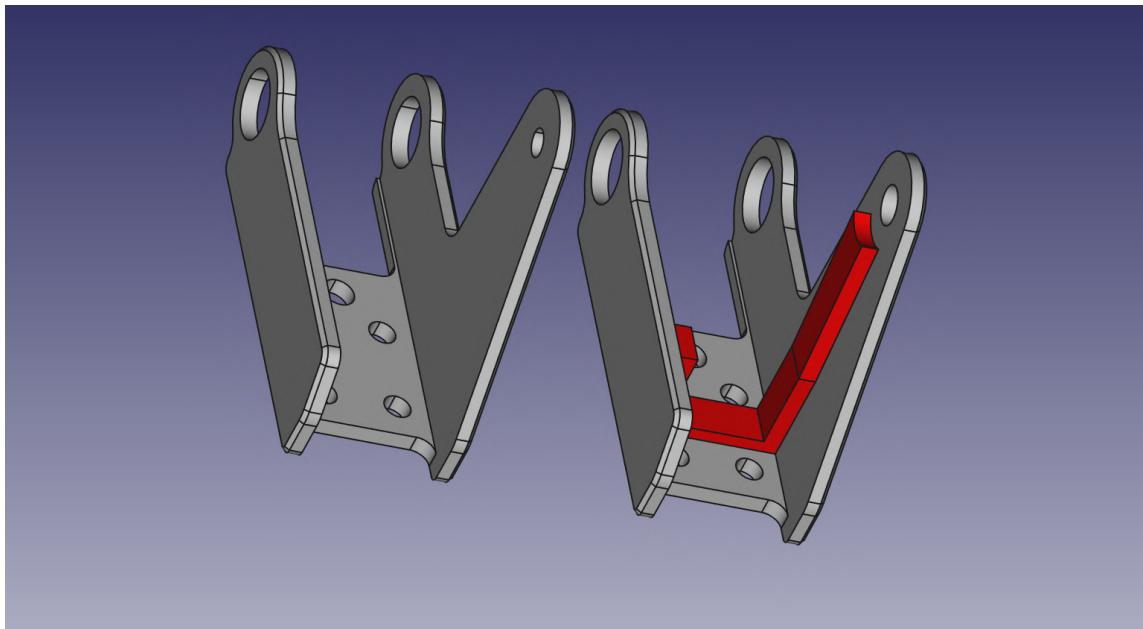


Figura 6.26: *End-effector* inicial (izquierda). *End-effector* modificado (derecha).

Como se puede observar en la figura 6.26, la modificación consiste en la adición de un refuerzo en las partes menos resistentes. Esta pieza fue modificada tras observar la fragilidad de las paredes laterales cuando estas se rompieron en la pieza sin refuerzo tras ser impresas.

Cabe destacar que, pese a que una gran parte de las piezas no han sido mencionadas de

manera concreta en este apartado, todas ellas han sido revisadas de manera individual y han sido editadas para posibilitar el uso de los tornillos, rodamientos y ejes disponibles.

Más concretamente, los tornillos de $4 \times 16\text{ mm}$ en el caso de las piezas que han sido construidas desde cero por parte del equipo de desarrollo, y tornillos de $3 \times 10\text{ mm}$ y $3 \times 20\text{ mm}$ según el caso, en piezas editadas.

En último lugar se han adaptado las ranuras para los rodamientos de 8 mm de diámetro externo que han sido adquiridos para el proyecto.

Tras añadir dichas piezas al diseño general y completar este con las piezas originales que no han sido modificadas, se obtiene el diseño definitivo del brazo robótico el cual se puede observar en la figura 6.27:

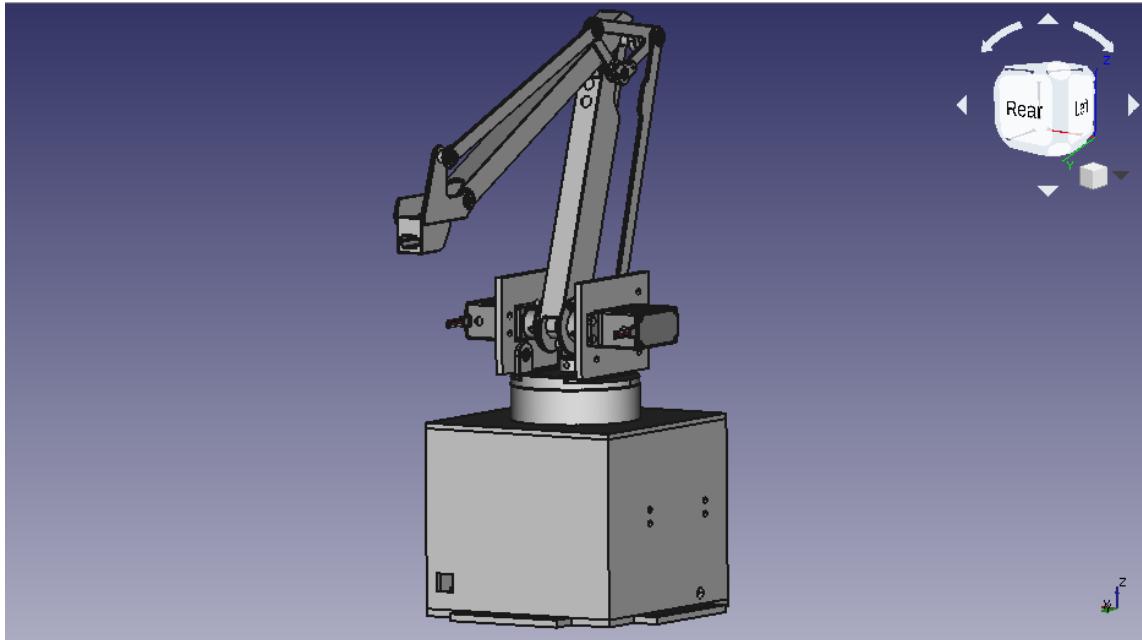


Figura 6.27: Brazo robótico completo tras las modificaciones.

6.2. Construcción del brazo

A continuación se procede a explicar la construcción del brazo robótico. Para ello, se emplearán los diseños 3D y ciertas imágenes del brazo real en sus etapas de construcción.

La razón de no explicar el proceso íntegro con imágenes reales del brazo es debido a que no hay suficientes documentos gráficos para exemplificar el proceso completo y ciertas etapas de la construcción quedarían sin poder ser documentadas.

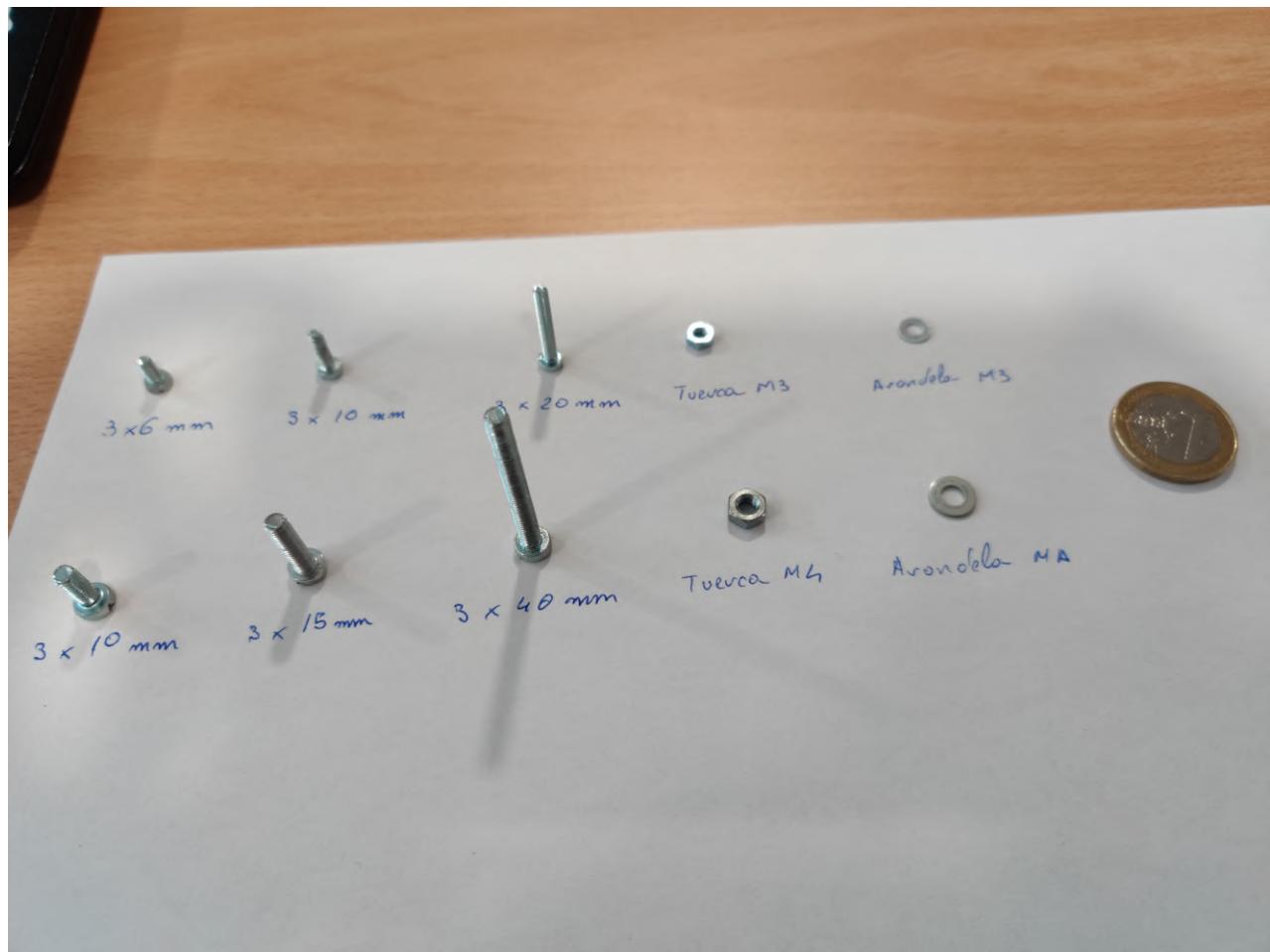


Figura 6.28: Elementos mecánicos externos.

En la figura 6.28 se pueden observar los distintos tornillos, tuercas y arandelas que se emplearán a lo largo de la construcción del brazo. Esta figura puede servir como referencia para tener una imagen real de los diferentes elementos externos que se nombrarán a lo largo de la explicación.

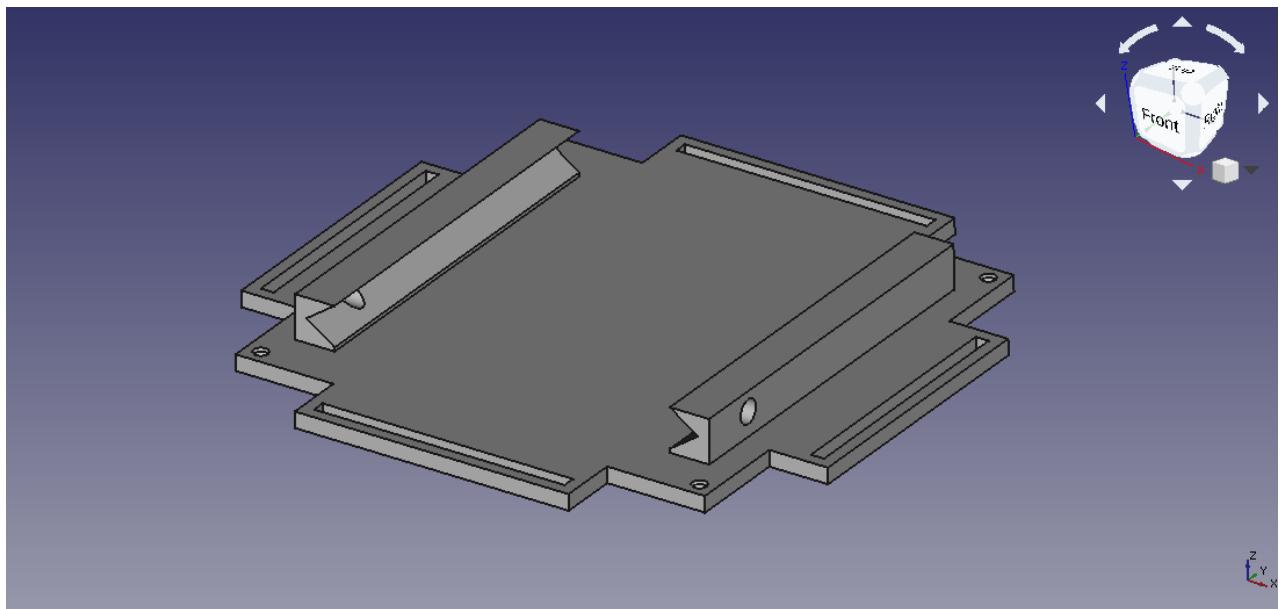


Figura 6.29: Base de la caja del brazo robótico.

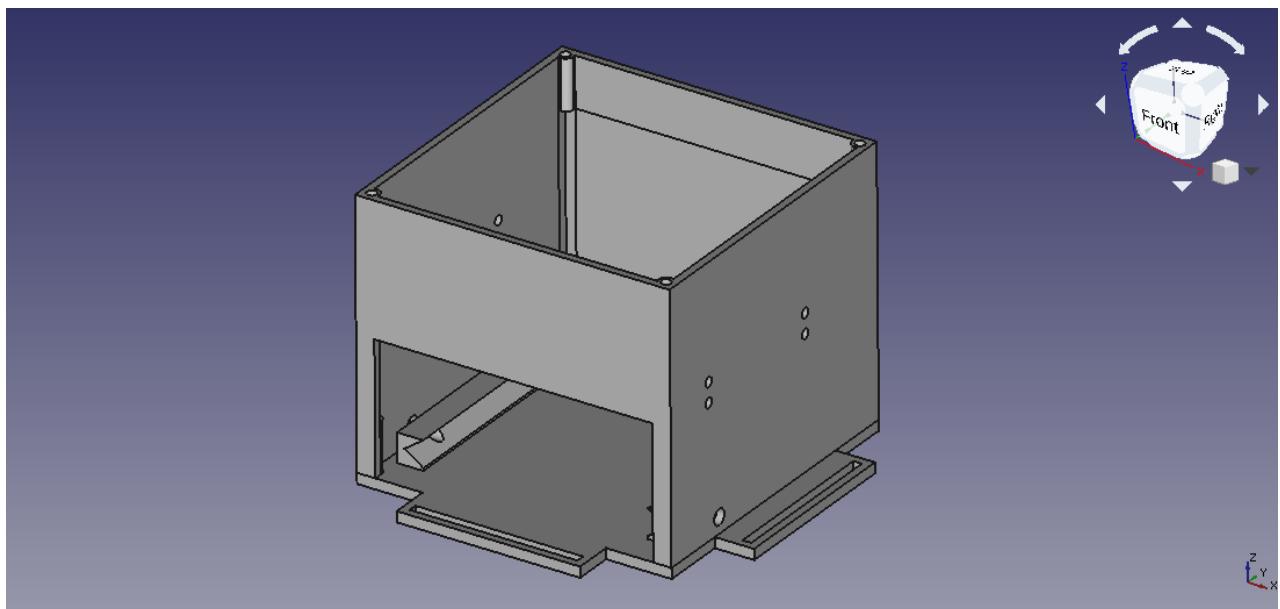


Figura 6.30: Base y paredes de la caja del brazo robótico.

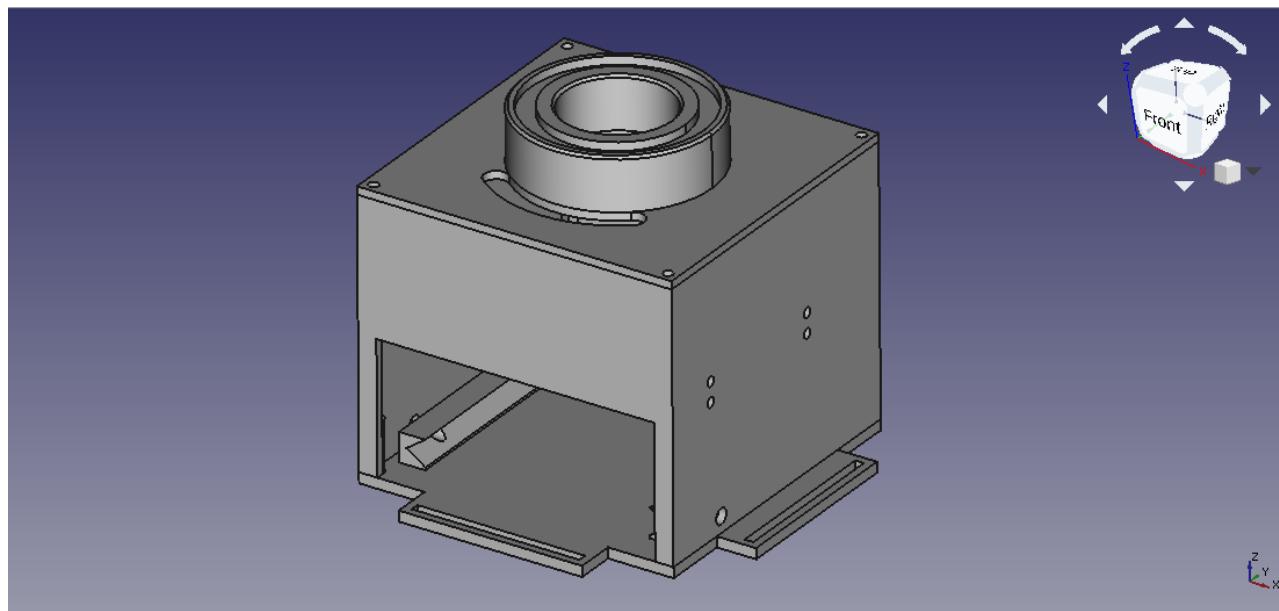


Figura 6.31: Caja completa del brazo robótico.

Como observamos en las figuras 6.29, 6.30 y 6.31, la caja está compuesta por 3 piezas y estas se ensamblan verticalmente una encima de otra mediante tornillos de $4 \times 15\text{ mm}$. Cabe destacar que esta parte de la estructura es inmóvil y sirve como soporte para la parte móvil.

A continuación se muestra el proceso de ensamblaje de los componentes que se encuentran en el interior de la caja.

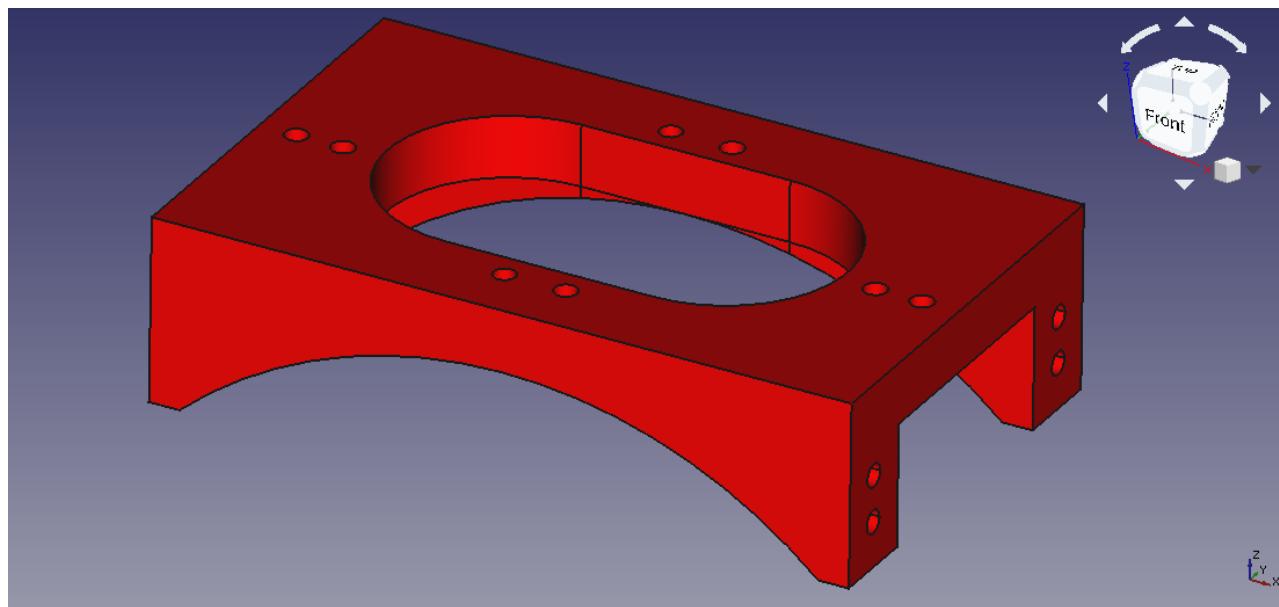


Figura 6.32: Sujeción del motor de la base.

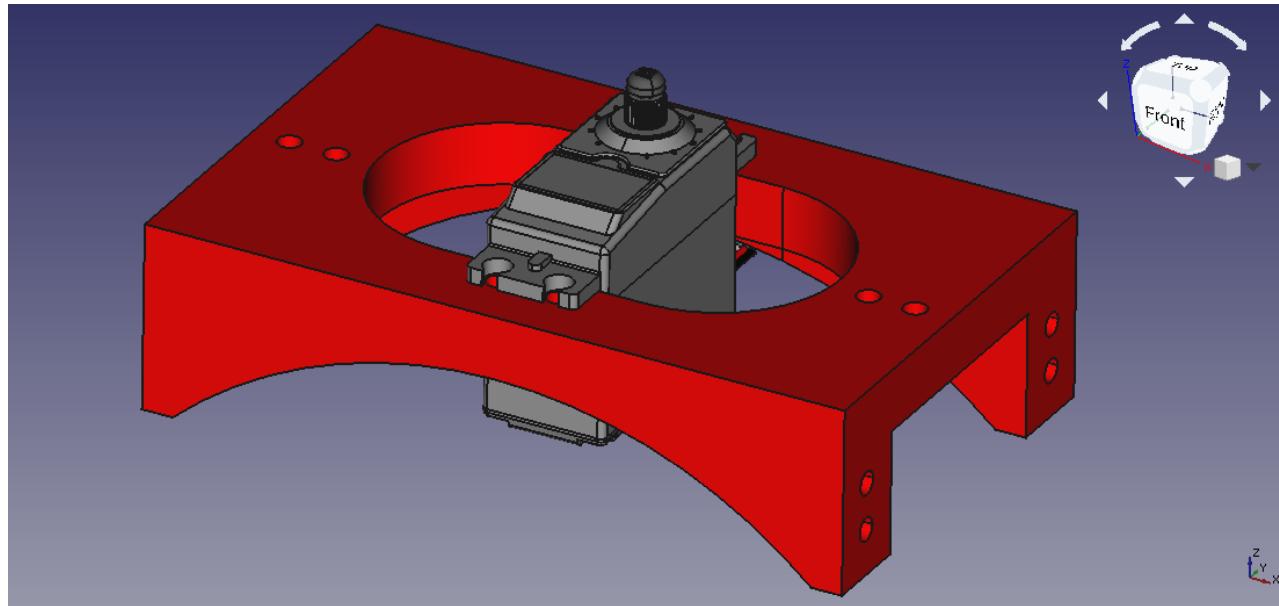


Figura 6.33: Motor de la base ensamblado en su soporte.

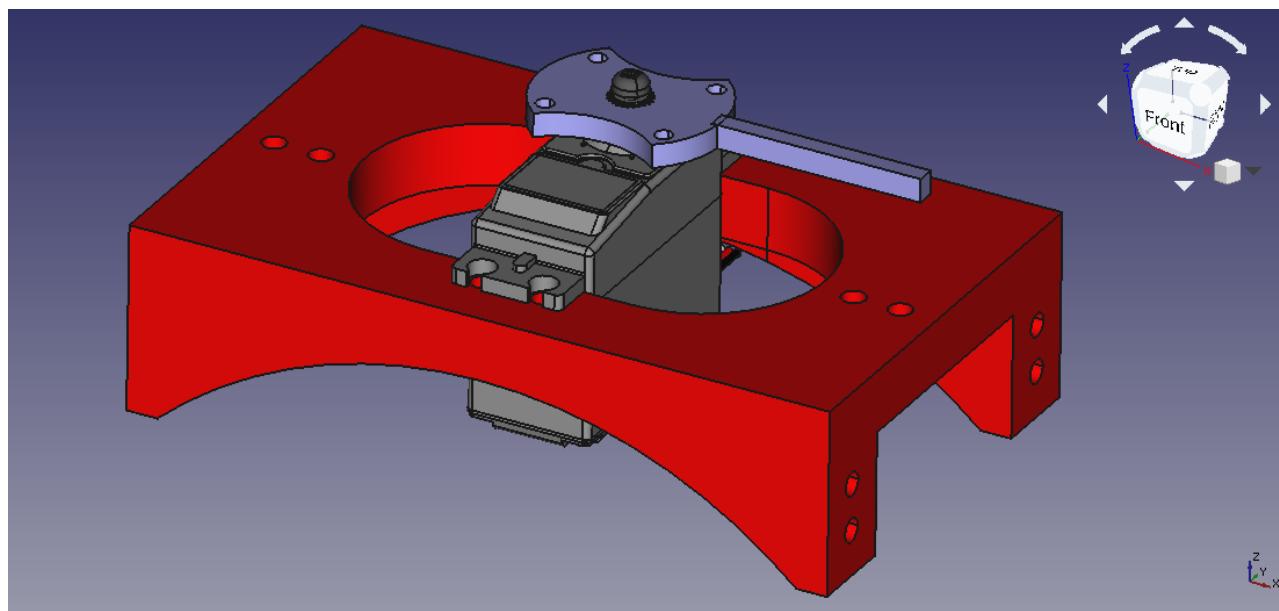


Figura 6.34: Primera pieza del sistema de transmisión del movimiento.

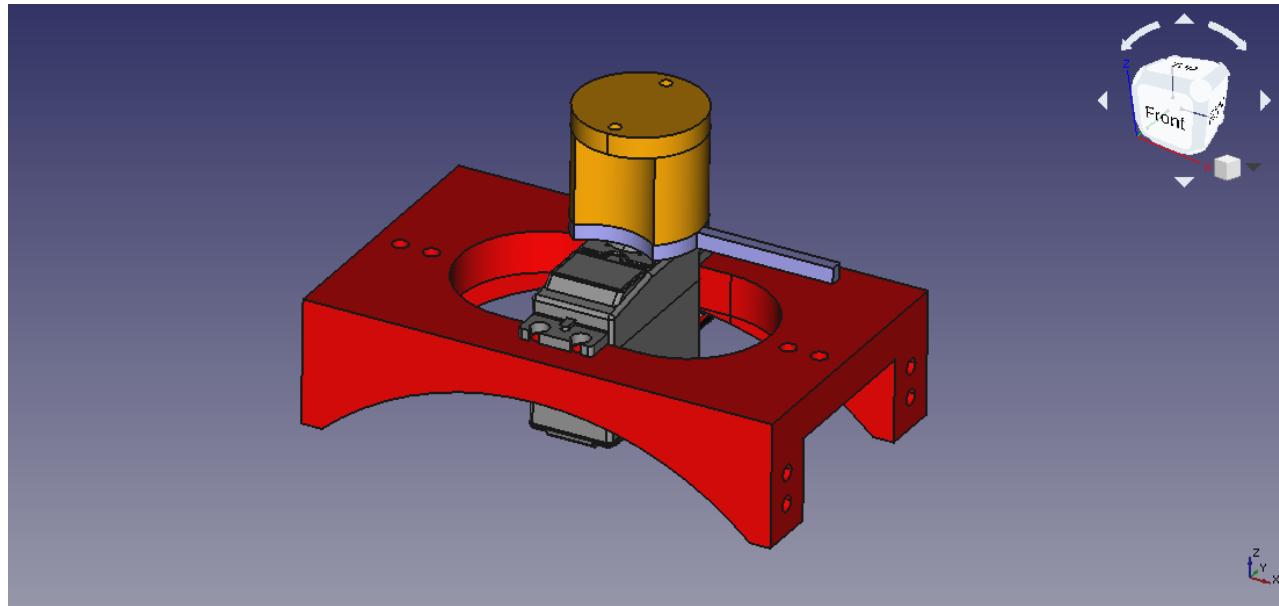


Figura 6.35: Segunda pieza del sistema de transmisión del movimiento.

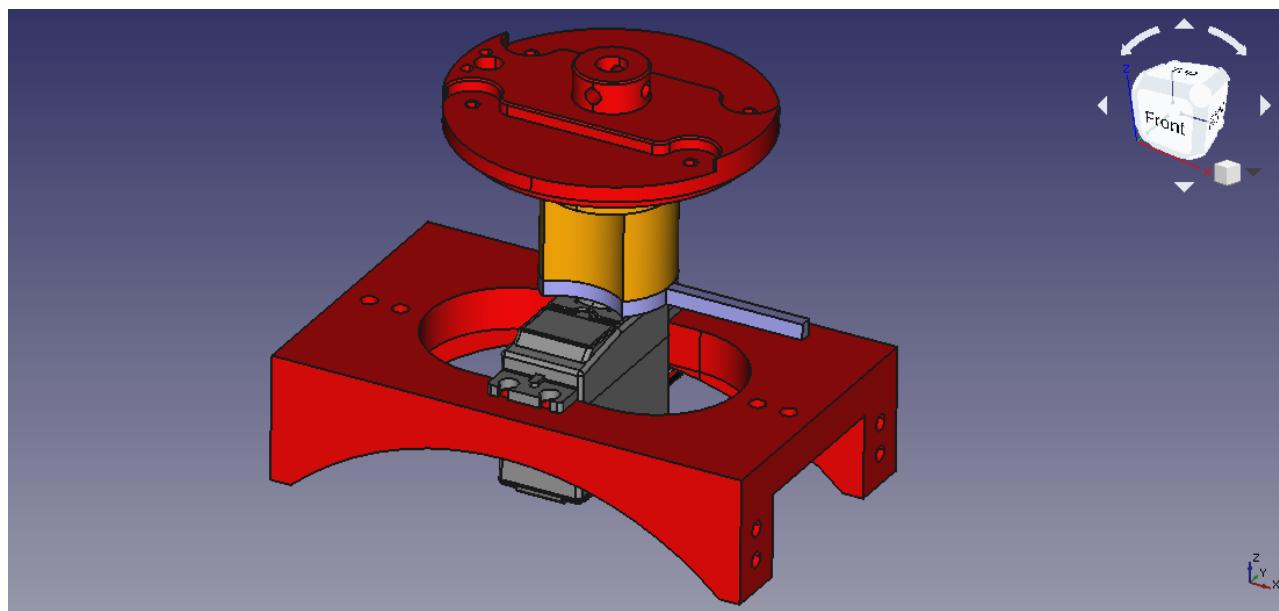


Figura 6.36: Tercera pieza del sistema de transmisión del movimiento.

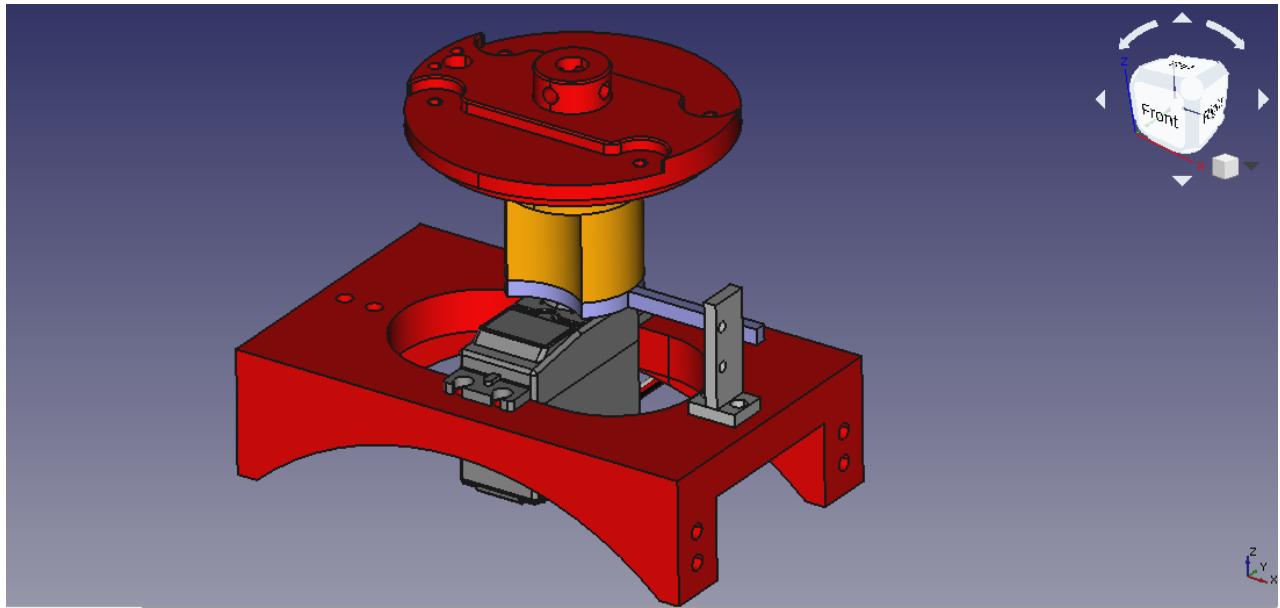


Figura 6.37: Cadena de transmisión final con soporte para fin de carrera.

En las figuras 6.32, 6.33, 6.34, 6.35, 6.36 y 6.37 se pueden ver los componentes que servirán para transmitir el movimiento desde el motor a la base giratoria, donde irá ensamblado el brazo robótico. Para asegurar el soporte a las paredes y posteriormente el motor al soporte se han empleado tornillos de $4 \times 15\text{mm}$. Para los componentes que servirán para transmitir el movimiento, se han empleado tornillos de $3 \times 10\text{ mm}$ debido a que las piezas son más pequeñas y es necesario que los tornillos ocupen menos espacio dentro de ellas. Esto es debido a que, con agujeros demasiado grandes, la integridad estructural de la pieza podría verse comprometida.

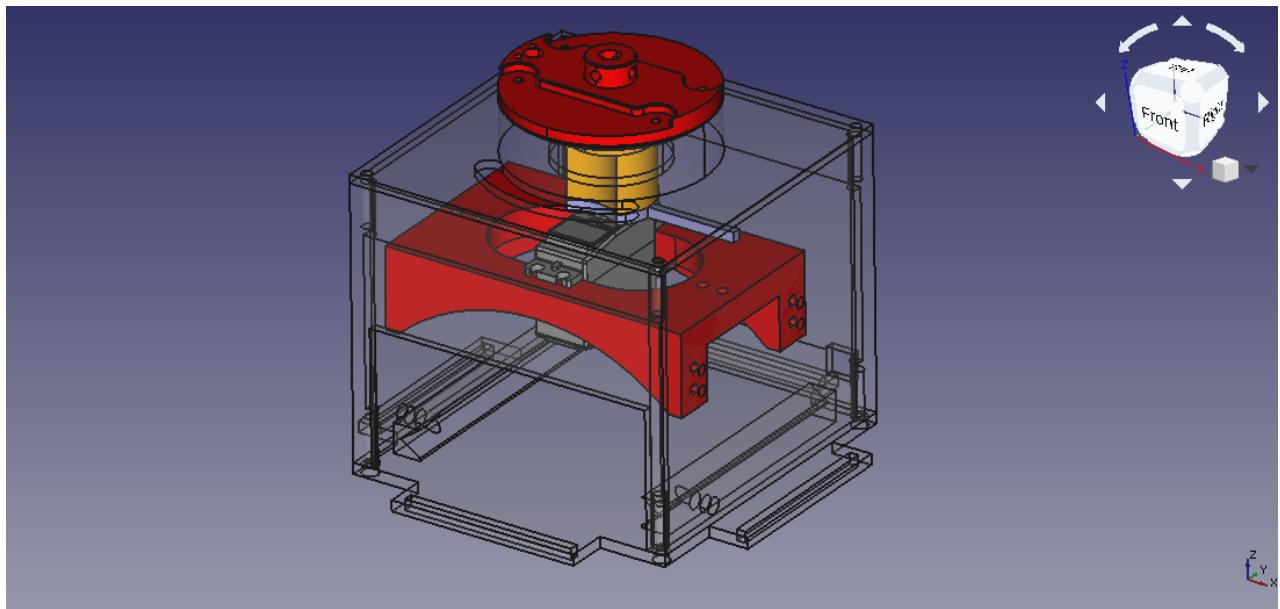


Figura 6.38: Interior de la caja.

Sobre la base rotatoria superior será montado ahora el sistema de motores que se encargará

de mover el brazo en el eje vertical.

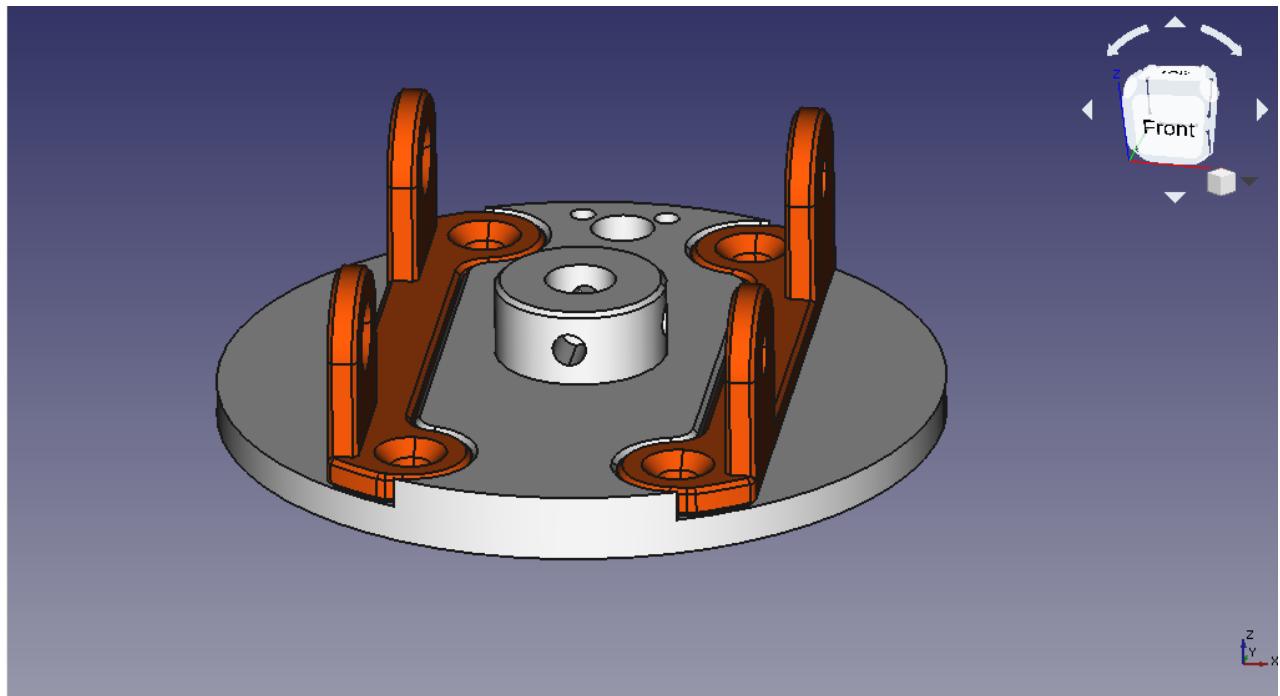


Figura 6.39: Base rotatoria con pletinas.

En la figura 6.39 se observa que las pletinas que sujetarán los motores a la base encajan en las hendiduras que existen en la base rotatoria. Estas pletinas son aseguradas a la base rotatoria mediante tornillos $3 \times 6\text{ mm}$.

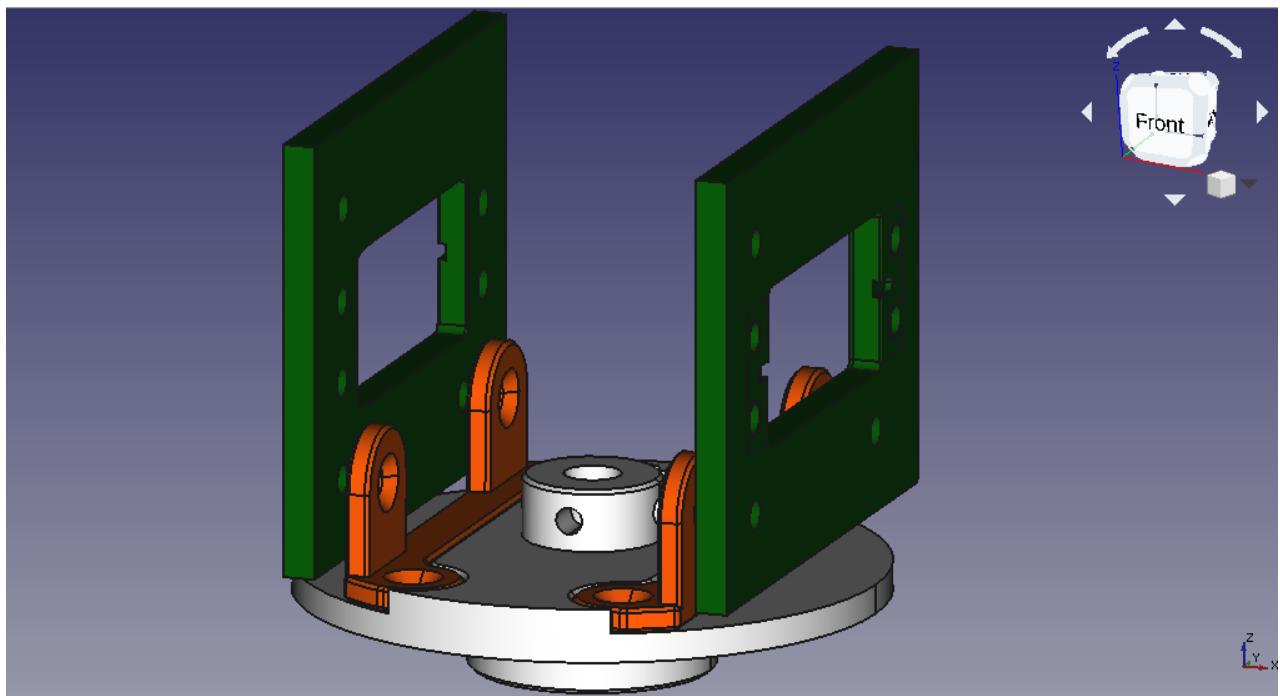


Figura 6.40: Soporte para los motores laterales.

A continuación los soportes de los motores son añadidos a las pletinas y son asegurados a estas mediante tornillos de $4 \times 15\text{ mm}$ con tuercas y arandelas. Entre la pletina y el soporte se puede observar una pequeña distancia, la cual se debe a un separador que existe en la construcción real pero que no aparece en el diseño 3D.

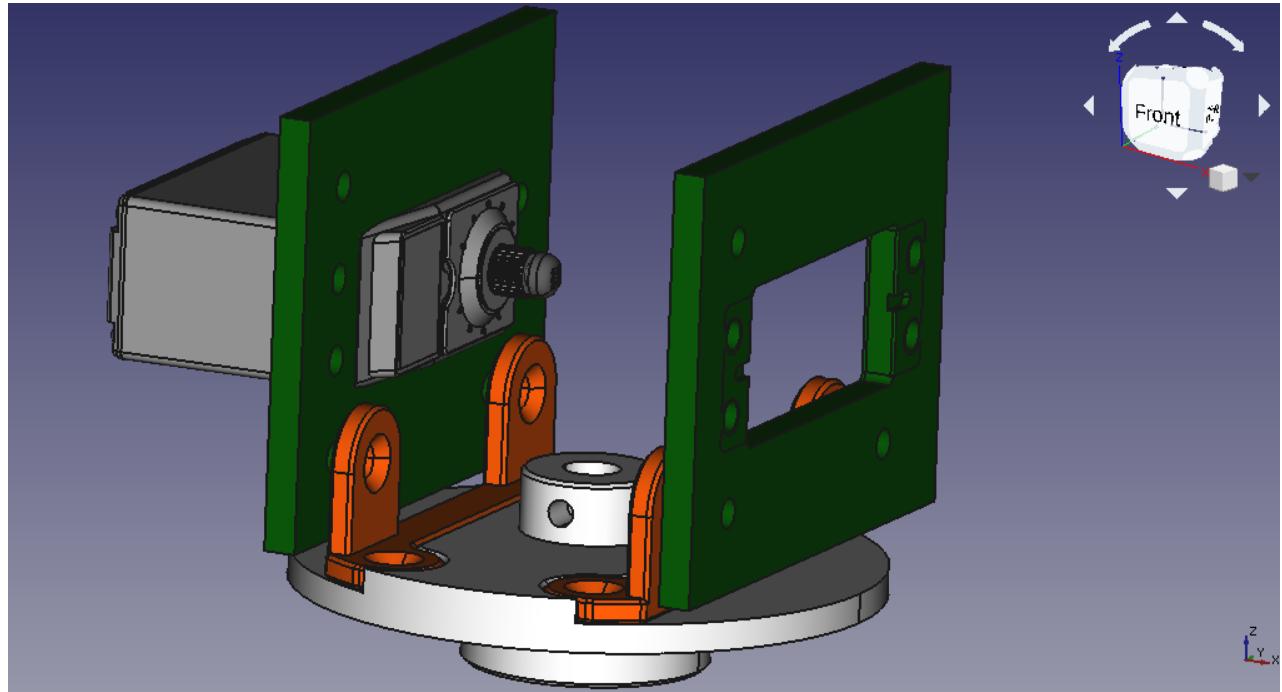


Figura 6.41: Se añade uno de los motores.

Finalmente, uno de los motores es añadido a su soporte y es asegurado a este mediante tornillos de $4 \times 10\text{ mm}$ con una tuerca.

Por otro lado, es ensamblada la parte superior del brazo según se indica a continuación:

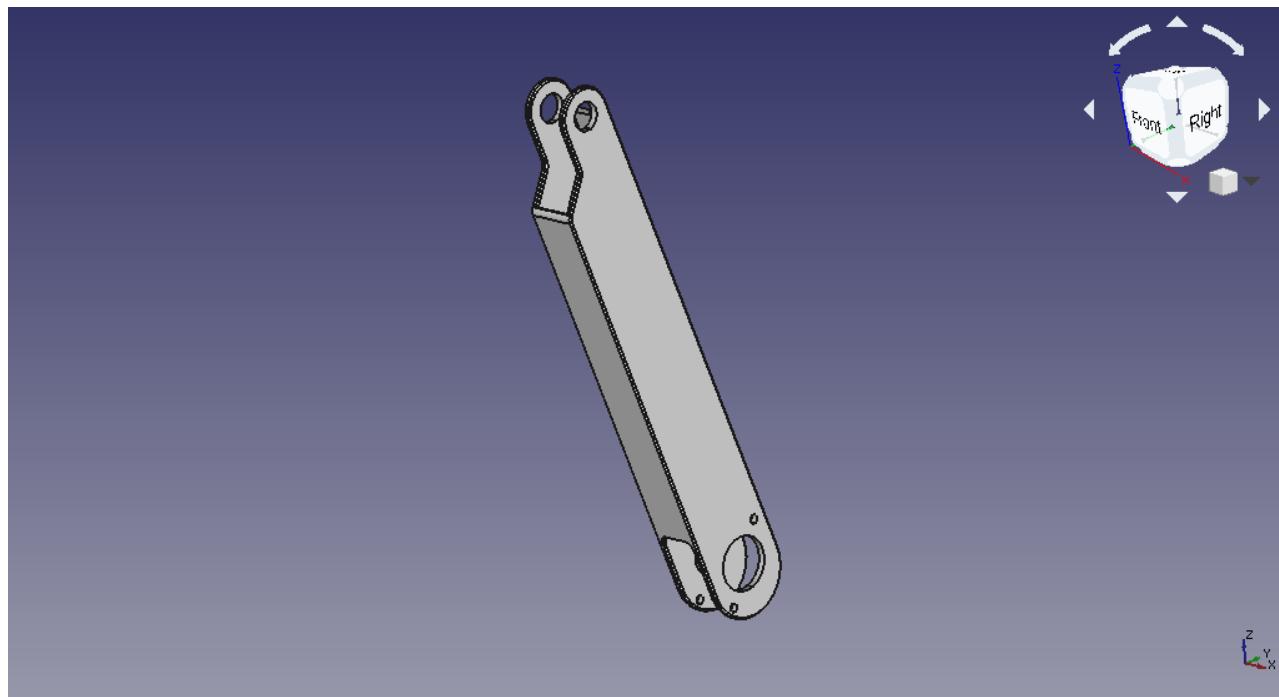


Figura 6.42: Segmento central del brazo robótico.

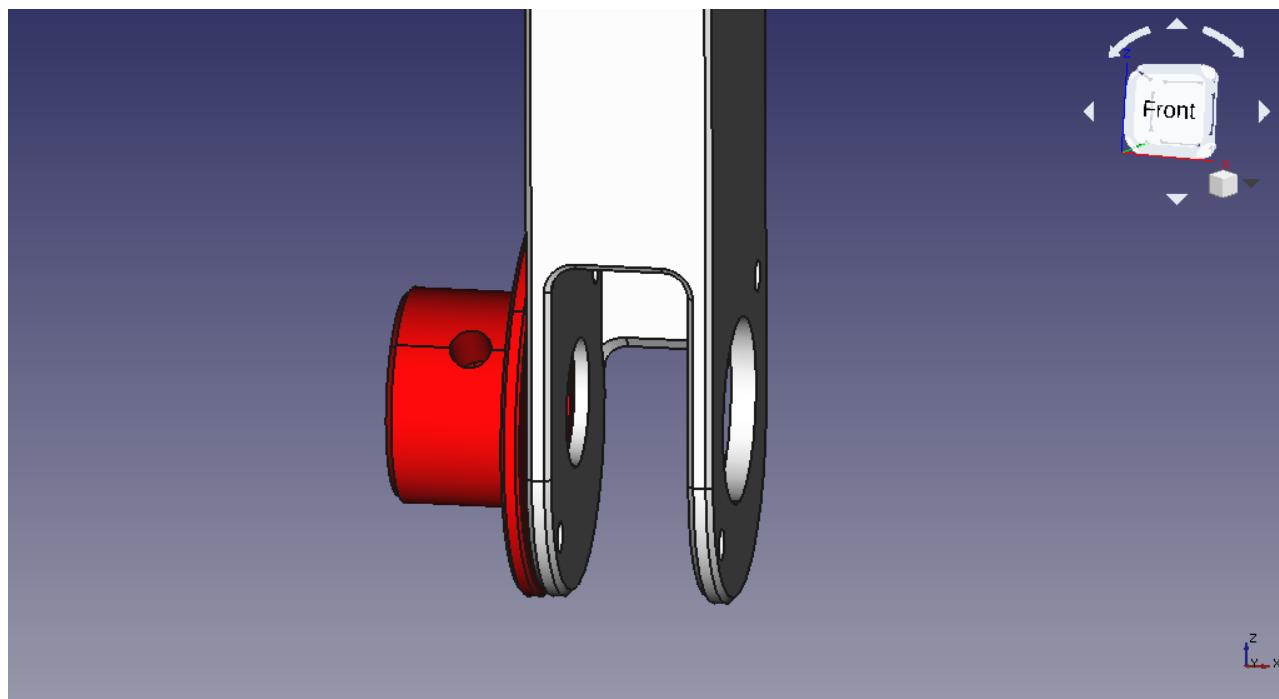


Figura 6.43: Pieza izquierda de la cadena de movimiento vertical.

En las figuras 6.42 y 6.43 se observa el montaje de la primera pieza de la cadena de movimiento vertical sobre el segmento central del brazo. Para asegurar la pieza al brazo se emplean tornillos de $3 \times 6\text{ mm}$.

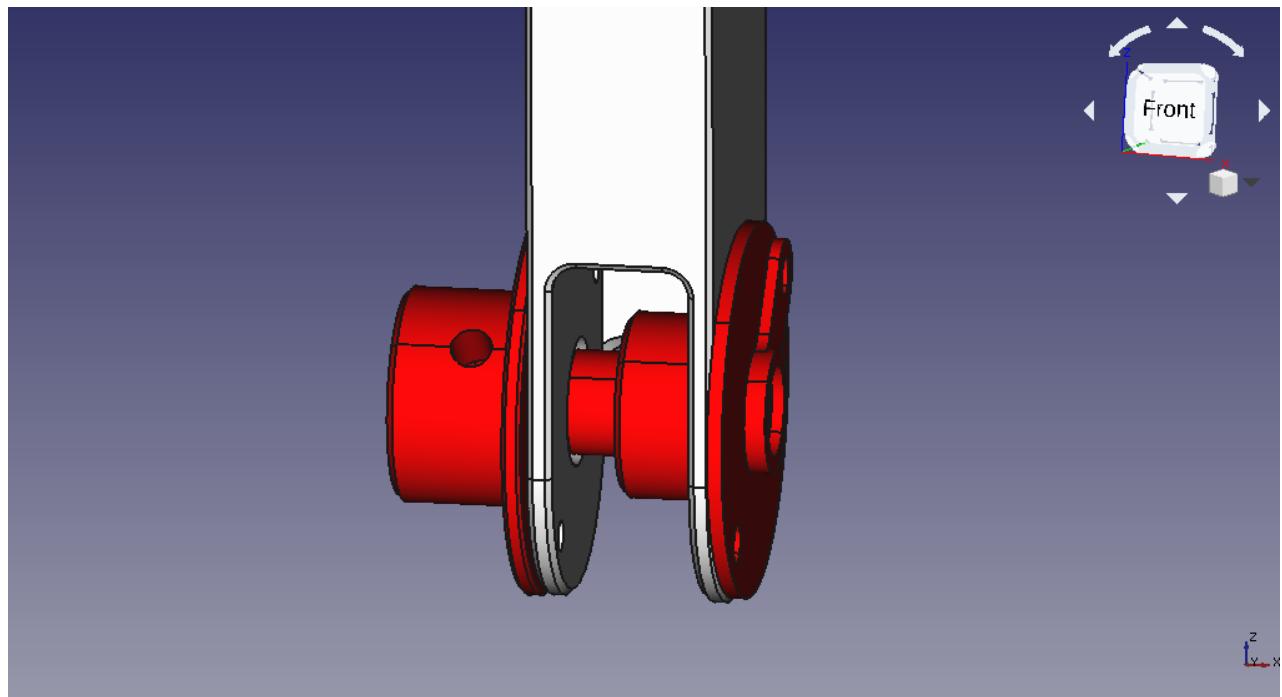


Figura 6.44: Pieza central de la cadena de movimiento vertical.

En la figura 6.44 se monta la pieza central empleando el mismo tipo de tornillos de tamaño $3 \times 6\text{ mm}$.

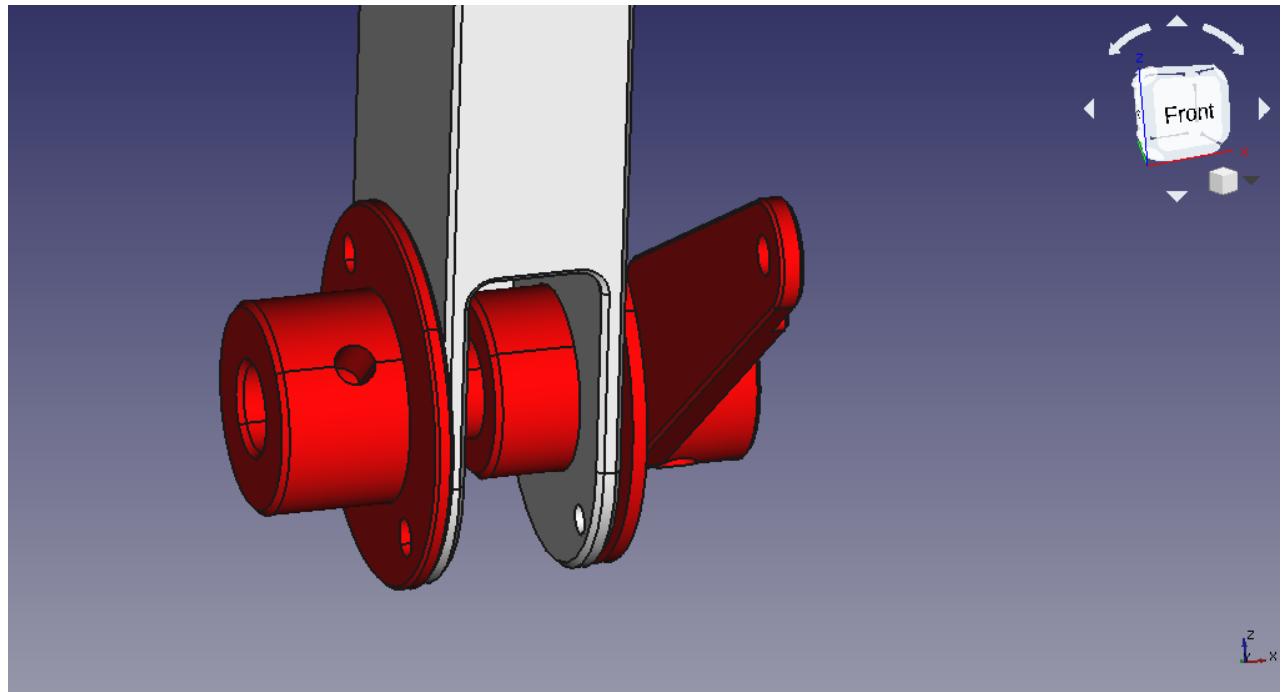


Figura 6.45: Pieza derecha de la cadena de movimiento vertical.

Finalmente, se añade la pieza derecha según se observa en la figura 6.45. Esta última es

soportada por un eje metálico que atraviesa las 3 piezas. Este eje se puede ver en la figura 6.46:

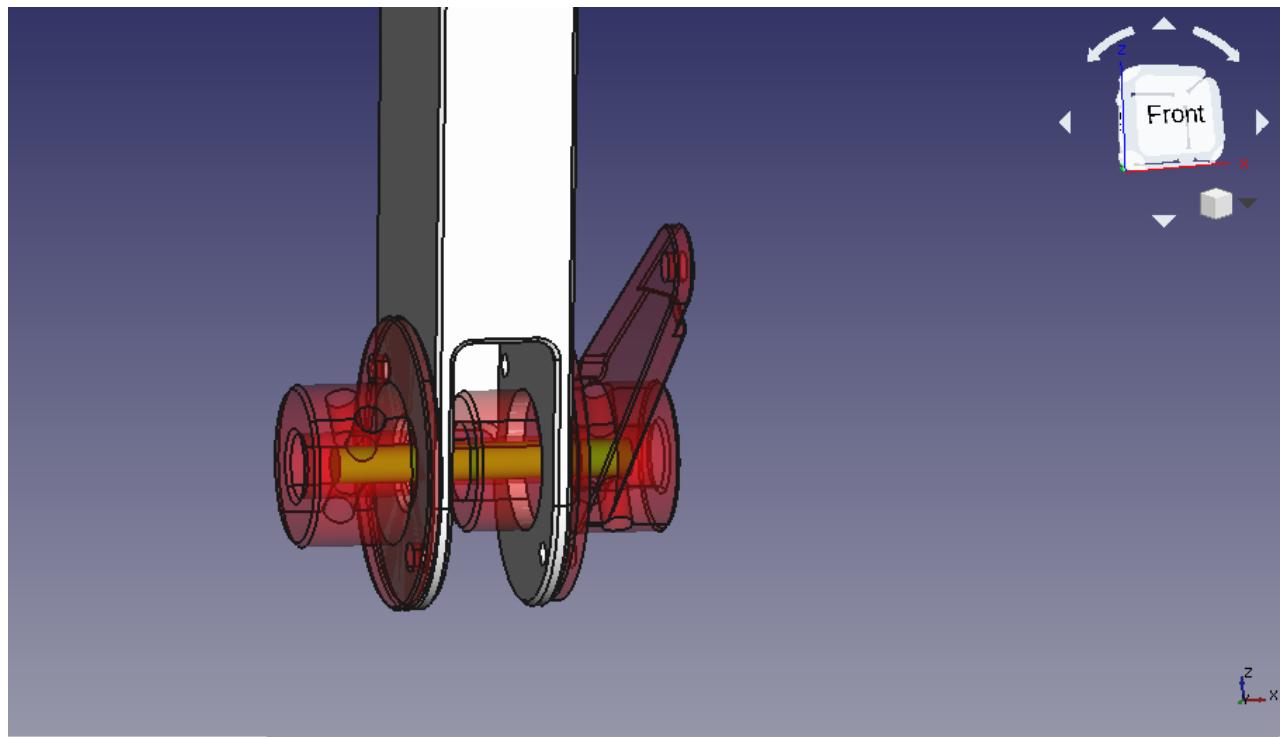


Figura 6.46: Eje metálico interno.

En la parte superior del segmento central se monta una pieza según se observa en la figura 6.47:

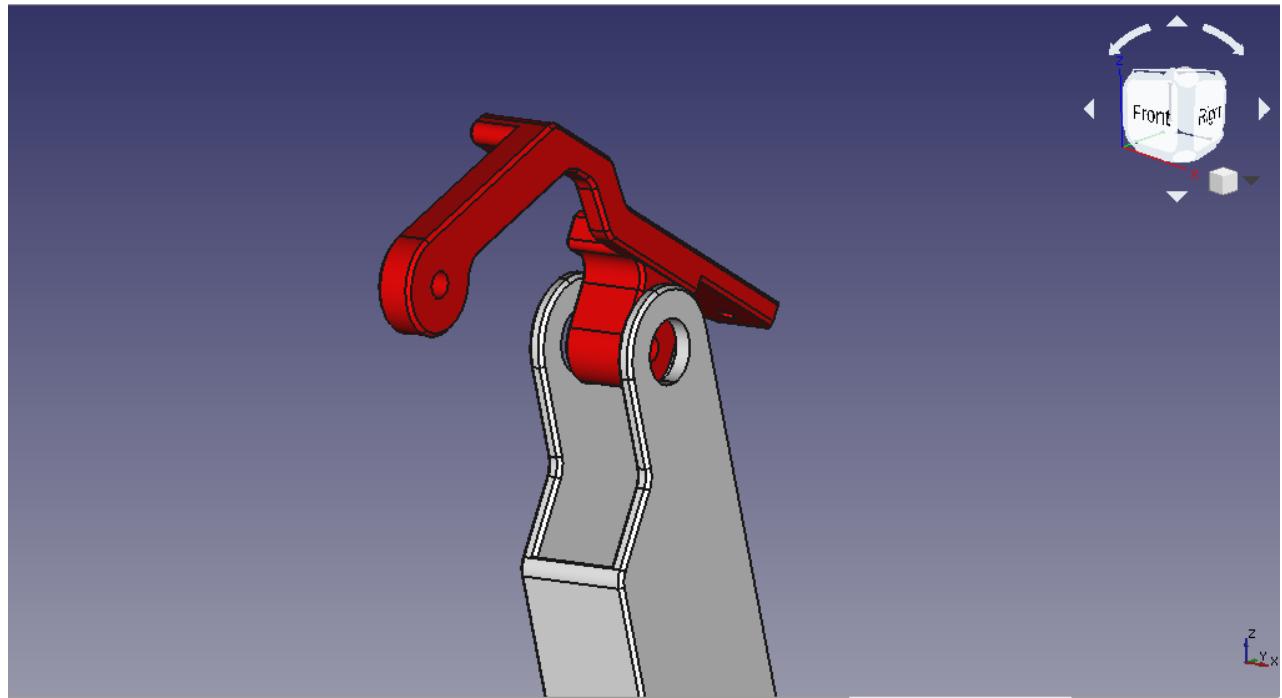


Figura 6.47: Pieza que sirve para unir el segmento inferior con el superior.

Esta pieza se moverá de manera libre sobre un tornillo $4 \times 15\text{ mm}$ que la atraviesa a modo de eje y que apoya sobre el segmento inferior en dos rodamientos, los cuales se encuentran en las ranuras circulares que se observan en la figura 6.47.

Empleando esta pieza es posible asegurar el antebrazo según se ve en la figura 6.48. También se pueden apreciar los rodamientos de los que se hablaba en el párrafo anterior.

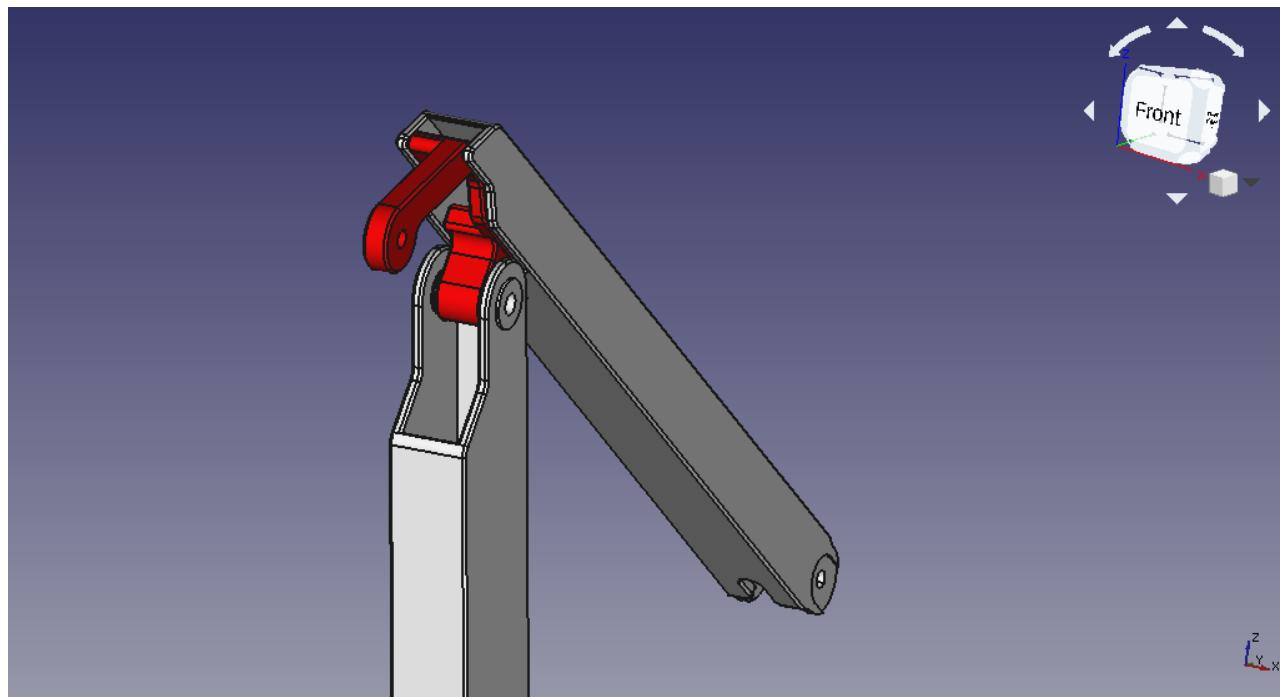


Figura 6.48: Antebrazo montado en la cadena articulada principal.

Finalmente, se añade a la cadena articulada principal el soporte del *end-effector* (figura 6.49). Este se mantiene unido al antebrazo mediante un tornillo de $4 \times 15\text{ mm}$ el cual sirve de eje libre que lo atraviesa y se apoya en los dos rodamientos que se observan de color naranja en la figura mencionada anteriormente.

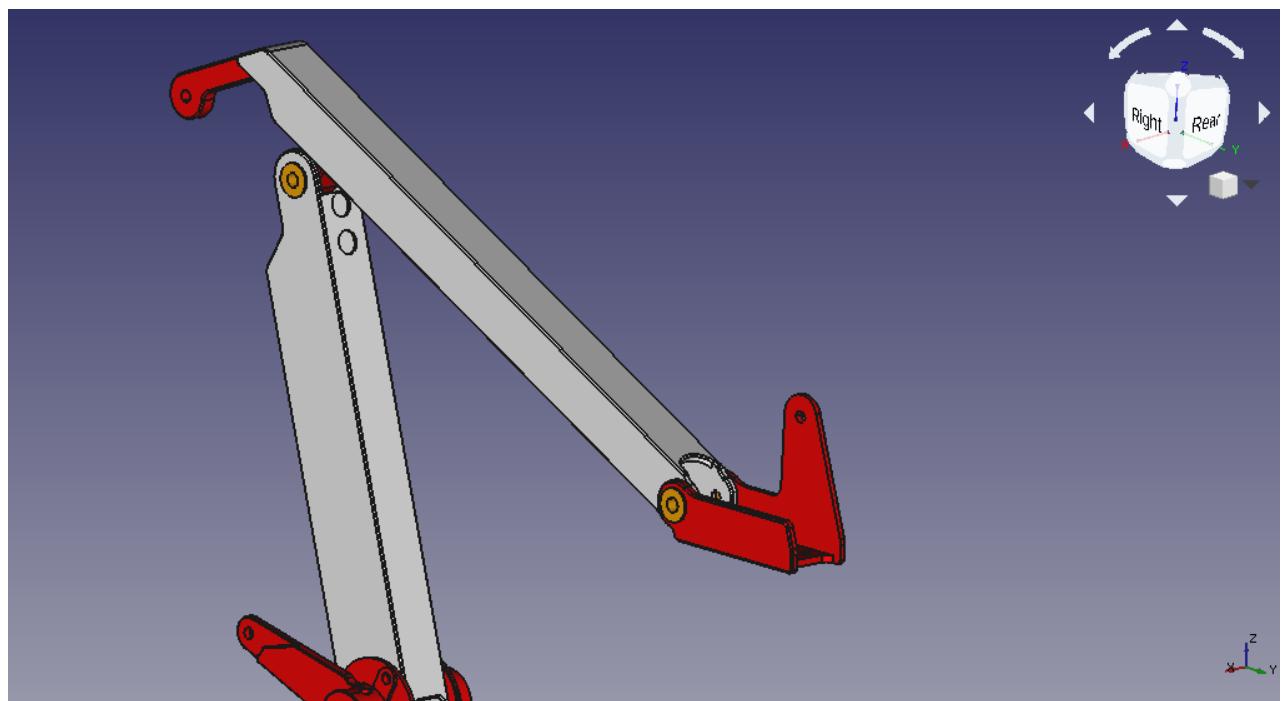


Figura 6.49: Soporte del *end-effector*.

También se añade al soporte del *end-effector* la sujeción para un posible motor que podría operar una pinza (figura 6.50). Para asegurar esta pieza sobre el soporte, se emplean tornillos de $3 \times 6\text{ mm}$.

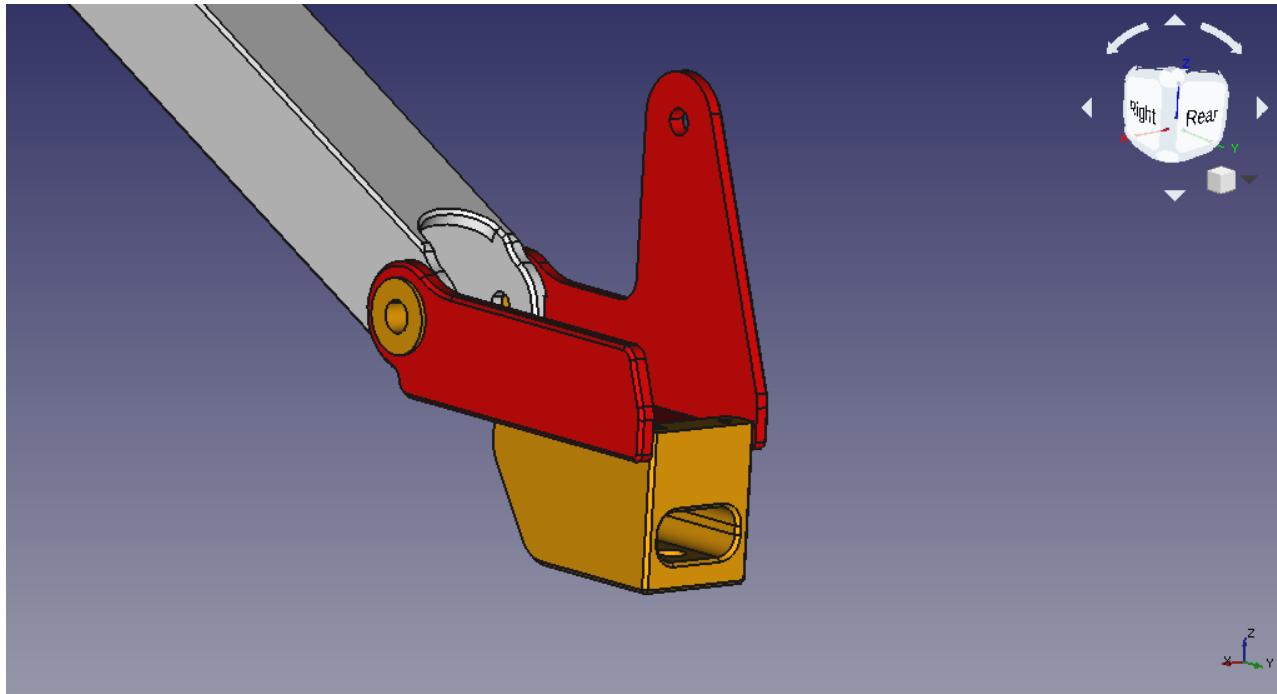


Figura 6.50: Soporte del motor que actúa sobre el *end-effector*.

Llegados a este punto es preciso ensamblar las cadenas articuladas auxiliares las cuales permiten transmitir movimientos al antebrazo y mantener el *end-effector* paralelo al suelo.

Primero se ensambla la cadena auxiliar derecha, según se observa en la figura 6.51:

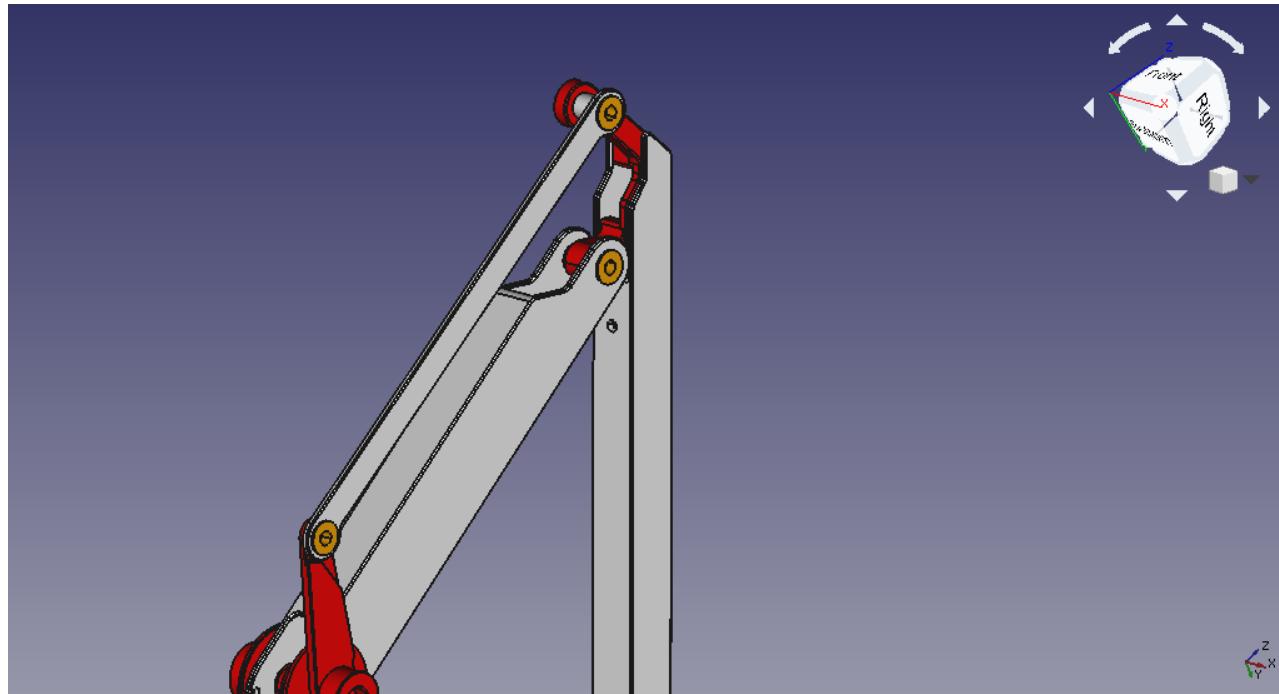


Figura 6.51: Cadena articulada auxiliar derecha.

La cadena auxiliar derecha se compone de una sola varilla que se ensambla en una de las piezas de la cadena de movimiento vertical, en la parte inferior del brazo, mediante un tornillo de $4 \times 10\text{ mm}$. En la parte superior, la varilla se ensambla en la junta de unión del brazo con el antebrazo mediante un tornillo de $4 \times 15\text{ mm}$. Se puede observar que en ambos casos para permitir que los tornillos actúen como eje libre, estos se apoyan sobre rodamientos.

En segundo lugar, se ensambla la cadena auxiliar izquierda. Se empieza añadiendo el triángulo de unión de las dos varillas que se explicarán más adelante. Este triángulo se observa en la figura 6.52:

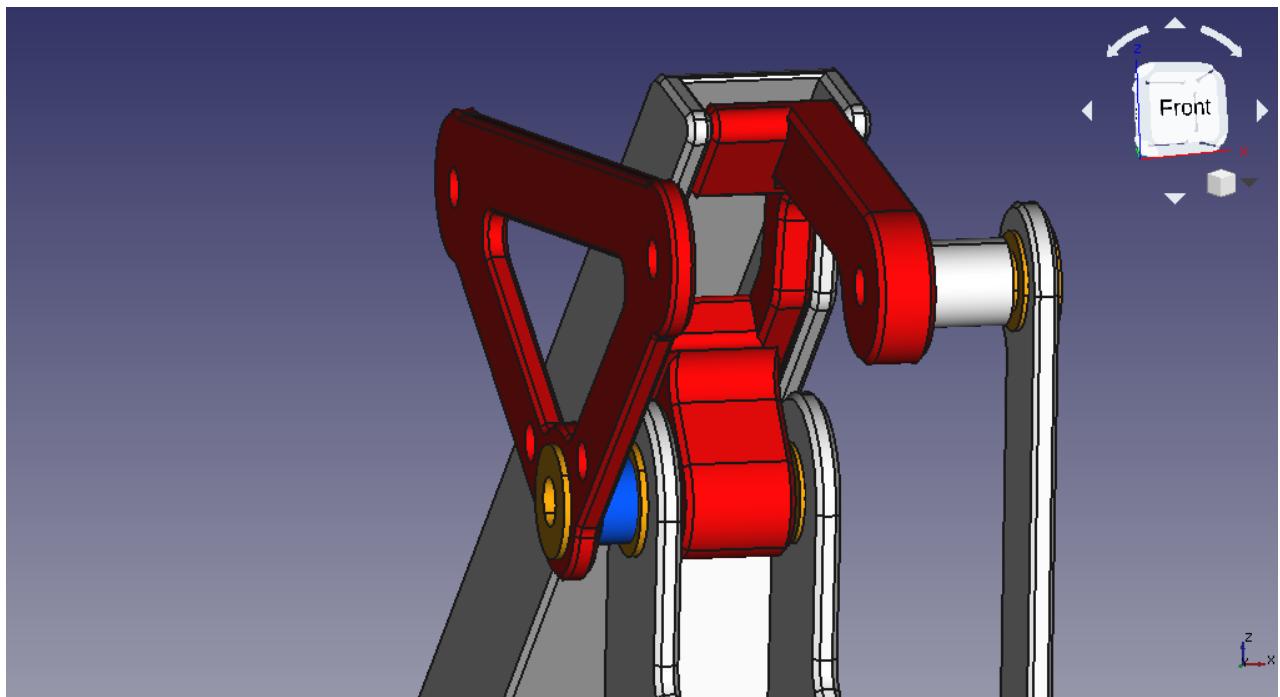


Figura 6.52: Triangulo de unión del las varillas de la cadena auxiliar izquierda.

Este triángulo se sujeta sobre el mismo eje que la junta de unión. Se observa en azul un separador y en naranja el rodamiento gracias al cual se consigue que el tornillo sirva como eje libre.

Sobre este triángulo se añade una primera varilla que lo une con uno de los soportes de los motores de la base (figura 6.53):

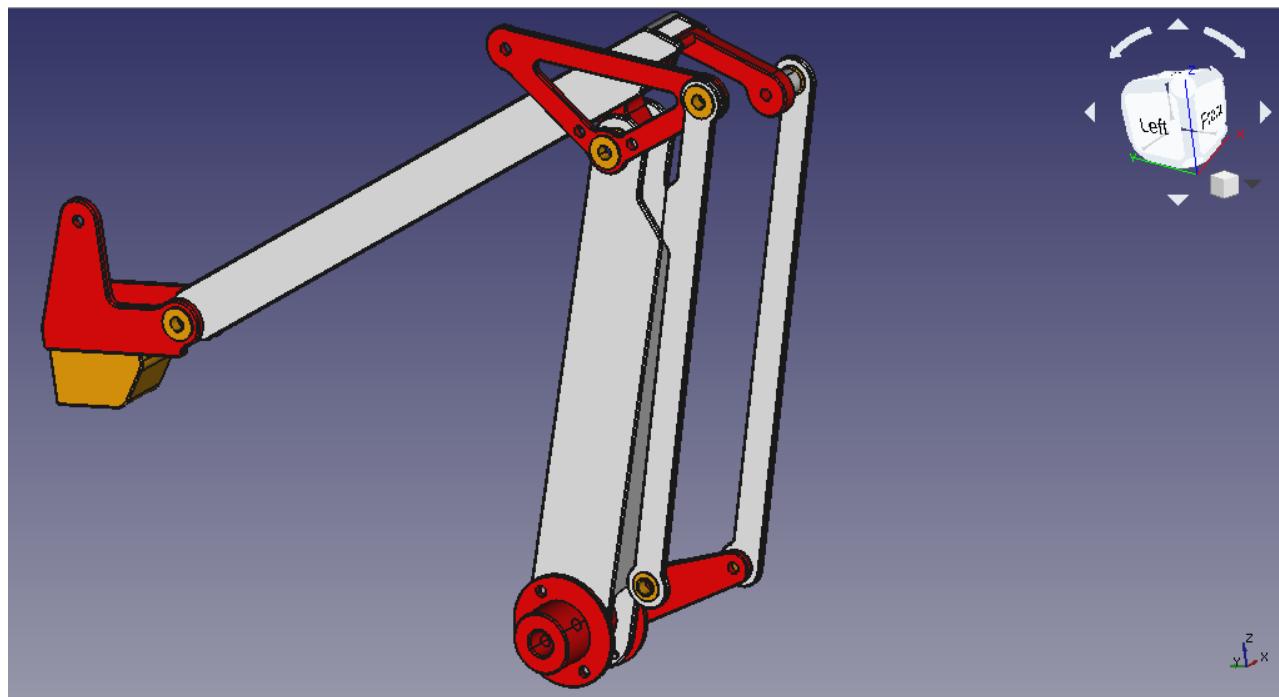


Figura 6.53: Varilla inferior de la cadena auxiliar izquierda.

Para sujetar la varilla al triángulo se emplea un tornillo de $4 \times 10\text{ mm}$ el cual apoya sobre un rodamiento. En el soporte inferior del motor se sujeta mediante un tornillo de $4 \times 40\text{ mm}$.

Finalmente se une la varilla superior con el *end-effector* mediante un tornillo de $4 \times 10\text{ mm}$ y un rodamiento. En el triángulo se emplea el mismo método de sujeción. Esto se puede observar en la imagen 6.54:

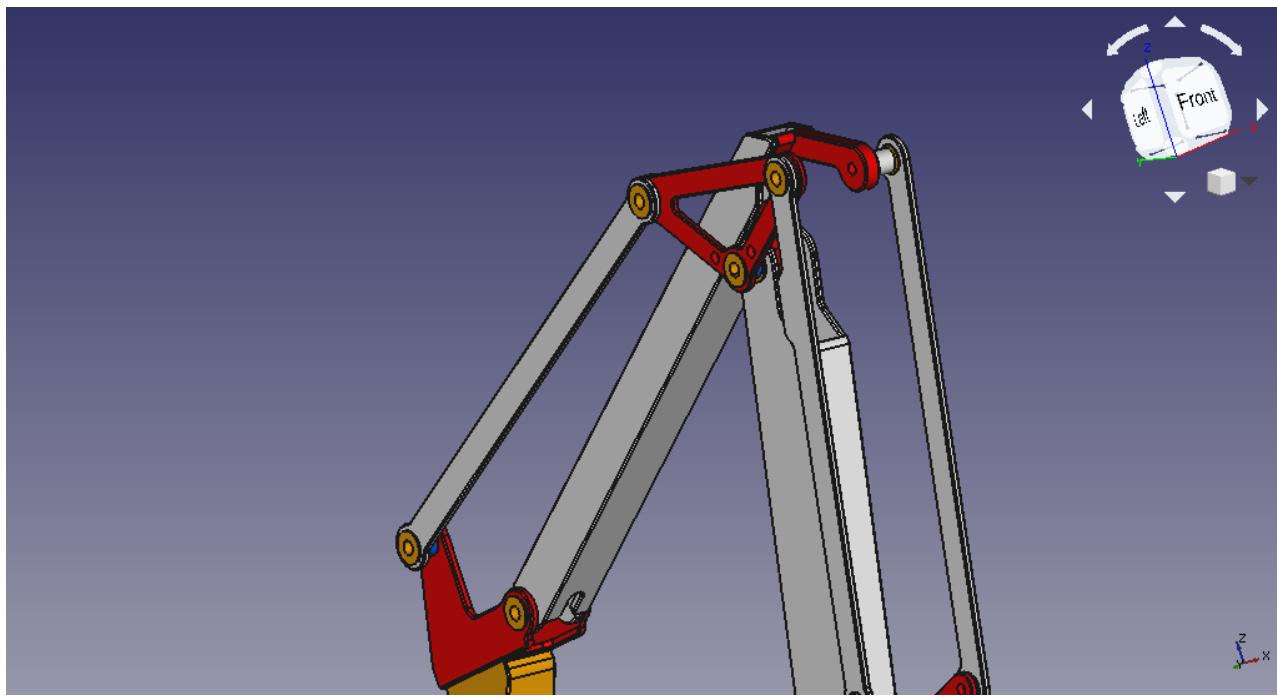


Figura 6.54: Varilla superior de la cadena auxiliar izquierda.

Con esto se finaliza la construcción de la parte superior del brazo robótico y ya es posible introducirlo en la base rotatoria, según se puede ver en la figura 6.55:

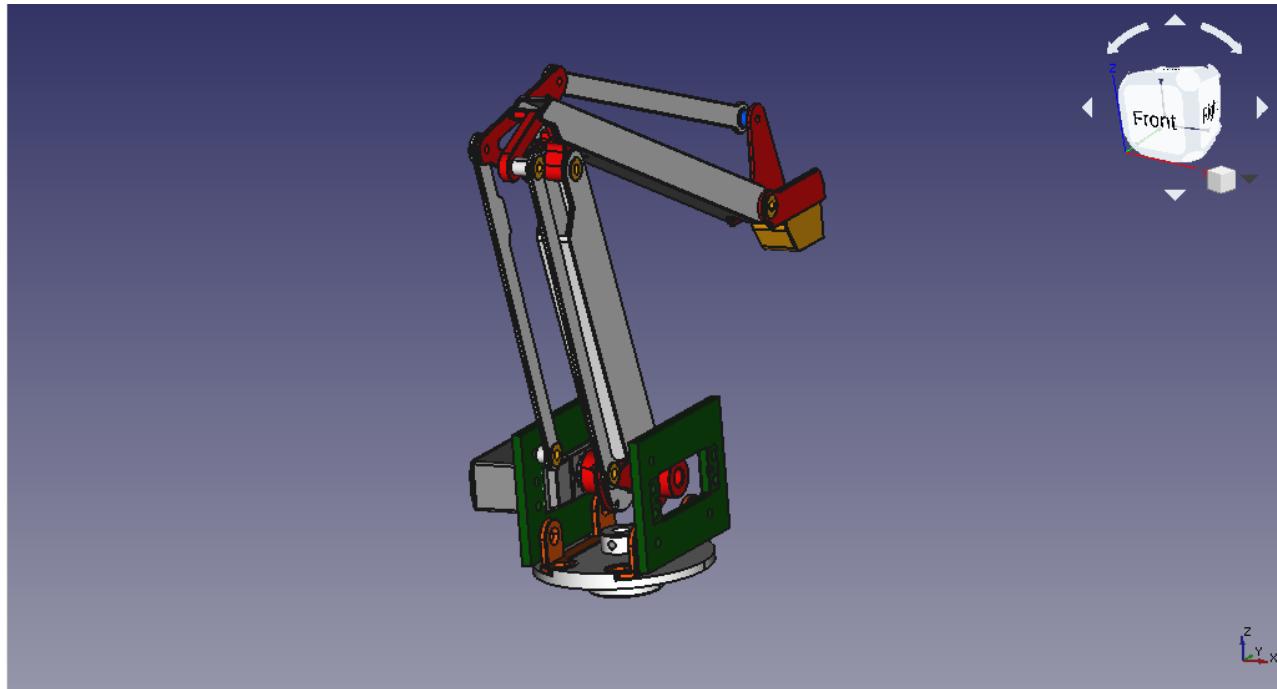


Figura 6.55: La parte superior el brazo robótico ensamblada en la base rotativa.

Finalmente, se coloca el último motor y se deja la parte superior del brazo fijada entre los dos.

De esta manera, la parte superior queda asegurada en la base rotatoria y la construcción está completa.

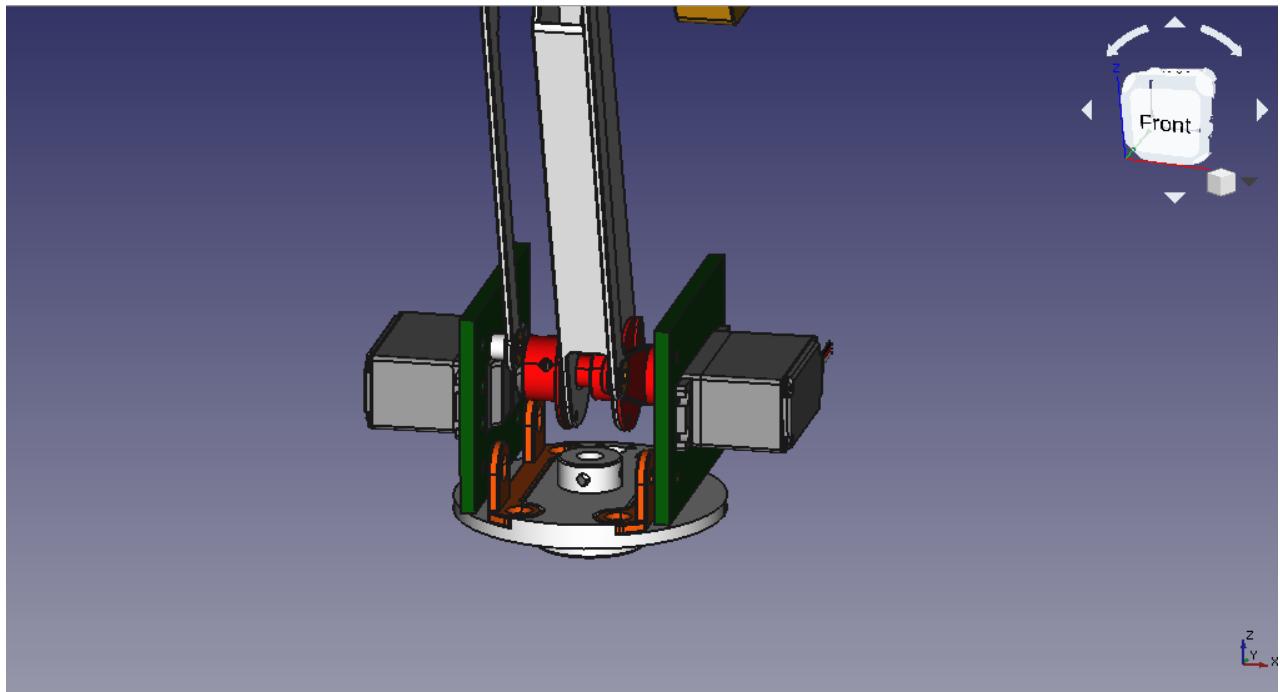


Figura 6.56: Brazo robotico completo.

6.3. Configuración mecánica del brazo

La estructura del brazo es pantográfica, lo cual significa que es un sistema de enlaces mecánicos que reproduce el movimiento de un punto de una articulación en un segundo punto, normalmente a un tamaño o bien más pequeño o bien más grande, como se puede apreciar en la figura 6.57. Se originó en el siglo XVII y su aplicación más conocida es como instrumento de dibujo.

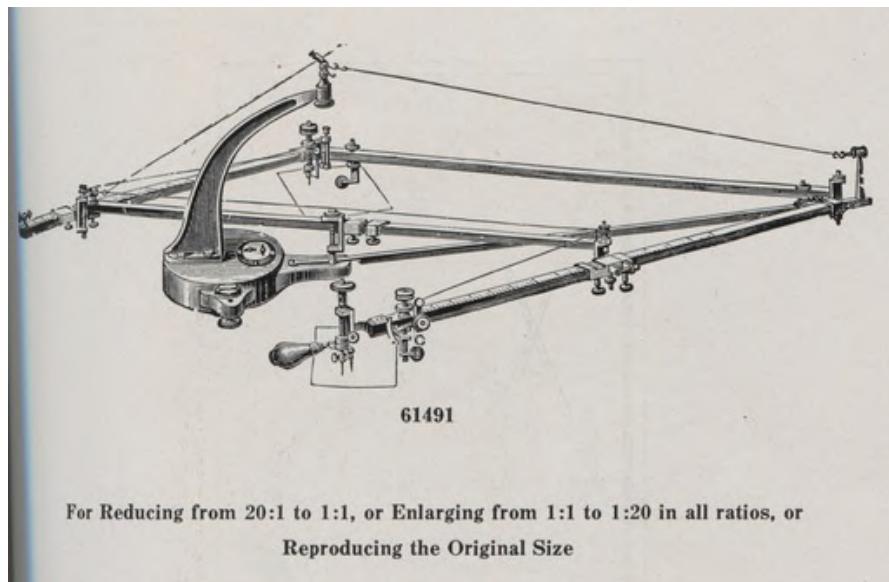


Figura 6.57: Una estructura pantográfica que transmite el movimiento de un punto al siguiente [34].

El hecho de que la estructura del brazo sea pantográfica permite controlar todas las articulaciones mediante motores ubicados en la base. Esto es de especial importancia ya que hace posible que las articulaciones finales no carguen con el peso de los motores, permitiendo emplear materiales como el plástico para la construcción de la estructura y, además, dando capacidad al brazo para levantar cargas más pesadas.

Otro beneficio derivado de la ubicación de los motores en la base es la alta estabilidad del brazo, ya que siendo estos los componentes más pesados y estando ubicados en la base se consigue un centro de gravedad muy cercano a la superficie de apoyo, permitiendo así un amplio rango de movimientos.

Otra característica a destacar es que debido a la estructura pantográfica, el *end-effector* mantiene siempre un ángulo perpendicular con la superficie de apoyo del brazo. Esto supone, por un lado, perder ciertos grados de libertad pero por otro, simplifica la estructura y abarata el coste de producción.

6.4. Microcontrolador utilizado

Para el desarrollo de la placa que conforma S2 se ha empleado un microcontrolador “dsPIC33EP512GM604”. Los motivos por los cuales se ha usado este modelo son los siguientes:

- En primera instancia, la cantidad y la precisión de los canales PWM que este ofrece, ya que son suficientes para poder controlar todos los motores y su precisión permite generar una señal adecuada para controlar a cada uno de ellos.
- Por otro lado, debido a la naturaleza de los cálculos que se deben realizar para convertir posiciones cartesianas a ángulos y viceversa, el *Digital Signal Processor* (DSP) facilita

la obtención de los valores de las diferentes operaciones matriciales que permiten esta conversión.

- Otro aspecto importante es la posibilidad de almacenar hasta 512KB de memoria de programa.
- Por último, se ha elegido un DSP de Microchip debido a que todos los integrantes del grupo de desarrollo tiene experiencia previa con este fabricante. Además, dicho fabricante proporciona documentación extensa sobre sus productos.

Una parte crítica del proyecto es la precisión y el control de los motores. En este aspecto, los canales PWM del DSP permiten generar ciclos de trabajo a partir de un registro de 16 bits. Para una frecuencia de $50Hz$ (período de $20ms$), un giro de 360° supondría una duración del nivel alto del PWM de $5ms$. Es decir, se podrá controlar la rotación entre 0° y 360° con $2^{16}/4 = 16384\ bits$ de control. Esto supone que obtendremos una precisión máxima de $360/16384 = 0'02197^\circ$, lo cual es suficiente para el proyecto.

Cabe destacar, que si bien los motores son capaces de girar 360° sexagesimales, debido a las limitaciones geométricas del brazo, los ángulos de giro se ven limitados en la aplicación concreta que se les dará.

En cuanto a las operaciones matemáticas, el DSP cuenta con un circuito *Phase Loop Lock* (PLL), el cual permite incrementar la frecuencia del oscilador para conseguir de esta manera una mayor cantidad de instrucciones por segundo y, por tanto, un mayor volumen de operaciones.

En el momento de comenzar el proyecto se podía suponer que el código no podría albergarse en la memoria que normalmente suministran microcontroladores de menores prestaciones.

Este microcontrolador, gracias a que dispone de varios puertos UART permite recibir las órdenes y los movimientos necesarios desde S1. Desde S2, se realizarán las operaciones matemáticas necesarias y después se encargará de generar las señales PWM para mover los motores de tal manera que el brazo robótico quede en la posición deseada.

6.5. Desarrollo y componentes de la PCB

6.5.1. Objetivos

El desarrollo de una PCB se considera una de las partes esenciales de este proyecto y por lo tanto su importancia es máxima.

El objetivo principal que persigue el diseñar y construir una PCB en este proyecto es el de dotar al sistema de un centro de cómputo principal, el cual es utilizado para procesar las órdenes de S1, realizar los cálculos pertinentes y ejecutar las acciones necesarias sobre la estructura del brazo robótico. La PCB por lo tanto se encarga de alojar el microcontrolador dsPIC, así como todos los periféricos necesarios para establecer la comunicación con S1, realizar el control de los actuadores y monitorizar el estado del manipulador.

6.5.2. Componentes principales

En general, se podría decir que los componentes de la PCB se clasifican en tres categorías principales:

- Componentes de alimentación eléctrica.
- Microcontrolador y componentes auxiliares para su correcto funcionamiento.
- Periféricos destinados a control de actuadores y canales de comunicación.

En primer lugar, los componentes de alimentación eléctrica son aquellos que forman el circuito de alimentación del microcontrolador así como de los actuadores. El circuito eléctrico de alimentación de la PCB se ha diseñado para poder alimentar de forma simultánea al microcontrolador y a cada uno de los cuatro servomotores y está formado por dos etapas:

- La PCB se conecta mediante una clema a una fuente de alimentación de 9V que puede suministrar una corriente máxima de 2A. Posteriormente, esta tensión de alimentación será reducida y adaptada para alimentar a cada una de las etapas de la PCB, es decir, al microcontrolador y servomotores.
- En la primera etapa se reduce el voltaje de alimentación principal a 6V, permitiendo que cada uno de los servomotores demande una corriente máxima de 0,4A aproximadamente, utilizando para ello un regulador LM317 por cada uno. Esta alimentación se realiza mediante clemas, las cuales se usan para conectar la alimentación de los motores.
- En la segunda etapa se reduce el voltaje de alimentación principal a 3,3V y estimándose un consumo máximo de 0,15A aproximadamente con el objetivo de alimentar el microcontrolador, utilizando en conjunto los reguladores L7805CV y A1117H. Esta alimentación se realiza mediante pistas únicamente.

En segundo lugar, el microcontrolador y sus componentes auxiliares representan el núcleo de la PCB:

- El dsPIC se encuentra localizado en el centro de la PCB y de él surgen todas las conexiones necesarias hacia los periféricos.
- Los componentes auxiliares del microcontrolador son componentes eléctricos que aseguran el correcto funcionamiento del dsPIC. En el caso específico de este microcontrolador, es recomendable incluir varios condensadores en sus pines de alimentación.

En último lugar, se han utilizado los siguientes periféricos:

- Cristal de cuarzo, el cual genera una señal de reloj precisa y de buena calidad. Su frecuencia es de $7,3728MHz$ y será recibida por el microcontrolador para ser usada como la señal de reloj principal del sistema.

- Puerto de programación, donde se puede conectar la sonda de programación del microcontrolador y por lo tanto es un elemento con especial importancia y relevancia dentro del sistema.
- Puerto entrada salida: para recibir señales digitales y analógicas, las cuales son procesadas por el microcontrolador. En este proyecto, estos puertos se utilizan para monitorizar los finales de carrera de la estructura del brazo robótico.
- Puerto PWM: capaz de generar señales PWM, es decir, señales digitales con un ancho de pulso variable, las cuales son necesarias para controlar los servomotores.
- UART, donde se establece un canal de comunicación *hardware* con S1, el cual se usa para recibir órdenes, movimientos y realimentar su resultado de vuelta a S1.
- LEDs de estado, los cuales muestran información del sistema usando tres diodos LED.

Mediante esta descripción general de la PCB y sus componentes se pretende brindar una idea global de la misma, así como de cuál es su papel dentro del proyecto. En los apartados siguientes se describe en términos técnicos los elementos de esta PCB, así como el proceso de diseño y fabricación llevado a cabo.

6.5.3. Diseño lógico y diagrama esquemático

El primer paso llevado a cabo durante el diseño de la PCB ha sido realizar un diseño lógico de alto nivel, en el cual se muestran las conexiones lógicas que existen entre los componentes; se trata, por lo tanto, del diseño con mayor nivel de abstracción y que tiene como objetivo establecer la primera toma de contacto con el plano de la PCB.

El diseño lógico se lleva a cabo mediante un diagrama esquemático que contiene dos tipos de elementos: huella lógica de cada uno de los componentes y las conexiones entre ellos. Este diagrama se ha llevado a cabo utilizando la herramienta “*Schematic Layout Editor*”, incluida en KiCad.

El diagrama esquemático está dividido en dos partes principales, las cuales facilitan la compresión del mismo:

- Diagrama esquemático del circuito de alimentación.
- Diagrama esquemático del microcontrolador y sus periféricos.

En primer lugar, se procede a describir detalladamente el diagrama esquemático del circuito de alimentación, el cual contiene las dos etapas de alimentación mencionadas anteriormente en el punto 6.5.2, siendo los principales componentes usados los reguladores de voltaje LM317, L7805CV y AZ1117H.

La clema principal de alimentación se conecta a una fuente de 9V que debe proporcionar una corriente máxima de 2A aproximadamente.

Conectados directamente a la clema principal, se encuentran los reguladores de tensión correspondientes a las dos etapas de alimentación:

- La primera está formada por cuatro reguladores LM317, los cuales alimentan cada uno de los servomotores. En dicha etapa se reduce el voltaje de 9V a 5,5V.
- La segunda está formada por un regulador de tensión L7805CV y un AZ1117H. En dicha etapa de alimentación se reduce el voltaje de 9V a 5V usando el primer regulador, y de 5V a 3,3V usando el segundo.

En particular, el LM317 es un regulador convencional que recibe una tensión continua de entrada de entre 3V – 40V y provee una tensión continua salida de entre 1,25V – 37V. Además, la relación entre la tensión de entrada y salida depende del dimensionado de dos resistencias auxiliares. El conexionado sugerido por el fabricante es el siguiente (ver imagen 6.58) y se ha obtenido del *datasheet* del regulador:

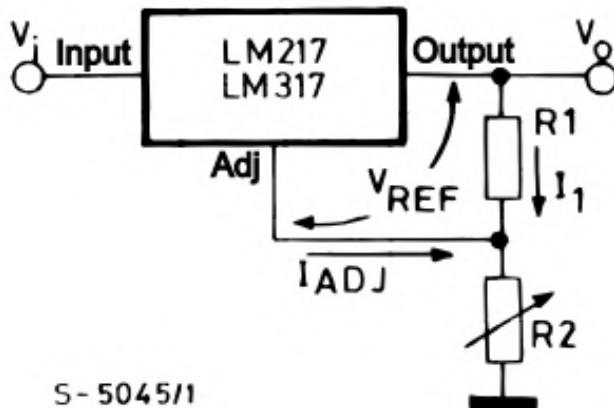


Figura 6.58: Diagrama de conexionado del LM317 [35].

La ecuación de funcionamiento que ofrece el fabricante para el regulador LM317 es la siguiente:

$$V_{OUT} = V_{REF} \cdot \left(1 + \frac{R_2}{R_1} \right) + I_{ADJ} \cdot R_2 \quad (6.1)$$

La ecuación anterior (ver ecuación 6.1) debe ser usada para realizar los cálculos pertinentes sobre el valor de las resistencias R_2 y R_1 . Se tienen en cuenta, además, dos observaciones necesarias para la correcta aplicación de la ecuación anterior:

- Por construcción, el fabricante establece el valor $V_{REF} = 1,25V$.
- Por motivos de construcción del regulador, la corriente I_{ADJ} tiene un valor máximo de $100\mu A$ y, por lo tanto, el término de la ecuación que la involucra puede ser despreciado en muchas ocasiones.

Se obtiene entonces que la ecuación funcional (ecuación 6.2) a utilizar en el cálculo de R_1 y R_2 es:

$$V_{OUT} = 1,25V \cdot \left(1 + \frac{R_2}{R_1} \right) \quad (6.2)$$

Teniendo en cuenta los voltajes de alimentación requeridos para las dos etapas de alimentación de la PCB ($\sim 5,5V$ para la primera, $\sim 3,3V$ para la segunda), se han realizado los siguientes cálculos para los valores de la resistencia R_1 y R_2 :

- Primera etapa, alimentación de servomotores a $5,5V$ requeridos:

$$5,5V = 1,25V \cdot \left(1 + \frac{R_2}{R_1} \right) \quad (6.3)$$

Por disponibilidad de materiales, se ha decidido escoger los valores 150Ω y 500Ω para las resistencias R_1 y R_2 respectivamente. Tomando en cuenta dichos valores y utilizando la ecuación anterior, se obtiene una reducción de voltaje de $9V$ a $5,41V$, voltaje suficiente para alimentar los servomotores.

Aplicando el conexionado recomendado por el fabricante y los cálculos para el valor de las resistencias se obtiene finalmente el diagrama esquemático del circuito de alimentación de los servomotores (imagen 6.59):

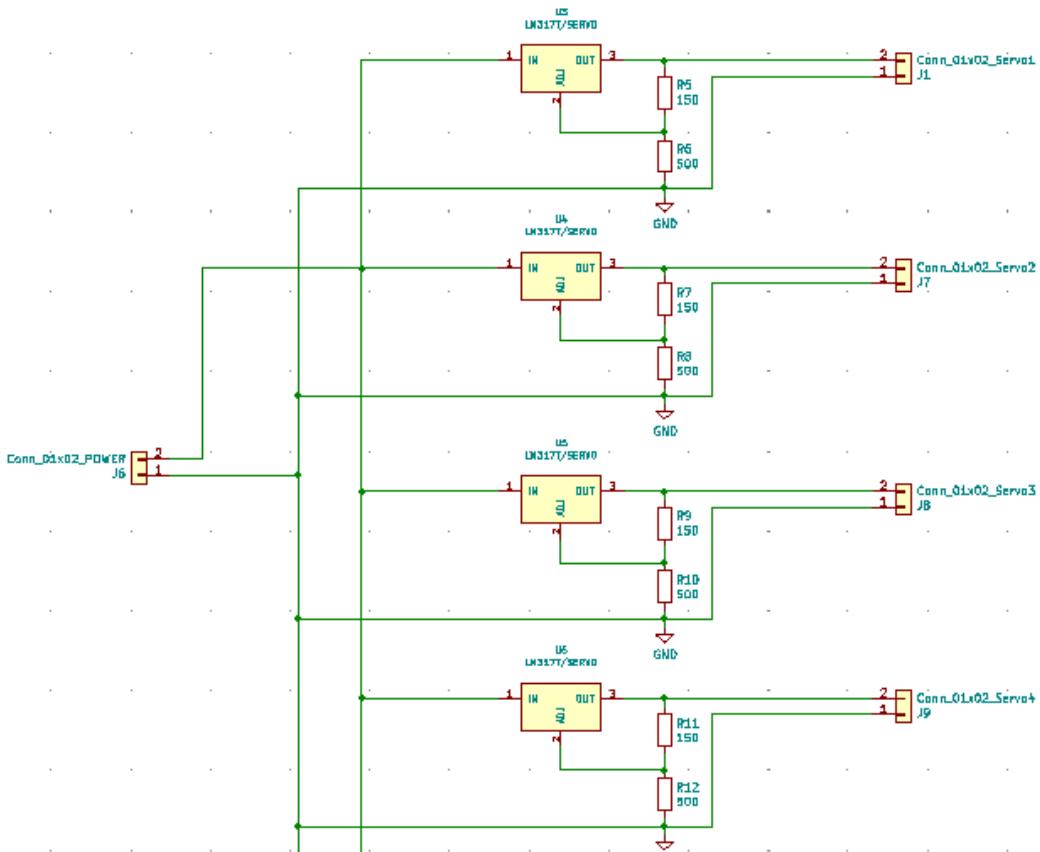


Figura 6.59: Diagrama esquemático del circuito de alimentación de los servomotores.

El funcionamiento de los reguladores L7805CV y AZ1117H es sencillo de comprender:

- El regulador L7805CV recibe un voltaje de entre $7V - 35V$ y provee un voltaje de salida fijo de $5V$. Su conexionado se realiza utilizando dos condensadores auxiliares, tal y como se describe en el *datasheet* del componente:

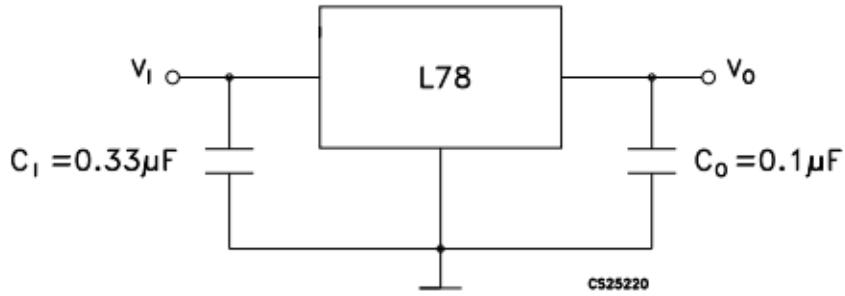


Figura 6.60: Diagrama de conexionado del regulador L7805 [36].

- El regulador AZ1117H recibe un voltaje de hasta 15V y provee un voltaje de salida fijo de 3,3V. Su conexionado se realiza de la misma forma que para el regulador anterior, tal y como se describe en el *datasheet* del componente:

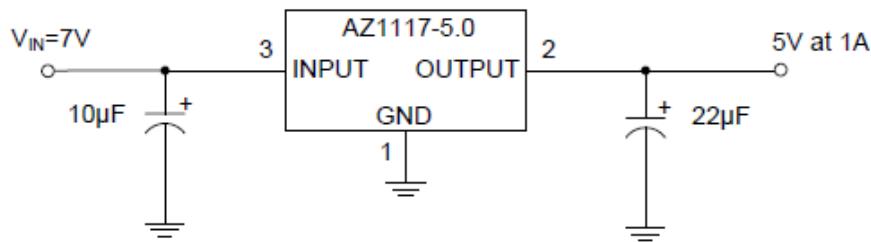


Figura 6.61: Diagrama de conexionado del regulador AZ1117H [37].

En el diagrama anterior (imagen 6.61) se utiliza el modelo que ofrece regulación de voltaje a 5V. Sin embargo, el modelo usado en el proyecto es el que ofrece regulación de voltaje a 3,3V, siendo su conexionado exactamente igual.

Teniendo en cuenta la información expuesta anteriormente, se ha decidido conectar en serie ambos reguladores, consiguiendo de esta forma una regulación de 9V a 5V y posteriormente una regulación de 5V a 3,3V. El diagrama esquemático final es el siguiente^{1,2} (ver imagen 6.62):

¹No se realiza un análisis más detallado de "bypassz desacoplo, porque no se presuponen escenarios comprometedores ni en la alimentación del sistema ni en las cargas del mismo

²Todos los reguladores trabajan lejos de sus márgenes de potencia máxima por lo que no es necesario instalar disipadores térmicos para evacuar calor

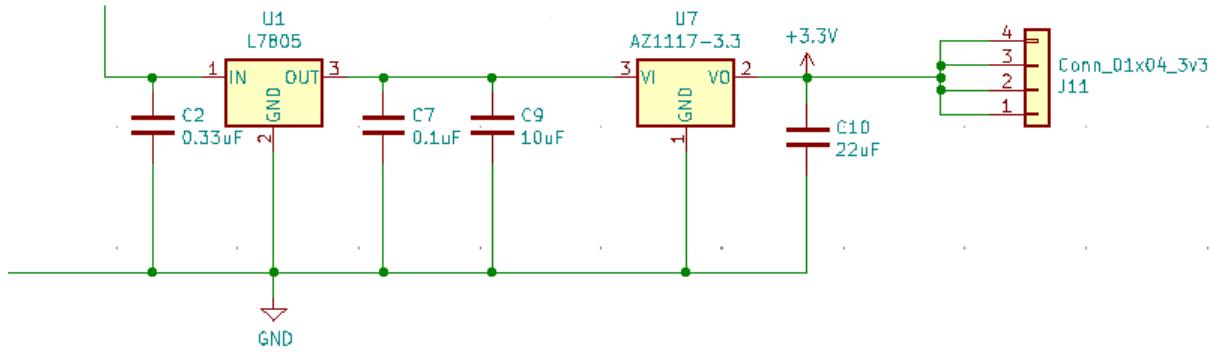


Figura 6.62: Diagrama esquemático de la etapa de alimentación del microcontrolador.

Es importante destacar tres aspectos:

- La primera etapa de alimentación incluye clemas para su conexiónado con los servomotores.
- La segunda etapa de alimentación incluye un puerto de cuatro pines, los cuales se usan para alimentar los micro-interruptores y el microcontrolador.
- El conexionado de todos los reguladores se ha realizado en paralelo, dedicando un regulador para cada servomotor así como para el microcontrolador. El objetivo de esta configuración es garantizar una vía de alimentación relativamente independientes para cada componente, reduciendo las interferencias de alimentación entre los reguladores y diferenciando la etapa de alimentación de los servomotores y microcontrolador.

En segundo lugar se procede a describir detalladamente el diagrama esquemático del microcontrolador y sus periféricos. A continuación se muestra el diagrama esquemático final y posteriormente se detallará cada uno de los periféricos:

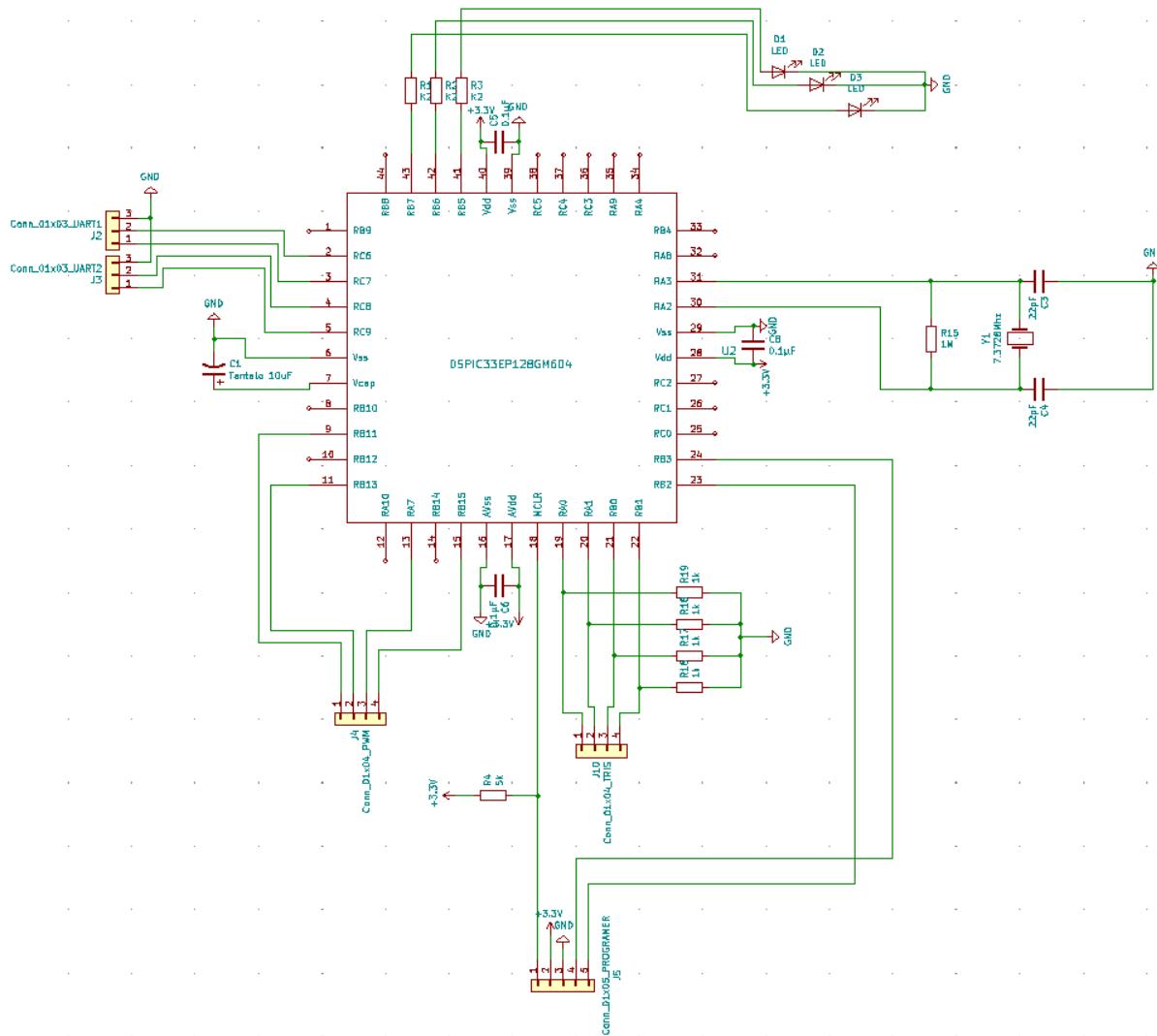


Figura 6.63: Diagrama esquemático del microcontrolador y sus periféricos.

Primeramente, es importante describir los condensadores auxiliares recomendables para el correcto funcionamiento del microcontrolador, los cuales se encuentran conectados en los pines de alimentación del mismo. Su conexionado es sugerido por el fabricante en el *datasheet* según el siguiente esquema (ver imagen 6.64):

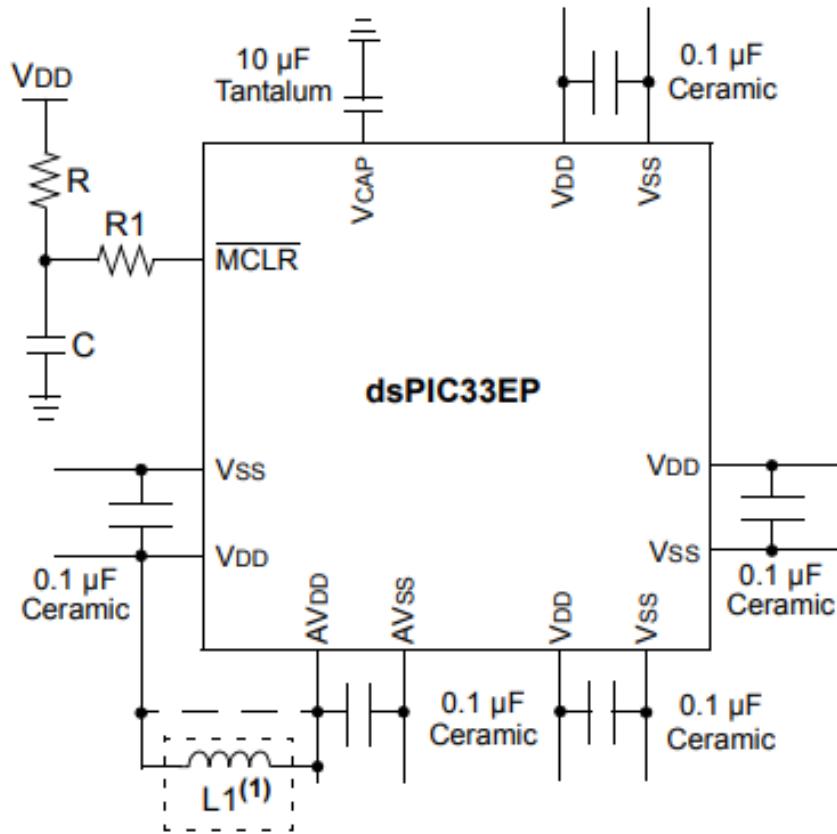


Figura 6.64: Conexionado mínimo del microcontrolador [38].

Todos los condensadores han sido conectados a los pines descritos por el fabricante y escogidos teniendo en cuenta las características técnicas de los mismos, también descritas en el *datasheet*.

A continuación, se describe el conexionado del resto de puertos y periféricos:

- Cristal de cuarzo, en términos técnicos, oscila a $7,3728MHz$, frecuencia perfectamente válida para el microcontrolador usado en el proyecto:

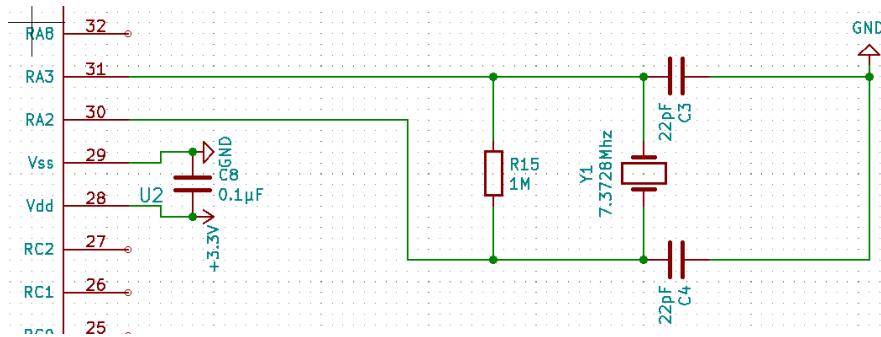


Figura 6.65: Diagrama esquemático del conexionado del generador de señales.

Su conexionado se realiza con los pines 32 y 31 del microcontrolador, siguiendo la estructura de la imagen anterior (ver imagen 6.65), empleando también una resistencia de

$1M\Omega$ y dos condensadores de $22pF$.

- Puerto de programación mediante sonda, el cual debe tener una estructura específica y se describe en el *datasheet* de la misma (ver imagen 6.66):

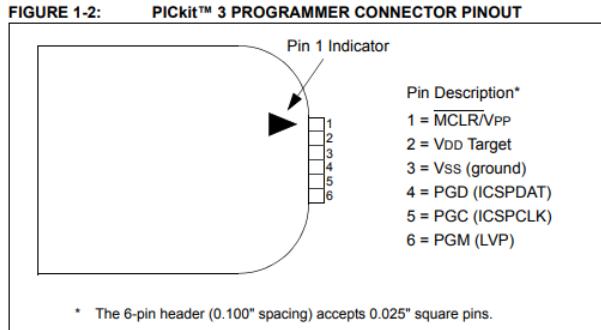
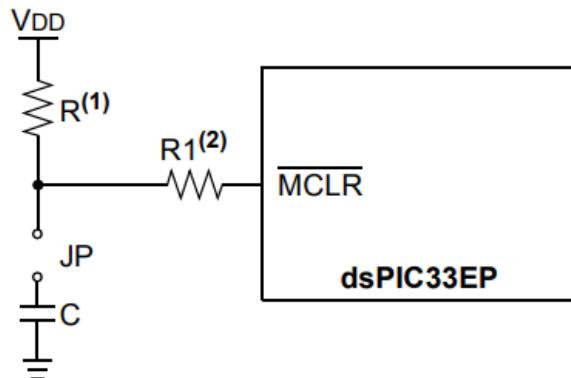


Figura 6.66: *Pinout* del conector de la sonda de programación [38].

Cabe destacar que el pin 18 o \overline{MCLR}/V_{PP} debe tener un conexionado específico en el que se emplea una resistencia *pull-up*; esta estructura de conexión se muestra en el *datasheet* del microcontrolador (ver imagen 6.67):



- Note 1:** $R \leq 10 \text{ k}\Omega$ is recommended. A suggested starting value is $10 \text{ k}\Omega$. Ensure that the MCLR pin V_{IH} and V_{IL} specifications are met.
- 2:** $R1 \leq 470\Omega$ will limit any current flowing into MCLR from the external capacitor, C, in the event of MCLR pin breakdown due to Electrostatic Discharge (ESD) or Electrical Overstress (EOS). Ensure that the MCLR pin V_{IH} and V_{IL} specifications are met.

Figura 6.67: Conexión del pin \overline{MCLR}/V_{PP} [38].

En este proyecto se ha decidido no incluir el *jumper* sugerido para conexión del pin $MCLR/V_{PP}$ y por lo tanto R_1 no se añade en el diagrama esquemático.

Teniendo en cuenta lo anteriormente mencionado, el conexionado final del puerto de programación mediante sonda es el siguiente (ver imagen 6.68):

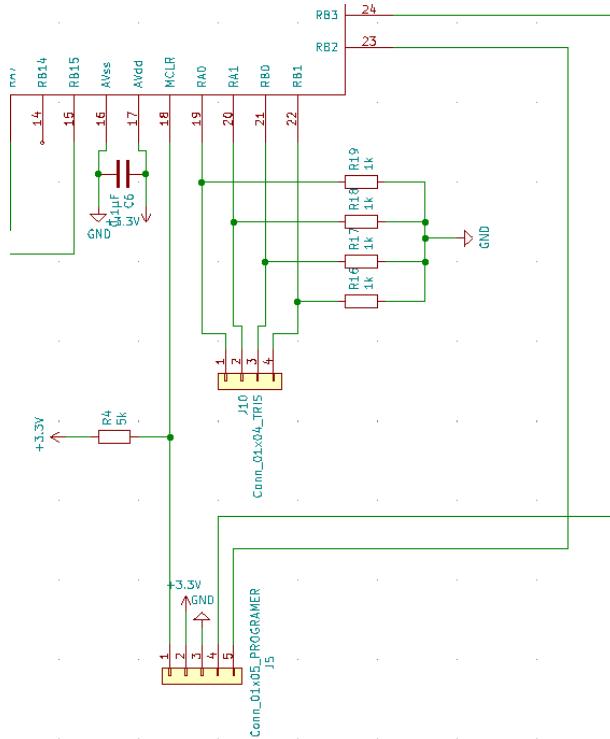


Figura 6.68: Diagrama esquemático del puerto de programación.

- Puerto PORT, usado para detectar si el brazo robótico se encuentra en uno de sus finales de carrera o zonas límite de movimiento. Mediante dicha señal y dado que estos micro-interruptores se encuentran conectados a los pines 19, 20, 21 y 22 del microcontrolador, se puede realizar la lectura de los mismos y detectar el estado de micro-interruptor. De esta manera, se consigue saber cuándo el manipulador ha alcanzado o no un final de carrera.

A continuación se muestra un esquema de lo mencionado anteriormente:

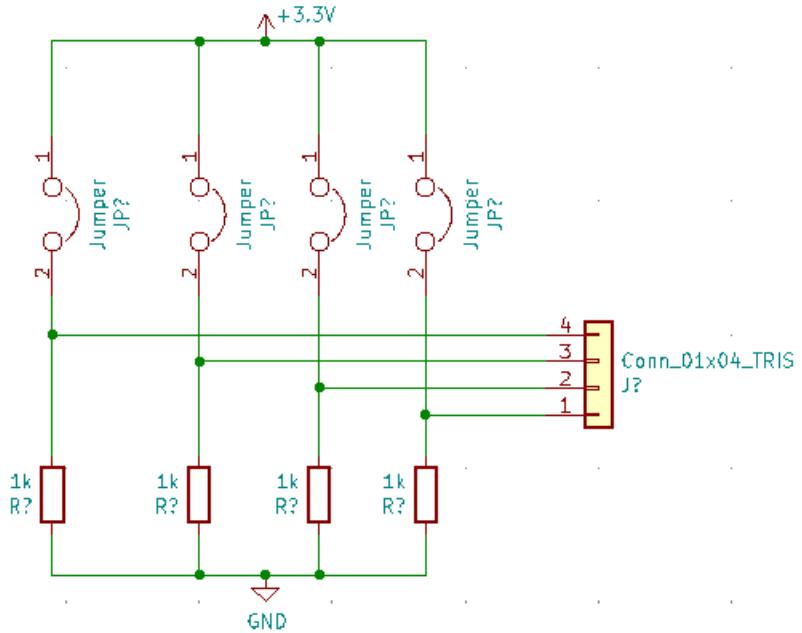


Figura 6.69: Circuito lógico para los finales de carrera.

Mediante el conexionado anterior (ver imagen 6.69), si el micro-interruptor está abierto se recibe un nivel bajo por el pin del microcontrolador, mientras que si el micro-interruptor está cerrado se recibe un nivel alto por el pin del microcontrolador.

Cabe destacar que tanto la resistencia *pull-down* como la conexión a tierra se incluyen en la PCB. Sin embargo, los micro-interruptores y su conexión a V_{DD} se encuentran localizados en la estructura del brazo robótico.

El diagrama esquemático que implementa esta funcionalidad es el siguiente (imagen 6.70):

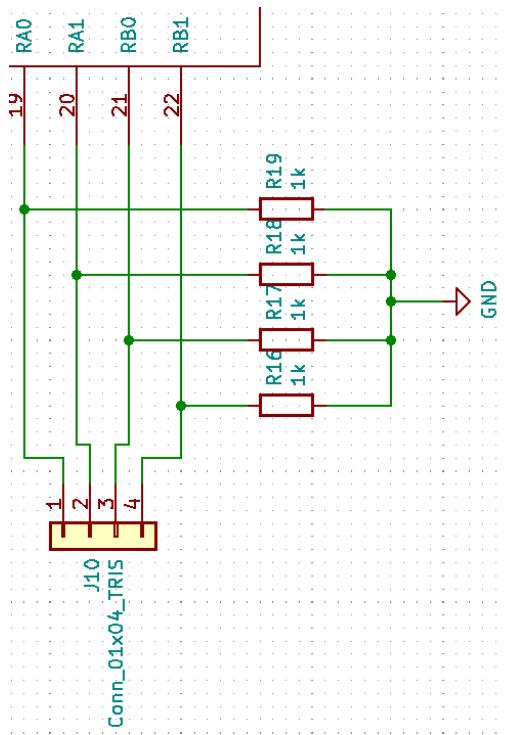


Figura 6.70: Diagrama esquemático del puerto TRIS.

- Puerto PWM, para el control de servomotores controlando así la posición angular de los mismos.

Los generadores de señal PWM de este microcontrolador poseen las siguientes características técnicas relevantes:

- El microcontrolador ofrece 6 generadores de señal PWM, cada uno de los cuales puede generar dos señales PWM.
 - Cada uno de los generadores posee un registro de 16 bits para la selección de la duración del ciclo de trabajo.

El esquema mostrado por el fabricante para los generadores PWM es el siguiente (ver imagen 6.71):

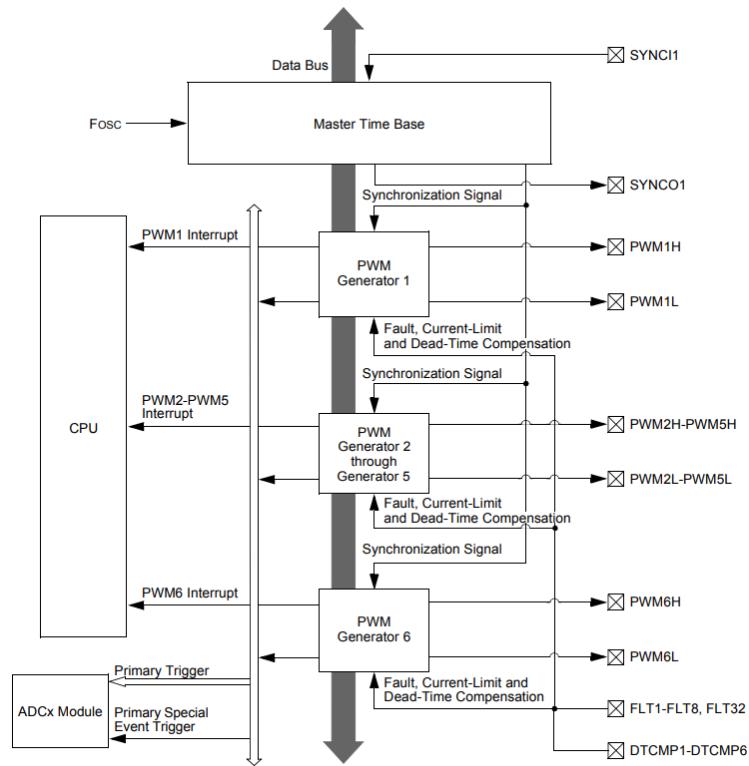


Figura 6.71: Esquema del generador PWM [38].

Mediante la información obtenida del *pinout* del microcontrolador, se tiene que los generadores de PWM 1, 4, 2 y 3 tienen asignados los pines 15, 13, 11 y 9, respectivamente.

El conexionado final del puerto PWM en el diagrama esquemático es el siguiente (imagen 6.72):

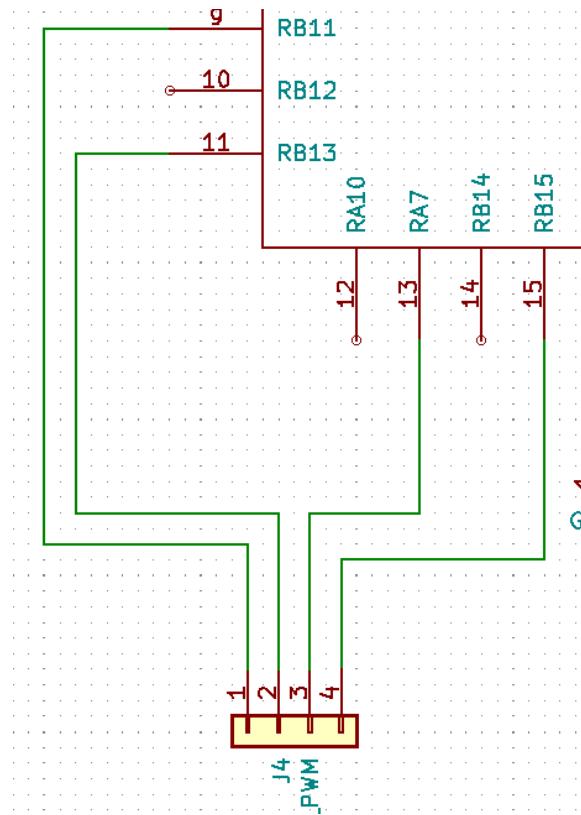


Figura 6.72: Diagrama esquemático del puerto PWM.

- Puertos UART, donde se establece un canal de comunicación *hardware* asíncrono en el cual existen diversas configuraciones en cuanto a formato de transmisión de bits y velocidades de comunicación. Suele ser un método de comunicación muy usado en microcontroladores y dispositivos *hardware* en general. Cada UART utiliza un puerto con tres conexiones: emisor (T_X), receptor (R_X) y tierra (GND).

El esquema simplificado de este periférico en el *datasheet* es el siguiente (imagen 6.73):

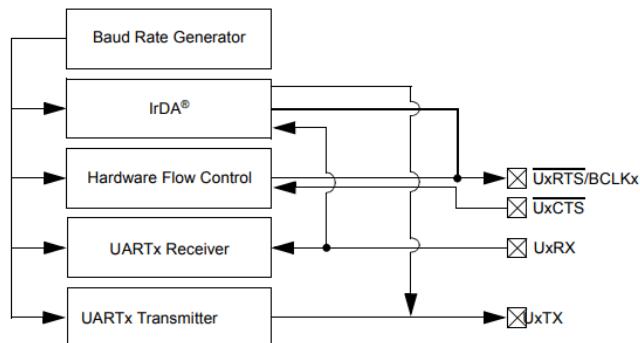


Figura 6.73: Esquema del periférico UART [38].

Se ha tomado la decisión de incluir dos puertos UART independientes en la PCB que se ha desarrollado, con el objetivo de dedicar uno de los canales a envío y recepción de instrucciones, mientras que el otro se usa para realizar labores de depuración y pruebas.

El conexionado de los puertos UART se realiza mediante pines reconfigurables del microcontrolador, en este caso se han utilizado los pines 2 y 3 para el primer canal, además de los pines 3 y 4 para el segundo canal. El diagrama esquemático final es el siguiente (imagen 6.74):

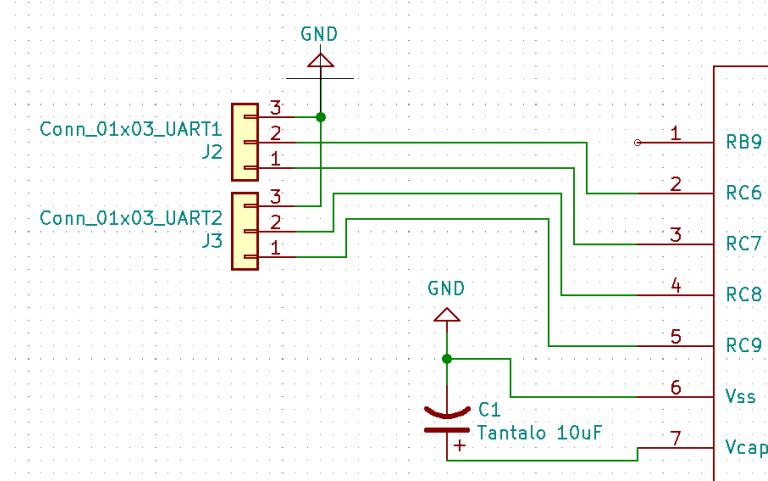


Figura 6.74: Diagrama esquemático de los puertos UART.

- LEDs de estado que muestran el estado del brazo robótico y demás aspectos del sistema. Su conexionado es realizado utilizando los pinos reconfigurables 41, 42 y 43, los cuales pueden ser usados para habilitar una salida digital. Se evalúa un nivel alto para encender el LED mientras que un nivel bajo para apagarlo.

A continuación se muestra el diagrama esquemático del circuito (imagen 6.75):

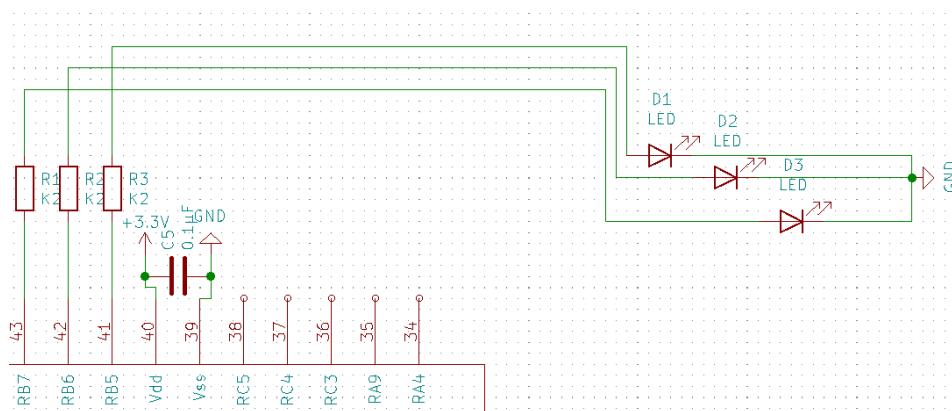


Figura 6.75: Diagrama esquemático de los LEDs.

Teniendo en cuenta que la corriente máxima suministrada por el microcontrolador es de $18mA$ para salida digital y el que voltaje de funcionamiento es de $3,3V$, el pin suministraría como mucho una potencia de $0,0594W$. Se ha decidido que una potencia adecuada a suministrar sería un 80% de la máxima, es decir $0,0475W$. Para cumplir esta restricción, se ha calculado el valor ideal de la resistencia del esquema anterior y su valor recomendado es de entre 180Ω y 200Ω .

En conclusión, una vista completa sobre el diagrama esquemático del proyecto es la siguiente (ver imagen 6.76):

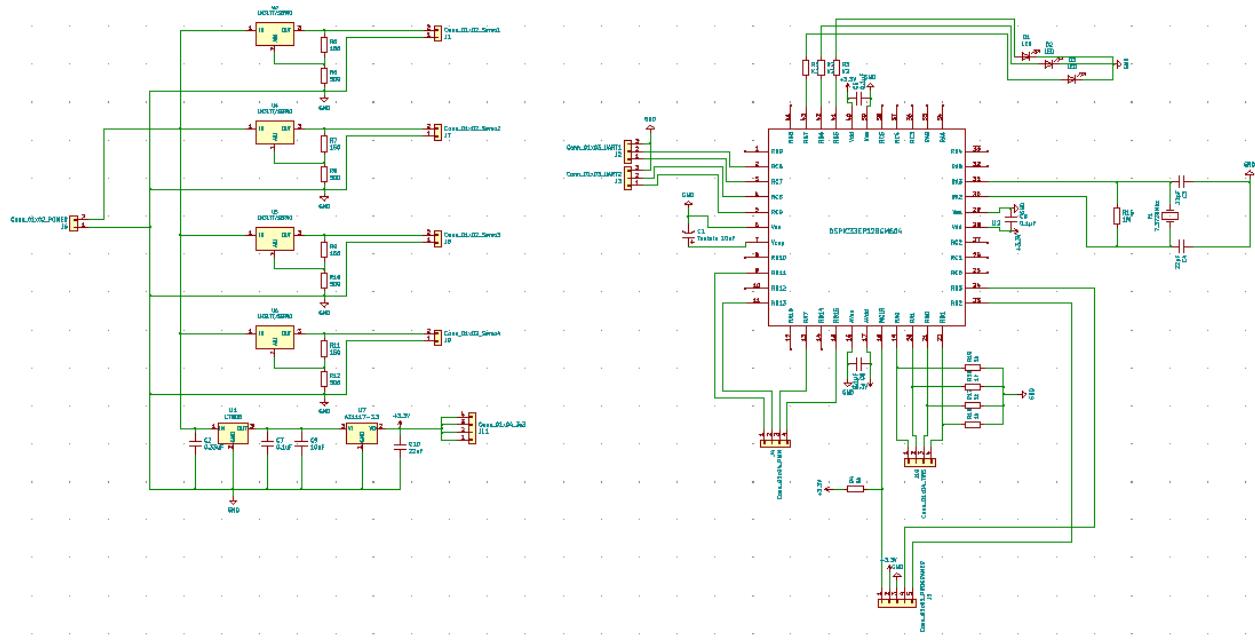


Figura 6.76: Diagrama esquemático completo.

6.5.4. Conversión del diagrama esquemático a diagrama físico

Tal y como se ha descrito en el apartado anterior, el diseño lógico de la PCB se implementa mediante el diagrama esquemático, y su objetivo es el de describir las huellas y conexiones lógicas de los componentes; sin embargo, este diseño es de alto nivel y no es implementable directamente en términos físicos.

El siguiente paso tras completar el diagrama esquemático es transformar este diseño lógico en un diseño físico más cercano a la implementación real.

El diseño físico de una PCB debe ser obtenido directamente de la información establecida en el diseño lógico y, por lo tanto, se tiene que transformar el diagrama esquemático en un diagrama físico, en el cual se deben contemplar los aspectos físicos de los componentes y sus conexiones, además de sus aspectos lógicos.

El objetivo principal del diseño y diagrama físico es el de plasmar la realidad física de los componentes y sus conexiones a partir de un diagrama esquemático, en el cual no se contemplan los aspectos físicos para simplificar el diseño inicial. En general, el diseño y diagrama físico es más complejo y difícil de comprender. Por ello, normalmente la primera etapa del diseño comienza con el diseño lógico y diagrama esquemático.

En general, el proceso que se debe llevar a cabo para obtener el diagrama físico a partir de un diagrama esquemático consta de varios pasos:

- Asignación de huellas físicas a cada uno de los componentes lógicos.

- Generar un listado de redes en el cual se especifiquen las conexiones que existen entre todos los componentes.
- Importar ambos elementos anteriores a la herramienta de creación del diagrama físico y comenzar el diseño.

Tal y como se ha mencionado anteriormente, en primer lugar se debe asignar una huella física a cada uno de los componentes lógicos del diagrama esquemático. Este proceso se realiza en la herramienta “*Schematic Layout Editor*” incluida en KiCad, utilizando la opción de “asignar huellas a símbolos del sistema” disponible en la barra de herramientas situada en la parte superior de la ventana:

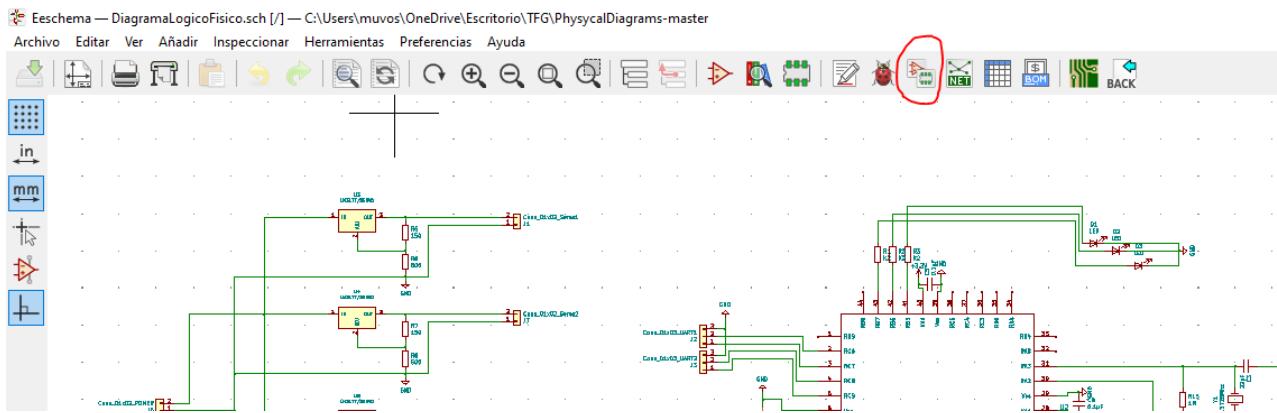


Figura 6.77: Herramienta de asignación de huellas.

Accediendo al menú de dicha herramienta (ver imagen 6.77), se encuentra una lista de los componentes del diagrama esquemático a los cuales se les debe asignar una huella física. Las huellas físicas que se deben asignar a cada componente lógico pueden ser seleccionadas de las extensa librerías que ofrece KiCad, o bien, ser diseñada y personalizada por el usuario.

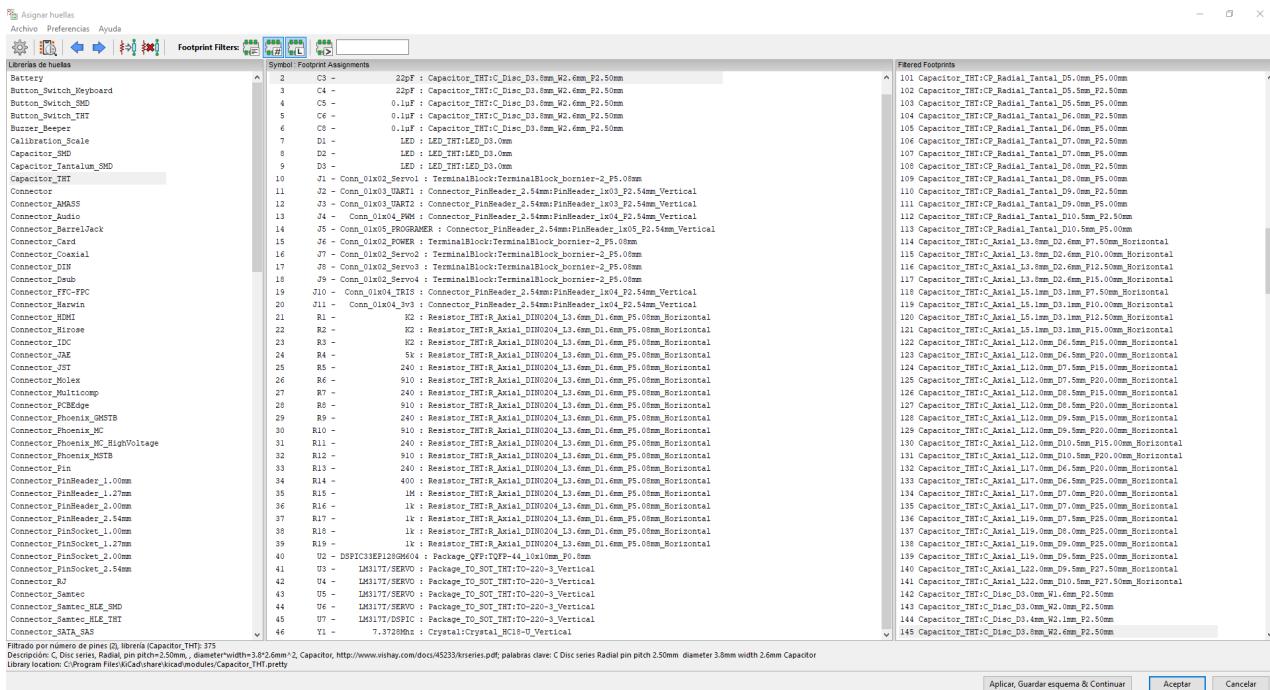


Figura 6.78: Ventana de asignación de huellas físicas.

En la imagen anterior (imagen 6.78) se muestra la ventana de asignación de huellas físicas, la cual está dividida en tres secciones:

- La sección izquierda muestra las librerías de componentes de KiCad.
- La sección central muestra los componentes lógicos del diagrama esquemático, seguidos de la huella física que tienen asignados.
- La sección derecha muestra las huellas físicas que cumplen los filtros establecidos dependiendo del componente lógico. Estos filtros suelen ser: nombre, número de pines y librería.

Como ejemplo, en la imagen anterior (imagen 6.78) se ha realizado una búsqueda dentro de la librería “capacitor THT”, ya que se quieren utilizar condensadores de agujero pasante en el diseño físico y, por lo tanto, en la sección derecha de la ventana se muestran las huellas susceptibles de ser usadas, filtradas por número de pines y librería.

Una vez se ha elegido la huella que se quiere utilizar en el diseño físico, esta se asigna al componente esquemático y puede ser visualizada (imagen 6.79):



Figura 6.79: Huella física de un condensador usado en la PCB.

Existen numerosos aspectos que afectan a la decisión de qué huella física asignarle a cada componente lógico y principalmente depende del tipo de implementación que se vaya a realizar en la PCB. Normalmente, estas decisiones se deben tomar a través de la información técnica suministrada por el fabricante en los diversos *datasheets*.

Un ejemplo a destacar de lo recién mencionado es la huella que se ha asignado al microcontrolador dsPIC utilizado, el cual dispone de diversos encapsulados. Dichos encapsulados se muestran de forma detallada en el *datasheet*.

En este proyecto, se ha decidido utilizar el encapsulado de 44 pines de soldadura superficial y de tipo “*Thin Quad Flat Package*” (TQFP), cuya descripción en el *datasheet* es la siguiente (ver imagen 6.80):

Pin Diagrams

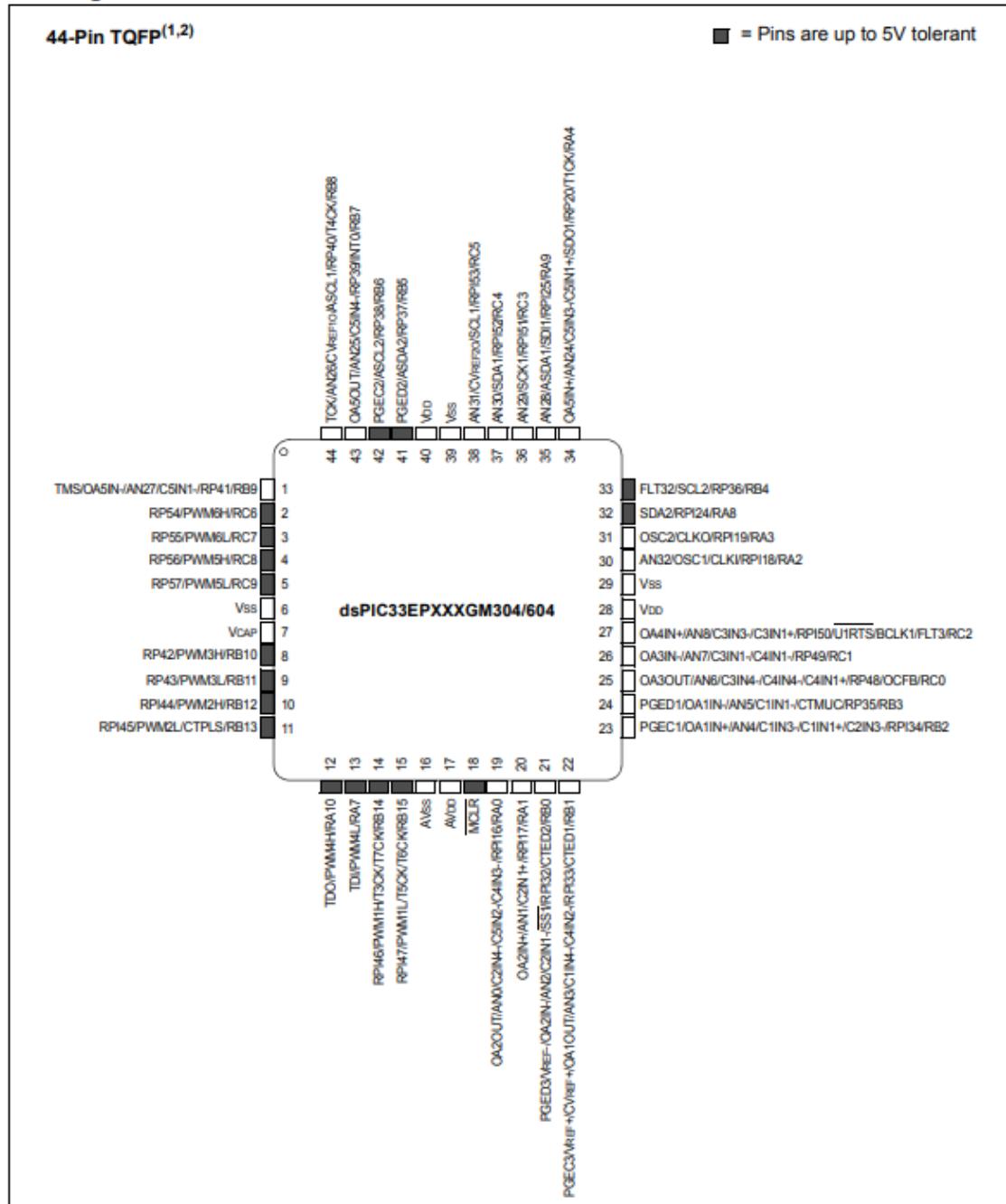


Figura 6.80: Encapsulado elegido para el microcontrolador [38].

La huella asignada al componente lógico es la siguiente (ver imagen 6.81) y se encuentra en la librería QFP incluida por defecto en KiCad:

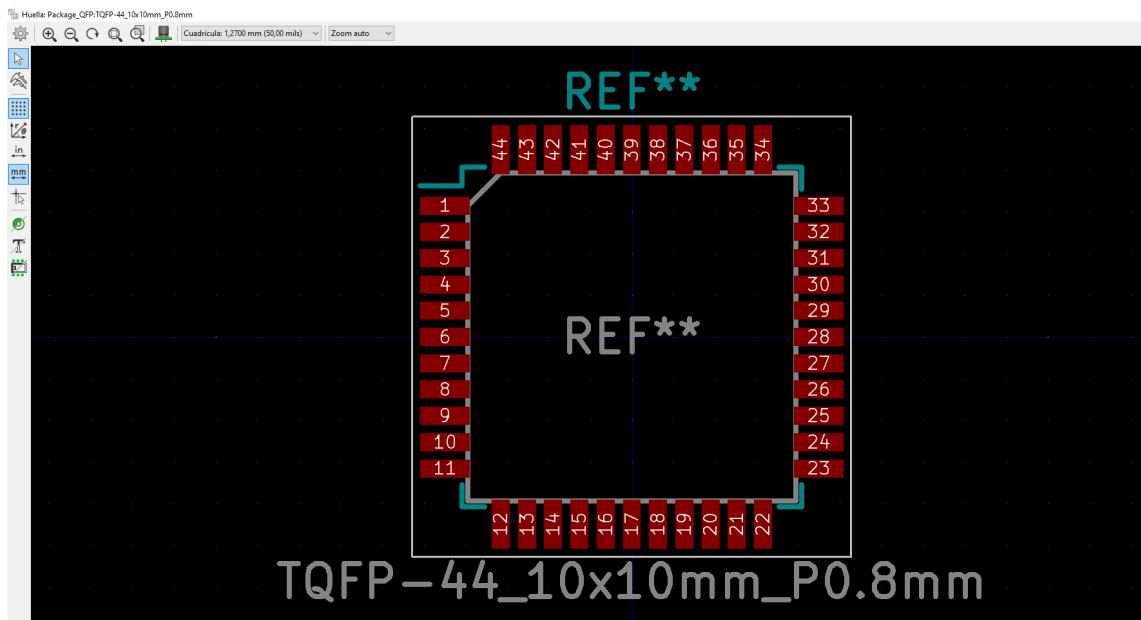


Figura 6.81: Huella física asignada al microcontrolador.

Cabe destacar que muchas de las huellas pueden ser usadas para distintos componentes de distintos fabricantes, ya que los encapsulados están estandarizados. Sin embargo, para evitar errores, siempre se deben realizar comprobaciones con respecto a las dimensiones, número de pines, etc.

Una vez se ha realizado el primer paso, se debe generar un listado de redes de los componentes lógicos usados, sus huellas físicas asignadas y las conexiones existentes entre todos ellos. Este listado de redes se genera usando la herramienta “Generar listado de redes” disponible en la barra de herramientas de la parte superior de la ventana:

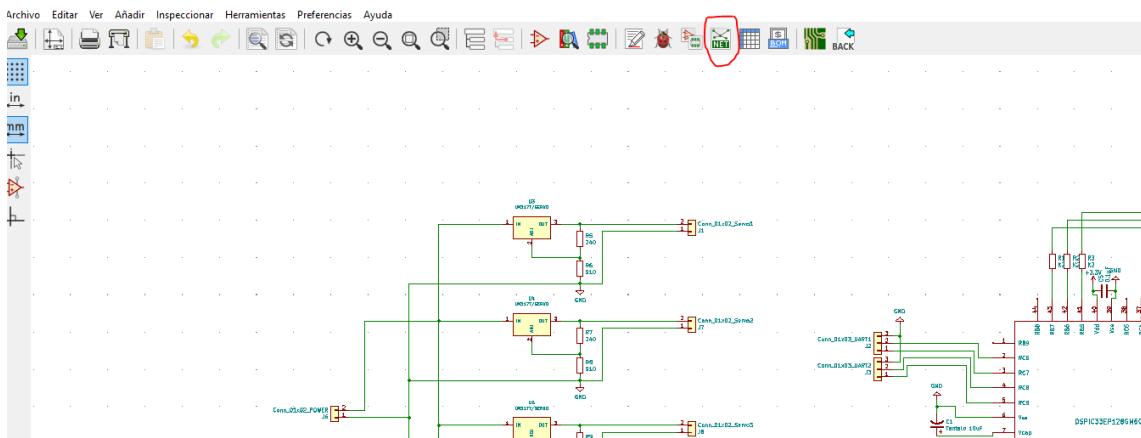


Figura 6.82: Herramienta de generado de listado de redes.

Utilizando la herramienta señalizada anteriormente (ver imagen 6.82) se puede generar un archivo con extensión “.net” que almacena el listado de componentes y conexiones del diagrama esquemático. Su contenido es de la siguiente forma (ver imagen 6.83):

```
(components
  (comp (ref R15)
    (value 1M)
    (footprint Resistor_THT:R_Axial_DIN0204_L3.6mm_D1.6mm_P5.08mm_Horizontal)
    (datasheet ~)
    (libsource (lib Device) (part R) (description Resistor))
    (sheetpath (names /) (tstamps /))
    (tstamp 5EA0CFA9))
  (comp (ref R1)
    (value K2)
    (footprint Resistor_THT:R_Axial_DIN0204_L3.6mm_D1.6mm_P5.08mm_Horizontal)
    (datasheet ~)
    (libsource (lib Device) (part R) (description Resistor))
    (sheetpath (names /) (tstamps /))
    (tstamp 5E722A02))
  (comp (ref C1)
    (value "Tantalum 10uF")
    (footprint Capacitor_THT:C_Disc_D3.8mm_W2.6mm_P2.50mm)
    (datasheet ~)
    (libsource (lib Device) (part CP1) (description "Polarized capacitor, US symbol"))
    (sheetpath (names /) (tstamps /))
    (tstamp 5E88CB39)))

```

Figura 6.83: Archivo de listado de redes.

Como último paso necesario para poder transformar el diagrama esquemático a diagrama físico se debe importar el listado de redes generado en el paso anterior a la herramienta de diseño físico. En este caso, la herramienta elegida ha sido “PCBnew” y se encuentra incluida dentro de KiCad. Esta herramienta se puede ejecutar usando la barra de herramientas de la parte superior (ver imagen 6.84):

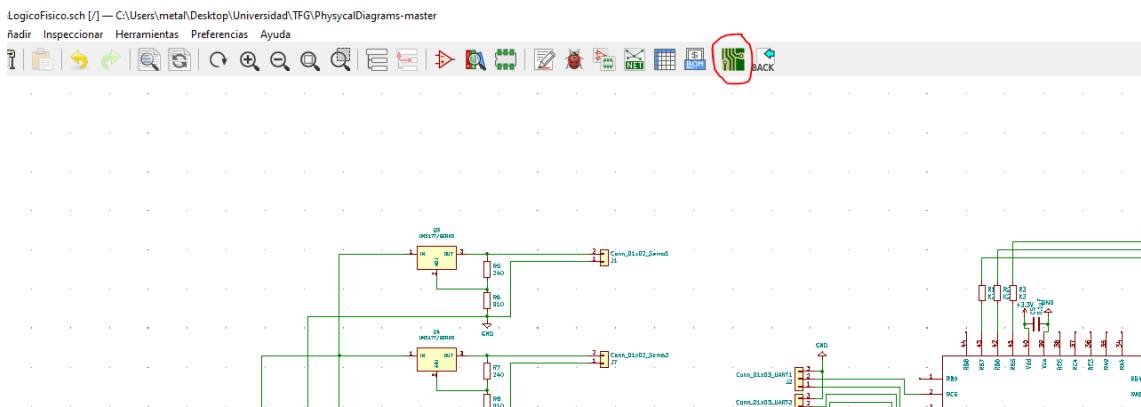


Figura 6.84: Acceso directo a la herramienta “PCBnew”.

Una vez se ha accedido a “PCBnew”, se debe importar el listado de redes generado anteriormente, usando el ícono designado para ello en la barra de herramientas superior (ver imagen 6.85):

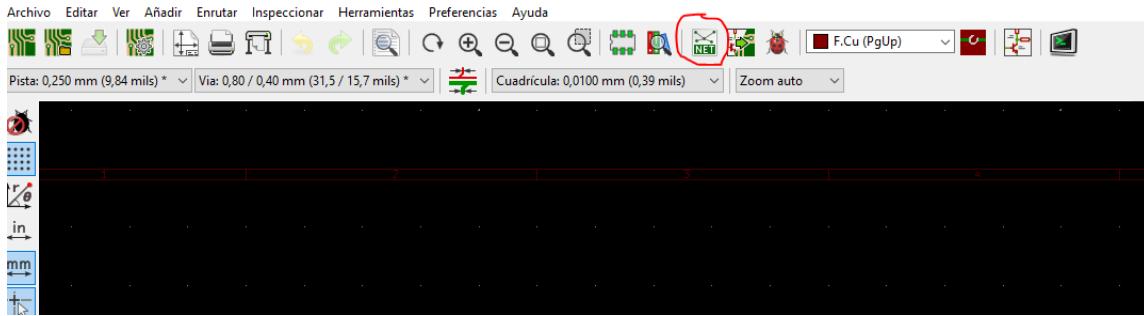


Figura 6.85: Herramienta de importado de listado de redes.

Para importar el listado de redes generado anteriormente, basta con buscar la ubicación del archivo, seleccionarlo y hacer clic en “actualizar PCB”. Al hacer esto, todos los componentes del diagrama esquemático son importados a “PCBnew”, y su apariencia es la huella física asignada anteriormente:

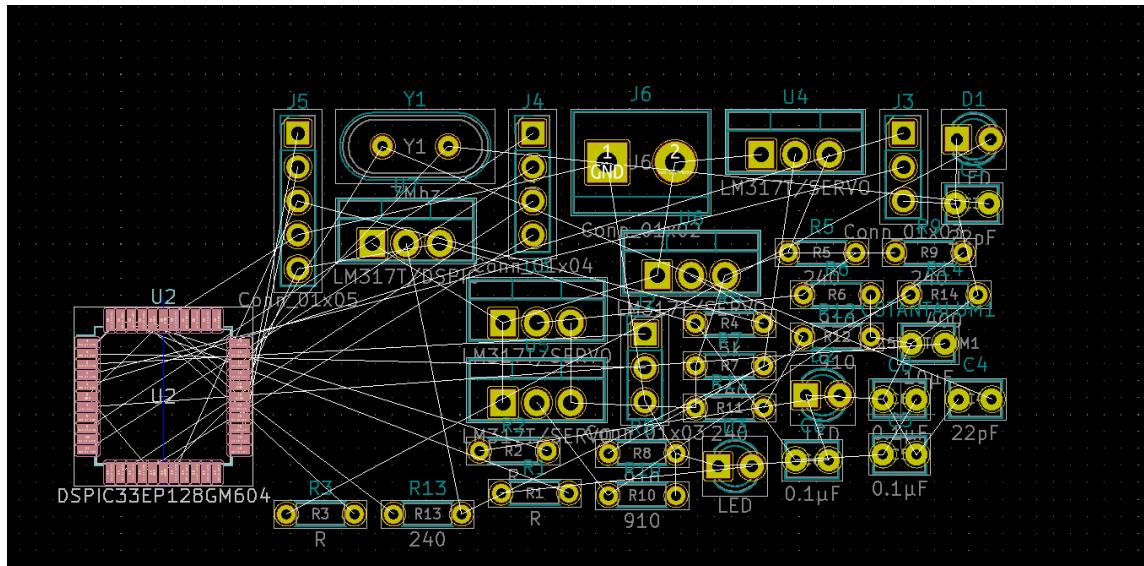


Figura 6.86: Situación inicial del diseño nada mas importar los componentes físicos.

Inicialmente, se muestra la huella física de los componentes y sus conexiones lógica pero se encuentran desordenados. Es recomendable reorganizarlos para verificar si todos ellos han sido importados correctamente.

El primer paso para comenzar el diseño físico de la PCB es distribuir los componentes por el plano, tratando de visualizar cuál va a ser la ubicación futura de los componentes en la placa de circuito impreso y cuál van a ser las dimensiones de la misma.

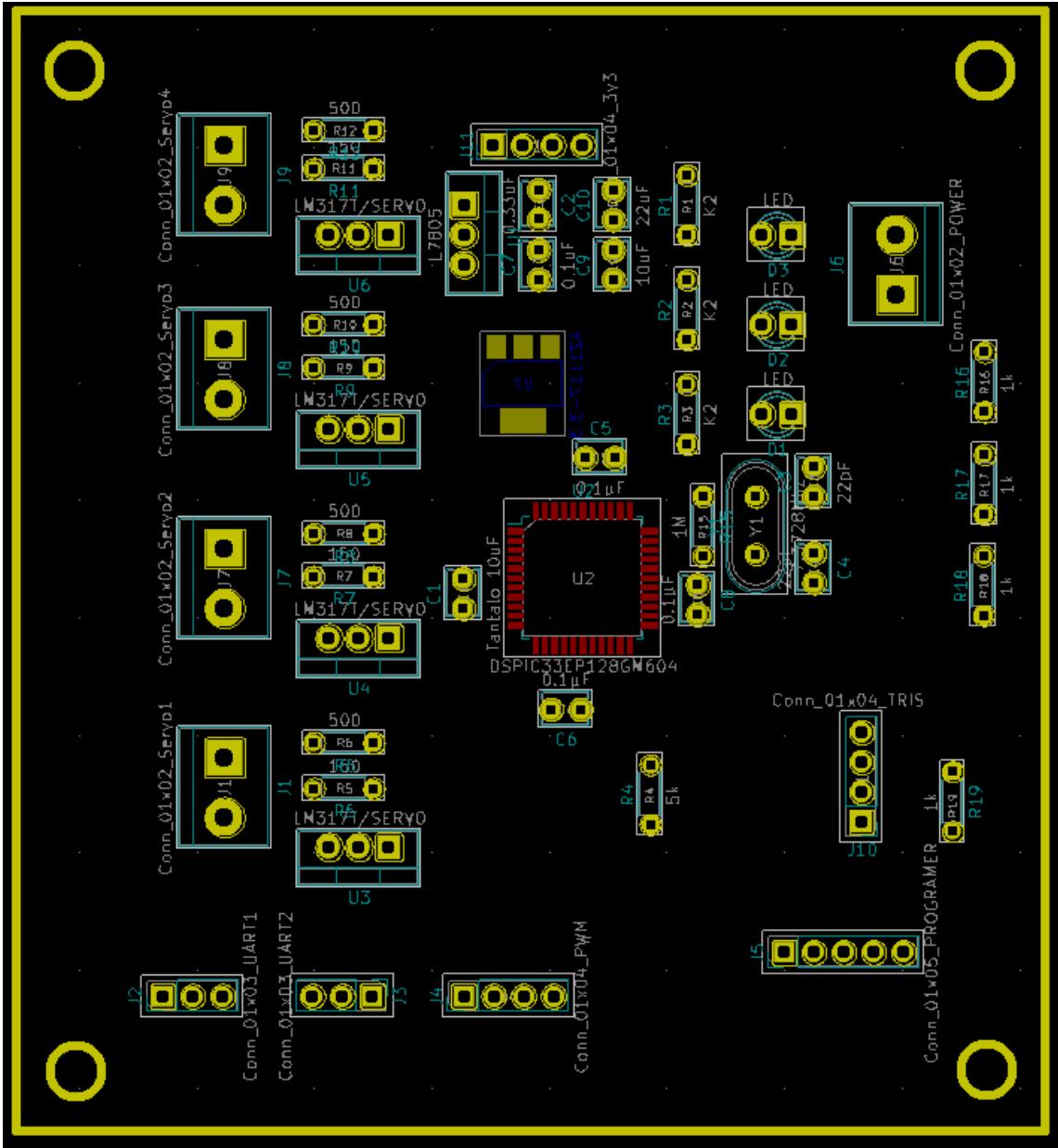


Figura 6.87: Distribución inicial de los componentes.

Es conveniente definir un contorno a la placa, el cual puede ser incluido en el diagrama usando la herramienta de dibujado de líneas y la herramienta de medición de distancias.

Por último, es importante remarcar que la distribución de los componentes queda a elección del diseñador de la PCB. Sin embargo, se debe de tratar de tener en cuenta factores como el tamaño deseado para la PCB, la localización de los componentes con respecto a sus conexiones cercanas, el proceso de fabricación a usar, etc.

6.5.5. Conexionado de los componentes mediante pistas

Una vez se ha creado el contorno de la PCB y se han distribuido los componentes de la forma deseada, se deben realizar las conexiones físicas entre los componentes.

Dado que se trata de una placa de circuito impreso, las conexiones lógicas entre los componentes se corresponden con pistas de cobre en el diagrama físico.

El proceso de conexionado mediante pistas se denomina “enrutado” y puede ser realizado de manera automática o manual. La complejidad del proceso de enrutado puede ser más o menos elevada en función del numero de componentes, capas que se utilicen para pistas, ubicación de los componentes, dimensiones de la PCB, etc. Se considera que el proceso de enrutado ha sido completado con éxito cuando todas las conexiones lógicas han sido realizadas y no existen conflictos o choques entre las pistas, así como soldaduras no realizablemente.

Durante el proceso de enrutado de las pistas de una PCB es habitual combinar herramientas de enrutado automático con enrutado manual, dado que, normalmente, las herramientas de enrutado automático suelen encontrar soluciones exitosas al proceso de enrutado, sin embargo no suelen ser óptimas y pueden requerir alguna modificación manual por parte del diseñador.

Para este proyecto se ha decidido realizar un proceso de enrutado íntegramente manual, debido a que, a pesar de haber intentado utilizar la herramienta “FreeRouting” de enrutado automático, no se ha obtenido un resultado apropiado y era bastante complejo.

El proceso de enrutado se ha realizado utilizando ambas capas de la PCB, esto quiere decir que se han trazado pistas en la capa de soldadura y en la capa de componentes.

En relación con lo anterior, las dos capas mencionadas anteriormente cumplen una función específica:

- La capa superior o de componentes contiene mayoritariamente las pistas de comunicación y alimentación del microcontrolador, ya que este componente es de tipo “SMD”.
- La capa inferior o de soldadura contiene mayoritariamente el resto de pistas de alimentación de la PCB y las soldaduras de la mayoría de componentes.
- Para comunicar las pistas de ambas capas se han realizado vías de conexión en los casos necesarios. Un ejemplo de su uso está en el conexionado de los pines del microcontrolador en la capa superior con las pistas de los conectores, los cuales se encuentran soldados en la capa inferior de soldadura.

Otro de los aspectos claves a la hora de enrutar una PCB es escoger un ancho de pista adecuado. Algunos de los factores que afectan a esta decisión son los siguientes:

- Intensidad de corriente que conducirá la pista. Distinción entre pistas de alimentación y de comunicación de señales digitales.
- Requerimientos de tamaño de la PCB, ya sea por número de pistas, espacio disponible, tamaño de los pines de conexión de componentes, etc.
- Limitaciones físicas y de precisión del proceso de fabricación de la PCB.

- Otros factores como la resistividad del material de la pista, el aumento de temperatura máximo tolerado por la misma o su longitud aproximada, son factores determinantes para el ancho de la pista.

KiCad incluye una herramienta denominada “PCB Calculator” la cual permite realizar cálculos sobre diversos aspectos relacionados con la PCB, entre ellos, el ancho de pista. Esta herramienta realiza el cálculo del ancho de pistas en función de los factores mencionados anteriormente y su interfaz es la siguiente (imagen 6.88):

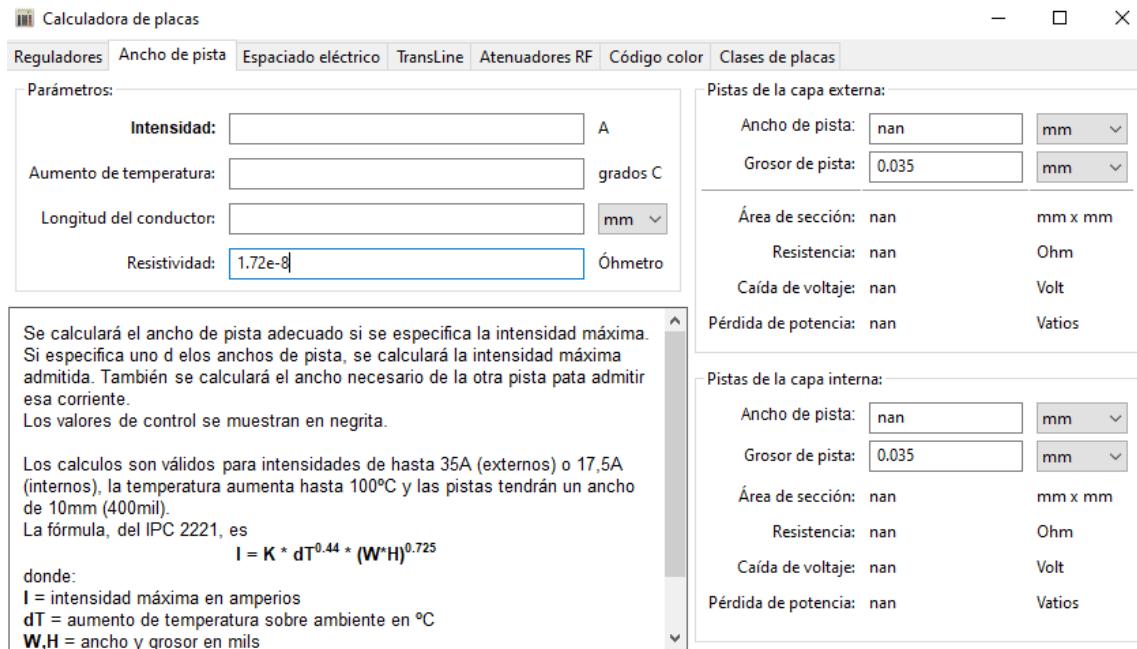


Figura 6.88: Ventana principal de “PCB Calculator”.

Mediante la herramienta anterior (imagen 6.88) se puede realizar el cálculo para los dos tipos de pistas usadas en esta PCB: pistas de alimentación y pistas de comunicación. Para el cálculo de ambos tipos, se asume un aumento de temperatura máximo de 10 °C, una longitud del conductor de 20cm y un valor de resistividad de $1,72 \cdot 10^{-8} \Omega/\text{metro}$.

En primer lugar, se considera que las pistas de alimentación de la PCB conducen 2A de intensidad y un voltaje de 9V máximo. Teniendo en cuenta dichos datos, el ancho de pista obtenido es el siguiente (imagen 6.89):

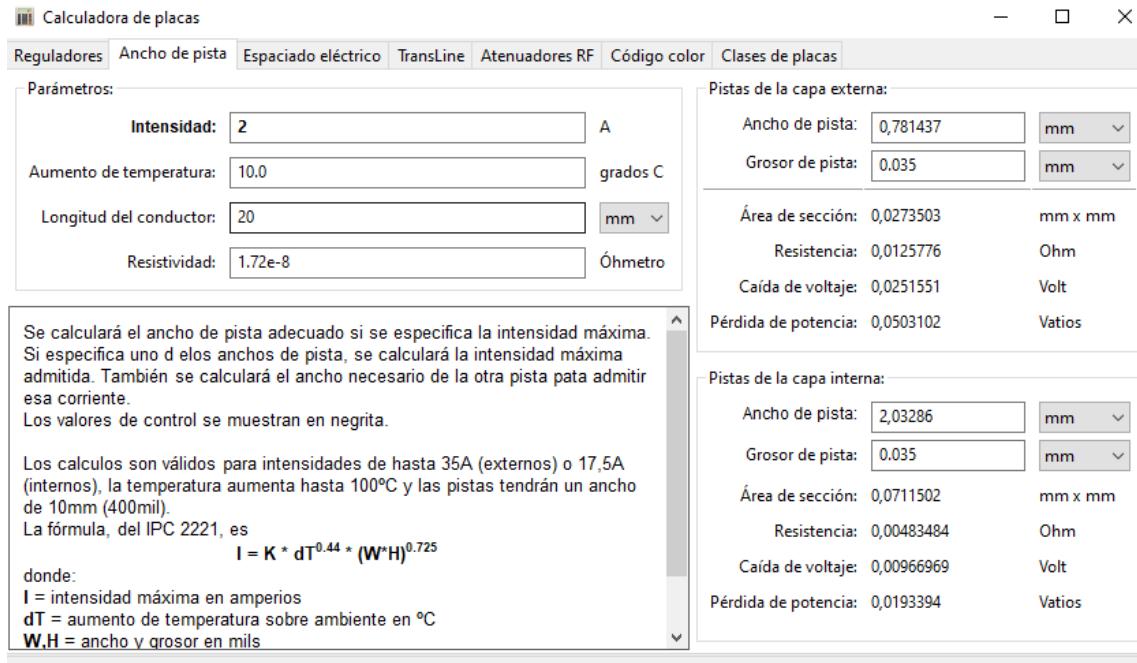


Figura 6.89: Cálculo del ancho de pistas de alimentación.

Se obtiene una ancho de pista de 0,78mm para las pistas de alimentación, por comodidad de approxima este valor de ancho a 0,8mm.

En segundo lugar, se considera que las pistas de comunicación del microcontrolador, conducen 0,25A y 3,3V máximo. Teniendo en cuenta dichos datos, el ancho de pista obtenido es el siguiente (imagen 6.90):

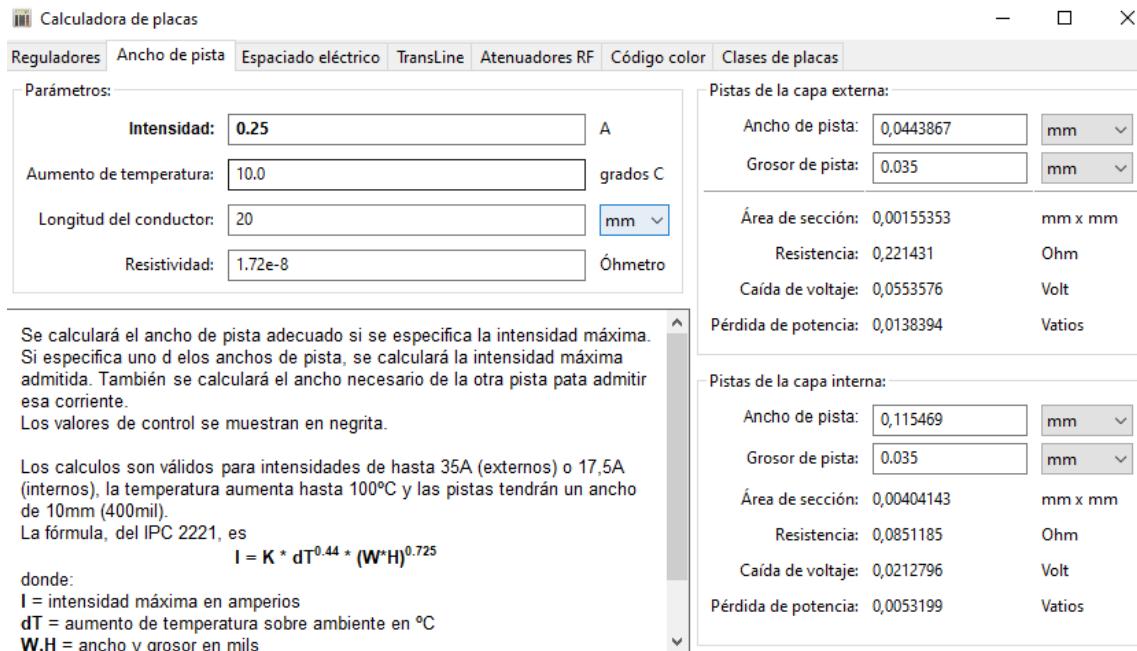


Figura 6.90: Cálculo del ancho de pistas de comunicación.

Se obtiene un ancho de pista de $0,044\text{mm}$ para las pistas de comunicación, sin embargo, se ha decidido no trazar pistas con un ancho menor a $0,4\text{mm}$, debido principalmente a que se pueden producir errores en el proceso de fabricación.

En este momento, cabe destacar que el proceso de fabricación llevado a cabo para construir la placa, es de carácter artesanal, no industrial.

Asumiendo un ancho mínimo de pista de $0,4\text{mm}$, las pistas de comunicación están sobredimensionadas por motivos justificados y por lo tanto se obtiene el siguiente cálculo (imagen 6.91) :

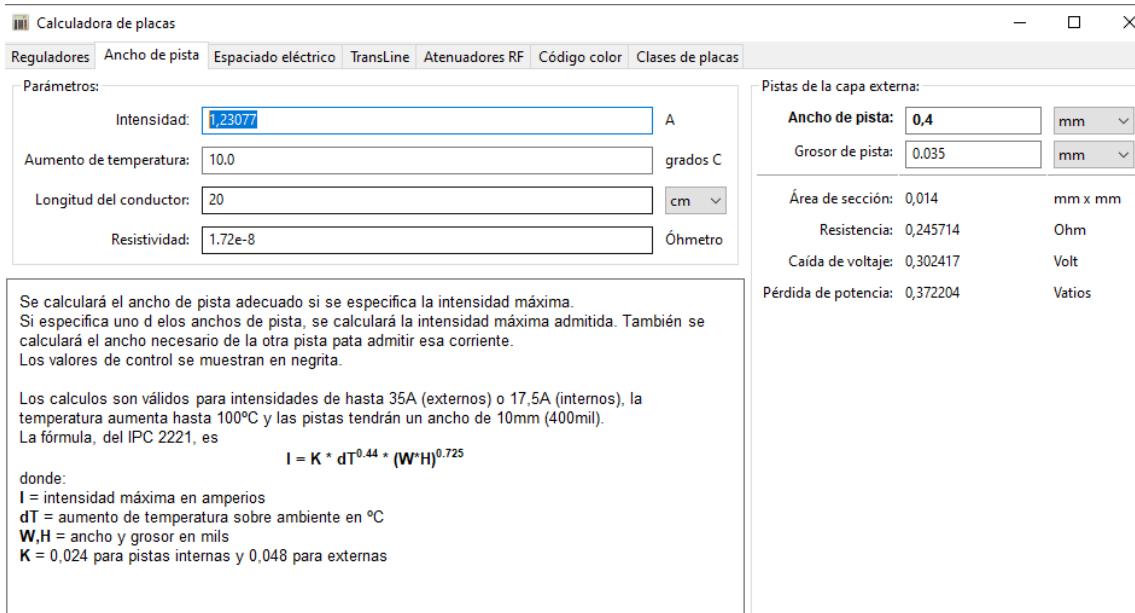


Figura 6.91: Cálculo inverso del ancho de pistas de comunicación.

Las pistas de comunicación tienen finalmente un ancho de $0,4\text{mm}$ y debido a su sobredimensionado, soportan una corriente de $1,23\text{A}$, la cual se sitúa muy por encima de la corriente que circulará por las mismas ($0,25\text{A}$).

A continuación se muestra la distribución final de los componentes físicos dentro del contorno de la PCB, aún sin haber trazado las pistas de conexión:

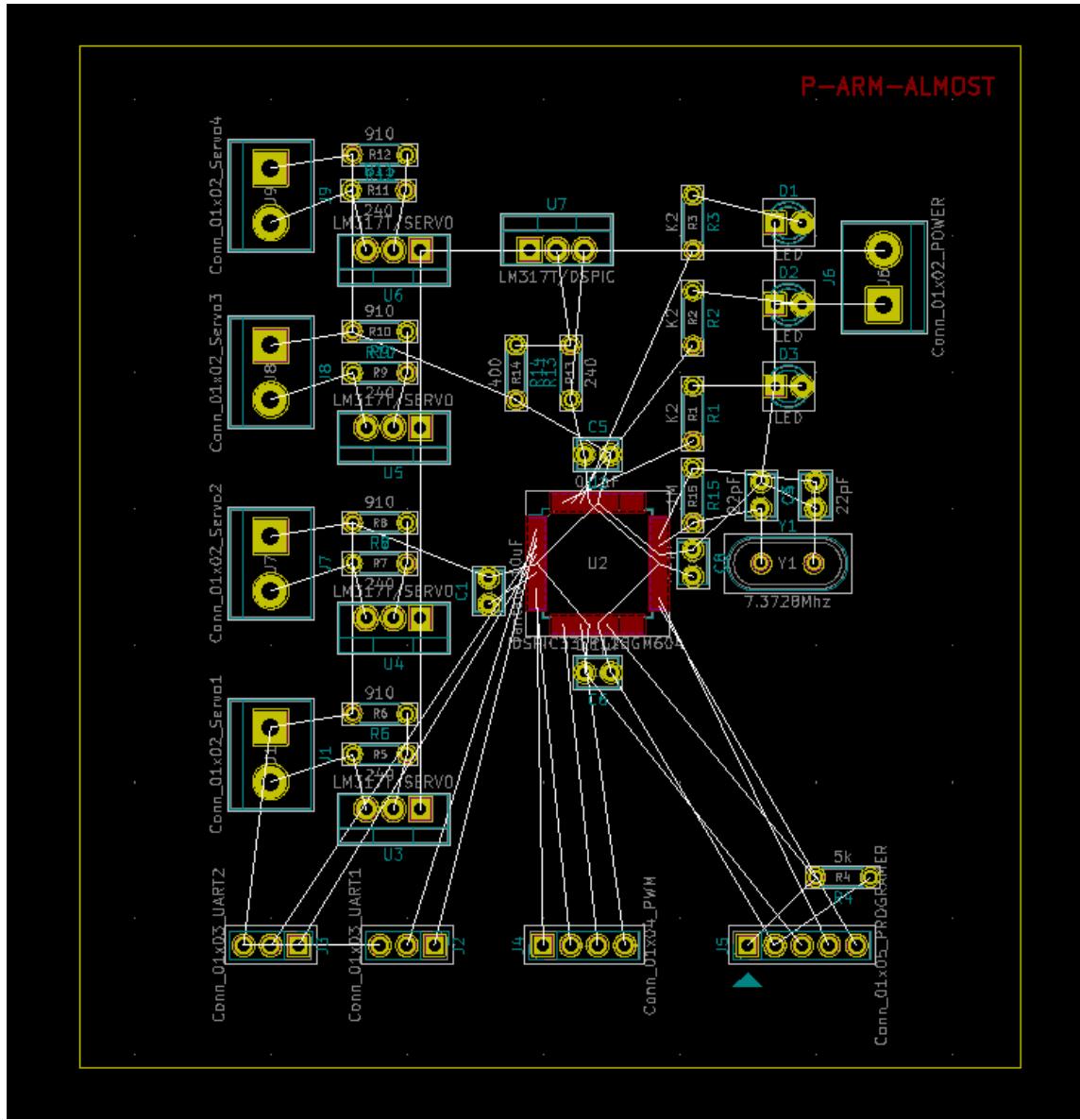


Figura 6.92: Distribución final de los componentes físicos.

Tal y como puede verse en la imagen anterior (imagen 6.92), las líneas blancas representan las conexiones físicas entre los componentes y por lo tanto, deben ser sustituidas por pistas de cobre. En este diagrama se han generado conexiones inexistentes en el diagrama esquemático, ya que se contemplan conexiones físicas que a nivel lógico no son necesarias.

Tras realizar el proceso de enrutado en ambas caras, el diagrama físico final obtenido es el siguiente (ver imagen 6.93):

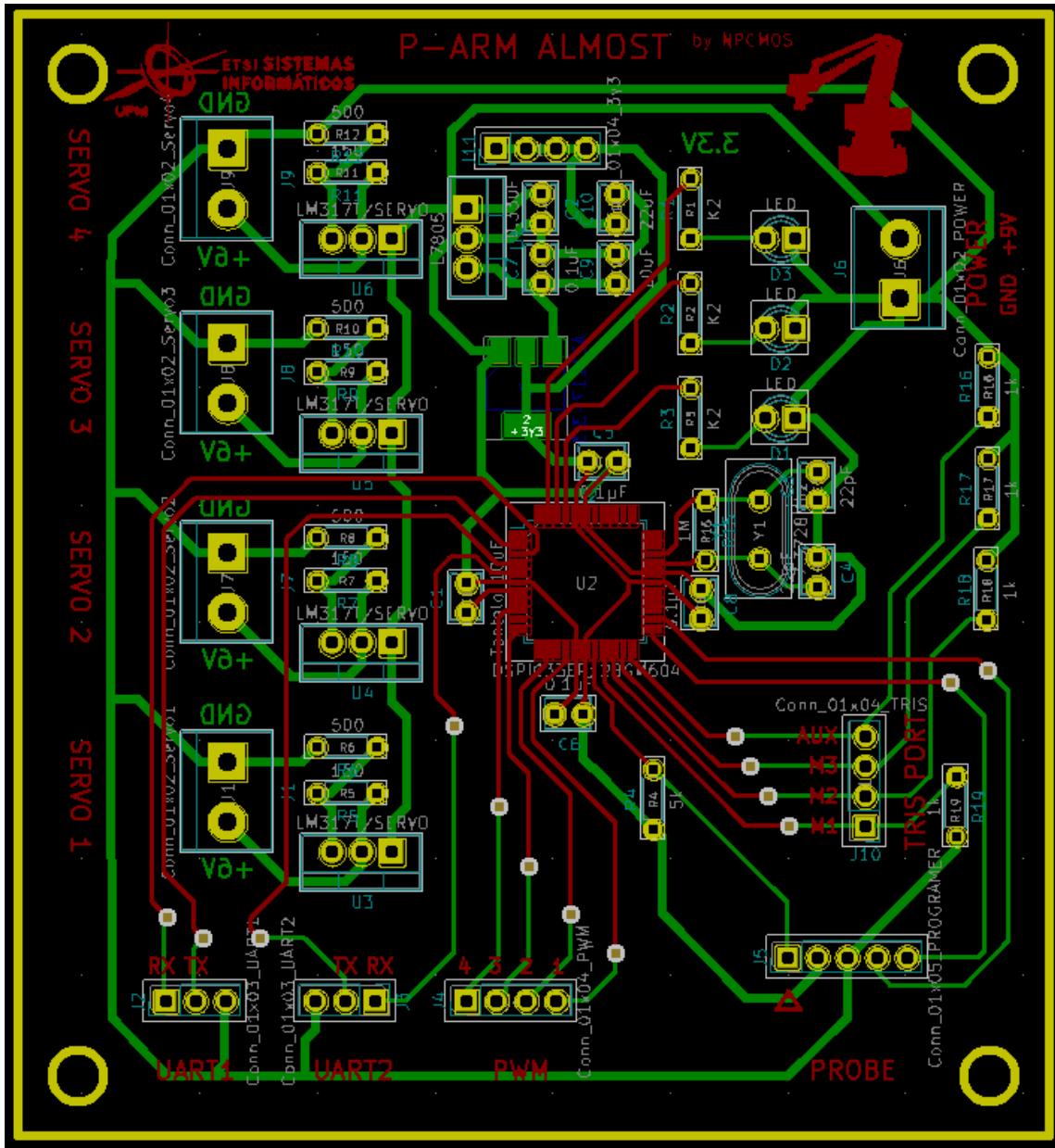


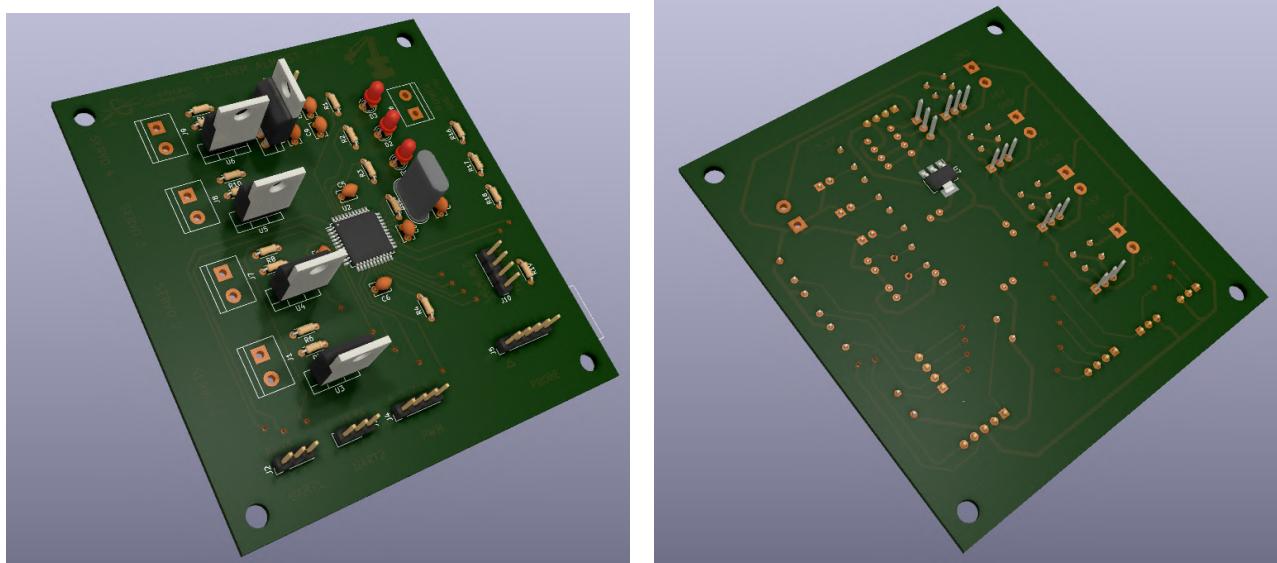
Figura 6.93: Diagrama físico final.

Cabe destacar varios aspectos:

- Las pistas de color rojo se corresponden con la cara frontal o de componentes.
- Las pistas de color verde se corresponden con la cara trasera o de soldadura.
- Se han incluido marcas de serigrafiado adecuadas para identificar correctamente la PCB y su interfaz.
- Se han incluido orificios de mecanizado para la futura sujeción de la PCB.
- En la parte superior izquierda y derecha de la capa frontal se han añadido los logos de la ETSISI y el *pArm*.

- En la parte central de la capa frontal se ha incluido el nombre de la placa (P-ARM ALMOST) y el nombre del grupo de ingenieros (NPCMOS).

Utilizando el visualizador 3D de KiCad se puede obtener un representación cercana a la realidad de como será la PCB al fabricarse (ver imagen 6.94) :



(a) Capa frontal

(b) Capa trasera

Figura 6.94: Representación 3D del diseño físico.

6.5.6. Verificaciones realizadas al diseño lógico y físico

Dado que el proceso de diseño de la PCB es uno de los elementos críticos dentro del proyecto, es necesario llevar a cabo una serie de verificaciones durante dicho proceso para minimizar el riesgo de fallos.

En primer lugar, se deben realizar las verificaciones del diseño lógico de la PCB, ya que es el primer diagrama que se realiza. Como aporta información de las conexiones lógicas entre los distintos componentes de la PCB, las verificaciones a realizar irán destinadas a comprobar la corrección de dichas conexiones y de los componentes empleados.

A continuación, se muestra una lista de las principales verificaciones realizadas en el diagrama lógico:

- Verificar si se han incluido en la PCB todos los componentes deseados.
- Verificar si se ha incluido el circuito de alimentación de la PCB, y por lo tanto, comprobar si todos los componentes están alimentados correctamente.
- Verificar si se ha incluido el conexionado mínimo recomendado por el fabricante para asegurar un correcto funcionamiento del microcontrolador y sus periféricos.

- Verificar si el conexionado de los dispositivos periféricos, puertos de conexión y demás componentes es correcto y, además, se ha realizado a los pines adecuados del microcontrolador.
- Verificar si se ha asignado una huella física correcta a cada uno de los componentes lógicos, la cual se utilizará posteriormente en el diagrama físico.

Tras realizar las revisiones anteriores se considera que el diagrama lógico está en un estado de madurez suficiente como para poder crear el diagrama físico a partir de él y, posteriormente, realizar también las revisiones del mismo.

Dichas verificaciones a realizar van destinadas a comprobar los aspectos físicos de los componentes, conexiones mediante pistas, requerimientos estructurales, dimensiones, etc., donde se comprueba:

- Verificar que todos los componentes del diagrama lógico aparecen en el diagrama físico al importar la lista de redes.
- Verificar que todas las conexiones lógicas entre componentes del diagrama lógico aparecen representadas en el diagrama físico.
- Verificar que todas las conexiones que aparecen en el diseño físico se corresponden con conexiones establecidas en el diagrama lógico.
- Verificar que todas las huellas físicas de los componentes lógicos son correctas y su distribución de pines es la deseada. Se debe contrastar esta información usando el *datasheet* de cada uno de los componentes.
- Verificar que todas las conexiones lógicas se han realizado, y por lo tanto, se corresponden con una conexión mediante pista en el diagrama físico.
- Verificar que todos los componentes están alimentados y que las pistas de alimentación tienen el tamaño adecuado.
- Verificar que todas las pistas de comunicación tienen el origen y destino adecuado, y que su ancho es el adecuado.
- Verificar que no existen pistas con un ancho menor de $0,3mm$, ya que el proceso de fabricación podría fallar por debajo de este tamaño.
- Verificar que la separación mínima entre pistas es como mínimo de $0,3mm$ para pistas de alimentación y $0,25mm$ para pistas de comunicación digital.
- Verificar que los componentes de soldadura superficial no tienen contactos bajo su superficie que toquen pistas no deseadas.
- Verificar que se han incluido marcas de serigrafía que permitan identificar a la PCB y su interfaz de forma clara.
- Verificar que se ha incluido el mecanizado de sujeción adecuado.

- Verificar que no existan pistas que contacten con partes metálicas del mecanizado de sujeción.
- Verificar que todas las soldaduras de los componentes SMD y THT sean realizables físicamente.
- Verificar que el conector de programación dispone de espacio suficiente para realizar la conexión de la sonda.
- Verificar que las vías y *pads* tienen un diámetro mínimo de 1mm con agujero de 0,5mm.

KiCad ofrece una herramienta llamada “Comprobar reglas de diseño” (ver imagen 6.95), la cual puede ser utilizada para facilitar el proceso de verificación del diseño físico. Utilizando esta herramienta se pueden verificar automáticamente factores como el ancho de pista mínimo permitido, separación mínima entre pistas, conexiones no realizadas, etc. A pesar de poder realizar estas verificaciones de forma automática, se han realizado todas las verificaciones de forma manual para también reducir el riesgo de fallos.

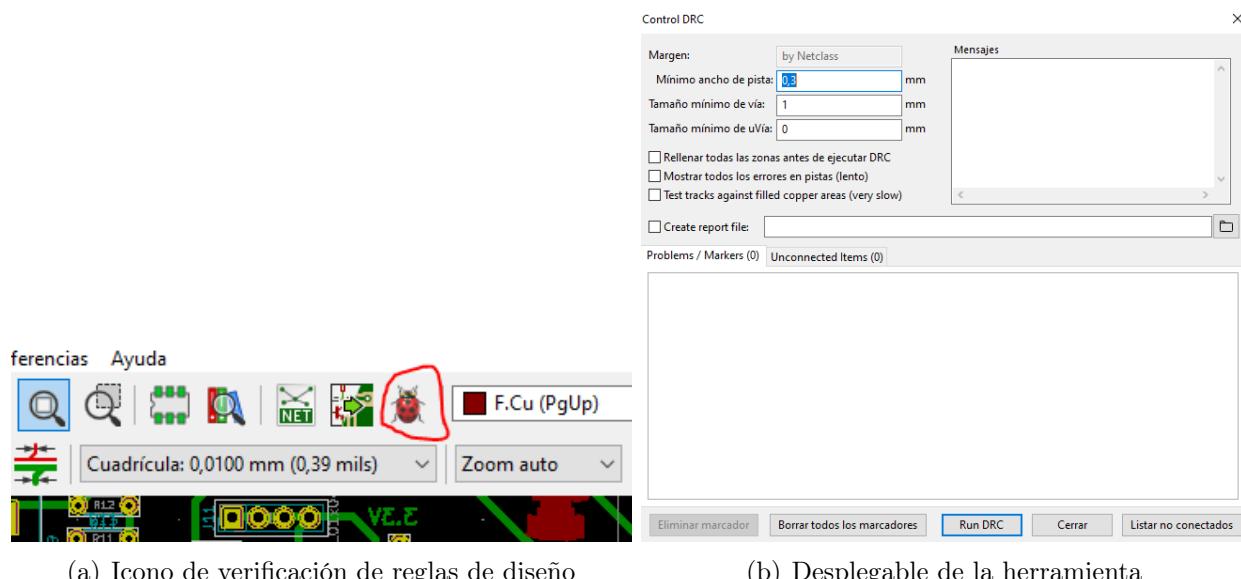


Figura 6.95: Herramienta de verificación de reglas de diseño.

Tras realizar todas las verificaciones anteriormente mencionadas en el diagrama físico, se han subsanado los errores encontrados previamente a realizar la fabricación de la PCB; se considera por lo tanto que el proceso de verificación del diseño ha sido exitoso.

6.5.7. Construcción

Una vez se ha finalizado el proceso de diseño de la PCB, se comienza el proceso de fabricación de la misma. Dicho proceso consta de diversas etapas, las cuales van desde la preparación de los materiales hasta la obtención del prototipo final. Todas estas etapas se detallan a continuación una por una.

En primer lugar, cabe destacar que el proceso de fabricación seleccionado para construir la PCB ha sido la fotolitografía. Dicho proceso consiste en transferir un patrón de un circuito desde una fotomáscara a una placa de prototipado positiva mediante una serie de procesos lumínicos y químicos. Posteriormente, una vez se tiene impreso el circuito en la placa de prototipado, se procede al taladrado de orificios y soldado de los componentes del circuito.

A continuación, se presentan los conceptos básicos empleados durante el proceso de fotolitografía y cuyo entendimiento es fundamental para comprender el proceso de fabricación:

- El proceso de fotolitografía parte necesariamente de una placa de prototipado de fibra de vidrio, la cual posee las siguientes capas de materiales:

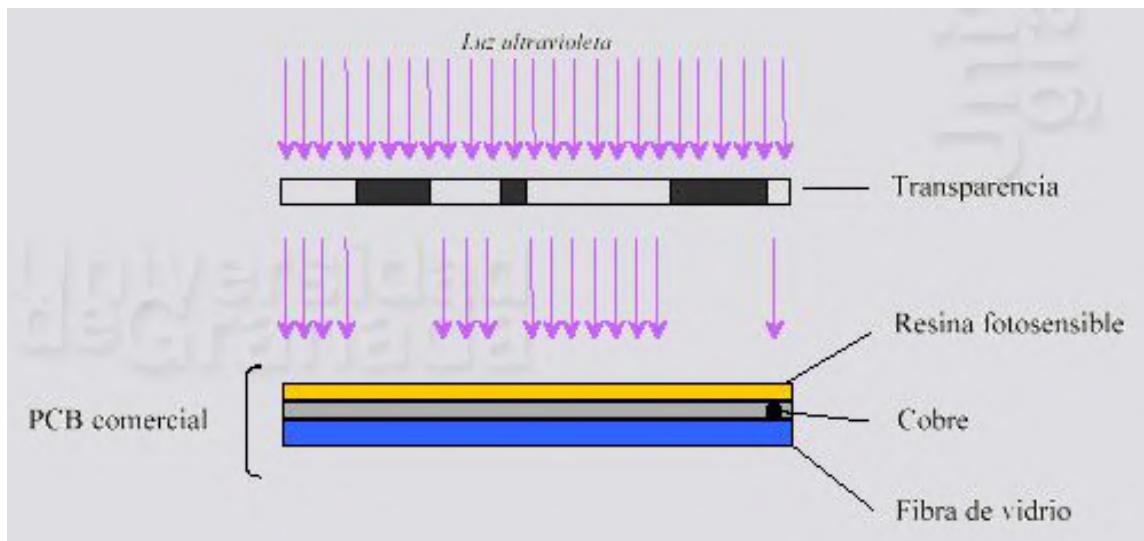


Figura 6.96: Estructura de la placa de prototipado [39].

Tal y como se puede ver en la imagen anterior (imagen 6.96), la placa de prototipado de fibra de vidrio posee tres capas: fibra de vidrio como material base encima de la cual hay una capa de cobre y, protegiendo la anterior, una capa de resina fotosensible a la luz ultravioleta.

El aspecto clave de esta estructura en capas recae en la capa de resina fotosensible, la cual tiene como objetivo capturar el patrón del circuito de la fotomáscara o transparencia. El funcionamiento de esta resina depende de si la placa de prototipado es positiva o negativa:

- En las placas de prototipado positivas, la resina fotosensible que es insolada con luz ultravioleta reaccionará correctamente con el revelador, y por lo tanto desaparecerá; mientras que la resina que no es insolada permanecerá tras el revelado.
- En las placas de prototipado negativas, la resina fotosensible que es insolada con luz ultravioleta se convierte en resistente al revelador y por tanto permanecerá tras el proceso de revelado; mientras que la resina que no es insolada con luz ultravioleta, reaccionará correctamente con el revelador, y por lo tanto desaparecerá.

En este proyecto se han utilizado placas de prototipado positivas de fibra de vidrio.

- Mediante el proceso de insolado con luz ultravioleta, la fotomáscara plasma el patrón del circuito a imprimir en la resina fotosensible:

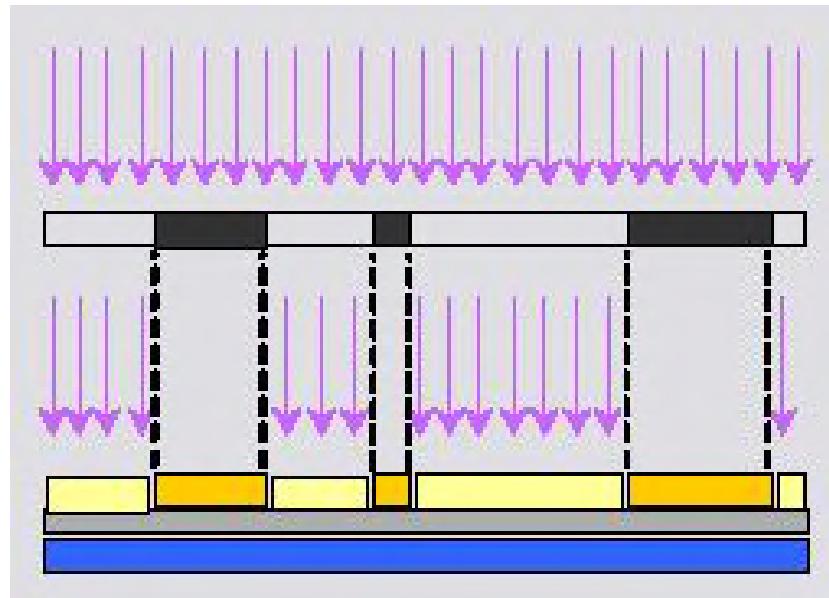


Figura 6.97: Tratado de las resinas mediante insolado [39].

Tal y como se puede ver en la imagen anterior (imagen 6.97), las superficies de la resina que han sido insoladas (amarillo claro), se convierten en reactivas al revelador, ya que en esa zona de la fotomáscara existe una transparencia. En las zonas en las cuales la transparencia es opaca e impide el paso de la luz ultravioleta la resina no se ve afectada y se mantiene no reactiva con el revelador.

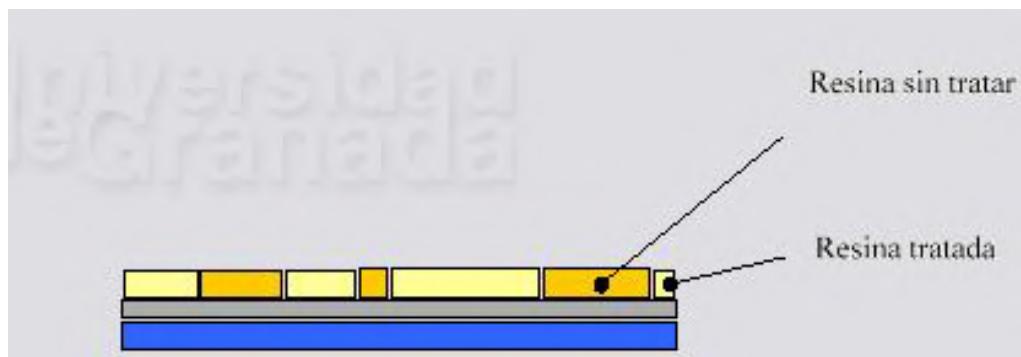


Figura 6.98: Resultado tras el insolado [39].

- Mediante el proceso de revelado, se elimina la resina que fue tratada en el proceso de insolación (ver imagen 6.99):

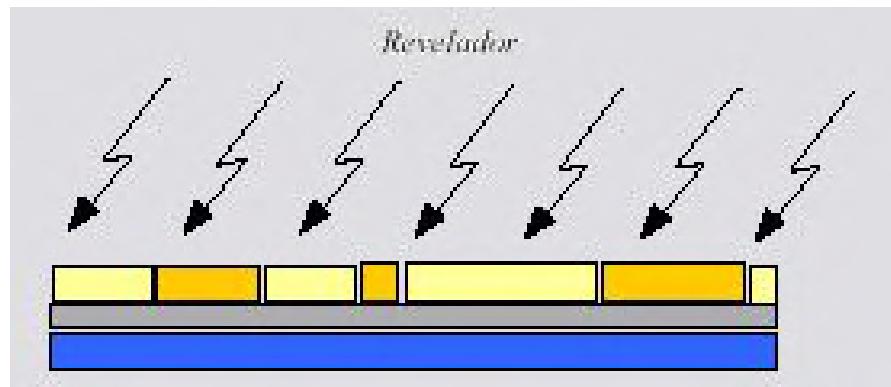


Figura 6.99: Proceso de revelado [39].

Tras someter la PCB al proceso de revelado, se obtiene el siguiente resultado (imagen 6.100):

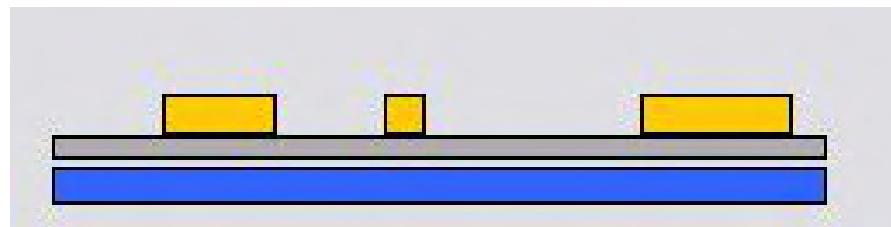


Figura 6.100: Resultado tras revelado [39].

El proceso de revelado elimina la resina que había sido insolada, ya que se trata de una placa positiva. En estas zonas, la capa de cobre queda expuesta y sin ninguna protección. Por el contrario, en las zonas que no fueron insoladas, el revelado no actúa y por lo tanto la resina inicial permanece protegiendo la capa de cobre.

- Mediante el proceso de atacado, se eliminan las superficies de la capa de cobre sobrantes, es decir, aquellas que no están protegidas por resina:

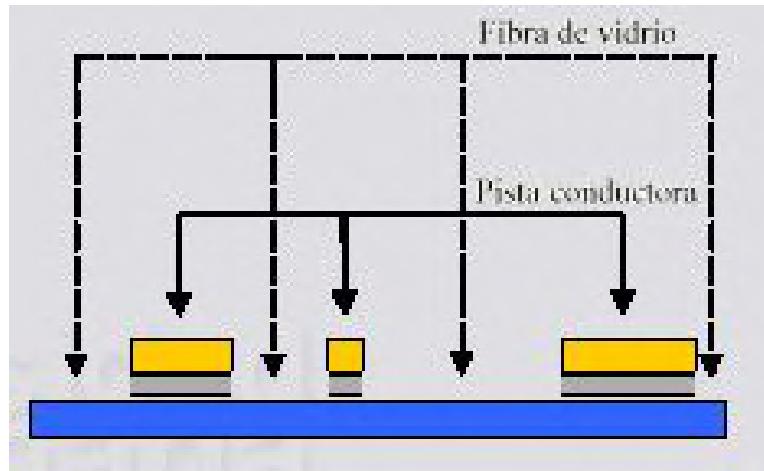


Figura 6.101: Resultado tras atacado [39].

Tal y como se puede observar en la imagen anterior (imagen 6.101), la capa de cobre solo permanece en los lugares que estaban protegidos por la resina inicial; este patrón coincide con el circuito de la fotomáscara. Para retirar la resina sobrante, basta con aclarar la PCB con alcohol, quedando las pistas del circuito perfectamente delimitadas e impresas en la placa de fibra de vidrio.

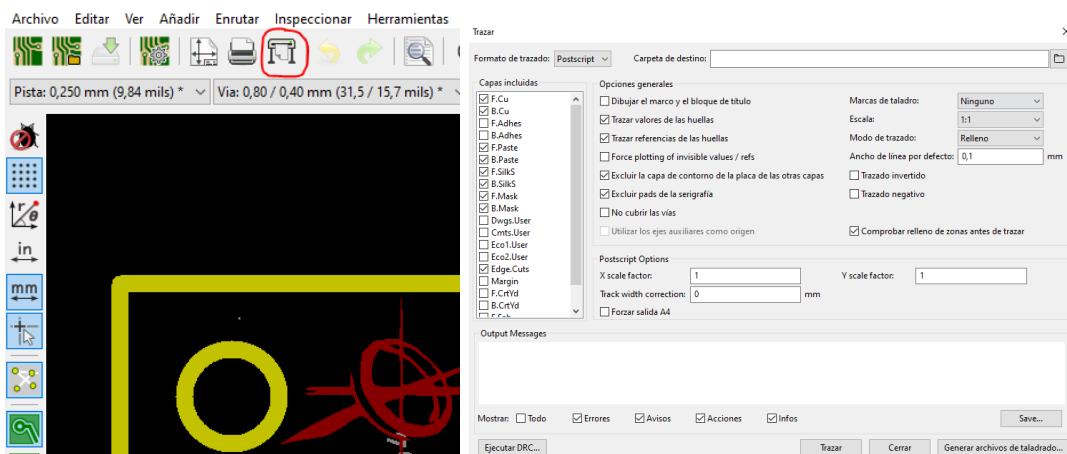
Existen otros procesos de fabricación para PCBs que ofrecen un resultado muy parecido al deseado en este proyecto, por ejemplo fabricación mediante máquina CNC, sin embargo, por motivos de disponibilidad de materiales y maquinaria, se ha decidido utilizar la fotolitografía.

A continuación se enumeran las diversas etapas del proceso de fabricación, ordenadas cronológicamente desde el comienzo del proceso hasta su finalización:

1. Generación de las fotomáscaras a partir del diagrama físico de la PCB.

El diagrama físico representa de forma precisa cual es la apariencia física de la PCB, las huellas de sus componentes, pistas de conexión, marcas de mecanizado, serigrafía, orificios y límites. Es precisamente toda esta información la que se quiere plasmar mediante fotolitografía en la placa de prototipado positiva que contendrá el circuito impreso.

Utilizando la herramienta “trazar placa” disponible en KiCad, se puede generar las máscaras asociadas al diagrama físico:

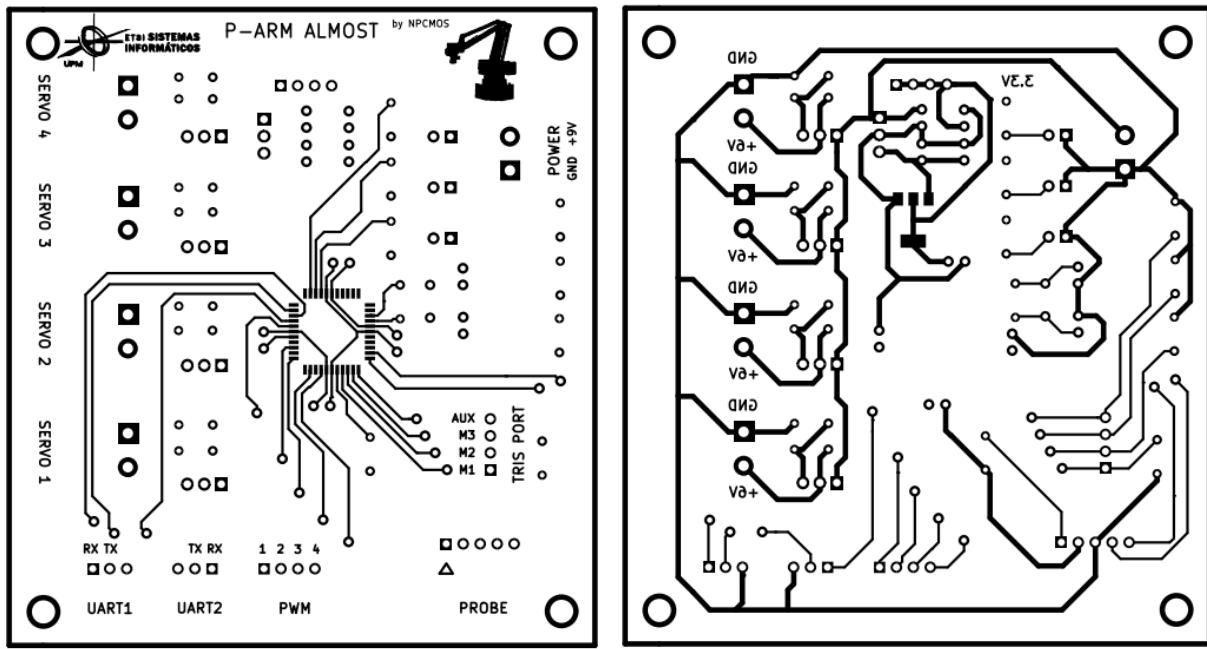


(a) Icono de trazado de placas

(b) Desplegable de la herramienta

Figura 6.102: Herramienta de trazado de placas.

Las fotomáscaras obtenidas tras el uso de dicha herramienta son las siguientes (ver imágenes 6.103):



(a) Máscara frontal.

(b) Máscara trasera.

Figura 6.103: Fotomáscara final de la PCB.

Con el fin de poder utilizar dichas máscaras en el proceso de fotolitografía, es necesario imprimirlas en papel de transparencia, puesto que deben dejar pasar la luz en los lugares que no contienen el patrón del circuito a fotolitografiar.

Para imprimir dichas máscaras en papel de transparencia, primeramente se imprimen en papel convencional y posteriormente se fotocopian a un papel de transparencia, obteniendo el resultado siguiente (ver imagen 6.104):



(a) Máscaras de papel y sus transparencias.

(b) Recorte de las máscaras.

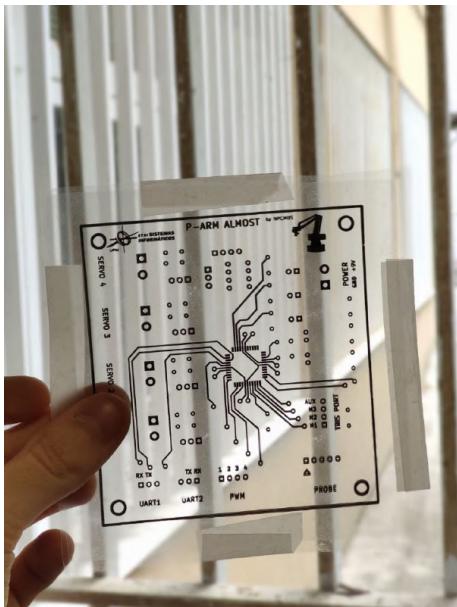
Figura 6.104: Proceso de impresión y recorte de las fotomáscaras.

Otro de los aspectos a destacar y que es vital en relación a la impresión de las transparencias es que, dado que todas las impresoras producen cierta distorsión en los documentos

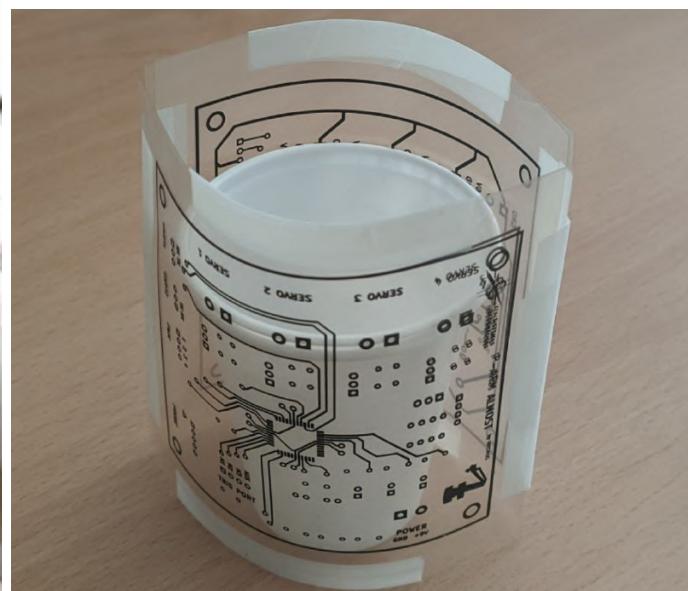
que imprimen, se debe incluir una corrección de escala *XY*, en particular para este proyecto de 1,037 para calibrar correctamente la impresión. En caso contrario, se puede producir algún pequeño desajuste en el encaje de los componentes una vez esté fabricado el circuito.

Dado que el objetivo principal de dichas fotomáscaras es el de transferir el patrón del circuito de la PCB a la placa de prototipado positiva, se deben ensamblar de tal forma que permitan introducir en su interior la misma, para que posteriormente durante el proceso de insolación, se consiga fotolitografiar el circuito en ambas caras de la PCB.

A continuación se muestra el ensamblado final de la funda construida con las fotomáscaras:



(a) Funda de transparencias.



(b) Demostración de su capacidad de contener a la placa.

Figura 6.105: Fotomáscara final.

Cabe destacar que se han incluido dos transparencias superpuestas por cada cara de la fotomáscara para aumentar la opacidad que se genera al transferir el patrón del circuito a la placa de prototipado positiva. Estas transparencias deben estar superpuestas con una exactitud muy elevada para impedir desfases en la impresión del circuito.

2. Una vez se ha completado el generado de la fotomáscara, se debe realizar el proceso de insolación de la placa de prototipado positiva. Durante este proceso, se realiza la transferencia del patrón del circuito desde la fotomáscara a la placa de prototipado mediante la exposición de la misma a luz ultravioleta.

A continuación se muestra el proceso de insolación de la placa de prototipado positiva, el cual se ha realizado usando una maquina insoladora:



(a) Placa de prototipado colocada dentro de la fotomáscara.

(b) Ajuste de la máquina de insolación.

Figura 6.106: Proceso de insolación de la PCB usando las fotomáscaras.

Tras el proceso de insolación, la placa de prototipado positiva habrá sido expuesta a la luz ultravioleta durante un minuto y, por lo tanto, el patrón del circuito habrá sido transferido a la superficie de resina de la placa de prototipado:

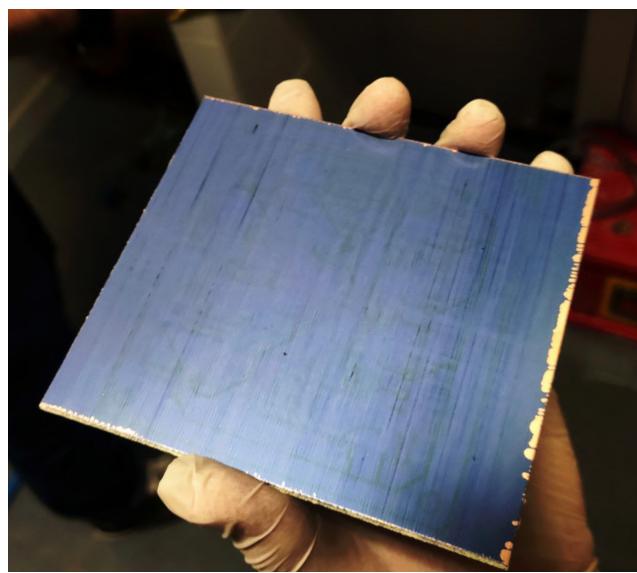


Figura 6.107: Placa de prototipado positiva insolada.

3. Cuando se ha completado el proceso de insolación de la placa de prototipado se debe llevar a cabo el proceso de revelado de la PCB.

El proceso de revelado es un proceso químico mediante el cual se consigue eliminar la resina protectora de la placa de prototipado que ha sido expuesta a la luz ultravioleta durante el proceso de insolación. Esta resina que es eliminada coincide con la superficie de la PCB que no contiene pistas del circuito impreso, gracias a la fotomáscara.

Este proceso puede llevarse a cabo con un revelador universal indicado para realizar este tipo de labores, a continuación se muestra el empleado en este proyecto:

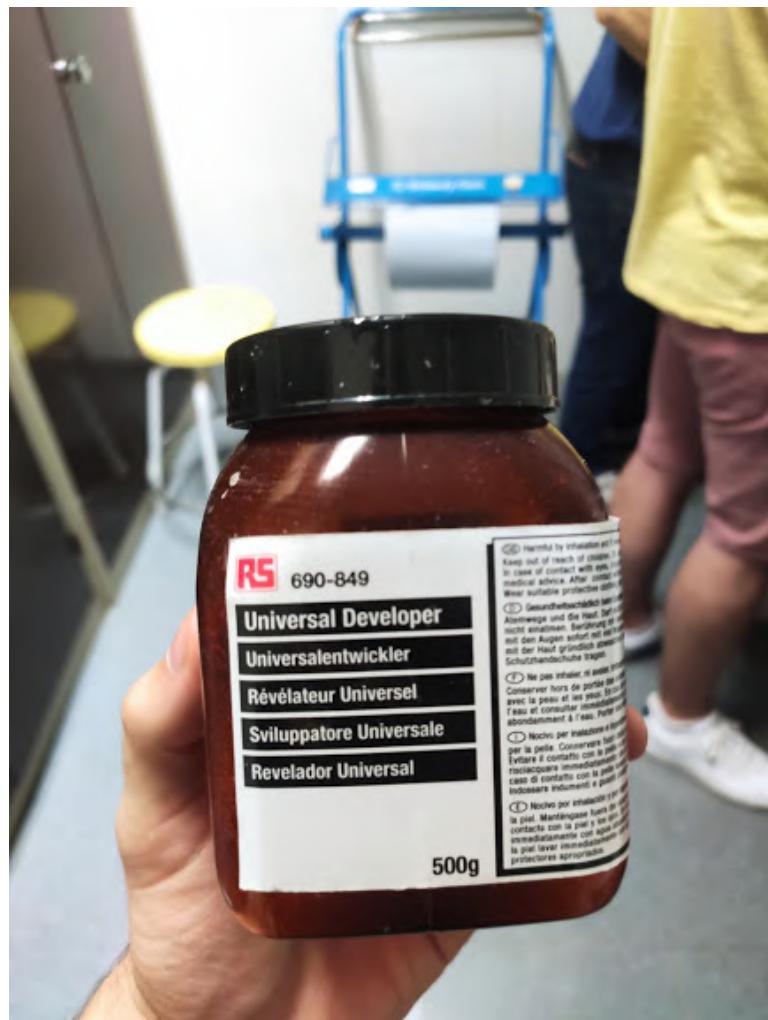


Figura 6.108: Revelador universal empleado.

El proceso de revelado de la PCB se lleva a cabo disolviendo el revelador anteriormente mencionado en agua y, posteriormente, sumergiendo la PCB en esta disolución:

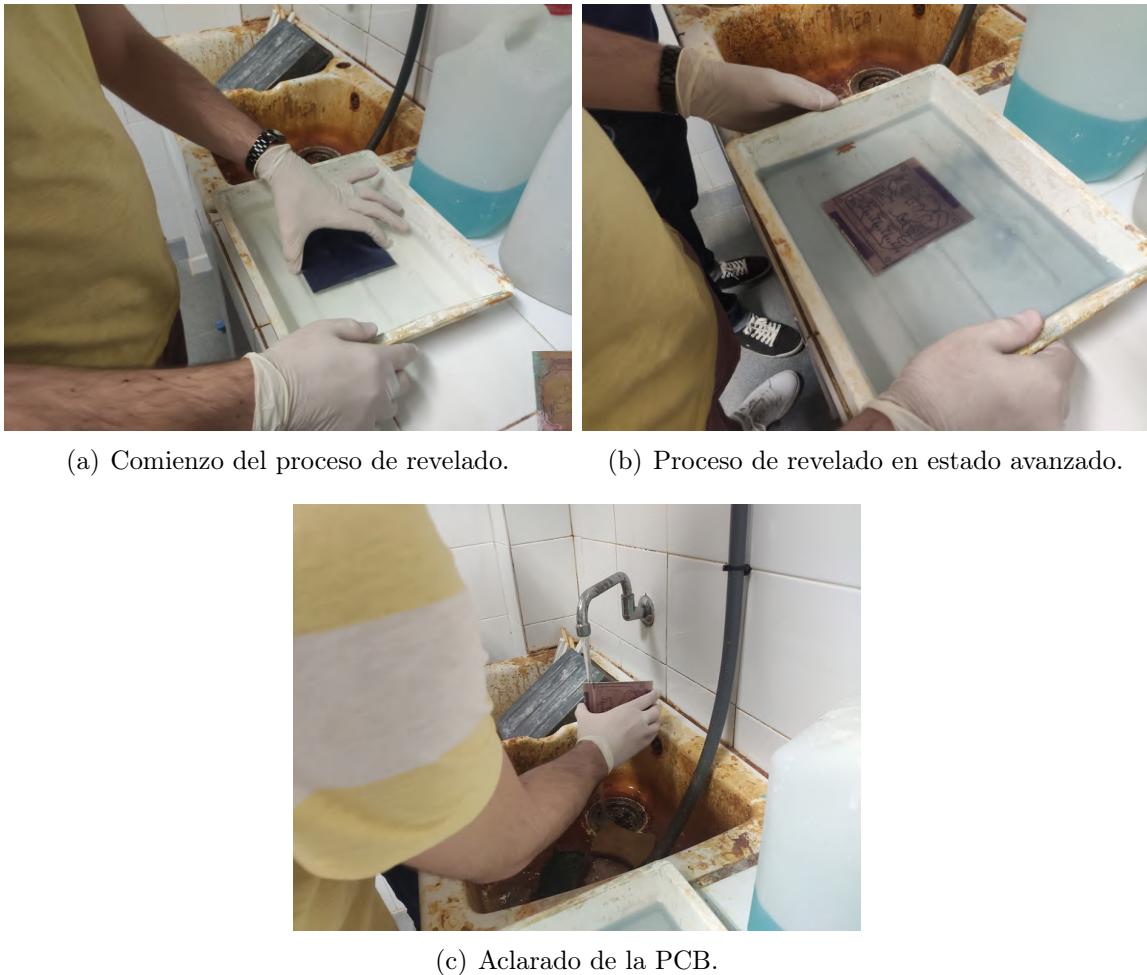


Figura 6.109: Proceso de revelado de la PCB.

Durante el proceso de revelado se tienen que generar movimientos delicados de la disolución reveladora para que esta haga un efecto adecuado y consiga eliminar la resina sobrante. Cuando el proceso ha finalizado, basta con aclarar la PCB con agua sin tocar el patrón del circuito que ha quedado revelado.

4. Tras revelar el patrón del circuito de la PCB, se debe realizar el proceso de atacado.

En la etapa anterior del proceso de fabricación, las zonas de la PCB que no fueron insoladas con luz ultravioleta quedan cubiertas de resina protectora, mientras que en las parte de la PCB que fueron expuestas a luz ultravioleta, la capa de cobre queda expuesta y sin cubrir por ningún tipo de resina ya que el proceso de revelado ha eliminado dicha resina sobrante.

La etapa de atacado del proceso de fabricación consiste en someter a la PCB a un proceso químico, mediante el cual se elimina el cobre de la placa de prototipado a través del atacado mediante una disolución de persulfato de amonio. Las zonas de la PCB en las cuales el cobre está expuesto resultarán atacadas por la disolución y, por lo tanto, el cobre desaparecerá. Por el contrario, las zonas de la PCB en las cuales el cobre está protegido por una capa de resina, no serán atacadas y el cobre permanecerá.

A continuación, se muestran algunas imágenes (imágenes 6.110) del proceso de atacado mediante ácido:

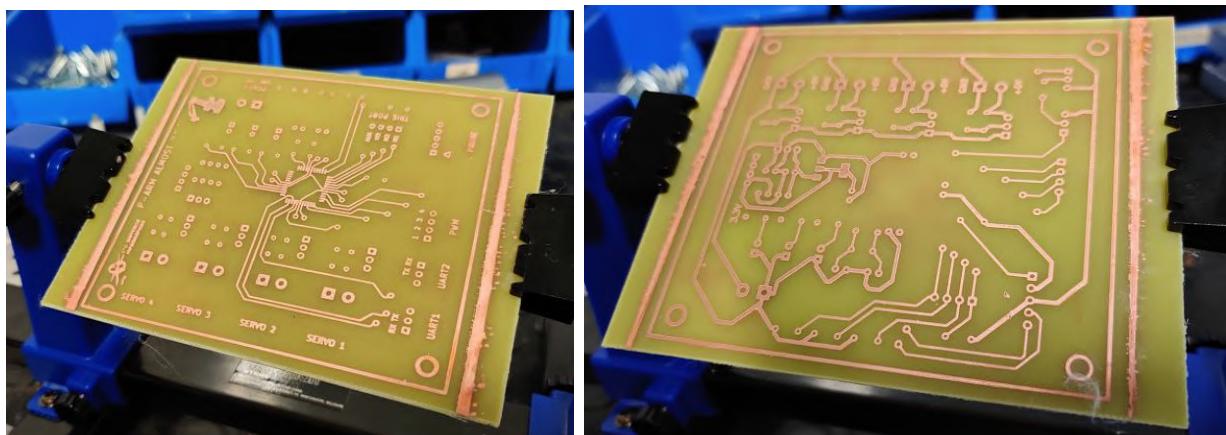


(a) Recipiente contenedor de la disolución de atacado. (b) PCB siendo sometida al proceso de atacado.

Figura 6.110: Proceso de atacado de la PCB.

Tras el proceso de atacado, el cual lleva entre una y dos horas, se debe aclarar y limpiar el circuito impreso con alcohol para eliminar los restos de resina sobrantes, así como las impurezas producidas por la disolución de atacado.

Al finalizar esta etapa del proceso de fabricación, se obtiene una PCB en la cual aparece el patrón del circuito de las fotomáscaras impreso en cobre:



(a) Parte frontal de la PCB.

(b) Parte trasera de la PCB.

Figura 6.111: Circuito impreso final tras el proceso de atacado.

5. Es recomendable realizar una revisión general de la PCB en busca de posibles imperfecciones que se hayan podido producir durante el proceso de impresión del circuito.

Normalmente, durante el proceso de atacado, se pueden producir imperfecciones en el trazado de las pistas debido a que la disolución de atacado puede producir una corrosión

irregular de la capa de cobre. Este tipo de imperfecciones suelen ser cortos entre pistas, fracturas de pista, etc.

Para realizar el proceso de comprobación de imperfecciones es necesario utilizar un polímetro que pueda funcionar en modo detección de cortos para que, de esta forma, se pueda determinar cuándo la conductividad de las pistas es correcta. También es recomendable usar una lupa de aumento o similar para mejorar la visibilidad de las imperfecciones:



Figura 6.112: Integrante del equipo verificando cortos.

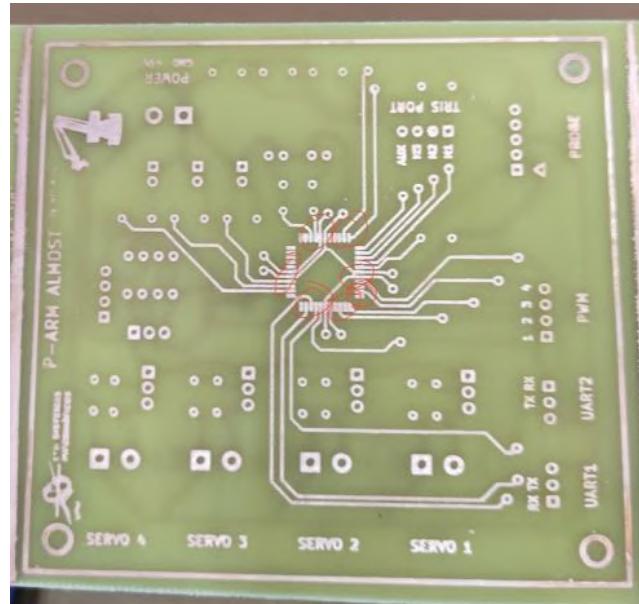


Figura 6.113: Algunos de los cortos detectados.

En caso de encontrar algún corto entre dos pistas que no deberían estar conectadas, basta con utilizar un útil con punta para rasgar la superficie del punto de unión entre ambas.

En caso de encontrar alguna fractura en una de las pistas, basta con realizar una soldadura de empalme utilizando un hilo de grapinar o similar en el punto de fractura.

Ambas aproximaciones propuestas anteriormente son soluciones artesanales y no puede aplicarse cuando las imperfecciones son demasiado grandes o complejas de resolver, siendo mejor en este caso repetir el proceso de fotolitografía.

6. Se debe realizar taladrado de los orificios de la PCB y el guillotinado de los márgenes de la misma.

El taladrado de todos los orificios de la PCB, es decir, *pads* de componentes, vías y mecanizado de sujeción, se debe realizar con un taladro que cuente con una broca del tamaño adecuado, además de un soporte que aporte la estabilidad suficiente:



(a) Taladro y soporte usado.

(b) Taladrado de los distintos orificios para componentes.

Figura 6.114: Taladrado de la PCB.

El recorte de la PCB se realiza con una herramienta de guillotinado que tenga precisión y permita realizarlos de forma limpia de la longitud necesaria. Estos cortes se realizan siguiendo los límites establecidos en el diagrama físico de la PCB.

Tras el proceso de guillotinado y taladrado se obtiene el siguiente resultado:

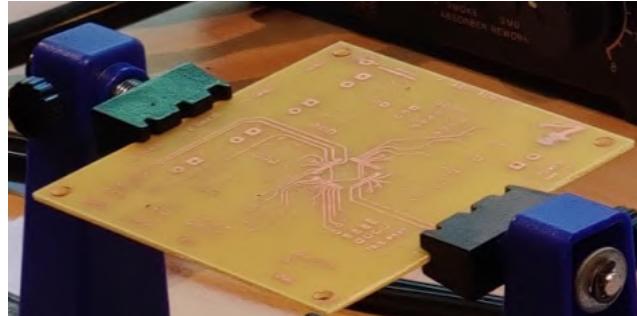


Figura 6.115: PCB con orificios taladrados y margenes guillotinados.

7. Soldar el componente principal de la PCB, es decir, el microcontrolador:

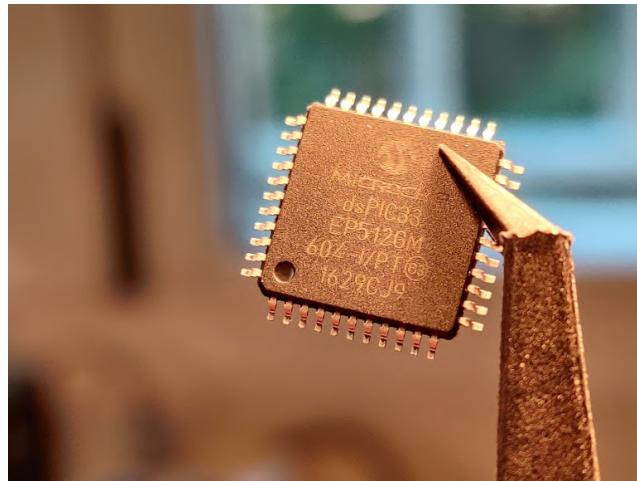


Figura 6.116: Plano detallado del microcontrolador.

Tal y como se puede ver en la imagen 6.116, el dsPIC utilizado tiene un encapsulado destinado a soldadura superficial (SMD). Normalmente, este tipo de componentes deben ser colocados y soldados los primeros, ya que el proceso de colocación y soldado de los mismos es delicado y requiere máxima precisión.

El soldado de este tipo de componentes se realiza mediante el aplicado de una pasta de soldadura, la cual se debe colocar en el punto de contacto de los pines del componente con las pistas de cobre:

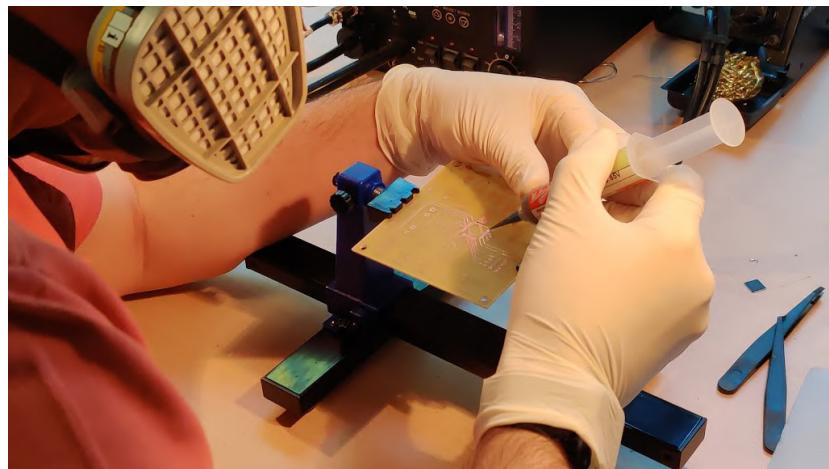


Figura 6.117: Aplicado de la pasta de soldadura.

El aplicado de la pasta de soldadura debe llevarse a cabo con la protección adecuada, ya que es altamente tóxica.

Tras aplicarla, se debe colocar el microcontrolador de forma precisa y cautelosa, cerciorándose de que cada uno de los pines realiza contacto correctamente con las pistas de la PCB:

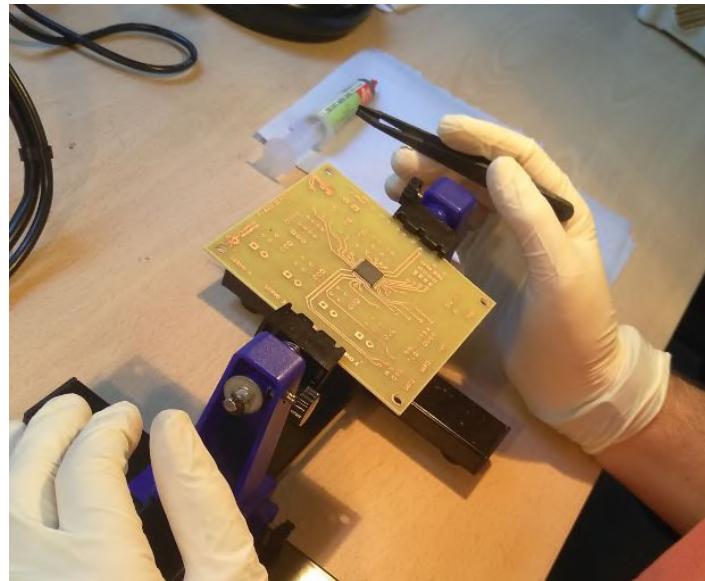
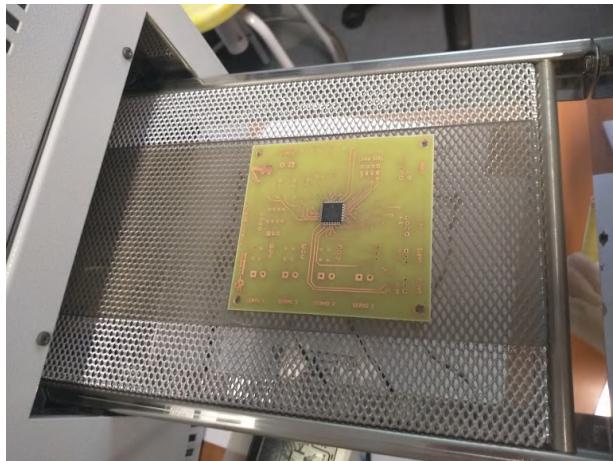
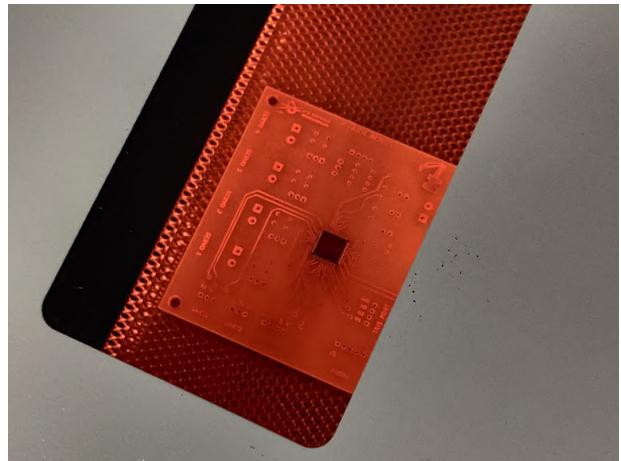


Figura 6.118: Colocado del microcontrolador sobre la pasta de soldadura.

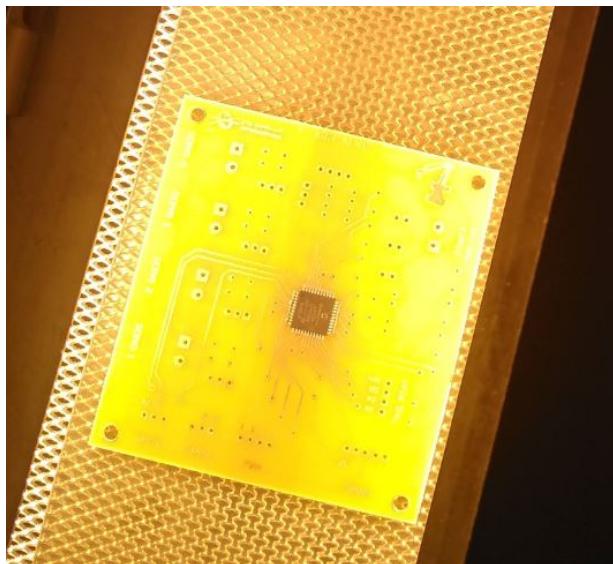
Tras colocar la pasta de soldadura, se debe introducir la PCB en un horno para que la pasta de soldadura se endurezca y selle por completo:



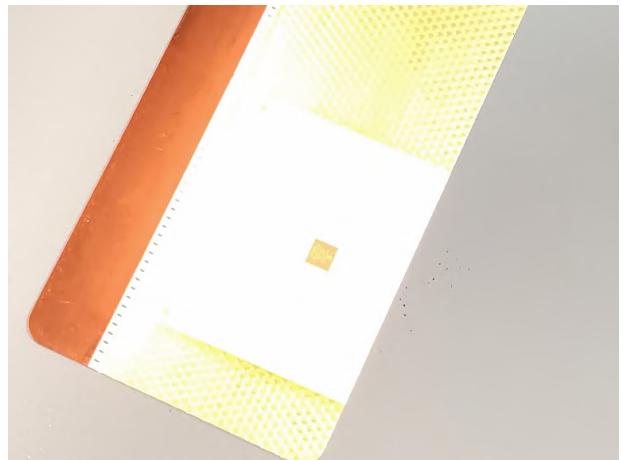
(a) Preparado de la placa en el horno.



(b) Precalentado del horno a 260 °C.



(c) Proceso de endurecimiento (1) - se deja reposar la pasta para que se solidifique.



(d) Proceso de endurecimiento (2) - se aplican golpes de calor para fijar la pasta.

Figura 6.119: Diferentes etapas del proceso de horneado.

Este proceso debe respetar un perfil de calentamiento en varias etapas que persiguen la actuación de las resinas de difusión y posteriormente la realización de las soldaduras propiamente dichas. Los límites de temperatura y los tiempos de cada etapa deben establecerse siguiendo las recomendaciones de los fabricantes de los componentes y de la pasta de soldadura.

Tras el proceso de horneado, se debe dejar enfriar la PCB. El resultado final tras el horneado es el siguiente:

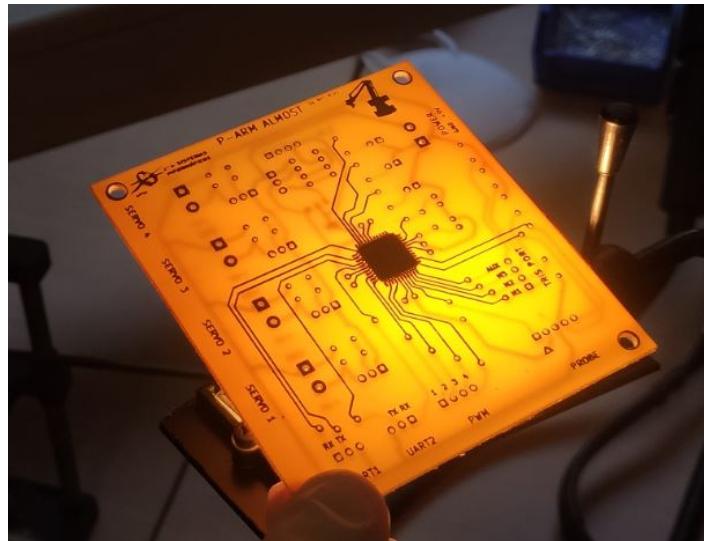


Figura 6.120: Imagen a contra luz de la PCB.

El microcontrolador se encuentra totalmente fijado y soldado superficialmente con las pistas de conexión gracias a la pasta de soldadura. En este momento, es recomendable verificar si existen cortos entre los pines del microcontrolador así como si la conductividad de los mismos es correcta a lo largo de las pistas:

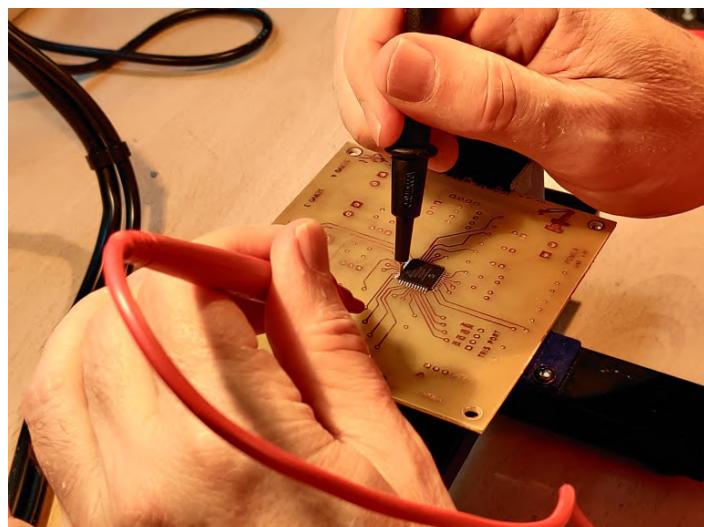
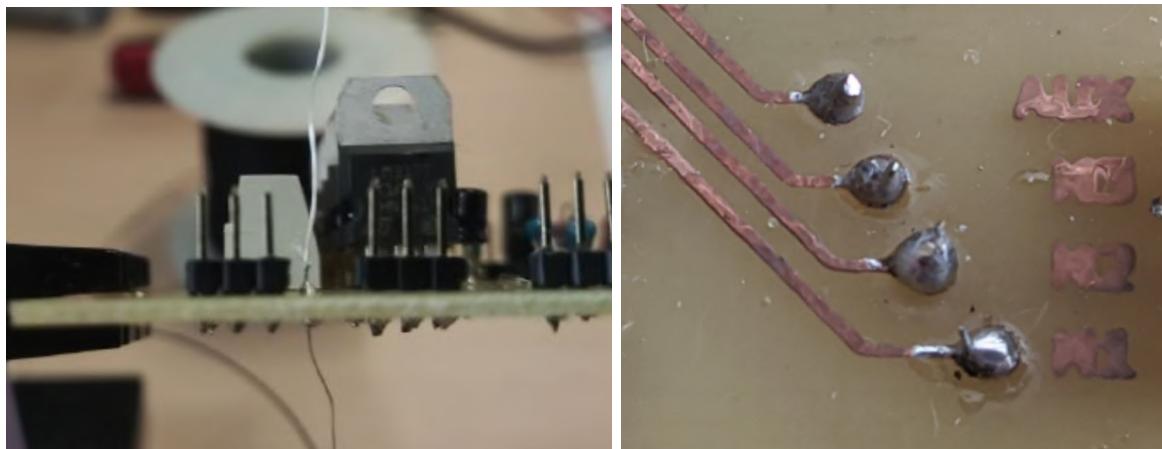


Figura 6.121: Comprobación de cortos y conductividad de las pistas del microcontrolador.

8. Soldado de las vías de conexión entre las pistas de la capa superior e inferior.

En ausencia de herramientas para remachar agujeros pasantes, basta con introducir un hilo de grapinar por el orificio de la vía, realizar una pequeña soldadura con ambas pistas superior e inferior y, por último, cortar el hilo excedente:



(a) Vía con hilo de grapinar sin cortar.

(b) Aspecto final de la vía.

Figura 6.122: Conexiónado de las vías de la PCB.

9. En último lugar, se deben soldar el resto de componentes de la PCB.

Los componentes restantes tienen un encapsulado de agujero pasante (THT), a excepción del regulador A1117Z, el cual es de tipo SMD. Sin embargo, sus pines son de gran tamaño y no es necesario utilizar pasta de soldadura.

Debido a lo anterior, todos estos componentes se pueden soldar de forma convencional, utilizando un soldador y estaño:

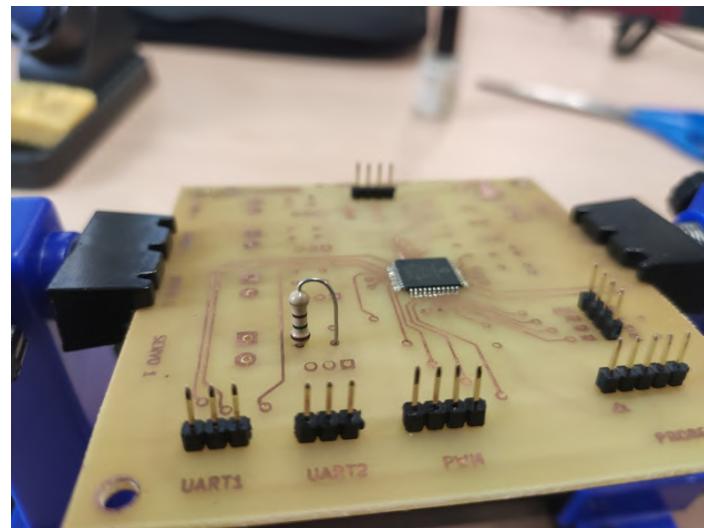


Figura 6.123: Comienzo del proceso de soldadura.

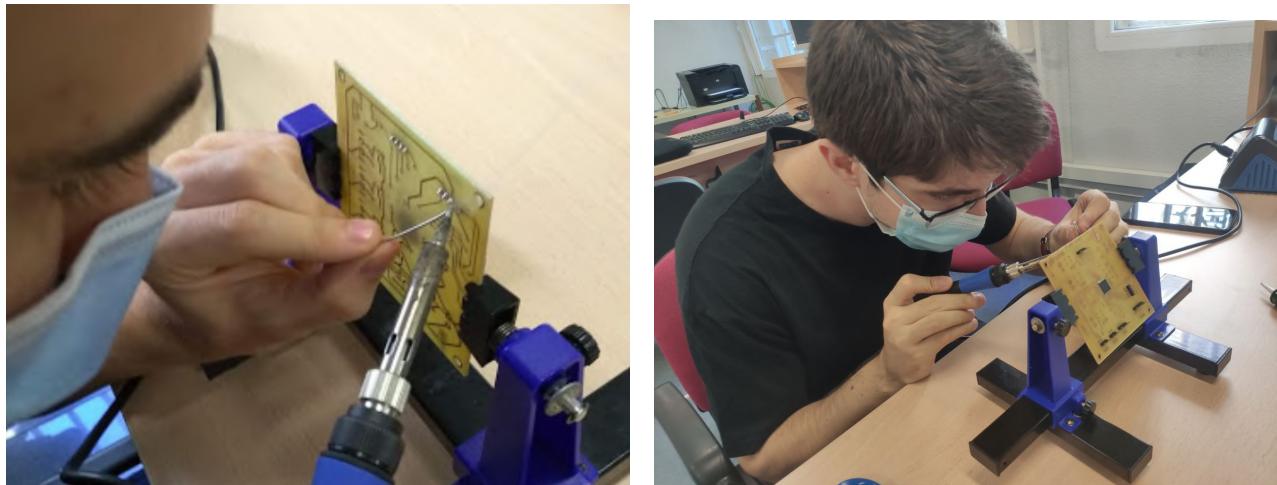


Figura 6.124: Integrantes del equipo soldando componentes.

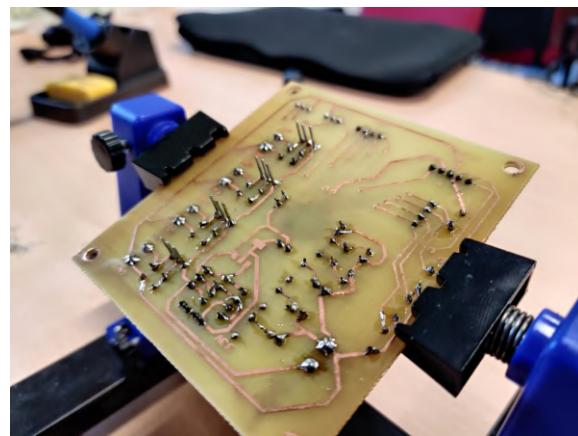
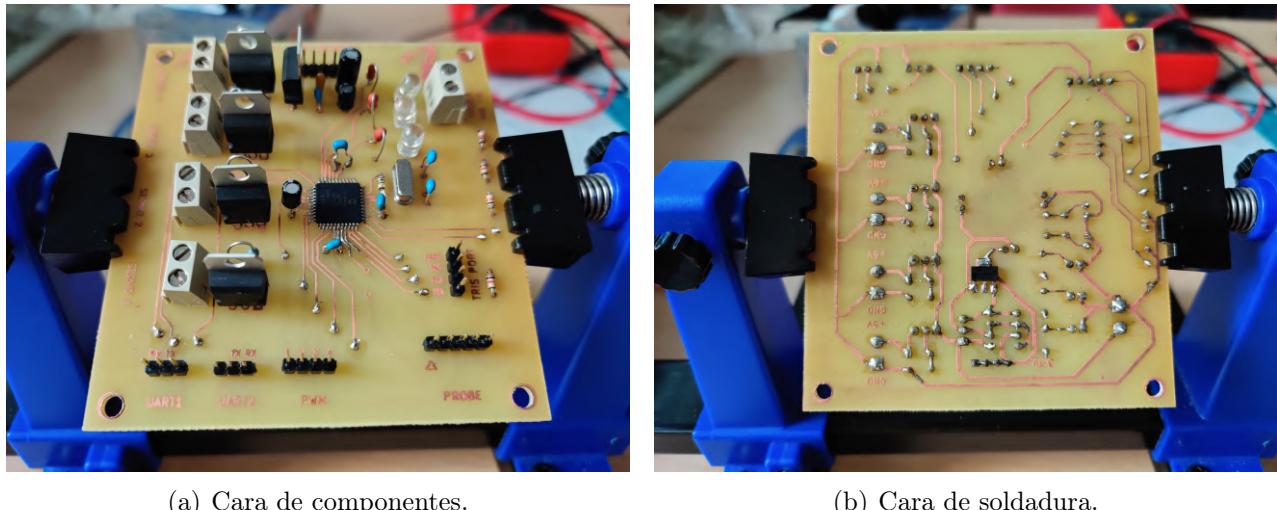


Figura 6.125: Proceso de soldadura en curso.

Tras finalizar el proceso de soldadura de los componentes y de haber recortado el exceso de sus pines, se obtiene el resultado final:



(a) Cara de componentes.

(b) Cara de soldadura.

Figura 6.126: Integrantes del equipo soldando componentes.

Cabe destacar que, tras el proceso de soldadura es recomendable hacer las siguientes comprobaciones:

- Verificar si las soldaduras son limpias y brillantes.
- Verificar si las soldaduras conducen correctamente
- Eliminar excesos de estaño que puedan producir cortos.

En este punto se considera que el proceso de fabricación de la PCB ha terminado, a falta de realizar el proceso de verificación pendiente y que se muestra en el siguiente apartado.

6.5.8. Verificaciones del prototipo construido

Tras completar la fabricación de la PCB, es recomendable realizar algunas comprobaciones mínimas para asegurarse de que el prototipo construido opera:

- Al completar la impresión del circuito y antes de soldar los componentes se debe verificar la correcta conductividad de las pistas de cobre, así como la búsqueda de cortocircuitos o fracturas. Para esta labor se debe utilizar un voltímetro.
- Posteriormente, pero antes de soldar los componentes, se debe verificar la correcta conducción de las distintas redes según lo descrito en el diagrama lógico. Para esta labor se debe utilizar un voltímetro.
- En el momento de soldar los componentes, se debe verificar su correcta inserción en los orificios de los *pads*, verificando también que su orientación y colocación de pines sea la correcta.

- Tras realizar todos las verificaciones anteriores y una vez se ha completado el proceso de soldadura de los componentes, se debe realizar el primer encendido del microcontrolador y el testeo de todos los componentes de la PCB.

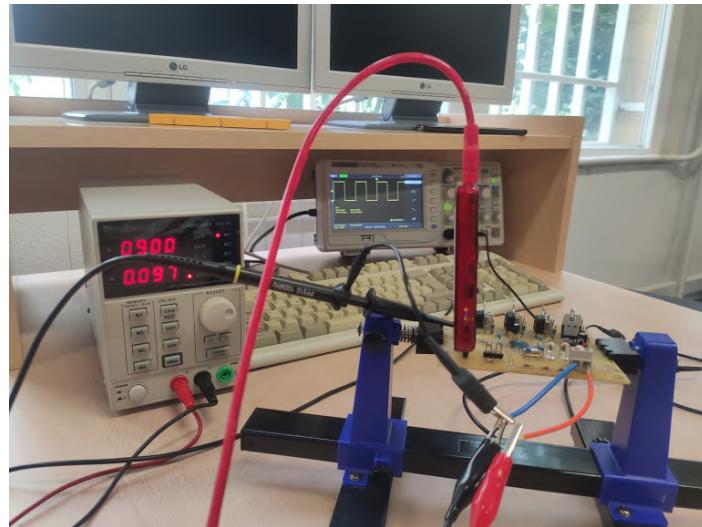


Figura 6.127: Primer encendido de la PCB usando un código de prueba del PWM.

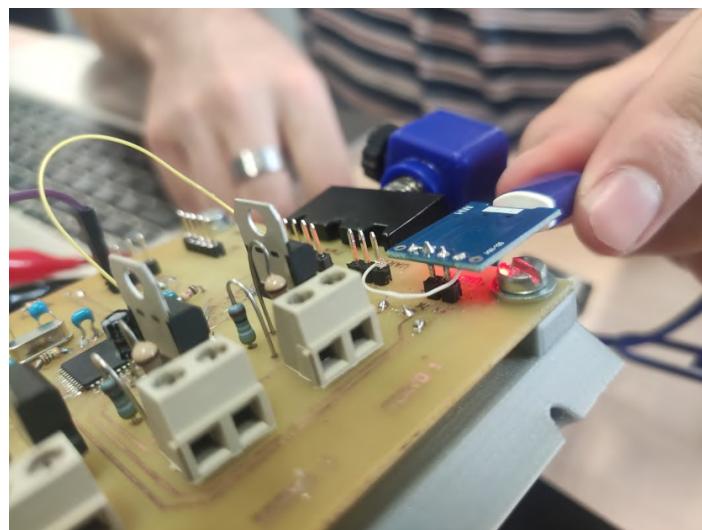


Figura 6.128: Prueba del funcionamiento de la UART.

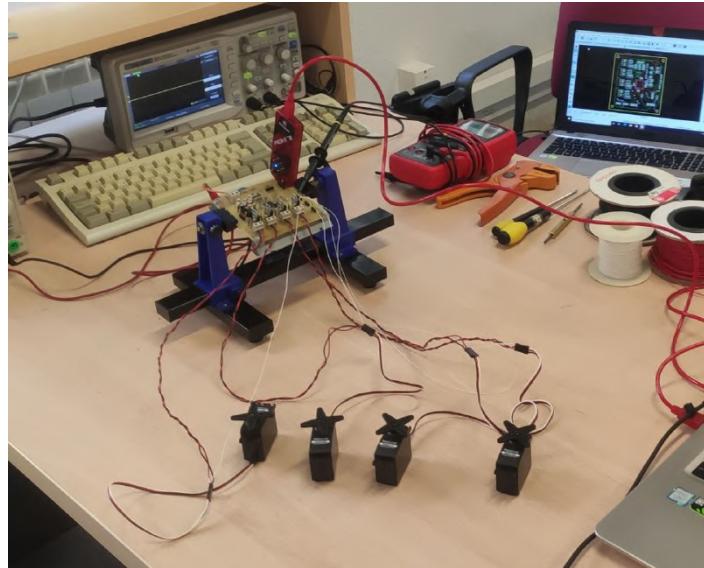


Figura 6.129: Prueba del funcionamiento de los servomotores.

Todas las pruebas realizadas dieron buen resultado y gracias a ellas se descubrieron alguno de los contratiempos descritos en el apartado siguiente.

6.5.9. Contratiempos ocurridos y soluciones planteadas

El proceso de fabricación de la PCB ha sido uno de las etapas más críticas dentro del proyecto y se considera importante comentar algunos de los contratiempos que han ocurrido, puesto que estos han tenido un impacto considerable en el desarrollo del proyecto.

En general, estos contratiempos han ralentizado el proceso de fabricación. Sin embargo, mediante su detección y correcta solución, se ha obtenido una PCB mucho más robusta y segura, ya que se han revelado aspectos negativos que de otra forma habrían pasado desapercibidos.

A continuación se muestran los principales contratiempos ocurridos y las soluciones que se han llevado a cabo para enmendarlos:

- El primer contratiempo sucedido durante el proceso de fabricación está relacionado con el proceso de atacado de la PCB.

La efectividad de esta disolución de atacado sobre la PCB puede verse afectada en función de la temperatura a la que se realiza el proceso, saturación de la disolución y otros factores específicos.

Debido a lo expuesto anteriormente, se realizaron dos intentos fallidos de atacado de la PCB, en los cuales la disolución no consiguió eliminar la capa de cobre de los lugares que no estaban protegidas por resina. El resultado obtenido fue el siguiente:

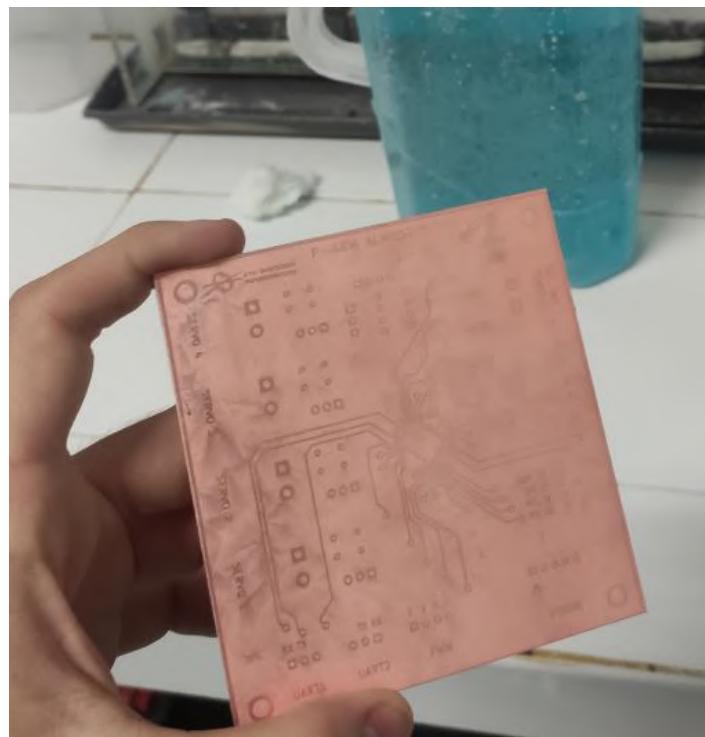


Figura 6.130: PCB tras proceso de atacado fallido.

Tras cada uno de los intentos fallidos, se tuvo que repetir el proceso de insolación de las fotomáscaras sobre una nueva placa positiva de prototipado.

Después de dos intentos fallidos, se decidió desechar la disolución de atacado, ya que esta llevaba bastante tiempo fabricada y podría estar en mal estado. Posteriormente, se creó una nueva disolución de atacado:



Figura 6.131: Proceso de creación de la disolución de atacado.

Tras el atacado con la nueva disolución, el proceso tuvo éxito y el circuito se imprimió correctamente en la placa de prototipado.

A continuación se muestran las PCB obtenidas durante esta etapa:

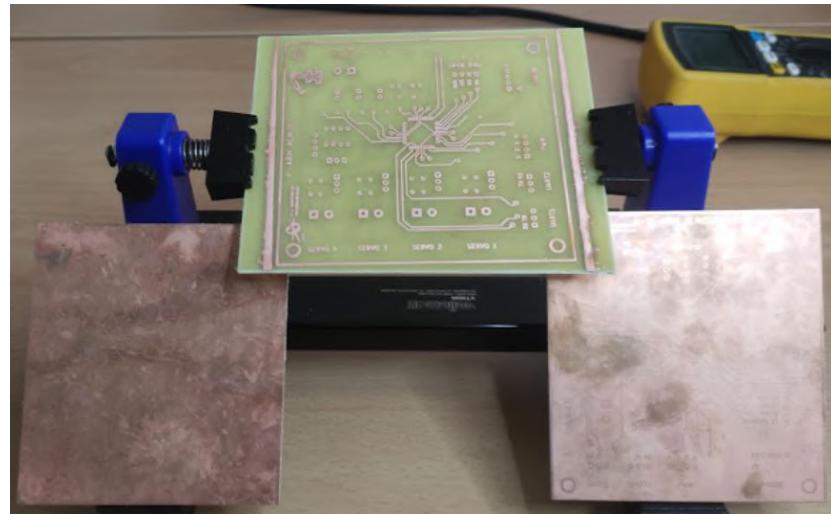


Figura 6.132: Intentos de fabricación de la PCB.

Tal y como se puede observar, en el centro de la imagen se encuentra la PCB exitosa y en los laterales los intentos fallidos, en los cuales el cobre no se eliminó adecuadamente.

- Otro de los contratiempos sucedidos fue detectado tras el proceso de soldado de los componentes, inmediatamente después de realizar el primer encendido del microcontrolador. Este contratiempo afecta al periférico PWM por partida doble, ya que se detectó una fractura en una de las pistas de comunicación del puerto PWM y, a su vez, se comprobó que el módulo PWM interno que usaba esta pista también estaba defectuoso y era inservible.

En la siguiente imagen (ver imagen 6.133) se puede apreciar la fractura en la pista de cobre:



Figura 6.133: Fractura en pista de cobre.

En primer lugar, se procedió a realizar una nueva conexión de la pista usando hilo de grapinar:



Figura 6.134: Parcheo de la pista usando hilo de grapinar.

Tras verificar que la conductividad del nuevo hilo era correcta, se detectó que el módulo PWM seguía sin funcionar a pesar de que la fractura se había solucionado. Posteriormente, tras realizar algunas pruebas al respecto usando un osciloscopio, se concluyó que el módulo estaba defectuoso y no funcionaba. Esto además fue comprobado y corroborado por el tutor, por lo que se optó por utilizar un hilo de grapinar para reconectar el puerto PWM inservible a otro de los pines del microcontrolador, el cual usaba un módulo PWM que funcionaba correctamente y que no estaba siendo usado. Se obtuvo el siguiente resultado:

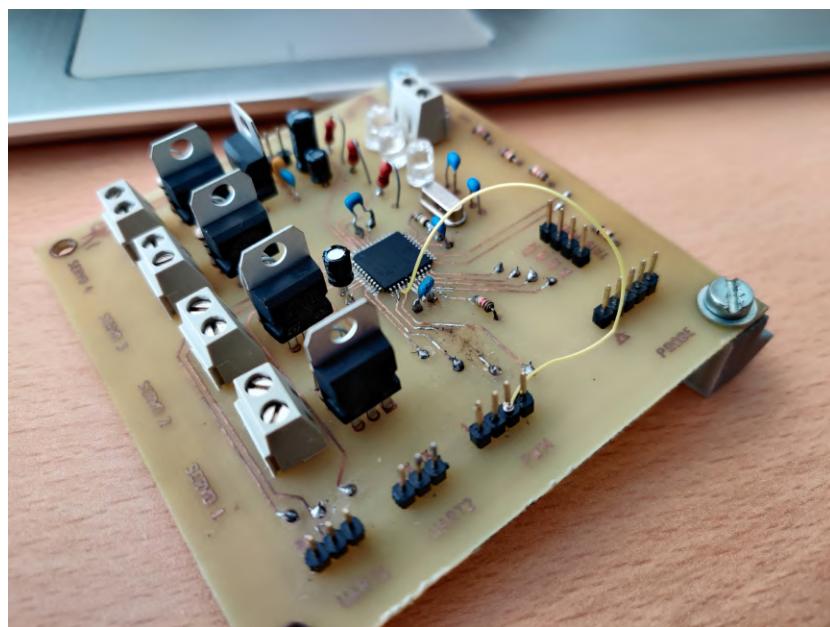


Figura 6.135: Reconexión del nuevo módulo PWM al puerto.

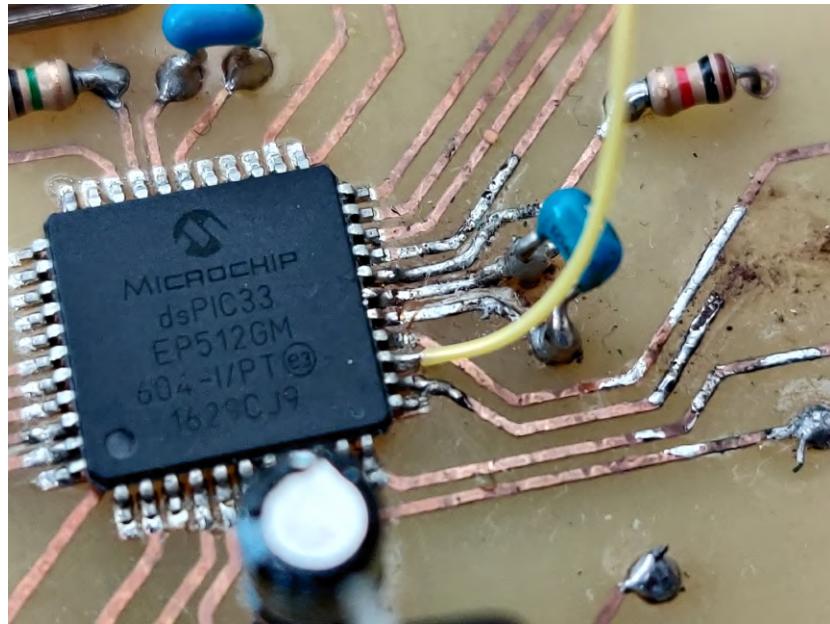


Figura 6.136: Soldadura del hilo de reconexión con el nuevo pin.

Tal y como se puede observar en las figuras anteriores (imágenes 6.135 y 6.136), el reducido tamaño de la soldadura requirió que este proceso fuese realizado con máxima precisión. Además, por precaución, se eliminó parte de la pista original para evitar errores en la señal.

A pesar de que esta solución es de carácter artesanal y lo recomendable hubiese sido fabricar de nuevo la PCB, se tomó la decisión de realizar este arreglo para reducir el impacto del contratiempo en el proyecto, ya que además se comprobó un buen resultado y fue suficientemente segura.

- El último de los contratiempos fue encontrado durante el proceso de soldado de la PCB y afecta al circuito de alimentación de los motores.

Tal y como se puede ver en otros apartados de este documento, los servomotores son alimentados mediante una clema, la cual obtiene su voltaje de los reguladores LM317 que adaptan el voltaje de alimentación de entrada.

Tras el soldado de la clema y los reguladores, se alimentó a la PCB para verificar que el voltaje de salida por las clemas fuese el requerido. Al realizar la medición del voltaje que había en los contactos de las clemas usando un voltímetro se detectó que este era muy superior al que debería ser.

El voltaje que deberían ofrecer las clemas es de $5,41V$ y, sin embargo al realizar la medición, se obtuvieron $7,1V$, un valor intolerable ya que los servomotores no pueden soportarlo. Tras revisar la zona del circuito de alimentación, es decir, clemas y reguladores de tensión, se detectó que los reguladores de tensión LM317 no estaban funcionando correctamente.

Posteriormente, se procedió a realizar una revisión en profundidad de la zona, pistas cercanas, las soldaduras de las clemas y reguladores, sin embargo, no se encontró ningún fallo de fabricación o ensamblaje.

Dado que el fallo no parecía deberse al proceso de fabricación, se procedió a revisar de nuevo los diseños lógico y físico de la PCB. Tras esta revisión, se concluyó que el conexionado del diagrama lógico era correcto pero la huella física elegida para el regulador de tensión LM317 en el diagrama físico presentaba un orden en los pines diferente al del utilizado en el prototipo. Se muestran las imágenes en las que se ve claramente esta diferencia:

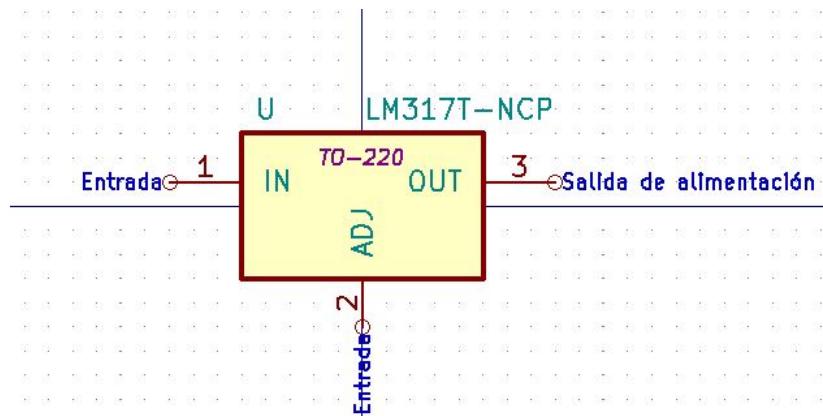


Figura 6.137: Esquemático lógico del LM317.

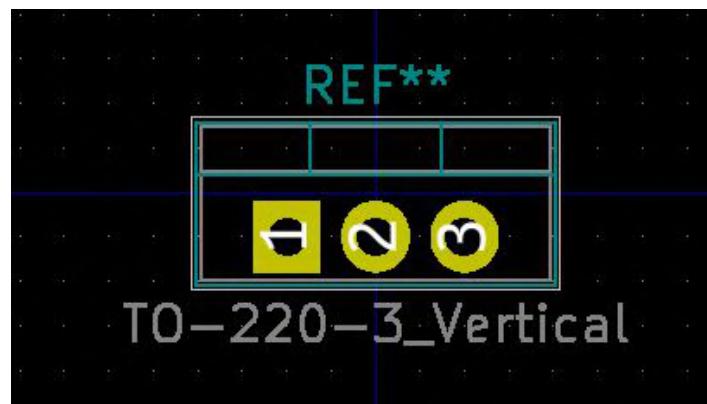


Figura 6.138: Huella física del LM317.

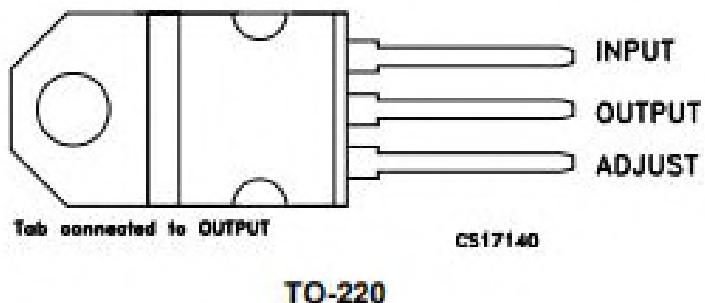


Figura 6.139: Encapsulado físico del LM317.

Tal y como se puede observar en las imágenes anteriores (imágenes 6.137, 6.138 y 6.139), se comprueba que el diagrama físico de la PCB contenía un error en el conexionado de los reguladores LM317 que alimentan los servomotores y, por ello, el voltaje de salida de las clemas era de 7,1V y no de 5,41V, tal y como se había diseñado.

El fallo cometido, fue seleccionar incorrectamente la variante de TO-220 necesaria para el regulador utilizado y no haber realizado correctamente el proceso de verificación en cuanto a orden en los pines en el diseño físico.

Como primer paso para solucionar dicho problema, se procedió a editar la huella física del componente y se intercambiaron los pines:

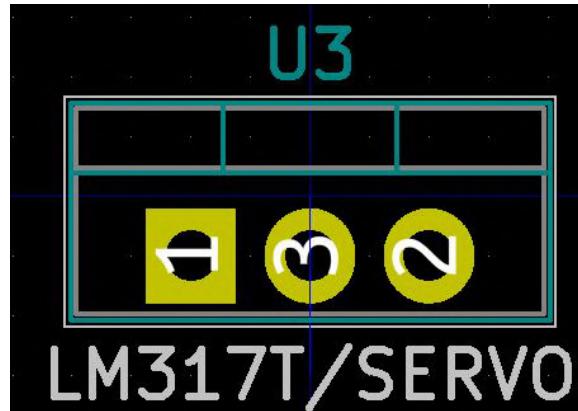


Figura 6.140: Huella física del LM317 con pines reordenados.

Puesto que se habían intercambiado dos pines de la huella física, las pistas de conexión de los mismos debían ser reconectadas para arreglar el problema:

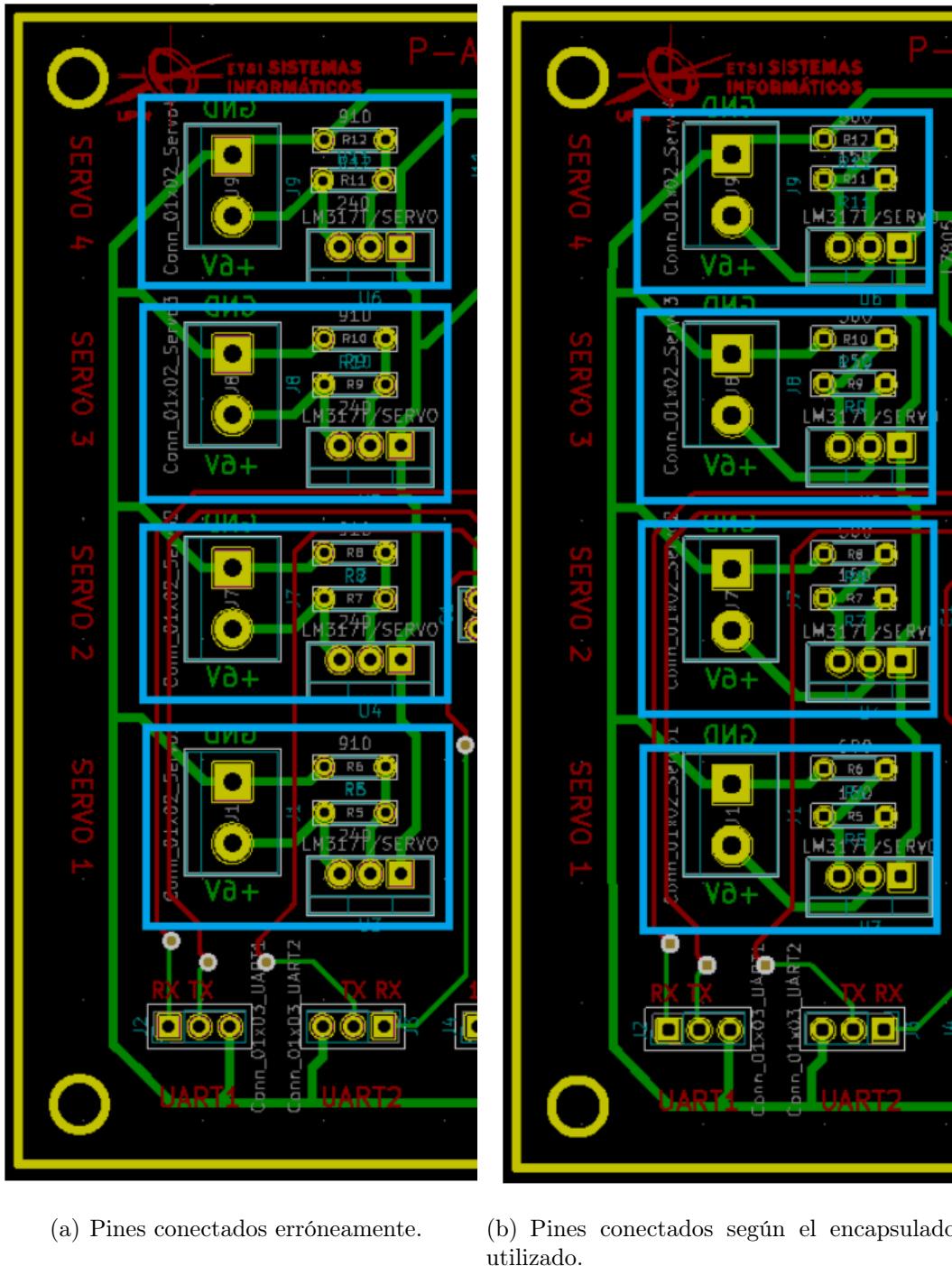


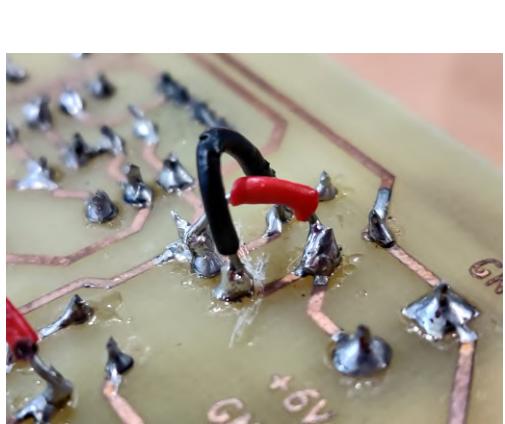
Figura 6.141: Modificación del circuito de alimentación de los servomotores, marcada en azul.

Esta modificación del diagrama físico de la PCB hecha a posteriori ya se contempla en el apartado 6.5.5 de la memoria, dado que en dicho apartado se muestra la versión final del diagrama físico.

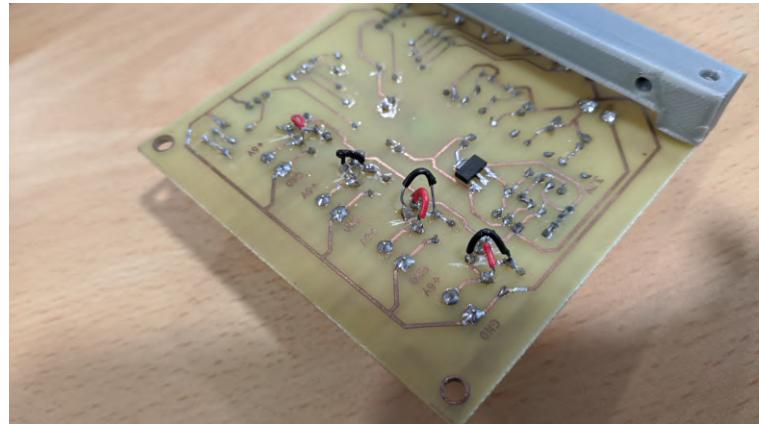
Como segundo paso para solucionar el problema y puesto que la PCB ya había sido fabricada siguiendo el diagrama físico que contenía el fallo se debía de ingeniar alguna solución para el prototipo construido. A pesar de que se planteó la posibilidad de fa-

bricar de nuevo la PCB, se consideró que este proceso causaría demasiado retraso en el desarrollo del proyecto y por lo tanto se optó por ingeniar otra solución directamente sobre el prototipo que estaba fabricado y ensamblado.

Puesto que el fallo consistía en dos pines que deberían estar intercambiados, bastaría con intercambiar las pistas de comunicación que conectaban con ellos. Se decidió, de esta forma, fracturar las pistas existentes para después colocar dos puentes, intercambiando de esta forma el conexionado de los pines con sus pistas (ver imágenes 6.142).



(a) Vista detallada del puente.



(b) Vista de los cuatro puentes realizados

Figura 6.142: Puentes entre pistas de los reguladores de tensión LM317.

Para la realización se emplearon cables de tamaño reducido que actuaban como puente entre las pistas. Por su tamaño y dado que el soldado de los mismos se realizó a mano, se considera que el resultado podría ser mejorable, sin embargo, esta solución dió buenos resultados y corrigió correctamente el voltaje de salida de los reguladores de tensión LM317.

A pesar de todos los errores cometidos durante el proceso de fabricación y diseño de la PCB, los integrantes del equipo han sido capaces de:

- Ingeniar soluciones apropiadas que generasen el mínimo impacto en el desarrollo del proyecto.
- Implementar dichas soluciones en el prototipo ya fabricado.
- Verificar que la corrección es apropiada y que daba el resultado buscado y esperado en un principio.

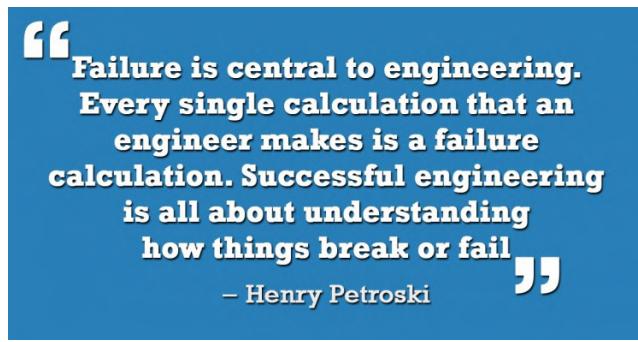


Figura 6.143: Reflexión [40].

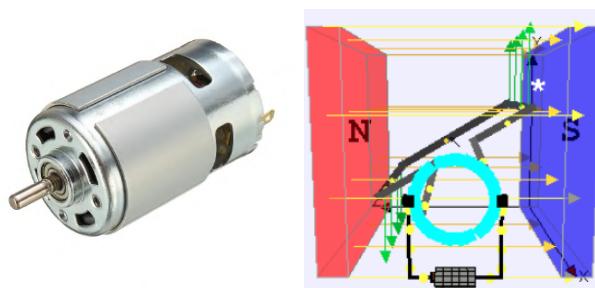
6.6. Motores empleados (actuadores)

Dado que en este proyecto se ha planteado la construcción de un manipulador o brazo robótico, los únicos actuadores empleados han sido motores, los cuales dotan de movilidad a la estructura física del brazo.

Cabe destacar que, debido a la forma de la estructura física del brazo y dado a que el mismo tiene tres articulaciones móviles, se han empleado tres motores principales en cada una de ellas y un motor auxiliar en el extremo del manipulador.

Existen varios tipos de motores eléctricos que pueden ser usados para dotar de movilidad a proyectos de robótica de pequeña escala. Sin embargo los principales tipos se pueden agrupar en las siguientes tres categorías:

- Motores de corriente continua: son los motores eléctricos más sencillos y básicos. Debido a esto, realizar el control de la posición angular del eje y su velocidad de rotación es complicado y requiere aplicar técnicas de control de lazo cerrado. Además, el control físico de este tipo de motores se lleva a cabo mediante una señal PWM actuando sobre un puente H.

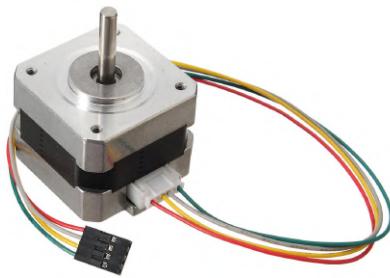


(a) Motor DC real [41] (b) Funcionamiento [42]

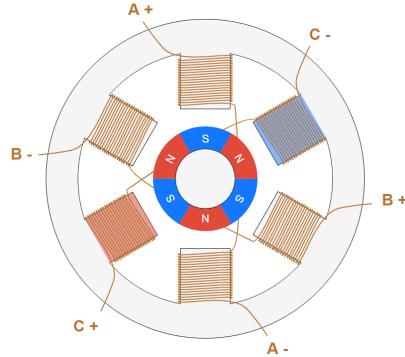
Figura 6.144: Motor de corriente continua

- Motores paso a paso: se trata de motores eléctricos más complejos que ofrecen una precisión muy alta en cuanto al control de posición y velocidad, ya que descomponen su

movimiento en pasos de longitud constante. En este tipo de motores se puede realizar control de velocidad y posición del eje del motor mediante técnicas de control de lazo abierto, dado que en este tipo de motores se controla el número de pasos que da el motor, así como cada cuánto tiempo se produce un paso. Este tipo de motores necesitan un manejador para gestionar las señales de cada fase de las bobinas del estator, que puede requerir algo más de esfuerzo del necesario para otros motores más sencillos.



(a) Motor real [43]



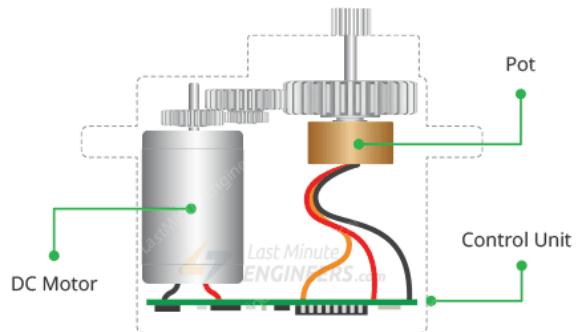
(b) Funcionamiento [44]

Figura 6.145: Motor paso a paso

- Servomotores: se trata de motores de corriente continua que incorporan un sistema de control de posición de lazo cerrado. Por ello, este tipo de motores ofrecen un control muy simple de la posición angular del eje del motor. A través de una señal PWM enviada al motor, se puede establecer una posición consigna que el eje del motor debe cumplir. Estos motores incluyen un sensor de posición (encoder, potenciómetro solidario al eje o similar) que determina la posición angular del eje del motor y una unidad de control que verifica la posición actual del eje en comparación con la posición de consigna establecida, realizando las correcciones necesarias hasta alcanzar dicha posición angular.



(a) Servomotor real [45]



(b) Funcionamiento [46]

Figura 6.146: Servomotor de corriente continua

Tras analizar los diferentes tipos de motores anteriormente expuestos, se ha decidido utilizar

servomotores para dotar de movilidad al brazo robótico. Esta decisión se fundamenta en los siguientes motivos:

- A diferencia de los motores paso a paso o motores de corriente continua, no se suele necesitar ningún tipo de circuito externo, driver o puente H para controlar un servomotor; únicamente se debe alimentar el motor y proporcionar una señal de control.
- Este tipo de motores ofrece un control de posición preciso y simple mediante una señal PWM. A pesar de que dicho control de posición se realiza mediante lazo cerrado internamente dentro del motor, desde un punto de vista externo, no se necesita ningún tipo de realimentación externa.

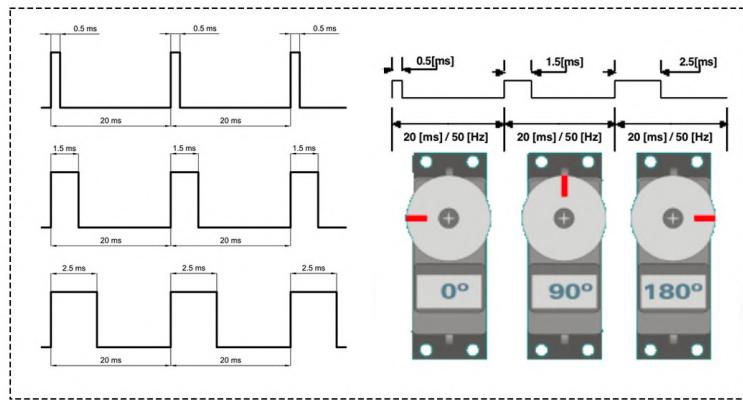


Figura 6.147: Ejemplo genérico de control de posición mediante señal PWM [47].

- Se trata de motores que se adaptan muy bien para proyectos de robótica de pequeña escala, debido a su bajo coste y sencillez de uso.
- Este tipo de motores está muy extendido en el mercado y existen numerosos modelos con diferentes potencias, tamaños, etc.

Es importante destacar que existen dos tipos de servomotores:

- Servomotores de giro limitado: son aquellos servomotores que tienen un rango de rotación limitado, el cual suele ser normalmente de 180º. Son el tipo de servomotor más sencillo.
- Servomotores de giro continuo: son aquellos servomotores que tienen rango completo de giro, es decir, pueden realizar giros sin limitación de recorrido.

Dado que ninguna de las articulaciones del motores está diseñada para realizar giros de más de 180º, se han empleado servomotores de giro limitado.

Otro de los datos que es importante clarificar antes de tomar la decisión de que motores van a ser usados en un proyecto de robótica, es la carga máxima que va a tener que desplazar el manipulador robótico. Este dato afecta principalmente al diseño de la estructura física del brazo y a la potencia de los motores escogidos, en especial, el torque que ejercen.

Finalmente, el modelo de servomotor elegido para las articulaciones ha sido el *Parallax 900-00005 Standard Servo* el cual tiene las siguientes características técnicas:

- Servomotor de rango limitado de 180° .
- Control mediante señal PWM de 50Hz.
- Alimentación de entre $4V$ y $6V$, utilizando entre $15mA$ y $200mA$. Potencia nominal de $140mA$.
- Torque máximo ejercido de $27N \cdot cm$, es decir aproximadamente $2,75Kgf \cdot cm$.
- Conociendo el torque ejercido por los servomotores y área de trabajo del manipulador se pueden realizar algunos cálculos para deducir cual será la carga máxima que podrá soportar el brazo robótico:
 - En la zona de trabajo en la cual el brazo robótico está menos extendido, y por lo tanto situación en la que el esfuerzo es mínimo sobre la estructura del manipulador, los motores aplican su fuerza a 8.1 cm del extremo del robot. Dado que el torque es generado es de aproximadamente $2,75Kgf \cdot cm$, se podría levantar una masa de aproximadamente 300g.
 - En la zona de trabajo en la cual el brazo robótico está más extendido, y por lo tanto situación en la que el esfuerzo es máximo sobre la estructura del robot, los motores aplican su fuerza a 34.6 cm del extremo del robot. Dado que el torque es generado es de aproximadamente $2,75Kgf \cdot cm$, se podría levantar una masa de aproximadamente 80g.
 - Teniendo en cuenta los cálculos anteriores, se recomienda que la carga máxima del manipulador sea de entre 150g y 60g, siempre teniendo en cuenta las zonas de trabajo en las que se vaya a desplazar la carga para tener garantías de que el desplazamiento es seguro.
- Peso de 44g.
- Dimensiones 406 x 55,8 x 19 mm



Figura 6.148: Servomotor Parallax utilizado [48]

Teniendo en cuenta los datos técnicos anteriores, este modelo de servomotor se adapta perfectamente a las características del brazo robótico que se ha desarrollado, cumpliendo todas la cualidades deseadas para que el funcionamiento del brazo robótico sea correcto.

Capítulo 7

Software

El proyecto tiene una clara división en cuanto al SW implementado en función de los diferentes dispositivos usados para el control del brazo robótico. Por un lado, se ha desarrollado un SW de alto nivel implementado en Python, el cual permite la interacción directa de un operador humano con el brazo robótico mediante una interfaz de usuario. Este SW se ejecuta en S1.

Por otro lado, y de manera concurrente, se ha desarrollado un SW implementado en C, que será cargado en la placa de control y cuyo propósito es generar las señales necesarias para mover los motores, así como interpretar las órdenes de movimiento que lleguen de S1. Este SW se ejecuta en S2.

Para comunicar estos dos sistemas se ha desarrollado un pseudo lenguaje basado en GCode el cual servirá para agilizar las comunicaciones y simplificar el envío y recepción de datos como posiciones o mensajes de error.

Cabe destacar que la comunicación es completamente asíncrona. Esto se debe a que los dos sistemas se comunican mediante el estándar UART, el cual es asíncrono, y permite además poder seguir comunicando los dos sistemas mientras estos realizan otras tareas distintas a la comunicación. Por ejemplo, S2 debe ser capaz de poder escuchar el puerto UART mientras mueve los motores. Por otro lado, S1 debe ser capaz de seguir permitiendo la interacción con la interfaz de usuario mientras está mandando una orden a S1.

7.1. S1

El SW de S1 se ha desarrollado íntegramente usando el lenguaje Python y los editores de código Visual Studio Code, así como el IDE PyCharm.

Es de vital importancia destacar que el SW de S1 esta formado por dos secciones diferenciadas: código de la interfaz gráfica de usuario (GUI) y código que gestiona la lógica de las comunicaciones entre S1 y S2.

Puesto que esta división se considera esencial, cada una de las secciones del código de S1 será descrita en detalle de forma independiente en los apartados siguientes.

7.1.1. Interfaz Gráfica de Usuario (GUI)

La interfaz gráfica de usuario ha sido desarrollada haciendo uso de la biblioteca gráfica multiplataforma Qt, la cual se encuentra originalmente implementada en C++ y que ha sido adaptada a Python, siendo conocida también como *PyQt*. Esta librería permite un desarrollo rápido y sencillo de interfaces gráficas de usuario mediante el paradigma de programación orientada a objetos.

En primer lugar, es importante destacar las principales razones que han desembocado en la elección de esta librería:

- La curva de aprendizaje es rápida y permite desarrollar interfaces gráficas de forma ágil y eficaz.
- La librería dispone de abundante documentación, la cual facilita su uso en el proyecto.
- Qt cuenta con una gran variedad de componentes gráficos y herramientas, las cuales cubren de sobra las aspiraciones de este proyecto.
- Es una de las librerías más usadas por la comunidad de desarrolladores en la programación de interfaces gráficas empleando Python.
- Proporciona una herramienta de diseño gráfico de interfaces de usuario, llamada *Qt Designer*, que agiliza las etapas tempranas de desarrollo de la interfaz gráfica, sobre todo en cuanto al ámbito de la distribución gráfica de los distintos *widgets*, botones, desplegables, etc.

En términos generales, la interfaz gráfica de usuario se considera un componente esencial del proyecto y su función es la de facilitar la interacción del usuario con el brazo robótico. Tal y como se ha podido ver en apartados anteriores de esta memoria, la interfaz gráfica ofrece al usuario la posibilidad de:

- Controlar el movimiento del brazo robótico mediante el control de algunos parámetros del mismo, por ejemplo, mediante la posición angular de los servomotores o la posición cartesiana del *end-effector* del brazo.
- Visualizar gráficamente una previsualización de la posición del brazo antes de que el mismo realice el movimiento físicamente.
- Gestionar la comunicación de S1 y S2 a través de la selección de un puerto serie.
- Mostrar información sobre la ejecución de la aplicación y el estado del sistema.

En relación a lo anteriormente mencionado, se planteó un diseño teórico que aproximaba la apariencia de la interfaz gráfica de usuario según los requisitos planteados (7.1):

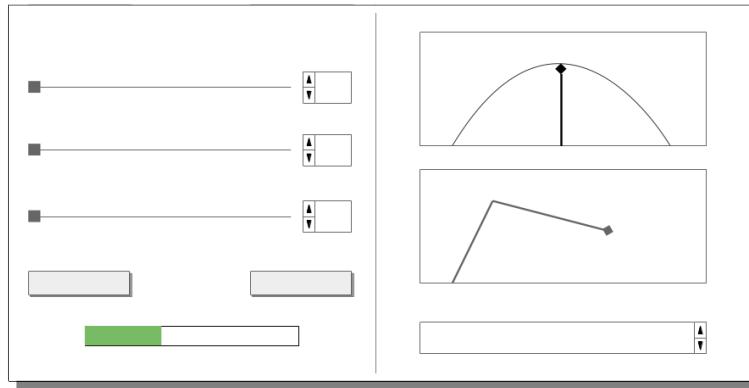


Figura 7.1: Diseño propuesto para la interfaz gráfica de usuario.

En el diseño anterior se pueden identificar algunos elementos cruciales que deben trasladarse inequívocamente a la interfaz gráfica de usuario final:

- *Sliders* que permitan al usuario visualizar y modificar los valores de las coordenadas angulares de los motores, así como de las coordenadas cartesianas del *end-effector*.
- *Spin Boxes* que permitan al usuario visualizar y modificar el valor numérico representado por el *Slider*.
- Botones que permitan al usuario ejecutar un movimiento, así como seleccionar el tipo de control de los parámetros del brazo (ángulos de giro o posición cartesiana).
- Representaciones gráficas de la posición del brazo robótico tras realizar un movimiento generado por el usuario.
- Una barra de progreso que muestre el estado del movimiento que está realizando el brazo robótico.
- Una consola que muestre los *logs* que genera la aplicación del sistema S1.

Teniendo en cuenta el diseño teórico propuesto previamente a desarrollar la interfaz gráfica de usuario, se procede a implementar dicho diseño utilizando la herramienta *Qt Designer* (figura 7.2):

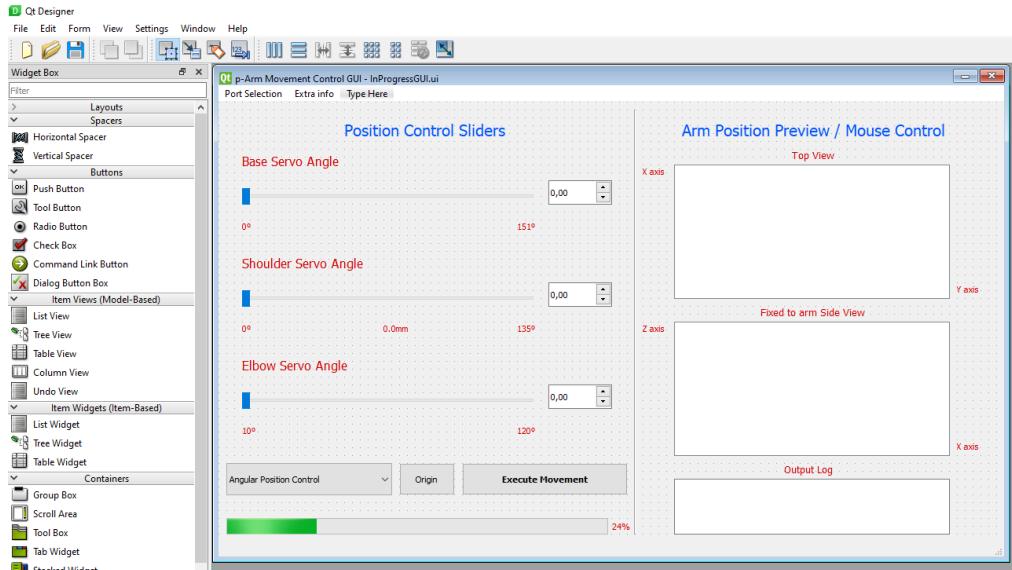


Figura 7.2: Diseño final para la interfaz gráfica de usuario.

Mediante la herramienta anterior, se puede diseñar gráficamente cuál va a ser la apariencia de la interfaz. Tal y como se puede ver en la parte izquierda de la imagen, Qt proporciona numerosos componentes gráficos de todos los tipos, los cuales pueden ser incluidos en el diseño con tan solo arrastrarlos al mismo. Esta herramienta de diseño, proporciona un archivo de salida XML que tiene extensión .ui y que posteriormente será cargado desde Python para la programación de la lógica que existe entre los distintos componentes gráficos, los cuales, son tratados como objetos de un cierto tipo en el código.

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
<class>MainWindow</class>
<widget class="QMainWindow" name="MainWindow">
<property name="geometry">
<rect>
<x>0</x>
<y>0</y>
<width>1000</width>
<height>600</height>
</rect>
</property>
<property name="minimumSize">
<size>
<width>1000</width>
<height>600</height>
</size>
</property>
<property name="maximumSize">
<size>
<width>1000</width>
<height>600</height>
</size>
</property>
<property name="palette">
```

Figura 7.3: Fragmento del archivo XML que describe la apariencia de la GUI.

A continuación se presenta una breve descripción de los componentes gráficos que aparecen en el diseño final de la interfaz de usuario, además, posteriormente se comentarán los detalles técnicos más relevantes en profundidad.

- Por un lado, en la sección izquierda de la interfaz gráfica de usuario se encuentran los siguientes componentes:
 - En la barra de herramientas de la parte superior se encuentran el menú desplegable de selección de puerto serie, así como el menú desplegable que da a conocer la documentación del proyecto al usuario.
 - En la parte central, tal y como describe el título “*Position Control Sliders*”, se encuentran los *Sliders* y *SpinBoxes* que permiten al usuario controlar el valor de las coordenadas angulares de los motores y las coordenadas cartesianas del *end-effector*.
 - Justo debajo de los *Sliders*, se encuentran tres botones que permiten al usuario: cambiar el tipo de coordenadas de los *Sliders*, devolver al brazo robótico a su posición inicial y ejecutar o cancelar un movimiento.
 - En la parte inferior de la mitad izquierda, se encuentra la barra de progreso que indica el estado del movimiento en el momento de la realización del mismo. Esta barra solo aparece mientras se está realizando un movimiento.
- Por otro lado, en la sección derecha de la interfaz gráfica de usuario se encuentran los siguientes componentes:
 - Tal y como describe el título, en esta parte de la interfaz se encuentran las dos representaciones gráficas de la posición del brazo robótico, las cuales además, permiten al usuario controlar la posición del brazo mediante movimientos del ratón.
 - Justo debajo de ambas representaciones, se encuentra la consola de salida que proporciona al usuario la información sobre los *logs* generados por la aplicación.

En relación a los componentes gráficos anteriores, es importante destacar algunas decisiones esenciales que han sido tomadas durante el desarrollo de la interfaz gráfica y que por lo tanto han condicionado el resultado final:

En primer lugar, inicialmente se disponía de unos rangos de valor teóricos para cada uno de los *Sliders* y *SpinBoxes*, los cuales, tras realizar algunas pruebas físicas con el prototipo del brazo robótico, han sido reajustados para cumplir con las limitaciones físicas y de seguridad de la estructura:

- Cuando los controles se encuentran en modo de coordenadas angulares, los tres *Sliders* y *SpinBoxes* representan a los ángulos de giro ($\theta_0, \theta_1, \theta_2$). Estos ángulos tienen los siguientes rangos de valores:
 - El ángulo θ_0 es ajustable entre 0° y 151° , que coincide con el rango de giro máximo del servomotor de la base.
 - El ángulo θ_1 es ajustable entre 0° y 135° , este rango se deduce de las limitaciones físicas de la estructura del brazo.

- El ángulo θ_2 es ajustable entre 10° y 120° , este rango se deduce de las limitaciones físicas de la estructura del brazo.



Figura 7.4: Controles en modo coordenadas angulares.

- Cuando los controles se encuentran en modo de coordenadas cartesianas, los tres *Sliders* y *SpinBoxes* representan a la posición cartesiana del *end-effector* del brazo (x, y, z). Estas coordenadas tienen los siguientes rangos de valores:

- La coordenada X es ajustable entre 0,0 mm y 346 mm, puesto que el *end-effector* siempre se sitúa delante de la base, coincidiendo con la parte positiva del eje X .
- La coordenada Y es ajustable entre -346 mm y 346 mm, puesto que el *end-effector* puede desplazarse hacia la derecha e izquierda de la base, coincidiendo con la parte positiva y negativa del eje Y .
- La coordenada Z es ajustable entre -106 mm y 360,6 mm, puesto que el *end-effector* puede desplazarse hacia arriba y abajo, teniendo como límite inferior la parte mas baja de la base del brazo y coincidiendo con la parte positiva y negativa del eje Z .

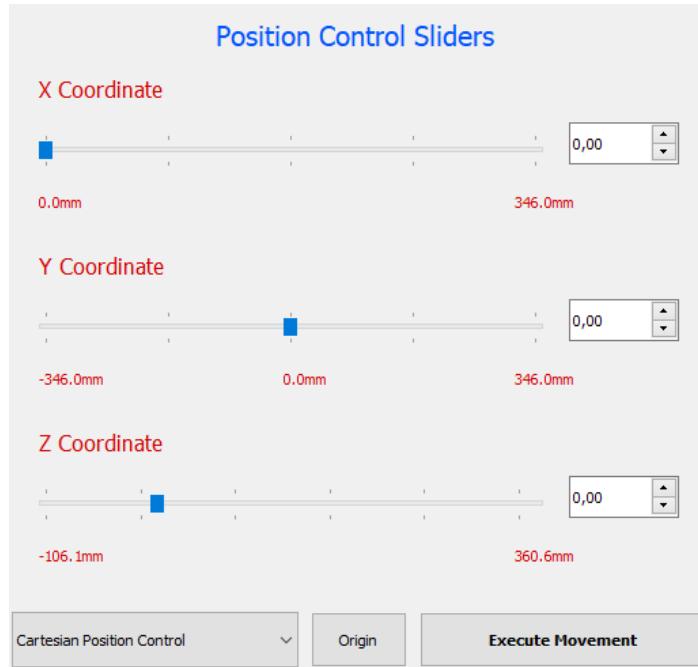


Figura 7.5: Controles en modo coordenadas cartesianas.

En segundo lugar y en relación con las representaciones gráficas del brazo robótico, se ha tomado la decisión de incluir dos vistas:

- Vista cenital o desde arriba: la cual muestra como el ángulo θ_0 o las coordenadas X e Y afectan a la posición del brazo. Esta vista es absoluta y no se encuentra fijada a la estructura del brazo robótico.
- Vista lateral o de perfil fijada: la cual muestra como los ángulos θ_1 y θ_2 o las coordenadas X , Y , Z afectan a la posición del brazo. Esta vista cuenta con una particularidad y es que se encuentra fijada a la estructura del robot y por lo tanto, no contempla el desplazamiento de profundidad del brazo robótico, es decir, el enfoque es siempre completamente paralelo al lateral del brazo.
- En ambas vistas se muestran los dos segmentos del brazo en colores distintos para así distinguirlos con claridad, así como la base sobre la que se encuentra situado el brazo.
- Ambas representaciones gráficas usan los modelos matemáticos de cinemática directa e inversa para realizar los cálculos que permiten dibujar los diferentes puntos y segmentos.

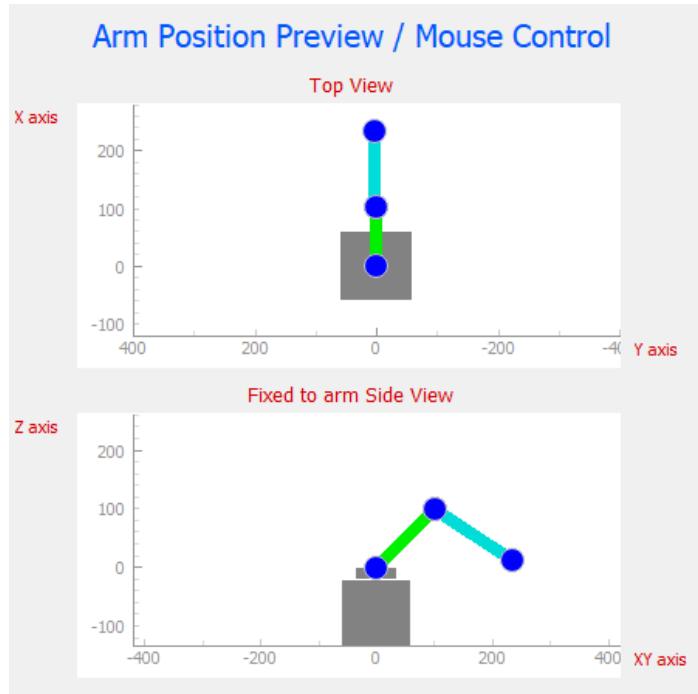


Figura 7.6: Representaciones gráficas de la posición del brazo.

Cabe destacar que, a pesar de que los controles del brazo pueden ser ajustados libremente por el usuario, se puede afirmar que el brazo robótico tiene un rango limitado por su estructura física y por ello, se ha programado un método que verifica si la posición que el usuario ha seleccionado es viable. En caso de serlo, el movimiento puede ser ejecutado y enviado como orden a S2, en caso contrario, la representación del brazo se colorea de color rojo y el botón de ejecutar movimiento se desactiva, de esta forma el usuario es notificado y debe modificar los valores de los controles. Continuando con el proceso de desarrollo de la interfaz gráfica de usuario y una vez se ha creado el diseño gráfico de la misma, el siguiente paso consiste en la programación de la lógica que existe entre los diferentes componentes gráficos, así como sus interacciones. Para la programación de la lógica entre los componentes gráficos, se hace uso de las señales que proporciona la librería Qt.

En la librería gráfica Qt, las señales representan el mecanismo principal para notificar cambios de estado y eventos que afectan a componentes gráficos de una interfaz. El concepto de señal puede ser asemejado con el concepto de interrupción, en el cual cuando sucede cierto evento, se dispara una señal asíncrona o notificación, que puede ser manejada por el programador y que dispara la realización de una cierta rutina. Uno de los ejemplos más claros de este mecanismo, es la señal `clicked()` de un botón del tipo `QPushButton` perteneciente a los *widget* que ofrece Qt: cuando se interactúa con el botón, se dispara una señal que permite al usuario ejecutar una función programada por el mismo tras este evento.

El concepto más importante tras las señales es el hecho de que Qt ofrece la posibilidad de conectar cada una de las posibles señales a la ejecución de un método programado por el desarrollador. De esta forma, mediante estas señales, se puede establecer una lógica que orquesta las interacciones entre los diferentes componentes gráficos, en base a los eventos que vayan ocurriendo.

Teniendo en cuenta el concepto anterior, el código de la interfaz gráfica de usuario consiste en un archivo Python desde el cual se carga el archivo XML que contiene el diseño gráfico de la interfaz, en el cual se han programado los diferentes métodos a ejecutar tras la recepción de las diferentes señales de los distintos componentes gráficos. Este archivo Python contiene a la clase `Ui`, en la cual se realiza la importación del archivo `.ui`, se realiza la configuración inicial de los diferentes componentes de la interfaz y se establece la conexión entre las señales y las rutinas de tratamiento de las mismas.

Como detalle final, es importante remarcar que el punto de enlace entre el código de la interfaz gráfica y el código de la lógica de comunicaciones se realiza mediante la clase denominada “*control interface*”, la cual se describe en el subapartado siguiente. Esta clase proporciona una serie de métodos que, al ser ejecutados dentro de las rutinas de gestión de señales, disparan y ejecutan los métodos necesarios del código de la lógica de comunicaciones que permiten el envío de las ordenes generadas por el usuario a través del canal de comunicación con S2

Finalmente, se muestra una imagen de la interfaz gráfica de usuario final en ejecución:



Figura 7.7: Interfaz de usuario final en ejecución

El código de la interfaz gráfica de usuario se muestra en el anexo y el archivo que lo contiene recibe el nombre de `GUI.py` (listado de código E.9).

7.1.2. Lógica de comunicaciones

Antes de proceder con la explicación del código conviene especificar ciertos aspectos destacados del SW.

Al tener que comunicar S1 con S2, es necesario que existan hilos de cómputo dedicados espe-

cíficamente a las labores de lectura y escritura en la UART. Esto se debe a que no se debe ocupar el hilo principal de cómputo con labores de comunicación ya que esto supondría bloquear la interacción con la interfaz gráfica mientras duren las comunicaciones. Por otro lado, es necesario un hilo especial para poder interactuar con uno de los elemento de la interfaz desde el hilo de comunicación.

Ante esta circunstancia se ha decidido que el SW se ejecutará en hasta 4 hilos de cómputo distintos y concurrentes.

Dichos hilos son:

- El hilo principal: en él se ejecuta la interfaz gráfica y la lógica de control siempre y cuando esta no suponga una comunicación con el S2.
- Hilos dedicados a comunicación: estos hilos se crean bajo demanda al iniciarse una comunicación con S2 y se destruyen tras terminar su labor. Gracias a estos hilos, la interfaz gráfica puede seguir ejecutándose en el hilo principal.
- Hilo dedicado al demonio de pulsos: este hilo se crea tras realizarse la sincronización inicial. Su cometido es mandar mensajes periódicos a través de la UART para indicarle a S2 que S1 aún sigue conectado y que no se ha desconectado.
- Hilo dedicado a la actualización de la barra de progreso: este hilo se encarga de actualizar la barra de progreso que representa el porcentaje del movimiento que ya se ha realizado, creándose en el hilo de comunicación. El motivo por el cual es necesario actualizarlo desde un hilo distinto al principal es que solo S2 conoce cuánto tiempo tardará en realizarse el movimiento y, dado que el hilo principal no puede acceder a este dato hasta que el futuro acabe, es necesario que la interacción se haga desde un hilo que pueda acceder al dato de manera independiente e inmediata.

El modelo de comunicación asíncrona plantea ciertos problemas.

Uno de ellos es el acceso concurrente a recursos. Esto se puede observar en el acceso al canal de comunicaciones. El hilo demonio escribe de manera periódica en el canal de comunicación un mensaje, el cual sirve de pulso. Puede ocurrir que, de manera concurrente, sea necesario que un hilo de comunicación envíe una orden de movimiento a S2.

Si el planificador de tareas del sistema operativo expulsa uno de los hilos en mitad de la comunicación y da paso al otro, los dos mensajes se mezclarían originando un fallo en la comunicación.

Para evitar esta situación, el acceso a la UART es bloqueado siempre que un hilo accede a este recurso y no se desbloquea hasta que el hilo ha terminado la comunicación.

Otro problema es la comunicación de mensajes entre hilos y la sincronización de recepción y envío de estos.

Este problema se origina ante la necesidad de que la interfaz gráfica se mantenga actualizada con los datos que se reciben en el hilo de comunicación. Para solucionarlo se han empleado futuros, los cuales pueden desencadenar la ejecución de una función en el hilo principal, cuando el futuro finaliza su ejecución.

Concretamente, el futuro que se ha implementado en S1 se encarga de ordenar el movimiento del brazo, monitorizarlo, recoger posibles errores, pedir las posiciones reales al acabar el movimiento y finalmente comunicarle dichas posiciones a la interfaz gráfica o bien devolver un error en caso de que este ocurra.

La lógica de control se ha dividido en varios paquetes para encapsular distintas funcionalidades del sistema por separado, que son: “communications”, “control”, “gcode”, “logger”, “security” y “utils”.

A continuación se procede a explicar la funcionalidad de cada uno de los paquetes a un nivel de abstracción alto, junto con una explicación más detallada de cada uno de los ficheros .py.

Además de las explicaciones dadas en esta sección, existen comentarios en el código que detallan el funcionamiento de cada una de las funciones contenidas dentro de los distintos ficheros .py.

A continuación se procede a explicar cada uno de los paquetes.

communications

Este paquete gestiona las comunicaciones desde y hacia S1. Su principal cometido es facilitar la operación y la lectura del *buffer* de recepción junto con la escritura por un puerto UART.

Contiene un único fichero, “connection.py” (anexo E.1), en el cual están definidas todas las funciones relacionadas con la lectura y la escritura a través del puerto UART.

Es aquí donde, mediante el uso de cierres de exclusión mutua, se consigue que el acceso a la UART se haga de manera individual por parte de los procesos.

Cabe destacar que para poder utilizar este puerto para la comunicación se ha hecho uso de la librería “pyserial”¹ de Python.

También se ha empleado la librería “threading” la cual da acceso a los cierres de exclusión mutua, llamados *locks* en dicha librería.

control

Este paquete contiene la lógica de control propiamente dicha. Sirve para implementar los métodos principales de movimiento, de gestión de la comunicación, de creación del demonio de pulsos y de sincronización inicial. Además, ofrece una interfaz lógica para que la interfaz de usuario pueda, a través de los botones que aparecen en pantalla, comunicarse con la lógica de control.

El paquete está conformado por 4 archivos:

- **control.py** (anexo E.2): el cual implementa los métodos de movimiento y petición de posiciones además del método que desencadena el proceso de sincronización inicial. Es

¹<https://pypi.org/project/pyserial/>

el archivo principal de control y sus funciones desencadenan llamadas a todos los demás paquetes para poder realizar funciones complejas.

- **control_interface.py** (anexo E.3): es una interfaz que implementa parte de los métodos de **control.py**. Este archivo es utilizado por la GUI para poder desencadenar acciones en S1 a partir de la interacción del operario con la interfaz gráfica.
- **control_management.py** (anexo E.4): ofrece funciones auxiliares que permiten a **control.py** monitorizar que las órdenes enviadas a S2 son realizadas con éxito o, en caso contrario, controlar los errores que se pudieran producir. Además, aquí se encuentran las funciones que hacen peticiones a S2.
- **heart_beat** (anexo E.5): ofrece funciones que permiten la instanciación de un objeto el cual envía un mensaje de manera periódica a través de la UART.

Cabe destacar que en este paquete se han utilizado librerías que implementan el uso de futuros en Python con el objetivo de poder monitorizar el movimiento de S2 sin necesidad de bloquear la interfaz de usuario.

gcode

Aquí se encuentran la lógica de interpretación y generación de las tramas de GCode que se transmiten entre S1 y S2.

Contiene 2 ficheros:

- **generator.py** (anexo E.6): este archivo contiene las funciones que, a partir de los valores que reciben como parámetros, generan las tramas en el formato adecuado para ser enviadas.
- **interpreter.py** (anexo E.7): analizador gramático de los *bytes* que hay en el *buffer* de S1. Para poder realizar esta labor, transforma los *bytes* en cadena de caracteres y posteriormente analiza dichas cadenas y las interpreta.

Se ha usado la librería “**typing**” para simplificar el tratamiento de los datos y la interpretación de las tramas.

logger

Este paquete permite llevar un registro de los datos y las acciones importantes que ocurren en S1. Genera un archivo en el que se guardan diferentes datos para poder hacer *debugging* y revisar *crashlogs*. Destaca el uso de la librería “**logging**” para poder generar registros de manera unificada y poder definir distintos niveles dentro de los registros, de manera que se pueden definir niveles que serán ignorados.

Contiene dos ficheros, **logger.py** (anexo E.11), el cual alberga las funciones necesarias para crear el archivo y operar con él, además de dar un formato estándar a las diferentes trazas;

y el fichero `PyQtHandler.py` (anexo E.12), el cual contiene un *wrapper* para que la librería `logging` también escriba los registros en la interfaz de usuario, en un *widget* de Qt.

security

A través de este paquete, S1 consigue autenticar al sistema S2 y viceversa.

Contiene un solo archivo, `rsa.py` (anexo E.13), el cual alberga las funciones necesarias para que, a partir de los números procedentes de S2, se pueda autenticar al emisor desde la trama recibida y poder reenviarla encriptada posteriormente.

Se ha usado la librería “`typing`” para simplificar el tratamiento de los datos y la interpretación de las tramas.

utils

En este paquete se encuentran todos los archivos auxiliares que no se puedan ubicar en ningún otro paquete.

Contiene un archivo, `error_data.py` (anexo E.14), el cual simplemente implementa un tipo de dato creado especialmente para poder contener de una manera más organizada los datos referentes a los errores provenientes de S2.

Se emplea la librería `collections` para permitir el uso de tipos de datos auxiliares.

Por otra parte, se declara una nueva estructura de datos en el fichero `atomics.py` (listado de código E.15). Se define una clase base `class Atomic(ABC, Generic[T])` que se diseña para que las clases hijas hereden de ella y establezcan el tipo de dato que van a contener, junto con las especializaciones requeridas.

Se definen dos clases hijas:

- `class AtomicFloat(Atomic[float])` para contener valores en coma flotante de forma atómica.
- `class AtomicInteger(Atomic[int])` para contener valores enteros y añadir además opciones para incrementar dicho valor.

La principal función de estas clases no es solo guardar un dato de forma atómica sino además poder utilizar la misma dirección de memoria a lo largo del programa, pudiendo tener acceso simultáneo al recurso.

7.1.3. Protocolo de autenticación

El protocolo de autenticación se ha creado para que los dos sistemas puedan reconocerse entre sí, permitiendo de esta manera asegurar que los datos que provengan del puerto elegido para la comunicación son efectivamente destinados al dispositivo y asegurando por tanto que los datos enviados a través de este mismo puerto llegarán al destino esperado.

Para realizar la autenticación se sigue el siguiente protocolo.

1. S2 genera dos números al iniciarse, n (el módulo) y e (la clave pública).
2. S1, al ser elegido un puerto de comunicación en la interfaz, manda una petición de autenticación a S2 por ese puerto empleando el GCode “I1”.
3. Si S2 efectivamente está conectado a ese puerto, recibe el mensaje y procede a mandar el número n y e en dos mensajes distintos. El primer mensaje en formato “I2 n ” y el segundo en formato “I3 e ”.
4. Posteriormente, S2 envía a S1 un mensaje cifrado con su clave privada (firma el mensaje).
5. Tras recibir este mensaje firmado, S1 utiliza la clave pública e para verificar el mensaje y autentificar a S2.
6. S1 envía el mensaje cifrado con la clave pública e de vuelta a S2.
7. S2 obtendrá el mensaje en claro utilizando su clave privada d y verificará que, en efecto, S1 es un “dispositivo de confianza”.

En cuanto los dispositivos se identifiquen, se permitirá el intercambio de datos a través del puerto seleccionado hasta que haya una desconexión de alguno de los dispositivos. En ese caso, al volver a realizar la conexión, los dispositivos tendrán que volver a autenticarse.

7.1.4. Pseudo-lenguaje de comunicación

Para poder comunicar los dispositivos entre si de manera eficiente, se ha planteado un pseudo-lenguaje basado en GCode para poder realizar el envío y la recepción de datos desde y hacia la placa.

El formato general del mensaje que se envía es el siguiente:

G1 X10 Y10 Z10

Donde la ‘G’ representa el tipo de trama y el número a su derecha el identificador individual de una trama concreta de ese tipo. Después, separado por un espacio, se tienen los parámetros de la trama. El número del parámetro dependerá del tipo concreto de orden.

Para este proyecto se han reservado 4 tipos de trama, a saber, tipos: I, J, G y M.

Las tramas de tipo ‘I’ sirven para gestionar la autenticación inicial de los dispositivos.

Las tramas de tipo ‘J’ comunican fallos o para confirmar un funcionamiento correcto del sistema.

Las tramas de tipo ‘G’ comunican valores de las posiciones tanto cartesianas como angulares.

Las tramas de tipo ‘M’ se emplean para pedir los valores de las posiciones cartesianas o angulares del brazo y para cancelar el movimiento de este.

A continuación se pasará a analizar cada una de las tramas por separado.

G0

Este tipo de trama sirve para comunicar una posición ya que es de tipo G. Más concretamente, como es G0, sirve para comunicar posiciones cartesianas.

Los parámetros de esta trama son X, Y y Z los cuales van seguidos de un valor numérico que representa la posición en cada uno de los respectivos ejes.

Por ejemplo, G0 X10 Y20 Z30 proveniente de S1 y con destino a S2 significa que S2 se ha de mover a las posiciones cartesianas $x = 10 \text{ mm}$, $y = 20 \text{ mm}$ y $z = 30 \text{ mm}$ con respecto a la posición inicial.

Por otro lado, si la misma trama fuese proveniente de S2 con destino a S1, esto significaría que se está comunicando a S1 la posición cartesiana real del brazo.

G1

Este tipo de trama sirve para comunicar una posición ya que es de tipo G. Más concretamente, como es G1, sirve para comunicar posiciones angulares.

Los parámetros de esta trama son X, Y y Z los cuales van seguidos de un valor numérico que representa el ángulo en θ_1 , θ_2 y θ_3 .

Por ejemplo, G1 X10 Y20 Z30 proveniente de S1 y con destino a S2 significa que S2 se ha de mover a las posiciones angulares $\theta_1 = 10^\circ$, $\theta_2 = 20^\circ$ y $\theta_3 = 30^\circ$.

Obsérvese que, pese a que los parámetros siguen siendo X, Y, y Z, en S1 se interpretan como ángulos, a diferencia de G0, donde se interpretaban como coordenadas cartesianas.

G28

Este tipo de trama sirve para comunicar una posición ya que es de tipo G. Más concretamente, como es G28, sirve para comunicar al brazo que debe volver a la posición de origen.

Esta trama no tiene parámetros ya que la posición de origen es conocida por el S2 y no hace falta comunicarla.

La trama G28 solo se envía actualmente desde S1 a S2.

M1

Este tipo de trama sirve para comunicar peticiones al brazo robótico ya que es de tipo M. Más concretamente, como es M1, sirve para pedir a S2 que cancele el movimiento que está ejecutando.

Esta trama no tiene parámetros ya que la petición de cancelar el movimiento es interpretada por si sola y no hace falta datos adicionales.

La trama M1 solo se envía actualmente desde S1 a S2

M114

Este tipo de trama sirve para comunicar peticiones al brazo robótico ya que es de tipo M. Más concretamente, como es M114, sirve para pedir a S2 que comunique la posición cartesiana actual en la que se encuentra el *end-effector*.

Esta trama no tiene parámetros ya que la petición de la posición cartesiana es interpretada por sí sola y no hacen falta datos adicionales.

La trama M114 solo se envía actualmente desde S1 a S2

M280

Este tipo de trama sirve para comunicar peticiones al brazo robótico ya que es de tipo M. Más concretamente, como es M280, sirve para pedir a S2 que comunique la posición angular actual en la que se encuentra el *end-effector*.

Esta trama no tiene parámetros ya que la petición de la posición angular es interpretada por sí sola y no hacen falta datos adicionales.

La trama M280 solo se envía actualmente desde S1 a S2.

I1

Este tipo de trama sirve para realizar el *handshake* inicial entre el S1 y el S2 ya que es de tipo I. Más concretamente, como es I1, sirve para al S2 que se identifique mandando su '*n*' (módulo) y '*e*' (exponente) para poder calcular su clave pública.

Esta trama no tiene parámetros ya que la petición de inicio del *handshake* es interpretada por sí sola y no hacen falta datos adicionales.

La trama I1 solo se envía actualmente desde S1 a S2.

I5

Este tipo de trama sirve para realizar el *handshake* inicial entre el S1 y el S2 ya que es de tipo I. Más concretamente, como es I5, sirve para enviar la trama firmada inicialmente por S2 de vuelta a este, sin firmar y cifrada.

El parámetro que tiene esta trama es una cadena de números que representan la trama sin firmar y cifrada.

La trama I5 solo se envía actualmente desde S1 a S2.

I6

Este tipo de trama sirve para hacer una petición al S2 de recalcular su claves con el objetivo de poder cambiar a un S1 distinto.

En el estado actual de desarrollo esta trama no se utiliza ya que solo tendría sentido si todas las comunicaciones entre los dos sistemas estuviesen cifradas. En el estado actual del proyecto, no lo están.

La trama I6 solo se envía actualmente desde S1 a S2.

7.1.5. Logs

Para conseguir mantener una traza del funcionamiento del S1 tanto en desarrollo como en producción, se ha decidido generar de manera automática ficheros que registren ciertos comportamientos del SW.

Estos ficheros contienen líneas de texto las cuales muestran datos del sistema y mensajes que describen el funcionamiento de este.

La estructura general de una línea de traza es la siguiente:

PID - ASCTIME | [NIVEL DE ERROR]: MENSAJE

7.2. S2

El S2 supone una parte fundamental en el desarrollo del proyecto ya que es el encargado de la gestión al completo del *pArm*. El SW que ejecuta se encuentra escrito puramente en C (en particular, C99) y se ha programado utilizando el paradigma de programación estructurada, utilizando subrutinas, secuencias, condiciones (**if**, **switch**) e iteradores (**for**, **while**)[49].

La estructura de la aplicación consta de los siguientes paquetes y ficheros:

arm – contiene el planificador de movimientos del *pArm*, definido por los ficheros **planner.h** (listado de código C.1) y **planner.c** (listado de código C.28).

gcode – contiene el intérprete de GCode que se utiliza principalmente para la comunicación entre S1 y S2. Se encuentra compuesto por los ficheros **gcode.h** (listado de código C.2) y **gcode.c** (listado de código C.29).

motor – este paquete aúna la lógica de control de los motores que componen el *pArm*.

Por una parte, se define un primer nivel de abstracción sobre el control de los servomotores implementado en los ficheros **servo.h** (listado de código C.3) y **servo.c** (listado de código C.30).

Sobre lo anterior, se define un segundo nivel de abstracción para el control de los servomotores que simplifica las operaciones a realizar sobre los mismos. Se encuentra implementado en los ficheros **motor.h** (listado de código C.4) y **motor.c** (listado de código C.31).

Finalmente, se establece un tercer nivel de abstracción que permite el control de los distintos motores mediante la cinemática directa (utilizando los ángulos $\{\theta_0, \theta_1, \theta_2\}$) y

mediante la cinemática inversa (utilizando los puntos $\{x, y, z\}$). La lógica se encuentra implementada en los ficheros `kinematics.h` (listado de código C.5) y `kinematics.c` (listado de código C.32).

`printf` – una implementación adaptada para trabajar con la placa de control dsPIC33E basada en la librería `mpaland/printf`² [50].

Se definen nuevos ficheros para la gestión de la librería y se encuentra estructurada en `io.h` (listado de código C.6), `printf_config.h` (listado de código C.7), `printf.h` (listado de código ??) y `printf.c` (listado de código ??).

`rsa` – paquete que recoge las principales funcionalidades del algoritmo RSA para realizar firma digital y encriptado de datos. Se encuentra implementado en los ficheros `rsa.h` (listado de código C.8) y `rsa.c` (listado de código C.33).

Además, contiene otras funcionalidades útiles como la generación de números pseudoaleatorios, implementada en los ficheros `rand.h` (listado de código C.9) y `rand.c` (listado de código C.34); los algoritmos de `clz` (“*count leading zeros*”) y `ctz` (“*count trailing zeros*”) implementadas en los ficheros `zeros.h` (listado de código C.10) y `zeros.c` (listado de código C.35); y un algoritmo de comprobación para saber si un número ‘*p*’ de 64 bits es primo, implementado en el fichero `primes.h` (listado de código C.11) y `primes.c` (listado de código C.36).

`sync` – un paquete que recoge funciones de sincronización entre distintos hilos de ejecución. Pese a que el dsPIC33E no cuenta con dicha funcionalidad, las rutinas de tratamiento de interrupciones se ejecutan en su propio contexto y podrían llegar a provocar una colisión con respecto al valor de ciertas variables.

Este paquete se utiliza principalmente para conocer cuándo los tres motores que afectan a la posición del robot han finalizado su movimiento, implementando un algoritmo de exclusión mútua, desarrollado en los ficheros `mutex.h` (listado de código C.12) y `mutex.c` (listado de código C.37), y el algoritmo de barrera, escrito en los ficheros `barrier.h` (listado de código C.13) y `barrier.c` (listado de código C.38).

`timers` – implementación de los *timers* que gestionan la posición y movimiento de los tres motores. Ofrecen una interfaz común para poder gestionar la sincronización entre ellos así como cuándo se empieza un movimiento y cuándo se finaliza.

El paquete se encuentra dividido en: `tmr3.h` (listado de código C.14) y `tmr3.c` (listado de código C.39); `tmr4.h` (listado de código C.15) y `tmr4.c` (listado de código C.40); y `tmr5.h` (listado de código C.16) y `tmr5.c` (listado de código C.41).

`utils` – distintas utilidades que se utilizan a lo largo de toda la aplicación. Se encuentran definidas utilidades para el manejo de *buffers* de tamaño arbitrario, implementado en los ficheros `buffer.h` (listado de código C.17) y `buffer.c` (listado de código C.42); constantes matemáticas o utilidades que se usan en tiempo de compilado por el preprocesador, implementado en el fichero `defs.h` (listado de código C.18); librería para el manejo del tiempo que ha pasado desde la ejecución de la aplicación, tanto en *ms* como en *μs*,

²<https://github.com/mpaland/printf>

implementado en los ficheros `time.h` (listado de código C.19) y `time.c` (listado de código C.43); definiciones de nuevos tipos de datos basados en estructuras, redefiniciones de tipos ya existentes y de constantes relacionadas a ellos, implementado en el fichero `types.h` (listado de código C.20); gestión de la salida estándar al puerto del microcontrolador para ser enviado vía UART, implementado en los ficheros `uart.h` (listado de código C.21) y `uart.c` (listado de código C.44); y finalmente distintas utilidades varias que no necesitan de ningún fichero específico para ellas ya que se consideran de carácter general, como puede ser la implementación de la rutina `foreach` en C, funciones de *delay* según el tiempo especificado, comprobaciones con respecto a valores tipo `double` (por ejemplo, si son `NaN`) y utilidades para mapear un valor entre un límite superior e inferior, implementado en los ficheros `utils.h` (listado de código C.22) y `utils.c` (listado de código C.45).

`arm_config.h` – paquete que define las constantes físicas del brazo robótico, implementado en el fichero C.23. Entre otros valores, se encuentran las longitudes de los segmentos inferior y superior del brazo (definidos anteriormente como $\overline{A_L}$ y $\overline{A_U}$), la altura de la base (B_h), etc.

`init` – rutinas de configuración e inicialización de toda la placa, además de algunas funciones complementarias para permitir habilitar y deshabilitar interrupciones.

En los ficheros `init.h` (listado de código C.24) e `init.c` (listado de código C.46) se encuentran rutinas para establecer la frecuencia de oscilación del reloj interno, los baudios a los que trabaja la UART, los distintos *timers* que se utilizan así como la inicialización de los pines de la placa y las interrupciones.

`interrupts` – paquete que engloba múltiples rutinas de tratamiento de interrupciones del sistema.

Implementadas en los ficheros `interrupts.h` (listado de código C.25) e `interrupts.c` (listado de código C.47), destacan principalmente las rutinas de tratamiento de los fines de carrera así como las de gestión de la UART, encargadas de asegurar una comunicación fiable entre ambos sistemas S1 y S2.

`pragmas.h` – implementación de la configuración básica del microcontrolador. En dicho fichero (listado de código C.26) se definen opciones como permitir la reasignación de puertos en tiempo de ejecución, *overclock* de la velocidad del reloj, etc.

`s_types.h` – definición de tipos básicos del sistema para facilitar el manejo de ciertos registros, en particular, el registro CORCON. Implementado en el fichero `system_types.h` (listado de código C.27).

`main.c` – punto de entrada de la ejecución del SW de S2. Aúna todos los ficheros y paquetes mencionados anteriormente y los coordina para que el sistema se ejecute según se espera.

Se compone principalmente de una rutina de inicialización (`setup()`) y del bucle principal (`loop()`), donde configura el sistema para que respete los parámetros establecidos y gestiona los distintos eventos recibidos por la UART. Configura además un modo *cli* que permite la interacción directa con el sistema sin necesidad de una interfaz de usuario así como de un modo *debug* que han de ser activados ambos en tiempo de compilación.

Por otra parte, la gestión de los errores recae sobre este fichero así como el *heartbeat* que mantiene activa la comunicación entre sistemas. Al ser además el coordinador de todo

el sistema, las órdenes recibidas por la UART son derivadas a otros paquetes para su interpretación y luego delegadas para realizar las órdenes especificadas, o indicar error en caso de que no sea un dispositivo verificado o si es una instrucción no implementada por el sistema.

Se encuentra implementado en el fichero `main.c` (listado de código C.48).

En los siguientes apartados se comentan puntos claves sobre cómo está diseñado este SW y se explicarán ciertas decisiones que han sido tomadas.

7.2.1. Inicialización del sistema

La inicialización del sistema cuenta de varias partes:

- Configuración del microcontrolador.
- Deshabilitar las interrupciones.
- Inicializar el reloj.
- Inicializar los pines y los puertos.
- Inicializar el módulo PWM.
- Inicializar el módulo UART.
- Establecer el modo de operación del CORCON.
- Habilitar de nuevo las interrupciones.

Toda esta lógica se aúna en la función `system_initialize`:

```
359 inline void system_initialize(void) {  
360     INTERRUPT_GlobalDisable();  
361     init_clock();  
362     init_ports();  
363     initPWM();  
364     initUART();  
365     SYSTEM_CORCONModeOperatingSet(CORCON_MODE_PORVALUES);  
366     INTERRUPT_GlobalEnable();  
367 }
```

Configuración del microcontrolador

La configuración del microcontrolador es una parte crucial a la hora de poder trabajar con el dispositivo ya que permite definir cómo se va a comportar ante ciertas situaciones.

La configuración se realiza mediante una serie de **pragmas** que definen tanto qué opciones están habilitadas como el modo de funcionamiento de ciertos componentes.

De todas las configuraciones establecidas, destacan las siguientes:

- `#pragma config PLLKEN = ON` – habilita el bit de bloqueo del PLL que permite saber cuándo un cambio en la frecuencia del oscilador es efectivo.
- `#pragma config IOL1WAY = OFF` – permite cambiar el modo de funcionamiento de los periféricos más de una única vez.
- `#pragma config FCKSM = CSECME` – permite cambiar la frecuencia del reloj y habilita el *Fail-Safe Clock Monitor*.
- `#pragma config PWMLOCK = OFF` – permite que un puerto PWM pueda ser utilizado para otros propósitos, como puede ser para la UART.

La lista completa de pragmas se define en el fichero `pragmas.h` (listado de código C.26).

Inicialización del reloj del sistema

La rutina de inicialización del reloj define la frecuencia a la que va a trabajar el sistema. Para este proyecto se cuenta con una frecuencia inicial de oscilación $F_{OSC} = 7,372\,8\text{ MHz}$ y se busca alcanzar una nueva frecuencia de oscilación $F'_{OSC} \approx 120\text{ MHz}$.

Según el manual del fabricante, la frecuencia máxima a la que podría trabajar el microcontrolador dsPIC33E sería de 140 MHz con una temperatura máxima de 85°C [51]. Para evitar alcanzar ese margen y que sea necesario utilizar un disipador, se establece la frecuencia 20 MHz por debajo de la máxima.

El cálculo de la nueva frecuencia de oscilación responde a la ecuación 7.1 provista por el manual [51]:

$$F_{OSC} = F_{IN} \cdot \frac{M}{N_1 + N_2} = F_{IN} \cdot \frac{PLLDIV + 2}{(PLLPRE + 2) \cdot 2(PLLPOST + 1)} \quad (7.1)$$

donde

$$\begin{cases} N_1 = PLLPRE + 2 \\ N_2 = 2 \cdot (PLLPOST + 1) \\ M = PLLDIV + 2 \end{cases}$$

En particular, se puede conseguir una frecuencia de oscilación de 119,808 MHz estableciendo los siguientes valores de N_1 , N_2 y M (ecuación 7.2):

$$F_{OSC} = 7,372\,8\text{ MHz} \cdot \frac{65}{2 \cdot 2} = 119,808\text{ MHz},$$

$$\begin{cases} N_1 = 2 \\ N_2 = 2 \\ M = 65 \end{cases} \quad (7.2)$$

De esta manera, se introducen en los registros PLLPRE, PLLPOST y PLLDIV los valores 0, 0 y 63 respectivamente. Una vez cambiados los registros se espera de forma activa a que cambie la frecuencia del reloj (OSWEN = 0 y LOCK = 1).

Esta lógica se encuentra en la función `init_clock`:

```

34 void init_clock(void) {
35 #ifndef CONFIG_SIMULATOR
36     // FRCDIV FRC/1; PLLPRE 2; DOZE 1:8; PLLPOST 1:2; DOZEN disabled; ROI disabled;
37     CLKDIV = 0x3000;
38     // TUN Center frequency;
39     OSCTUN = 0x00;
40     // ROON disabled; ROSEL FOSC; RODIV 0; ROSSLP disabled;
41     REFOCON = 0x00;
42     // Setup de PLL for reaching 40 MHz with a 7.3728 clock.
43     // Maximum speed is of 140 MHz as the maximum temperature
44     // of 85 °C implies 70 MIPS.
45     //
46     // For working at ~120 MHz:
47     // F_osc = F_in * M / (N1 * N2)
48     // F_cy = F_osc / 2
49     // F_osc ~= 120 MHz -> F_osc = 7.3728 * 65 / (2 * 2) = 119.808 MHz
50     // F_cy = F_osc / 2 = 59.904 MHz
51     //
52     // Then, setup the PLL's prescaler, postcaler and divisor
53     PLLFBD = 0x3F;
54     // AD1MD enabled; PWMMMD enabled; T3MD enabled; T4MD enabled; T1MD enabled; U2MD enabled;
55     // T2MD enabled; U1MD enabled; QEI1MD enabled; SPI2MD enabled; SPI1MD enabled; C2MD enabled;
56     // C1MD enabled; DCIMD enabled; T5MD enabled; I2C1MD enabled;
57     PMD1 = 0x00;
58     // OC5MD enabled; OC6MD enabled; OC7MD enabled; OC8MD enabled; OC1MD enabled; IC2MD enabled
59     ; OC2MD enabled; IC1MD enabled; OC3MD enabled; OC4MD enabled; IC6MD enabled; IC7MD enabled;
60     // IC5MD enabled; IC8MD enabled; IC4MD enabled; IC3MD enabled;
61     PMD2 = 0x00;
62     // ADC2MD enabled; PMPMD enabled; U3MD enabled; QEI2MD enabled; RTCCMD enabled; CMPMD
63     enabled; T9MD enabled; T8MD enabled; CRCMD enabled; T7MD enabled; I2C2MD enabled; T6MD
64     enabled;
65     PMD3 = 0x00;
66     // U4MD enabled; CTMUMD enabled; REFOMD enabled;
67     PMD4 = 0x00;
68     // PWM2MD enabled; PWM1MD enabled; PWM4MD enabled; SPI3MD enabled; PWM3MD enabled; PWM6MD
69     enabled; PWM5MD enabled;
70     PMD6 = 0x00;
71     // PTGMD enabled; DMA0MD enabled;
72     PMD7 = 0x00;
73     // CF no clock failure; NOSC FRCPLL; CLKLOCK unlocked; OSWEN Switch is Complete; IOLOCK not
74     // -active;
75     __builtin_write_OSCCONH((uint8_t) (0x01));
76     __builtin_write_OSCCONL((uint8_t) (0x01));

    // Wait for Clock switch to occur
77     while (OSCCONbits.OSWEN != 0);
78     // And wait for clock switching to happen
79     // First, wait for clock switch to occur
80     // and thenm wait the PLL to lock
81     while (OSCCONbits.LOCK != 1);

```

```
76 #endif  
77 }
```

Inicialización de los pines y puertos

En el dsPIC33E se utilizan múltiples pines y puertos que han de ser inicializados para el control de distintos periféricos. Los distintos pines y puertos del microcontrolador se muestran en la figura 7.8:

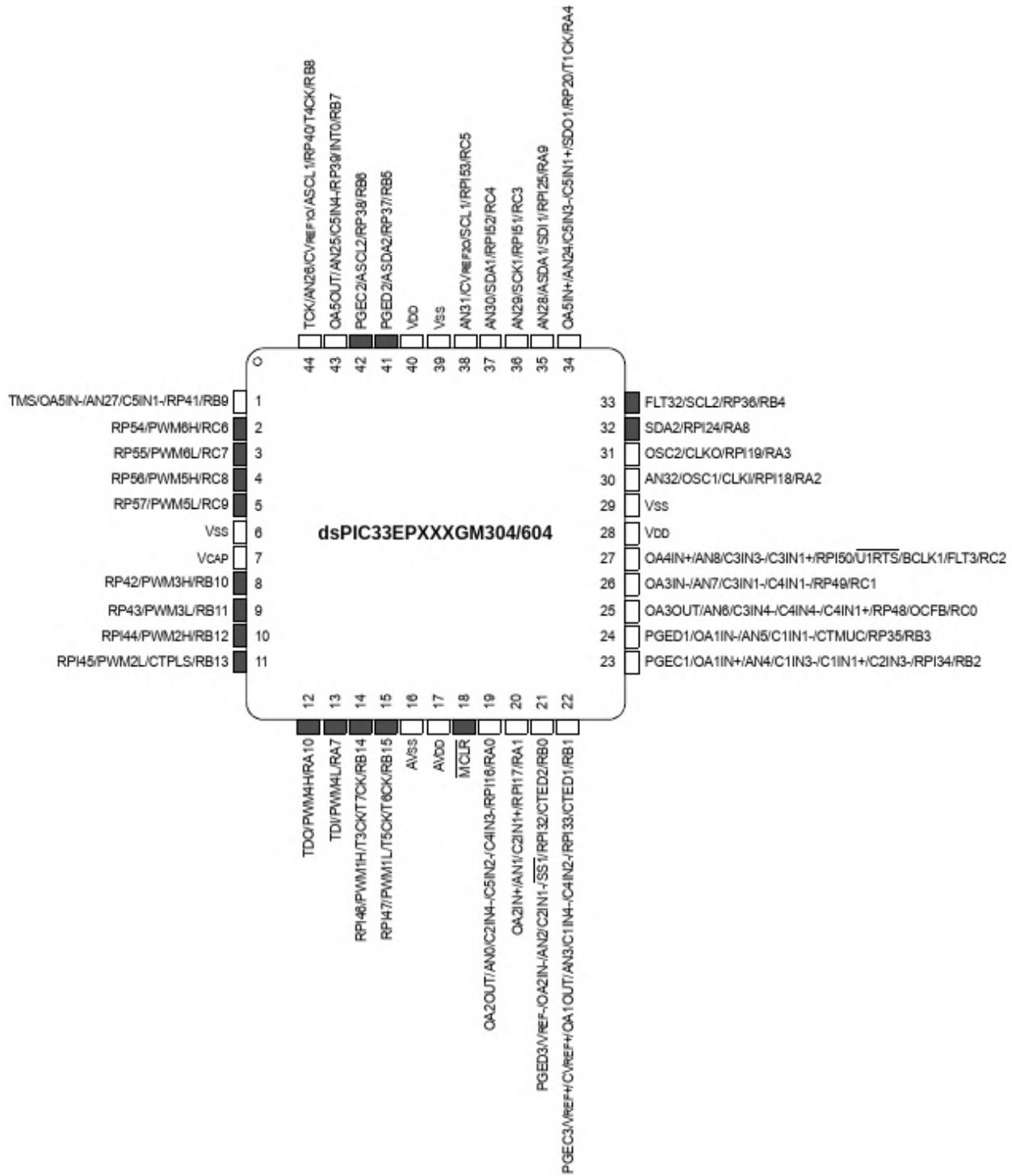


Figura 7.8: Vista esquemática del dsPIC33E [41].

En particular, se utilizan los pinos 41 – 44 para controlar los LEDs colocados en la placa, los pinos 19 – 22 para controlar los microinterruptores conectados que actúan como fin de carrera. Esto se encuentra en la función `init_ports`:

326 `void init_ports(void)`

```

327 {
328     //Digital Ports for micro-interruptors, set as input
329     TRISAbits.TRISA0 = 1;
330     TRISAbits.TRISA1 = 1;
331     TRISBbits.TRISB0 = 1;
332     TRISBbits.TRISB1 = 1;
333
334     //Input Change Notification Interrupt configuration
335     _CNIP = 5;          // priority (7 = highest)
336     _CNIE = 1;          // Enable CN interrupts
337     _CNIF = 0;          // Interrupt flag cleared
338     CNENBbits.CNIEB0 = 1;
339     CNENBbits.CNIEB1 = 1;
340     CNENAbits.CNIEA0 = 1;
341     CNENAbits.CNIEA1 = 1;
342
343
344     //Digital Ports for LED lights, set as output.
345     TRISBbits.TRISB5 = 0;
346     TRISBbits.TRISB6 = 0;
347     TRISBbits.TRISB7 = 0;
348     TRISBbits.TRISB8 = 0;
349
350     //Set I/O ports to digital, clear the analogic enable bit.
351     ANSELAbits.ANSA0 = 0;
352     ANSELAbits.ANSA1 = 0;
353     ANSELBbits.ANSB0 = 0;
354     ANSELBbits.ANSB1 = 0;
355     ANSELBbits.ANSB7 = 0;
356     ANSELBbits.ANSB8 = 0;
357 }

```

Por otro lado, se utilizan los pines 9, 11, 13 y 15 para controlar la señal PWM que controla los motores. Esto se realiza en la función `initPWM`:

```

158     TRISBbits.TRISB11 = 0; // PWM3L
159     TRISBbits.TRISB13 = 0; // PWM2L
160     TRISBbits.TRISB15 = 0; // PWM1L
161     TRISAbits.TRISA7 = 0; // PWM4L

```

Finalmente, para controlar la UART se utilizan los pines 3 y 2, los cuales son remapeables y es necesario establecer a qué puerto va cada uno y qué funcionalidad cumple.

Esto se realiza en la función `initUART`:

```

87     TRISCbits.TRISC7 = 1; // RC7 set as input
88     TRISCbits.TRISC6 = 0; // RC6 set as output
89
90     RPOR6bits.RP54R = 0b00001; // RC6->UART1:U1TX
91     RPINR18bits.U1RXR = 55;

```

Inicialización del módulo PWM

El módulo PWM requiere de una inicialización propia para establecer la frecuencia de funcionamiento del mismo. Los motores que se están utilizando son servomotores y se controlan mediante una señal de pulsos cada cierto tiempo, establecido por el diseño del fabricante.

Para el servomotor Parallax 900-00005, se requiere un periodo de 20 ms (imagen 7.9):

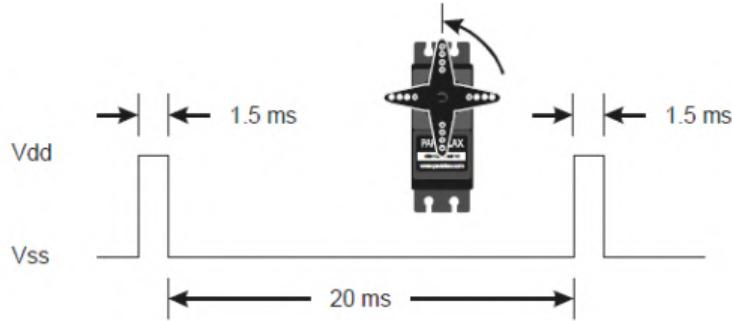


Figura 7.9: Periodo de la señal PWM que ha de ser enviada al servomotor [48].

En el dsPIC33E, el módulo PWM se configura estableciendo un valor en el registro PTPER, el cual respeta la siguiente ecuación (ecuación 7.3):

$$PTPER = \frac{F_{OSC}}{F_{PWM} \cdot PWM_{Prescaler}} \quad (7.3)$$

El registro PTPER es un registro de 16 bits, por lo que el valor más alto que puede contener es 65 536. Si el valor supera el máximo de dicho registro, se habrá de incrementar el prescalado para poder reducirlo. Según el manual [52], se encuentran disponibles prescalados desde 2^0 hasta 2^6 (incrementándose en potencias de dos).

De esta manera, para controlar los servomotores, se tiene que:

$$\left\{ \begin{array}{l} F_{PWM} = T_{PWM}^{-1} = \frac{1}{20} = 50 \text{ Hz} \\ F_{OSC} = 119,808 \text{ MHz} \\ PWM_{Prescaler} = 1 : 2^5 = 1 : 32 \equiv Prescaler = 101 \end{array} \right.$$

por lo que el valor a introducir en el registro PTPER es (ecuación 7.4):

$$PTPER = \frac{119,808 \text{ MHz}}{50 \text{ Hz} \cdot 32} = 37\,440 \quad (7.4)$$

Finalmente, se configuran los módulos PWM para trabajar en modo verdaderamente independiente. Esto es, se cuenta con dos señales PWM: **PWMxL** y **PWMxH**, que pueden funcionar en modo combinado de 16 bits, pero para tener más posibilidades se configuran ambas señales como señales independientes, esto es, funcionan sin alterar la salida que se produce en la otra.

Todo el código de inicialización se encuentra en la función **initPWM**:

```
157 void initPWM(void) {
158     TRISBbits.TRISB11 = 0; // PWM3L
159     TRISBbits.TRISB13 = 0; // PWM2L
160     TRISBbits.TRISB15 = 0; // PWM1L
161     TRISAbits.TRISA7 = 0; // PWM4L
162
163     PTCON2bits.PCLKDIV = 0b110; // Prescaler 1:32
164
165     // Setup PWM period - the motors have a
166     // minimum time in between pulses of 20ms,
167     // so the frequency must be of 50 Hz.
168     //
169     // F_osc = 119.808 MHz
170     // F_PWM = 50 Hz
171     // PWM_Prescaler = 64
172     // PTPER = F_osc / (F_PWM * PWM_Prescaler) --> PTPER = 119.808 MHz / (50 Hz * 32)
173     // = 37440 = PTPER --> F_PWM = 50.000 Hz
174     PTPER = 37440;
175
176     // Initialize independent time base to zero.
177     // As we are using PWMxL, we only use
178     // SPHASEx ports. If using PWMxH, just change
179     // SPHASEx to PHASEx ones.
180     SPHASE3 = 0;
181     SPHASE2 = 0;
182     SPHASE1 = 0;
183     SPHASE4 = 0;
184
185     // By default, set no duty cycle of programmed signals
186     SDC3 = 0;
187     SDC2 = 0;
188     SDC1 = 0;
189     SDC4 = 0;
190
191     // Disable Dead Time values
192     ALTDTR4 = 0;
193     ALTDTR3 = 0;
194     ALTDTR2 = 0;
195     ALTDTR1 = 0;
196
197     DTR4 = 0;
198     DTR3 = 0;
199     DTR2 = 0;
200     DTR1 = 0;
201
202     // True independent work mode, so then both PWMxH and
203     // PWMxL can be used independently
204     IOCON4bits.PMOD = 0b11;
205     IOCON3bits.PMOD = 0b11;
206     IOCON2bits.PMOD = 0b11;
207     IOCON1bits.PMOD = 0b11;
208
209     // Disable PWM fault input
210     FCLCON4bits.FLTMOD = 0b11;
211     FCLCON3bits.FLTMOD = 0b11;
```

```

212 FCLCON2bits.FLTMOD = 0b11;
213 FCLCON1bits.FLTMOD = 0b11;
214
215 // Do not swap LOW/HIGH values
216 IOCON4bits.SWAP = 0;
217 IOCON3bits.SWAP = 0;
218 IOCON2bits.SWAP = 0;
219 IOCON1bits.SWAP = 0;
220
221 // Set pins as PWM ones
222 IOCON4bits.PENL = 1;
223 IOCON3bits.PENL = 1;
224 IOCON2bits.PENL = 1;
225 IOCON1bits.PENL = 1;
226 // Disable high output as we are not using it
227 IOCON4bits.PENH = 0;
228 IOCON3bits.PENH = 0;
229 IOCON2bits.PENH = 0;
230 IOCON6bits.PENH = 0;
231 IOCON6bits.PENL = 0;
232 IOCON6bits.OVRENH = 1;
233 IOCON6bits.OVRENH = 1;
234
235 // Set PWM configurations to zero by default
236 PWMCON6 = 0;
237 PWMCON5 = 0;
238 PWMCON4 = 0;
239 PWMCON3 = 0;
240 PWMCON2 = 0;
241 PWMCON1 = 0;
242
243 // Disable dead time in-between output switches
244 PWMCON4bits.DTC = 0b10;
245 PWMCON3bits.DTC = 0b10;
246 PWMCON2bits.DTC = 0b10;
247 PWMCON1bits.DTC = 0b10;
248
249 // and enable the PWM module
250 PTCONbits.PTEN = 1;
251 }

```

Inicialización de la UART

El módulo UART permite la comunicación entre los dos sistemas S1 y S2. En el proceso de diseño se estableció la tasa de transmisión en 9 600 baud, por lo que se ha de configurar el dsPIC33E para que funcione a esta velocidad.

Según el manual del microcontrolador, el valor del *baudrate* se ha de establecer en el registro UxBRG y responde a la siguiente ecuación (ecuación 7.5):

$$UxBRG = \frac{F_{CY}}{16 \cdot \text{Baud rate}} - 1 \quad (7.5)$$

Según la configuración establecida anteriormente, se sabe que la frecuencia del ciclo de instrucción (F_{CY}) es:

$$F_{CY} = \frac{F_{OSC}}{2} = \frac{119,808 \text{ MHz}}{2} = 59,904 \text{ MHz}$$

por lo que se tiene que el valor del registro UxBRG es (ecuación 7.6):

$$\text{UxBRG} = \frac{59,904 \text{ MHz}}{16 \cdot 9600} - 1 = 389 \quad (7.6)$$

Por otra parte, se configuran múltiples parámetros de la UART para:

- Producir una interrupción en cada carácter recibido por R_X .
- El estado de inactividad no se invierta, esto es, tenga un nivel alto.
- Se utiliza el modo de velocidad estándar de la UART.
- Las transmisiones son de 8 bits sin bit de paridad.
- La transmisión de bits así como su recepción generan una interrupción.

Todo esto se encuentra definido en la función `initUART`:

```

79 void initUART(void) {
80     // Unlock the Peripheral Pin Selector (PPS)
81     // for allowing changes on TRIS ports without
82     // affecting expected device behavior.
83     // 0xBF is a shortcut for ~(1 << 6) == 191
84 #ifndef CONFIG_SIMULATOR
85     __builtin_write_OSCCONL(OSCCON & 0xBF); // unlock PPS
86 #endif
87     TRISCbits.TRISC7 = 1; // RC7 set as input
88     TRISCbits.TRISC6 = 0; // RC6 set as output
89
90     RPOR6bits.RP54R = 0b00001; // RC6->UART1:U1TX
91     RPINR18bits.U1RXR = 55;
92     // Lock again the PPS as we are done
93     // configuring the remappable ports.
94     // 0x40 is a shortcut for (1 << 6) == 64
95 #ifndef CONFIG_SIMULATOR
96     __builtin_write_OSCCONL(OSCCON | 0x40); // lock PPS
97 #endif
98
99     IEC0bits.U1TXIE = 0;
100    IEC0bits.U1RXIE = 0;
101
102    // Setup UART
103    // Stop on idle
104    U1MODEbits.USIDL = 1;
105    // Disable IrDA
106    U1MODEbits.IREN = 0;

```

```
107 // Use only TX and RX pins
108 // ignoring CTS, RTS and BCLK
109 U1MODEbits.UEN = 0b00;
110 // Do not wake-up with UART
111 U1MODEbits.WAKE = 0;
112 // Disable loopback mode
113 U1MODEbits.LPBACK = 0;
114 // Do not use automatic baudrate when receiving
115 U1MODEbits.ABAUD = 0;
116 // Disable polarity inversion. Idle state is '1'
117 U1MODEbits.URXINV = 0;
118 // Do not use high speed baudrate
119 U1MODEbits.BRGH = 0;
120 // 8 data bits without parity
121 U1MODEbits.PDSEL = 0b00;
122 // One stop bit
123 U1MODEbits.STSEL = 0;
124
125 // Calculate the baudrate using the following equation
126 // UxBRG = ((FCY / Desired Baud rate) / 16) - 1
127 // For 9600 bauds and FCY = 59.904E6, the obtained BRG is
128 // -> 389, and the obtained baudrate is: 9600, with an error
129 // of 0%
130 U1BRG = 0x185;
131
132 // Interrupt after one RX character is received;
133 // UTXISEL0 TX_ONE_CHAR; UTXINV disabled; OERR NO_ERROR_cleared; URXISEL RX_ONE_CHAR;
134 UTXBRK COMPLETED; UTXEN enabled; ADDEN disabled;
135 U1STA = 0x400;
136
137 // Enable UART TX Interrupt
138 IEC0bits.U1TXIE = 1;
139 IEC0bits.U1RXIE = 1;
140 IEC4bits.U1EIE = 1;
141 IFS0bits.U1RXIF = 0;
142 IFS0bits.U1TXIF = 0;
143 IFS4bits.U1EIF = 0;
144 IPC2bits.U1RXIP = 0b110;
145
146 //Make sure to set LAT bit corresponding to TxPin as high before UART initialization
147 LATCbits.LATC7 = 1;
148 LATCbits.LATC6 = 1;
149 U1MODEbits.UARTEN = 1; // enabling UART ON bit
150 U1STAbits.UTXEN = 1;
151
152 // Wait 105 uS (when baudrate is 9600) for a first
153 // transmission bit to be sent and detected, so then
154 // the UART can be used
155 DELAY_105uS;
156 }
```

7.2.2. Control de los componentes

Uno de los procesos básicos del *pArm* es el del control de los distintos componentes que componen a S2. En particular, se destacan.

- Los LEDs de control.
- Los fines de carrera.
- Los servomotores que componen el brazo.

Los diodos LED

El sistema de LEDs se emplea para comunicar ciertos errores y problemas que se han podido encontrar durante la inicialización de la placa. El acceso a estos componentes se realiza mediante la escritura de un nivel alto en el registro correspondiente o de un nivel bajo en dicho registro.

El manejo de los LEDs se realiza mediante:

- PORTBbits.RB5
- PORTBbits.RB6
- PORTBbits.RB7

Actualmente, los LEDs permanecen encendidos durante el proceso de inicialización del sistema y se apagan si este ha resultado exitoso. Si, por un casual, hubiera algún tipo de problema al iniciar el sistema los LEDs parpadearían indefinidamente indicando que se ha producido un error.

Fines de carrera

Para la calibración de los motores que componen el brazo se emplean microinterruptores que actúan como fines de carrera. Dichos microinterruptores se encuentran conectados a distintos pines y se ha configurado el SW para que genere una interrupción cada vez que el valor de uno de esos pines cambie.

El método de manejo de interrupciones de periféricos por cambio de valor no es específica, es decir, todos los periféricos generaría el mismo tipo de interrupción [53]. Por eso mismo, se emplea un sistema de *polling* por el cual, cada vez que se genera una interrupción, se comprueba el valor de todos los puertos de interés.

Como cada motor tiene, en principio, un posible fin de carrera asignado. Si bien es cierto que en la estructura 3D del brazo no se contempla esta opción, el código se ha diseñado para ser lo más genérico posible y que se pueda adaptar a futuras mejoras. Por esto mismo, la rutina de tratamiento de interrupción mapea el valor de los pines a un espacio de memoria que ya ha sido registrado previamente por el motor que lo va a utilizar:

```

121 void __attribute__((__interrupt__, no_auto_psv)) _CNInterrupt(void) {
122 #ifdef LIMIT_SWITCH_ENABLED
123     if (limit_switch_map != NULL) {
124         limit_switch_map[0] = PORTAbits.RA0;
125         limit_switch_map[1] = PORTAbits.RA1;
126         limit_switch_map[2] = PORTBbits.RB0;
127         limit_switch_map[3] = PORTBbits.RB1;
128     }
129 #endif
130     _CNIF = 0; // Clear the interruption flag
131 }
```

De esta forma se tiene constancia de cuándo un motor ha tocado con un fin de carrera, ya que su valor referenciado cambiará. Como la interrupción anterior se produce tanto si el motor cierra el contacto como si no, no es necesario realizar ningún código adicional que compruebe si el fin de carrera ya no está siendo activado.

Servomotores que componen el brazo

El control de los servomotores, como se explicó anteriormente, presenta tres niveles de abstracción:

- **servo.h** y **servo.c**, que sería el equivalente a el *driver* que interactúa directamente con la interfaz ofrecida por el motor.
- **motor.h** y **motor.c**, los cuales ofrecen un nivel de abstracción por encima del modelo anterior que permiten trabajar directamente con ángulos y obtener información sobre los servomotores.
- **planner.h** y **planner.c**, para interactuar con el elemento anterior mediante la planificación de ángulos, puntos y trayectorias, teniendo en cuenta la posición actual de los motores así como el tiempo que tardarán en ejecutar el movimiento solicitado.

En el fichero **servo.h** se define una estructura que define el *driver* del servomotor:

```

39 // Servo definition
40 typedef struct {
41     /**
42      * A reference to the register in which the duty cycle is stored. This
43      * way, it is possible to define multiple servo_t handling different motors.
44      */
45     volatile uint16_t *dutyCycleRegister;
46
47     /**
48      * A reference to the limit switch map used to indicate whether the
49      * servomotor has ended its movement by touching a physical limit switch.
50      * For using this field, LIMIT_SWITCH_ENABLED macro should be defined.
51      */
52     volatile uint_fast8_t *limit_switch_value;
53 }
```

```

54 /**
55 * The servo home position.
56 */
57 double64_t home;
58
59 /**
60 * The servo physical minimum angle.
61 */
62 double64_t min_angle;
63
64 /**
65 * The servo physical maximum angle.
66 */
67 double64_t max_angle;
68 } servo_t;

```

Dicha estructura define:

- **dutyCycleRegister** – el registro el cual gestiona el *duty cycle* que genera el módulo PWM que controla el motor. De esta manera, solo es necesaria la configuración inicial que establece los puertos y utilizar una referencia al registro en cuestión.

Por ejemplo, los valores utilizados en esta primera versión se corresponden a: `&SDC1`, `&SDC2`, etc.

- **limit_switch_value** – la dirección de memoria en la que se mapea el valor del fin de carrera con el motor al que está relacionado.

Por ejemplo, los valores utilizados en esta primera versión se corresponden a: `&limit_switch_map[0]`, `&limit_switch_map[1]`, etc.

- **home** – la posición en radianes en la que se encuentra la posición inicial del servomotor.

De esta manera, se puede enviar el brazo al origen de coordenadas accediendo directamente a este campo.

- **min_angle** – el ángulo mínimo al que puede girar el motor. Dicho ángulo no tiene por qué estar directamente relacionado con los que puede efectuar el brazo en sí sino a los limitantes de la estructura física. Por ello, pese a que el rango de movimiento real del motor es mucho más amplio el movimiento efectivo resulta más pequeño.

- **max_angle** – al igual que el caso anterior, el ángulo máximo de movimiento de alguno de los motores puede estar limitado y no corresponderse con el rango real.

Además, se ofrecen métodos complementarios para facilitar el manejo de esta estructura:

```

70 /**
71 * Writes an angle directly into the servo register.
72 *
73 * @param servo - the servo to move.
74 * @param angle_rad - the angle to move.
75 */
76 void SERVO_write_angle(const servo_t *servo, double64_t angle_rad);

```

```

77 /**
78 * Writes a value of milliseconds as the angle to move the servo. This
79 * milliseconds equals the duty cycle period.
80 *
81 */
82 * @param servo - the servo to move.
83 * @param ms - the milliseconds representing the angle.
84 */
85 void SERVO_write_milliseconds(const servo_t *servo, double64_t ms);
86
87 /**
88 * Writes directly a value into the servo's register. Use this method
89 * with caution.
90 *
91 * @param servo - the servo to move.
92 * @param dtc_value - the duty cycle register value.
93 */
94 void SERVO_write_value(const servo_t *servo, uint16_t dtc_value);
95
96 /**
97 * Obtains the equivalent milliseconds stamp representing the given angle.
98 *
99 * @param angle_rad - the angle to transform.
100 * @return double64_t representing the milliseconds.
101 */
102 double64_t SERVO_from_angle_to_ms(double64_t angle_rad);

```

Dichos métodos permiten el control de los servomotores mediante un ángulo en radianes, mediante una posición según el tiempo en milisegundos o directamente mediante un valor del *duty cycle*.

Para el primero, se realiza una conversión de radianes a milisegundos, ya que el motor está definido para trabajar bajo distintos valores de *duty cycle*. Según la documentación del fabricante, los posibles valores de control del servomotor se encuentran entre (figura 7.10):

BASIC Stamp Module	0.75 ms	1.5 ms (center)	2.25 ms
BS1	75	150	225
BS2, BS2e, BS2pe	375	750	1125
BS2sx, BS2px, BS2p24/40	938	1875	2813

Figura 7.10: Valores de *duty cycle* límites según el fabricante [48].

La conversión en sí consiste en hacer un mapeo del valor de entrada, el cual responde a la siguiente ecuación (ecuación 7.2.2):

$$\begin{cases} x \in [n, m] \\ o \in [k, v] \end{cases}$$

$$o = \frac{(x - n) \cdot (v - k)}{(m - n) + k}$$

Así, el mapeo se haría con una entrada $x \in [0, \pi]$ rad y una salida $o \in [0, 75, 2, 25]$ ms, obteniendo así el *duty cycle* buscado.

Con respecto a trabajar directamente con la posición según el tiempo y el valor del *duty cycle*, la conversión es algo más compleja y sigue la ecuación 7.7:

$$DTC = \frac{F_{OSC}}{\frac{1000}{ms} \cdot PWM_{Prescaler}} \quad (7.7)$$

De esta manera se pueden generar *duty cycles* que van acorde al tiempo que ha de durar a nivel alto dicho valor.

Con respecto a la abstracción de motor, se define una estructura que maneja el *driver* definido anteriormente:

```

41 typedef struct {
42     /**
43      * Pointer storing the driver that manages the servomotor.
44      */
45     servo_t *servoHandler;
46
47     /**
48      * Motor unique identifier.
49      */
50     const uint8_t id;
51
52     /**
53      * The current movement duration, in us.
54      */
55     volatile double64_t movement_duration;
56
57     /**
58      * The current motor angle in us.
59      */
60     volatile double64_t angle_us;
61
62     /**
63      * Flag indicating that the motor has finished its movement.
64      */
65     volatile bool movement_finished;
66
67     /**
68      * Flag indicating whether the movement is clockwise or anticlockwise.

```

```

69 */  

70 int8_t clockwise;  

71  

72 /**  

73 * Volatile counter indicating the elapsed time since the movement started.  

74 */  

75 volatile time_t current_movement_count;  

76  

77 /**  

78 * Function that initializes the timer attached to the motor.  

79 */  

80 TMR_func TMR_Start;  

81  

82 /**  

83 * Function that finishes the timer attached to the motor.  

84 */  

85 TMR_func TMR_Stop;  

86 } motor_t;

```

Dicha estructura define:

- La dirección de memoria del *driver* del servo.
- Un identificador único para el motor.
- La duración, si procede, del movimiento que está realizando actualmente el motor.
- El ángulo actual, en microsegundos, en el que se encuentra el motor.
- *Flag* para indicar si el movimiento actual se ha finalizado.
- *Flag* de control indicando si el movimiento es en el sentido de las agujas del reloj o en sentido contrario.
- Función que permite iniciar el *timer* dedicado al movimiento del motor.
- Función que permite detener el *timer* dedicado al movimiento del motor.

El paquete anterior define una interfaz con métodos útiles para permitir el manejo de los motores:

```

88 /**
89 * Moves the motor to the specified angle in radians.
90 *
91 * @param motor a pointer to the motor to move.
92 * @param angle_rad the radians to move.
93 */
94 void MOTOR_move(motor_t *motor, double64_t angle_rad);
95
96 /**
97 * Freezes the motor at the current position.
98 *
99 * @param motor a pointer to the motor to freeze.
100 */

```

```

101 void MOTOR_freeze(motor_t *motor);
102 /**
103  * Performs the motor calibration.
104  *
105  * @param motor a pointer to the motor to calibrate.
106  * @return EXIT_SUCCESS if calibration is OK or EXIT_FAILURE in other case.
107  */
108 char MOTOR_calibrate(motor_t *motor);
109
110 /**
111  * Gets the motor position as us.
112  *
113  * @param motor a pointer to the motor.
114  * @return the position in microseconds.
115  */
116 double64_t MOTOR_position_us(motor_t *motor);
117
118 /**
119  * Gets the motor actual position in radians.
120  *
121  * @param motor a pointer to the motor.
122  * @return the position in radians.
123  */
124 double64_t MOTOR_position_rad(motor_t *motor);
125
126 /**
127  * Gets the motor actual position in degrees.
128  *
129  * @param motor a pointer to the motor.
130  * @return the position in degrees.
131  */
132 double64_t MOTOR_position_deg(motor_t *motor);

```

Cuadro 7.1: Métodos de `motor` y descripciones.

Método	Descripción
<code>void MOTOR_move(motor_t *motor, double64_t angle_rad)</code>	Planifica un movimiento controlado del robot e inicializa los <i>timers</i> para el control de la posición actual.
<code>void MOTOR_freeze(motor_t *motor)</code>	Detiene inmediatamente el movimiento de los motores, manteniendo fija la posición actual. Deshabilita el <i>timer</i> si estaba habilitado.
<code>void MOTOR_calibrate(motor_t *motor)</code>	Según el motor y su fin de carrera, se calibra y se actualizan los valores de <code>min_angle</code> y <code>max_angle</code> .

<code>double64_t MOTOR_home(motor_t motors[MAX_MOTORS])</code>	Itera sobre los motores de la lista y los mueve a la posición inicial.
<code>double64_t MOTOR_position_us(motor_t *motor)</code>	Obtiene el valor actual de la posición del motor en microsegundos.
<code>double64_t MOTOR_position_rad(motor_t *motor)</code>	Obtiene el valor de la posición actual del motor en radianes.
<code>double64_t MOTOR_position_deg(motor_t *motor)</code>	Obtiene el valor de la posición actual del motor en grados.

Un punto interesante del manejo de la posición de los motores es el método empleado. Los servomotores que se están empleando no cuentan con ningún tipo de retroalimentación que indique la posición en la que se encuentra más allá del control que pueda llevar el desarrollador a nivel de aplicación. Además, el sistema S2 permite cancelar un movimiento en ejecución, por lo que es necesario saber en cada instante de tiempo la posición del motor. Una ventaja que presentan los servomotores es que, si trabajan dentro de los límites establecidos (no mueven más carga de la especificada, etc.) presentan una velocidad de giro que puede considerarse constante.

Se plantea entonces el sistema S2 para manejar de forma precisa el punto en el que se encuentra el motor. Para ello, se establece una precisión de microsegundo y se inicializa un contador que es el encargado de llevar el registro del tiempo que ha pasado desde que un motor ha iniciado su movimiento.

Para ello, se definen tres *timers* que son iniciados independientemente y cuando un motor comienza a realizar un giro. Todos reciben la dirección de memoria en la que se encuentra el motor que están manejando y siguen una implementación similar, la cual se muestra a continuación:

```

55 void __attribute__((interrupt, no_auto_psv)) _T3Interrupt(void) {
56     TMR3_count += 1L;
57
58     if (TMR3_count >= duration)
59         TMR3_Stop();
60
61     IFS0bits.T3IF = 0;
62 }
63
64 void TMR3_Start(void) {
65     /* Clear old value*/
66     TMR3_count = .0F;
67     duration = (int_fast32_t) TMR3_motor->movement_duration;
68
69     /*Enable the interrupt*/
70     IEC0bits.T3IE = 1;
71
72     IFS0bits.T3IF = 0;

```

```

73
74     /* Start the Timer */
75     T3CONbits.TON = 1;
76 }
77
78 void TMR3_Stop(void) {
79     TMR3_motor->movement_finished = true;
80     BARRIER_arrive(TMR3_barrier);
81     // If movement is clockwise then add the count to current angle_us
82     // else, the count must be substracted
83     TMR3_motor->angle_us += (TMR3_motor->clockwise * TMR3_count);
84     /* Stop the Timer */
85     T3CONbits.TON = 0;
86
87     /*Disable the interrupt*/
88     IEC0bits.T3IE = 0;
89 }
```

El método anterior es simple pero fundamental para llevar el control de la posición:

1. Por una parte, se suma cada vez que es invocado el valor 1,001 6 μs al contador de tiempo propio del *timer*. Esto es debido a que, con la frecuencia de oscilación actual no es posible conseguir un valor exacto de 1 μs .
2. Si la cuenta de tiempo es mayor o igual que la duración del movimiento del motor quiere decir que se ha llegado a la posición indicada. Se indica entonces que el movimiento ha finalizado, que el servomotor ha llegado a la posición de destino y se actualiza la posición actual del motor sumando (o restando) el tiempo que lleva activo el *timer* con la posición anterior. Finalmente, detiene el *timer*.
3. En cualquier caso, desactiva el *flag* de la interrupción para evitar entrar múltiples veces en la rutina de tratamiento.

De esta manera, si el usuario decide cancelar el movimiento se tiene de manera bastante precisa la posición del servomotor en ese instante de tiempo. Tras realizar un estudio estadístico con 30 muestras³ (ya que no se encontró ninguna información en la documentación del fabricante) se estima que la velocidad de giro del motor es:

$$M_{Speed} \approx 5\,245,275\,04 \mu\text{s}/^\circ$$

Con la relación anterior se pueden establecer *timers* que actúan cada 1,001 6 $^-{\text{s}}$ para saber con alta precisión el punto aproximado en el que el motor se encuentra. La precisión se podría mejorar si:

- Se cuenta con la velocidad de giro precisa de los motores, dada por el fabricante.

³Se tomaron 30 marcas temporales en las que el motor se hizo girar una cantidad predefinida de grados (en particular, 90°), y se midió el tiempo que se tardaba en realizar dicho movimiento. Se obtuvo una media de $\bar{x} = 0,457\text{s}$ y una desviación estándar de $S(x) \approx 0,015\,074\,813\,4\text{s}$, por lo que se estimó (teniendo en cuenta el posible fallo humano) que en hacer $1^\circ \approx 0,005\,245\,275\,7\text{s} \Rightarrow M_{Speed} \approx 190,647\,747\,9\text{ }^\circ/\text{s}$.

- Se incrementa la precisión del *timer* a nivel nanométrico. Esta alternativa se planteó pero la interrupción se realizaba con una frecuencia suficientemente elevada como para que el sistema se volviera inestable.

Finalmente, se encuentra la abstracción de **planner**. En dicha abstracción se aúnan todos los motores que componen el brazo en la siguiente estructura:

```

37 typedef struct {
38     /**
39      * pArm base motor, which moves along Y axis.
40      */
41     motor_t *base_motor;
42
43     /**
44      * pArm lower arm motor, which moves along X axis.
45      */
46     motor_t *lower_arm;
47
48     /**
49      * pArm upper motor, which moves along Z axis.
50      */
51     motor_t *upper_arm;
52
53     /**
54      * pArm end-effector motor.
55      */
56     motor_t *end_effector_arm;
57 }
58 /**
59  * Simple struct containing pointers to every pArm motor.
60  */
61 motors_t;

```

En dicha estructura se guarda la dirección de memoria de cada uno de los motores que componen el brazo. La razón de ser de la misma es recoger, de forma ordenada, la cantidad de motores disponibles y poder utilizarlo a lo largo del SW.

La interfaz **planner** ofrece un nuevo nivel de abstracción sobre los dos anteriores para poder planificar movimientos completos o bien moviendo los tres motores del brazo robótico según unos ángulos de entrada $\{\theta_0, \theta_1, \theta_2\}$ o bien mediante la posición en coordenadas cartesianas del *end-effector*: $P_{ee} = \{x, y, z\}$. Además, se ofrecen maneras de conocer la posición exacta tanto en coordenadas angulares como en coordenadas cartesianas de los tres motores a la vez o del *end-effector*, respectivamente.

Se definen pues los siguientes métodos:

```

65 #ifdef LIMIT_SWITCH_ENABLED
66 /**
67  * Initialize the planner so movements can be done.
68  *
69  * @param barrier - the planner synchronization barrier.
70  * @param switch_map - an array shared with {@link interrupts.h#_CNInterrupt}.
71  */
72 void PLANNER_init(barrier_t *barrier, uint_fast8_t switch_map[4]);

```

```
73 #else
74 /**
75 * Initialize the planner so movements can be done.
76 *
77 * @param barrier - the planner synchronization barrier.
78 */
79 void PLANNER_init(volatile barrier_t *barrier);
80 #endif
81
82 /**
83 * Moves every motor to its home position.
84 *
85 * @return the time, in us, that will pass until the movement is done.
86 */
87 double64_t PLANNER_go_home(void);
88
89 /**
90 * Moves to the specified position at point (x, y, z).
91 *
92 * @param xyz - the final position.
93 * @return the time, in us, that will pass until the movement is done.
94 */
95 double64_t PLANNER_move_xyz(point_t xyz);
96
97 /**
98 * Moves each motor to the specified angle (t0, t1, t2).
99 *
100 * @param angle - the motors' angles.
101 * @return the time, in us, that will pass until the movement is done.
102 */
103 double64_t PLANNER_move_angle(angle_t angle);
104
105 /**
106 * Moves the specified angle and waits until the movement is completed.
107 *
108 * @param angle - the motors' angles.
109 */
110 void PLANNER_move_waiting(angle_t angle);
111
112 /**
113 * Immediately stops any possible movement of the motors.
114 *
115 * @return EXIT_SUCCESS if there was any movement or EXIT_FAILURE if none.
116 */
117 uint8_t PLANNER_stop_moving(void);
118
119 /**
120 * Obtains the current end-effector position, in terms of (x, y, z).
121 *
122 * @return a pointer to the position.
123 */
124 point_t *PLANNER_get_position(void);
125
126 /**
127 * Obtains the current motor angles, in terms of (t0, t1, t2).
128 *
```

```

129 * @return a pointer to the angles.
130 */
131 angle_t *PLANNER_get_angles(void);

```

Cuadro 7.2: Métodos de planner y descripciones.

Método	Descripción
<code>void PLANNER_init(barrier_t *barrier)</code>	Inicializa la barrera que controla cuándo los motores han llegado a la posición final.
<code>double64_t PLANNER_go_home(void)</code>	Mueve a los tres motores a la posición inicial. Devuelve el tiempo esperado.
<code>double64_t PLANNER_move_xyz(point_t xyz)</code>	Según el punto dado $\{x, y, z\}$ mueve los motores para alcanzar dicha posición. Devuelve el tiempo esperado.
<code>double64_t PLANNER_move_angle(angle_t angle)</code>	Según el ángulo dado $\{\theta_0, \theta_1, \theta_2\}$ mueve los motores hasta alcanzar dicho ángulo. Devuelve el tiempo esperado.
<code>void PLANNER_move_waiting(angle_t angle)</code>	Mueve los motores bloqueando el hilo principal de ejecución.
<code>void PLANNER_stop_moving(void)</code>	Detiene el movimiento inmediatamente de los tres motores.
<code>point_t *PLANNER_get_position(void)</code>	Devuelve la coordenada $\{x, y, z\}$ del <i>end-effector</i> .
<code>angle_t *PLANNER_get_angles(void)</code>	Devuelve los ángulos $\{\theta_0, \theta_1, \theta_2\}$ del <i>end-effector</i> .

7.2.3. Cálculo de movimientos/trayectorias

El cálculo de los movimientos según los puntos recibidos se realizan utilizando tanto la cinemática directa como la cinemática inversa, explicadas previamente en el capítulo 5.

Dichas cinemáticas se implementan en el paquete `kinematics`, implementado en los listados de códigos C.5 y C.32.

Se definen dos estructuras que permiten manejar los ángulos y los puntos:

```

51 // Position definitions
52 #ifndef point_t

```

```

53
54 typedef struct {
55     double64_t x;
56     double64_t y;
57     double64_t z;
58 } point_t;
59 #define point_t point_t
60 #endif
61
62 // Angle definitions
63 #ifndef angle_t
64
65 typedef struct {
66     double64_t theta0;
67     double64_t theta1;
68     double64_t theta2;
69 } angle_t;
70 #define angle_t angle_t
71 #endif

```

Con dichos tipos de datos, las funciones dedicadas a la cinemática directa como la cinemática inversa computan el valor de entrada y actualizan la dirección de memoria en la que se espera obtener el valor resultante.

En principio, los métodos anteriores suelen funcionar correctamente pero se establecen unos casos en los cuales la obtención del valor falla:

- Cuando alguno de los ángulos provistos/obtenidos es **NaN**.
- Si alguno de los valores provistos en la función `arc cos` produce un resultado mayor a uno, provocando que dicha función falle.
- Si alguno de los valores de entrada no se encuentra dentro de los rangos de los motores.

En todos los casos anteriores, la función falla con un código de error `EXIT_FAILURE`.

7.2.4. Interpretación del pseudo-lenguaje

Las comunicaciones entre los sistemas S2 y S1 se realiza mediante UART pero los mensajes no se componen de un lenguaje propio sino que se utiliza una adaptación del GCode.

El GCode es el nombre que recibe habitualmente el lenguaje de programación más usado de control numérico, el cual posee múltiples implementaciones [54]. La estructura típica de un código en GCode se compone de una orden y una serie de parámetros que pueden ser opcionales. Según la web de RepRap, se definen los siguientes tipos de GCode [55]:

- **Gnnn** – Comando GCode estándar, como moverse hasta un punto.
- **Mnnn** – Comando definido por RepRap, como encender un ventilador.

- **Tnnn** – Seleccionar la herramienta nnn. En RepRap, las herramientas son extrusores.
- **Snnn** – Parámetro de comando, como la tensión enviada a un motor.
- **Pnnn** – Parámetro de comando, como el tiempo en milisegundos.
- **Xnnn** – Una coordenada X, normalmente para moverse a ella. Puede ser un número entero o racional.
- **Ynnn** – Una coordenada Y, normalmente para moverse a ella. Puede ser un número entero o racional.
- **Znnn** – Una coordenada Z, normalmente para moverse a ella. Puede ser un número entero o racional.
- **Innn** – Parámetro - Actualmente no utilizado.
- **Jnnn** – Parámetro - Actualmente no utilizado.
- **Fnnn** – *Feedrate* en mm por minuto. (Velocidad de movimiento del cabezal de impresión).
- **Rnnn** – Parámetro - usado para temperaturas.
- **Qnnn** – Parámetro - Actualmente no utilizado.
- **Ennn** – Longitud a extruir en mm. Ex exactamente como X, Y y Z, pero para la cantidad de filamento a extruir. Es común que los nuevos sistemas basados en pasos lo interpreten... Mejor: Skeinforge 40 y siguientes interpretan esto como la longitud absoluta de filamento insertado, no como la longitud de la extrusión que sale.
- **Nnnn** – Número de línea. Utilizado para pedir la repetición de la transmisión en caso de errores de comunicación.
- ***nnn** – Checksum. Usado para comprobar errores de comunicación.

En este proyecto se han utilizado los siguientes GCode:

- **Gnnn** – para indicar movimientos hasta un punto o coordenada angular.
- **Mnnn** – para indicar órdenes, como detener el movimiento u obtener la posición actual.
- **Innn** – órdenes relacionadas con el intercambio de mensajes en RSA.
- **Jnnn** – órdenes que indican distintos tipos de errores durante la ejecución.

Durante la ejecución, los caracteres recibidos por la UART son guardados en un *buffer* temporal de hasta 1024 caracteres. Una vez se recibe un salto de línea, el *buffer* es copiado a una cadena de caracteres de tamaño fijo, para no emplear más memoria de la necesaria. Dicha acción activa además un *flag* que indica que un nuevo mensaje ha sido recibido, delegando su tratamiento al bucle principal.

La gestión de los bytes recibidos se realiza en la rutina de tratamiento de interrupción de la UART:

```
57 void __attribute__((__interrupt__, no_auto_psv)) _U1RXInterrupt(void) {
58     IFS0bits.U1RXIF = 0;
59     if (U1STAbits.FERR == 1)
60         return;
61     if (U1STAbits.URXDA == 1) {
62         char received_val = U1RXREG;
63         PORTBbits.RB5 = received_val;
64 #ifdef CLI_MODE
65         printf("%c", received_val);
66 #endif
67         if (received_val == '\n' || received_val == '\r') {
68             if (urx_order == NULL) {
69 #ifdef DEBUG_ENABLED
70                 printf("[DEBUG]\tU1RX not initialized!\n");
71 #endif
72             }
73         }
74 #ifdef CLI_MODE
75         printf("\n");
76 #endif
77         uart_buffer[uart_chars++] = '\0';
78 #ifdef DEBUG_ENABLED
79         printf("[DEBUG]\tNew order received: '%s'\n", uart_buffer);
80 #endif
81         if (urx_order->order_buffer == NULL) {
82             urx_order->order_buffer = BUFFER_create(uart_chars);
83         }
84         if (urx_order->order_buffer->size != uart_chars) {
85             BUFFER_update_size(urx_order->order_buffer, uart_chars);
86         }
87         if (urx_order->order_buffer->buffer == NULL) {
88 #ifdef DEBUG_ENABLED
89             printf("[ERROR]\tFailed to allocate %dB for order!\n",
90                   (uart_chars * sizeof (char)));
91 #endif
92             BUFFER_free(urx_order->order_buffer);
93             uart_chars = 0U;
94             return;
95         }
96         strncpy(urx_order->order_buffer->buffer,
97                 uart_buffer,
98                 urx_order->order_buffer->size);
99         uart_chars = 0U;
100        urx_order->message_received = true;
101    } else {
102        uart_buffer[uart_chars++] = received_val;
103        if (uart_chars >= 1024) {
104            // UART buffer overflow...
105            // Release memory and ignore instruction
106            uart_chars = 0;
107            printf("J11\n");
108        }
109    }
110 }
```

y el GCode es interpretado en el paquete `gcode`, definido por los listados de código C.2 y C.29. Actualmente, S2 interpreta y entiende las siguientes órdenes (tabla 7.3):

Cuadro 7.3: Órdenes GCode interpretadas por S2.

Orden	Funcionalidad
G0 Xnnn Ynnn Znnn	Orden para dirigir el <i>end-effector</i> al punto $\{x, y, z\}$ indicado.
G1 Xnnn Ynnn Znnn	Orden para mover los motores los ángulos $\{\theta_0, \theta_1, \theta_2\}$ equivalentes a los parámetros recibidos.
G28	Orden para mover los motores a la posición inicial, es decir, posicionar el <i>end-effector</i> en $P_{ee} = \{0, 0, 0\}$.
M1	Parada incondicional. Detiene cualquier movimiento y fija los motores en la posición resultante. Contesta con otra orden M1 cuando se han detenido definitivamente los motores.
M114	Obtener la posición actual del <i>end-effector</i> . Responde con un código G0 junto con la posición.
M280	Obtener la posición actual a nivel de coordenadas angulares. Responde con un código G1 junto con las coordenadas angulares.
I1	Comando personalizado que indica la intención de obtener el módulo ‘n’ y la clave pública ‘e’ en RSA. Responde con las órdenes I2, I3, I4 en caso de que el dispositivo esté autorizado para pedir las claves. En otro caso, responde con un J8.
I2 n	Comando personalizado que indica el módulo ‘n’ de la clave RSA. Solo es invocado tras una orden I1 exitosa.
I3 e	Comando personalizado que indica la clave pública ‘e’ de la clave RSA. Solo es invocado tras una orden I1 exitosa.
I4 msg	Comando personalizado que envía un mensaje firmado con la clave privada ‘d’. Solo es invocado tras una orden I1 exitosa.
I5 msg	Comando personalizado que recibe un mensaje encriptado para verificar al emisor. Si la verificación es válida se responde con I5 sin ningún mensaje.
I6 msg	Comando personalizado que fuerza la generación de un nuevo par de claves RSA. Solo puede ser invocado tras autentificarse con el mensaje aleatorio enviado previamente cifrado.
I7 msg	Comando personalizado que funciona como <i>heartbeat</i> .
J1	Comando personalizado utilizado como ACK.
J2	Comando personalizado que indica un error de calibración en los motores.

J3	Comando personalizado que indica un error de GCode desconocido.
J4	Comando personalizado que indica un error de fuera de rango.
J5	Comando personalizado que indica un error de cancelación sin movimiento (no implementado).
J6	Comando personalizado que indica un error durante el <i>handshake</i> .
J7	Comando personalizado que indica un error de múltiples movimientos enviados.
J8	Comando personalizado que indica un error de falta de coordenadas para G0.
J9	Comando personalizado que indica un error de falta de coordenadas para G1.
J10	Comando personalizado que indica un error de dispositivo de no confianza.
J11	Comando personalizado que indica un error de <i>buffer overflow</i> de la UART.

Cuando se recibe una orden esta es guardada junto con toda la información relativa a ella en una estructura del tipo `order_t`:

```

123 #ifndef order_t
124
125 typedef struct {
126     /**
127      * Flag active when a new message is received through UART port.
128      * It is updated at <pre>interrupts.h#_U1RXInterrupt</pre>.
129      *
130      * @type bool
131      * @see interrupts.h#_U1RXInterrupt
132      */
133     bool message_received;
134
135     /**
136      * Buffer which contains the order received by the UART. It has fixed
137      * size so no extra space is used. This variable is updated at
138      * <pre>interrupts.h#_U1RXInterrupt</pre>.
139      *
140      * @type buffer_t
141      * @see interrupts.h#_U1RXInterrupt
142      */
143     buffer_t *order_buffer;
144 }
145 /**
146  * Order container with all the required information for managing
147  * the UART messages.
148  */
149 order_t;
150 #define order_t order_t
151 #endif

```

Dicha estructura contiene un `buffer_t` con la orden recibida y un `flag` indicando si hay una nueva orden lista para ser interpretada. El bucle principal de ejecución comprueba en cada iteración si hay algún mensaje pendiente y, en caso afirmativo, delega la interpretación al paquete `gcode` (listados de código C.2 y C.29). Una vez interpretada la línea recibida, se devuelve el control al bucle principal y se realizan las acciones oportunas.

7.2.5. *Heartbeat* y cifrado RSA

A la hora de mantener una comunicación activa y evitar el control del brazo por algún otro dispositivo se ha implementado el sistema de cifrado RSA tanto en el SW de S1 como en el de S2.

Mediante el intercambio de mensajes en GCode un dispositivo externo puede autenticarse contra la placa y esta verificarle, permitiendo seguir con las comunicaciones. Para ello, se sigue el siguiente esquema:

- Un dispositivo envía una orden I1 al sistema S2. Si dicho sistema no tiene todavía ningún dispositivo de confianza, acepta la orden I1. En otro caso, responderá con una orden J10.
- Si se acepta la orden I1 entonces se procede a enviar el módulo ‘n’ y la clave pública ‘e’:
 - El módulo se manda mediante una orden I2 n.
 - La clave pública se manda mediante una orden I3 e.
- Además, se manda un mensaje aleatorio firmado con la clave privada ‘d’ donde el otro dispositivo habrá de “desfirmarla” y volver a encriptarla, para mandarlo de vuelta al sistema S2.
 - El mensaje aleatorio firmado se manda con una orden I4 msg.
- El sistema S2 entonces queda a la espera de recibir el mensaje cifrado por el sistema S1, donde lo desencriptará y comprobará el valor obtenido. Si el mensaje coincide, se acepta al nuevo dispositivo como de confianza y se podrán seguir con las comunicaciones. Se desactiva la orden I1.
 - El mensaje cifrado ha de enviarse con un I5 msg, donde S2 responderá con la misma orden vacía.

En el siguiente diagrama (figura 7.11) se pretende mostrar el comportamiento anterior:

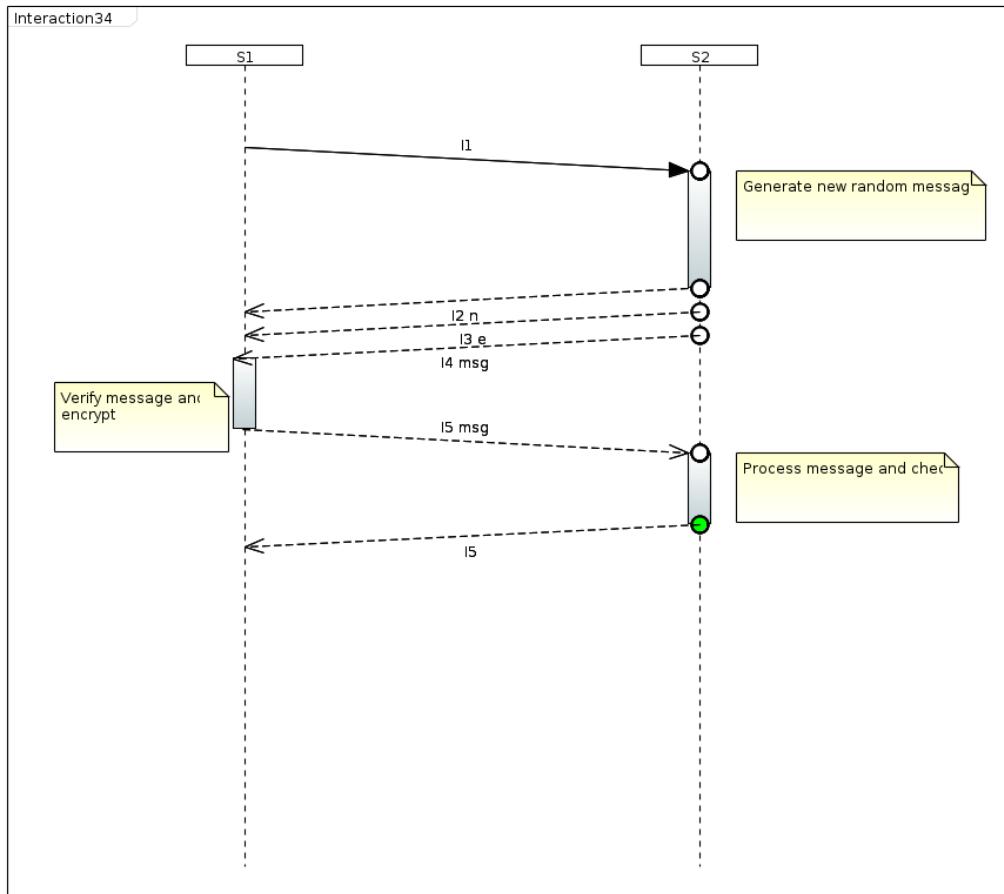


Figura 7.11: Diagrama de secuencia para el intercambio de las claves RSA.

Una vez se ha establecido la comunicación entre dispositivos es necesario mantenerla activa, esto es, enviar mensajes periódicamente. Para ello se ha definido una función de *heartbeat* que mantiene dicha comunicación activa. Cada 200 ms se espera un mensaje desde S1 del tipo I7 msg, con el mensaje aleatorio intercambiado anteriormente cifrado. S2 lo verificará y si coincide con el mensaje propio, actualiza el intervalo de tiempo.

Si se producen 5 fallos consecutivos (es decir, no hay mensajes válidos durante un segundo) se deja de confiar en el dispositivo y es necesario volver a hacer todo el proceso de intercambio de claves. Además, se generan nuevas claves RSA por mayor seguridad.

Esta lógica está implementada en el fichero `main.c` (listado de código C.48).

7.2.6. Opciones de compilación

Durante la compilación es necesario definir opciones adicionales a las básicas establecidas por defecto, ya que en otro caso la compilación fallará o no contendrá todas las características esperables.

Por una parte, es necesario definir que se trabaje con C99. Esto se hace definiendo la opción del compilador adicional `-std=gnu99`. Además, es recomendable trabajar con `double` de 64 bits, pero en otro caso se utilizarán tipos de datos propios para trabajar con esa precisión:

```
43 // Double precision double type
44 #ifndef double64_t
45 #if DBL_MANT_DIG < LDBL_MANT_DIG
46 typedef long double double64_t;
47 #else
48 typedef double double64_t;
49 #endif
50 #define double64_t double64_t
51 #endif
```

Es recomendable designar al menos 8 KB de memoria RAM dedicada a memoria *heap* para la creación de espacios de memoria reservados a variables de forma dinámica. Esto permite que funciones como `void* malloc(size_t)` o `void* realloc(void*, size_t)` no fallen y reserven la cantidad de memoria necesaria.

En lo referente a los modelos de memoria, se pueden establecer todos ellos como *small*: `code model`, `memory model`, `data model`. Por otra parte, es recomendable habilitar el nivel de optimización del código `-O2`, de forma que las librerías auxiliares que se utilizan son optimizadas para reducir el tamaño y mejorar el rendimiento. Además, se recomienda marcar las siguientes opciones:

- “*Do not override inline*”, para mantener la estructura de las funciones `inline`.
- “*Unroll loops*”, donde se aplica la técnica de optimización de bucles en donde, si se conoce de antemano el tamaño del mismo, se desenrolla en múltiples instrucciones que son más rápidas en tiempo de ejecución pero consumen más memoria de programa.
- “*Align arrays*”, para establecer el modelo de memoria que distribuye las posiciones que componen un *array*.

Para reducir el tamaño del código, se recomienda habilitar la opción del *linker* para eliminar los segmentos de código no utilizados en el programa, con la opción “*Remove unused sections*”.

Finalmente, se proveen las siguientes macros del compilador para habilitar distintas opciones a lo largo del código:

- `PRINTF_INCLUDE_CONFIG_H` – se usa una configuración propia para la librería adaptada de `printf`.
- `USE_CUSTOM_PRINTF` – se utiliza una versión personalizada de `printf` en lugar de la provista por `stdio.h`.
- `DEBUG_ENABLED` – añade nuevas opciones de depuración al código.
- `CONFIG_SIMULATOR` – si se está trabajando con el simulador, para deshabilitar opciones no usadas.
- `CLI_MODE` – para habilitar el modo de control mediante consola de comandos, comunicando directamente desde la UART.

- USE_MOTOR_TMRS – para usar un *timer* por cada motor en lugar del contador global. Permite obtener mayor precisión pero hay un *delay* apreciable en la ejecución de movimientos.
- LIMIT_SWITCH_ENABLED – utiliza los fines de carrera para detectar la posición de los motores.

En la tabla 7.4 se resumen las opciones habilitadas y empleadas:

Cuadro 7.4: Opciones de compilación definidas para S2.

Opción de compilación	Valor
Optimization level	2
Code model	Small
Data model	Small
Remove unused sections	<input checked="" type="checkbox"/>
Heap size	8 192
Use legacy libc	<input checked="" type="checkbox"/>
Fast floating point math	<input checked="" type="checkbox"/>
Additional options	-std=gnu99
Allow call optimization	<input checked="" type="checkbox"/>
Generate debug info	<input checked="" type="checkbox"/> (modo <i>debug</i>) – <input type="checkbox"/> (producción)
Use 64 bit double	<input checked="" type="checkbox"/>
Scalar mode	Small
Do not override 'inline'	<input checked="" type="checkbox"/>
Unroll loops	<input checked="" type="checkbox"/>
Align arrays	<input checked="" type="checkbox"/>
Smart IO forwarding level	1
Min stack size	16
Use local stack	<input checked="" type="checkbox"/>
Init data sections	<input checked="" type="checkbox"/>
Pack data template	<input checked="" type="checkbox"/>
Create handles	<input checked="" type="checkbox"/>
Create default ISR	<input checked="" type="checkbox"/>
Define common macros	[PRINTF_USE_CONFIG_H, USE_CUSTOM_PRINTF, DEBUG_ENABLED, CLI_MODE, CONFIG_SIMULATOR, USE_MOTOR_TMRS, LIMIT_SWITCH_ENABLED]

Capítulo 8

Impresión 3D

Las razones por las cuales se toma la decisión de fabricar la estructura del brazo mediante impresión 3D son:

- Cumplir con el objetivo de replicabilidad y asequibilidad: una de las bases del proyecto es que pueda ser reproducible a bajo coste tanto de recursos como de tiempo. Se decide por tanto construir la estructura física del brazo mediante técnicas de impresión 3D, ya que están altamente extendidas y son cada vez más asequibles.
- Características físicas del material: los plásticos utilizado en impresión 3D suelen ser ligeros y suficientemente resistentes para soportar las cargas para las que está pensado el manipulador.
- Disponibilidad de impresora 3D: dado que la Universidad es capaz de proveer al equipo con una impresora 3D, los costes del proyecto se abaratan si la estructura es realizada con los medios de los que la ya se disponen.
- Simplificar el proceso de mejora y personalización: debido a la naturaleza OS y OH del proyecto, se espera que las personas interesadas puedan contribuir a él, mejorándolo y/o personalizándolo. Además, la impresión 3D facilita estas acciones.

En particular, la impresora que la Universidad pone a disposición del equipo de trabajo es la “*Ultimaker 3 Extended*”, la cual es capaz de imprimir en una alta variedad de materiales, de los cuales destacan los siguientes:

- PLA [56]: este material permite imprimir con alta precisión dimensional y una resistencia a la tracción excepcional, el cual soporta grandes velocidades de impresión y es biodegradable, ya que se obtienen a partir de almidón de maíz, de yuca, mandioca o de caña de azúcar.
- Acrilonitrilo Butadieno Estireno (ABS) [57]: material que presenta buena adhesión entre capas y una resistencia a temperaturas de hasta 85 °C. Permite obtener buenos detalles estéticos.

- Ultimaker Nylon: este material es un tipo de poliamida basada en los polímeros plásticos PA6/66. Presenta una absorción de humedad reducida así como una capacidad considerable de resistencia ante tensiones mecánicas junto con un bajo coeficiente de fricción, haciéndolo un material ideal para construcciones mecánicas.
- CPE y CPE+: este material presenta una alta estabilidad dimensional, con buena resistencia al impacto y a la temperatura. Debido a su alta solidez y su estabilidad dimensional ofrece un buen rendimiento mecánico y gran resistencia al desgaste.

Debido a la naturaleza mecánica del proyecto, el equipo ha decidido emplear materiales con alta resistencia mecánica para las piezas móviles. El Ultimaker Nylon junto con el CPE cumplen con dicha característica.

Por otro lado, los componentes que no sean móviles como carcasas o piezas protectoras se imprimirán en PLA ya que, tras realizar pruebas, el equipo de desarrollo ha concluido que el material es lo suficientemente resistente para soportar los pesos a los que será sometido.

8.1. El proceso de impresión 3D

El proceso de impresión 3D se compone de varias etapas y de bastantes pruebas hasta que se consiguen resultados aceptables. En el momento de impresión, el brazo se compone de unas 27 piezas, pero se han tenido que imprimir un total de 75 piezas, lo que supone un 277.7% más en comparación con las piezas que componen al *pArm* (figura 8.1).



Figura 8.1: Algunas de las piezas que no se imprimieron correctamente.

Dicha cantidad de piezas fallidas se ha producido por distintas causas:

- Inexperiencia por parte del equipo de desarrollo, ya que era la primera vez que se trabajaba con impresoras de este estilo.
- Invalides de las piezas originales: como se ha comentado en el apartado de diseño 3D (sección 6.1), muchas de las piezas originales no eran válidas para ser impresas en 3D, por lo que fue necesario volver a imprimirlas tras comprobar que en efecto presentaban errores.
- Mal estado de uno de los *nozzle* de la impresora, en particular, el cabezal tipo BB de 0,4 mm.
- Mal estado del material de soporte de PVA, por lo que ciertas piezas no pudieron ser impresas.
- Falta de material de impresión para ciertas partes, como PVA y CPE.

Todos los sucedidos contratiempos se explican con mayor detalle en el apartado 11.4.

Tras las pruebas fallidas se fue perfeccionando el proceso de impresión 3D hasta llegar al punto en que todas las piezas que se imprimían salían bien en el primer intento, lo cual se explica a continuación.

8.1.1. El entorno de impresión 3D

El entorno de producción y preparación 3D es el Ultimaker Cura el cual ofrece una interfaz para poder posicionar piezas y realizar ajustes sobre las mismas (figura 8.2).

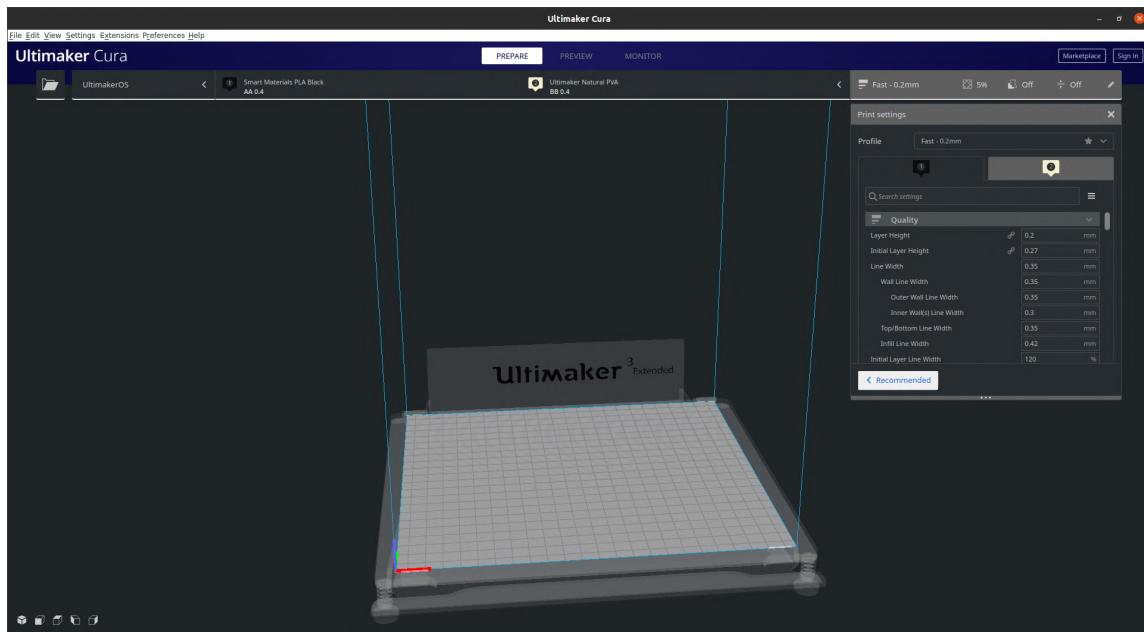


Figura 8.2: Entorno de producción de Cura.

La herramienta se encuentra disponible para todas las plataformas y su descarga puede hacerse desde la web oficial de Ultimaker¹.

Cuando se coloca una pieza, se ha de mover y preparar para ser impresa. La impresión siempre se hace desde las capas inferiores hasta las capas superiores, por lo que es posible imprimir múltiples piezas simultáneamente. Es recomendable que las piezas que se impriman no asciendan muy rápido verticalmente, ya que eso implicará que se necesita mayor soporte y estabilidad vertical. Por ejemplo, en la figura 8.3 se muestra una pieza que es casi completamente vertical, lo cual le supondrá un gran esfuerzo a la impresora a la hora de mantenerla estable y asegurar que salga bien.

¹<https://ultimaker.com/es/software/ultimaker-cura>

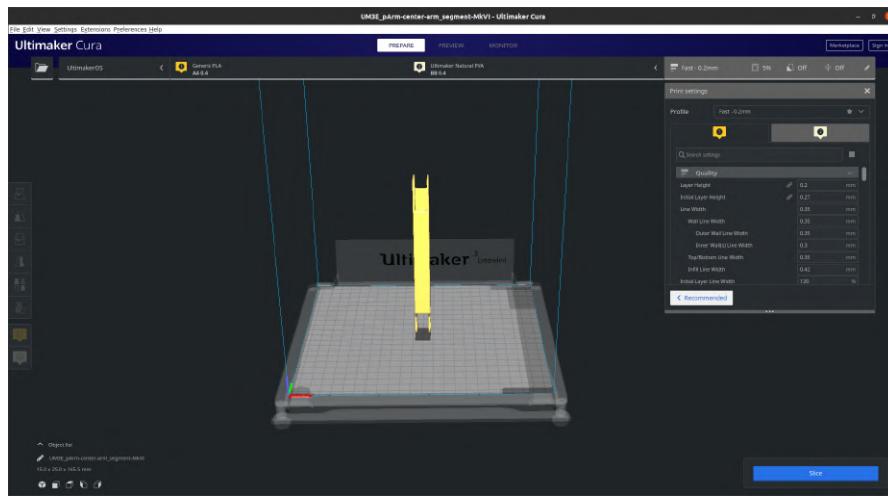


Figura 8.3: Pieza del *pArm* casi completamente vertical.

Para poder imprimir la pieza en esa posición correctamente es necesario añadir material de soporte ya que, en otro caso, es muy probable que se caiga.

Por lo general, es recomendable evitar el uso de dicho material por múltiples motivos:

- Es un material especialmente caro: una bobina de 750 g cuesta 85.14 €, por lo que interesa optimizar su uso.
- Es un material muy sensible al ambiente: al ser soluble, absorbe mucha humedad del ambiente y se acaba pudriendo si además está expuesto a la luz.
- A raíz de lo anterior, aunque el material parezca estar en buen estado, a la hora de imprimir pueden salir grumos de color oscuro, signos de que el material se ha podrido al ser expuesto a la temperatura del *nozzle*.
- Del mismo modo, en una impresión de larga duración el PVA se puede deteriorar con el tiempo, ya que estará expuesta a la temperatura constante del *build plate* así como al flujo constante de aire de los ventiladores del cabezal de impresión.
- Al utilizar doble extrusor pero no funcionar a la vez, los tiempos de impresión se multiplican ya que hay que recorrer dos veces la superficie que se ha impreso además de tener que dedicar tiempo a intercambiar los extrusores.

En el apartado 11.4 se explica con más detalle los problemas que se encontraron al imprimir en 3D utilizando dicho material y las soluciones que se plantearon.

Antes de utilizar doble extrusión para imprimir se puede modificar la posición y orientación de la pieza desde el SW de Cura para ver si se puede posicionar de forma que pueda ser impresa con un único material. La pieza anterior puede ser tumbada para que el crecimiento sea más progresivo, quedando de la siguiente forma (figura 8.4):

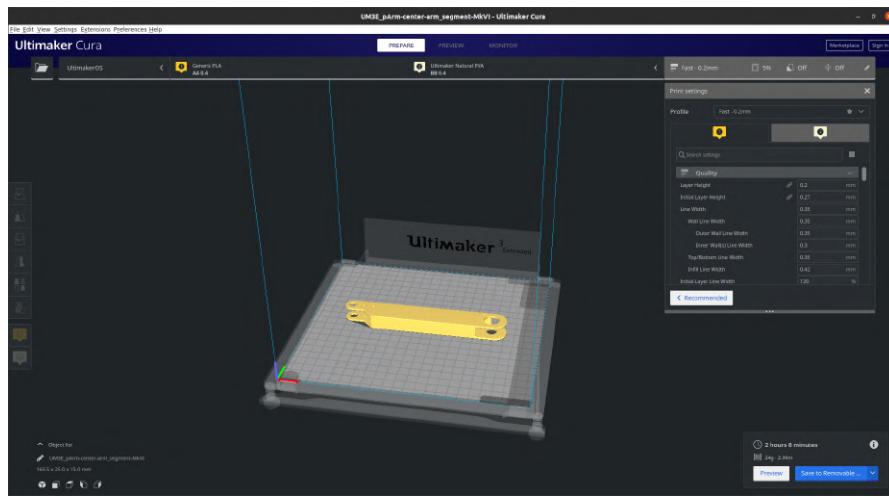


Figura 8.4: Pieza del *pArm* en otra posición para que sea más sencilla de imprimir.

Pese a haber colocado la pieza en dicha posición, el SW Cura indica que la impresora puede tener problemas al crearla. Esto se muestra marcando el segmento que se considera problemático de color rojo (figura 8.5):

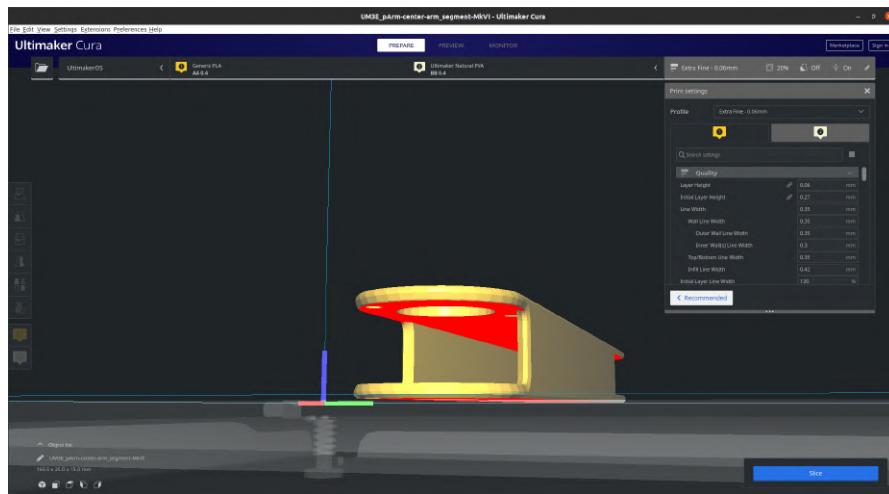


Figura 8.5: El “techo” de la pieza se marca de color rojo.

Esto es porque, al ser el “techo” de la pieza y al estar hueca, no hay una capa sobre la que apoyar el material plástico que completaría la figura. Por ende, es necesario añadir material de soporte (en PVA o PLA) de forma que se puede imprimir correctamente la pieza (figura 8.6):

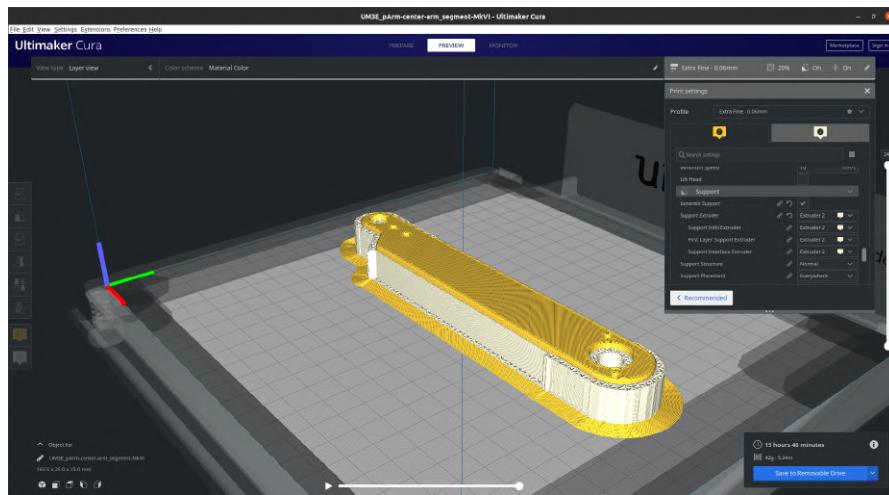


Figura 8.6: Con el material de soporte la pieza es imprimible.

Según el material en uso, se han de adaptar ciertas configuraciones para intentar que las impresiones salgan bien, las cuales se detallan a continuación.

8.1.2. Los parámetros de configuración

Como se ha mencionado anteriormente, el *pArm* está compuesto principalmente por:

- PLA para piezas de soporte, que no van a moverse sobre otras piezas.
- CPE para piezas mecánicas que necesitan transmitir fuerzas a través de la estructura mecánica del brazo.

Para el primero de los materiales, la configuración básica que ofrece el SW Cura es suficiente, ya que es de los materiales más utilizados a la hora de imprimir. En cambio, con el CPE en conjunto con PVA como material de soporte las opciones de impresión han de ser configuradas con cierto rigor.

El material de soporte puede resultar bastante problemático en impresiones de larga duración y acarrea consigo un mantenimiento que hay que realizar después de cada impresión. Las impresoras Ultimaker necesitan tener conectados ambos extrusores al cabezal de impresión, ya que utilizan el calor que pueden generar individualmente para crear retroalimentación positiva entre ellos, esto es, se contribuyen mutuamente para alcanzar la temperatura deseada (referencia en la imagen 8.7).



Figura 8.7: Vista del cabezal de impresión junto con los extrusores de la Ultimaker 3 [59].

Uno de los problemas de que el extrusor esté siempre contribuyendo a la temperatura y al flujo de aire se produce cuando además tiene plástico PVA en su interior. Como se ha mencionado anteriormente, el PVA absorbe con mucha facilidad la humedad del aire y, al someterlo a calor, se pudre. Por ello, las impresiones de larga duración suelen tener problemas con dicho material ya que durante el tiempo que no se está utilizando sigue absorbiendo humedad y, al estar en constante contacto con el calor, se empieza a pudrir, provocando que cuando se quiere hacer un soporte con dicho material no se pueda.

Después de muchas pruebas, se ha conseguido llegar finalmente a una configuración en la que la impresión con dicho material suele tener resultados satisfactorios. Dicha configuración se basa en:

- Retirar el plástico del *nozzle* cuando se ha dejado de imprimir, de forma que no está expuesto al calor de forma tan directa.
- Reducir la temperatura del extrusor cuando no se está utilizando. Esto permite evitar que se pudra el plástico que pudiera quedar en el *nozzle* y reduce el flujo de calor que llega al plástico retraído.
- Reducir la velocidad de impresión con PVA, de manera que cuando se extruye el material se hace de forma precisa, evitando posibles problemas al realizarlo a mayor velocidad.
- Aplicar una velocidad de giro del ventilador de impresión constante, entre el 1% y el 3%. De esta manera se contribuye al secado del PVA cuando ya ha sido extruído sin aplicar un flujo constante de aire al plástico que está en el *hot end*.
- Utilizar el *prime tower*, una construcción que permite limpiar el plástico que va a ser utilizado para imprimir.

- Retractar el eje *Z* cuando se hace el cambio de materiales o no se está imprimiendo. Pese a que esta característica no afecta directamente al PVA, se evitan posibles accidentes de que el cabezal de impresión pueda chocar con alguna pieza que esté siendo impresa.
- Reducir la temperatura del extrusor cuando se imprime PVA, unos grados por debajo de la de por defecto pero que todavía es útil para poder imprimir.

A continuación se detalla en la tabla 8.1 las características de impresión definidas para poder trabajar con múltiples materiales junto con el material de soporte:

Cuadro 8.1: Configuración de la Ultimaker 3 para generar material de soporte en PVA.

Configuración	Valor	Descripción
Layer height	[0,06 mm, 0,1 mm, 0,15 mm, 0,2 mm]	El tamaño de la capa de plástico que es utilizada para crecer verticalmente. Valores pequeños otorgan más precisión pero incrementan el tiempo de impresión.
Enable Ironing	<input checked="" type="checkbox"/>	Vuelve a recorrer la última capa impresa extruyendo muy poco material para suavizarla al tacto.
Infill density	20 % - 80 %	Cantidad de material introducido en capas huecas. Mayor material incrementa la resistencia pero aumenta el peso y el tiempo de impresión. Se recomienda no usar valores por encima del 50 %.
Infill pattern	<i>Tri-Hexagon</i>	El patrón de relleno del <i>infill</i> . Por defecto son triángulos, pero “ <i>tri-hexagon</i> ” gasta más material a cambio de mayor soporte físico.
Printing temperature	215 °C	Para el PVA, se establece la temperatura de impresión en el rango habitual.
Final printing temperature	200 °C	La temperatura final de acabado de impresión (para el PVA).
Standby temperature	60 °C	La temperatura que se establece en el extrusor cuando no se está utilizando activamente.
Enable retraction	<input checked="" type="checkbox"/>	Retractar el eje <i>Z</i> cuando el <i>nozzle</i> se mueve sobre un área sin objetos impresos.
Retraction at layer change	<input checked="" type="checkbox"/>	Retractar el <i>nozzle</i> cuando se avanza a la siguiente capa de impresión.
Retraction distance	4,5 mm	La cantidad de material que es recogido cuando se retracta el <i>nozzle</i> .
Avoid printed parts when travelling	<input checked="" type="checkbox"/>	Si es posible, evitar mover el cabezal de impresión sobre zonas ya impresas.

Avoid supports when travelling	<input checked="" type="checkbox"/>	Si es posible, evitar las partes impresas con material soporte.
Z hop when retracted	<input checked="" type="checkbox"/>	Cuando se mueve la cabeza de impresión, se baja además el <i>build plate</i> para evitar colisiones.
Z hop only over printed parts	<input checked="" type="checkbox"/>	Solo se realiza la retracción anterior si se mueve el cabezal sobre partes ya impresas.
Z hop after extruder switch	<input checked="" type="checkbox"/>	Retractar el eje Z cuando se cambia de extrusor.
Enable print cooling	<input checked="" type="checkbox"/>	Activa el ventilador que ayuda a solidificar el plástico recién impreso.
Regular fan speed	20 %	Velocidad a la que suele estar el ventilador.
Maximum fan speed	80 %	Velocidad máxima que puede alcanzar el ventilador.
Initial fan speed	0 %	Empieza con el ventilador apagado.
Regular fan speed at height	0,51 mm	Altura a la cual se activa el ventilador a la velocidad normal.
Regular fan speed at layer	6	Iniciar, si no se ha alcanzado todavía, los ventiladores a velocidad regular en esa capa de impresión.
Generate support	<input checked="" type="checkbox"/>	Genera una capa de soporte utilizando el extrusor especificado.
Support pattern	Triangles o Tri-Hexagon	Patrón que utiliza el extrusor para poner material de soporte.
Support density	50 %	La cantidad de soporte que es utilizado para sujetar una pieza.
Enable support interface	<input checked="" type="checkbox"/>	Se crea una serie de capas con <i>infill</i> de casi el 100 % para crear una superficie plana antes de imprimir el nuevo material sobre ella.
Enable support roof	<input checked="" type="checkbox"/>	Crea la capa mencionada anteriormente sobre el material de soporte, antes de imprimir sobre él.
Enable support floor	<input checked="" type="checkbox"/>	Crea la capa mencionada anteriormente sobre la cama caliente o la pieza impresa previamente, antes de extruir material de soporte.
Enable prime tower	<input checked="" type="checkbox"/>	Crea una torre en una esquina de la cama caliente que se utiliza principalmente para limpiar los cabezales después de no haber sido utilizados durante el cambio de material.

Capítulo 9

Calidad y pruebas

En este apartado se procede a explicar qué pruebas se han hecho en cada área del desarrollo del proyecto donde destacan:

- Diseño de las piezas usando SW de modelado 3D.
- Impresión de las piezas que conforman el brazo usando una impresora 3D.
- Pruebas post-impresión de las piezas.

Debido a la escasa experiencia del equipo de desarrollo con el diseño y la impresión 3D, este ha sido el área donde mas pruebas se han realizado.

9.1. Explicación de las pruebas

9.1.1. Pruebas en el diseño de las piezas 3D

Desde los momentos iniciales del proceso de impresión se observa que ciertas partes de las piezas impresas no concuerdan en tamaño con lo especificado en el diseño 3D. Este problema se hace más evidente en piezas que requieren de alta precisión como pueden ser los agujeros para los tornillos o los dientes de los engranajes.

Dado que existe esta diferencia entre el tamaño real y el tamaño del diseño, durante el desarrollo se realizan varias pruebas para determinar qué tamaño es necesario definir en el diseño para que, al imprimir, se obtenga la medida deseada.

Estas pruebas se han tenido que hacer para determinar el diámetro necesario para los tornillos de métricas M3 y M4 junto con el de los ejes y los rodamientos.

Las pruebas han consistido principalmente en imprimir piezas con varios agujeros de distinto diámetro con el objetivo de averiguar cuál longitud es la adecuada.

En la figura 9.4 se pueden ver distintas pruebas realizadas a tales efectos:

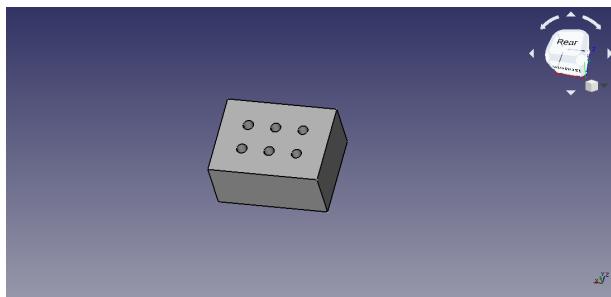


Figura 9.1: Pieza de prueba para tornillos M4.

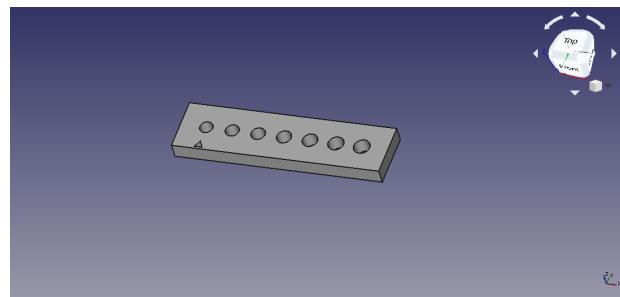


Figura 9.2: Pieza de prueba para tornillos M3.

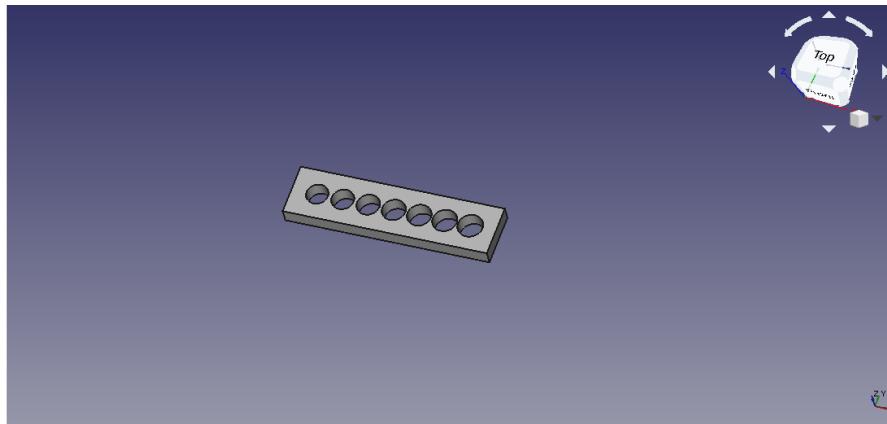


Figura 9.3: Pieza de prueba para ejes de 4 mm de diámetro.

Figura 9.4: Distintas pruebas realizadas para obtener el tamaño buscado para métricas M3 y M4 así como los ejes de 4 mm de diámetro.

Tras realizar estas pruebas se tomaron las siguientes decisiones técnicas:

- Para tornillos de métrica M3 (véase figura 9.2) el agujero del diseño 3D deberá tener 2,8 mm de diámetro. Con esta longitud se consigue suficiente material adicional en las paredes del agujero para que, tras utilizar la herramienta de hacer machos, quede una rosca suficientemente profunda.

Para diámetros menores, el macho no podía ser introducido en el agujero y por tanto no se podía hacer rosca mientras que, para diámetros mayores, el macho entraba con demasiada holgura o bien no había suficiente material adicional para crear una rosca resistente.

Cabe recordar que los tornillos empleados son metálicos y que el agujero es sobre material plástico, por lo que una rosca resistente es necesaria debido al desgaste que el tornillo ejerce sobre la montura.

- Para tornillos de métrica M4 (véase figura 9.1) el agujero del diseño 3D deberá tener 3,8 mm de diámetro. Las razones son las mismas que las expuestas anteriormente para el tornillo de métrica M3.

- En el caso de los ejes se siguió una metodología similar. Para un eje de 4 mm de diámetro se debía dejar una agujero de entre 3,9 mm y 4,1 mm de diámetro dependiendo de la holgura que se desee.

Por otro lado, se han tenido que hacer pruebas para determinar el módulo adecuado para el engranaje exterior que transmitiría el movimiento desde el eje motor hasta las piezas que se ensamblan con él.

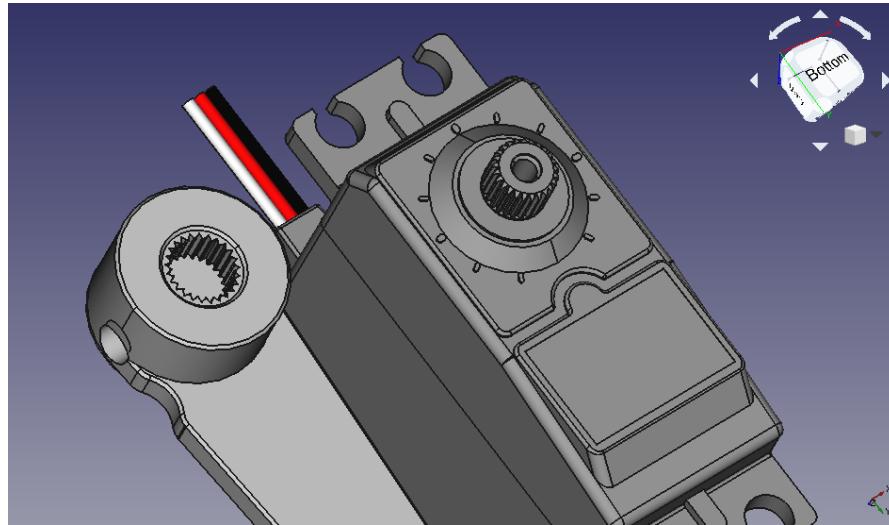


Figura 9.5: Modelos 3D del motor empleado junto a una pieza que se ensambla en su eje.

Como se observa en la figura 9.5, las piezas que van engranadas directamente en el eje han de tener un número determinado de dientes y un diámetro concreto.

La cantidad de dientes viene dada por el engranaje del motor (figura 9.6), el cual no puede ser modificado y es una restricción ajena al proceso de diseño. Sin embargo, se tuvieron que hacer pruebas para determinar el diámetro adecuado.



Figura 9.6: Cantidad de dientes presentes en los motores empleados.

Tras varias pruebas se concluyó que el módulo necesario ($mod = \mathcal{D}/\#dientes$) depende de si las capas que contienen los dientes son capas superiores o inferiores a la hora de imprimirlas. En capas inferiores el módulo ha de ser de 0,257 mm mientras que en capas superiores el módulo ha de ser 0,25 mm.

Para un módulo mayor los dientes de ambos engranajes no encajaban bien y por tanto el giro no se transmitía del eje a la pieza. Para un módulo menor el eje no entraba dentro de la pieza.

Esta diferencia en el valor del módulo se debía a que las primeras capas de la impresión 3D se realizan con distinta precisión con respecto a las capas más elevadas de la pieza, por lo que la medida ha de ser adaptada según la orientación en la que se quiera imprimir.

9.1.2. Pruebas en la impresión 3D

Pese a que no se realizaron demasiadas pruebas propiamente dichas, destacan los siguientes experimentos:

- Se imprimió una “seta” para probar el soporte con PVA, probando con distintas configuraciones y densidades. Esta prueba resultó satisfactoria y permitió avanzar en la configuración a usar para imprimir con PVA.
- Se imprimieron, tal como se ha comentado en la sección anterior, una pieza con distintos módulos para los engranajes. Con esta prueba se descubrió que las primeras capas de impresión constan de menos precisión por lo que, según se imprima en las primeras o últimas capas, habrá que utilizar un módulo distinto.

9.1.3. Pruebas post-impresión

Aprovechando ciertas piezas cuya impresión fue insatisfactoria, el equipo de desarrollo realizó pruebas destructivas para determinar el aguante ante esfuerzos de tracción, compresión y torsión.

Debido a que la estructura inferior del brazo debe soportar el peso de este, se estudió el aguante ante esfuerzos de compresión de las piezas macizas donde se concluyó que pueden aguantar pesos superiores a 50 kg sin deformarse.

Por otro lado, el esfuerzo de tracción se ha estudiado al completar la estructura del brazo, y se ha concluido que las piezas son capaces de soportar, al menos, el propio peso de la estructura.

Cabe destacar que en las piezas que son atravesadas por tornillos se han observado problemas en las roscas de poca profundidad. Al ser el acero un material mucho más duro que el plástico, este puede romper los surcos de las roscas si son muy superficiales, ya que el esfuerzo se realiza sobre un número más reducido de surcos que en el caso de una rosca más profunda. Concretamente, este problema aparece en las piezas que tienen tornillos para hacer presión sobre un eje.

Para solucionar este problema se ha decidido limar el eje en las zonas donde este entre en contacto con los tornillos, buscando conseguir una estructura cuadrada (en lugar de circular).

De esta manera, la fuerza que se ha de ejercer para mantener el eje fijo es menor y por tanto no se fuerzan las roscas.

Capítulo 10

Demostración

Timelapse de la impresión 3D de la pieza más grande

https://youtu.be/1YTt2AyJ_g8



Timelapse del proceso de fabricación de la placa

<https://youtu.be/WEvQMlzXDXw>



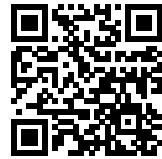
Movimiento simultáneo de los cuatro motores que pueden componer el pArm

<https://youtu.be/Y4dw6AtHFDk>



*p*Arm interpretando órdenes enviadas desde otro sistema

<https://youtu.be/MP4Z0DCgzCA>



Capítulo 11

Planificación, costes y tiempo empleado

En este capítulo se hará un análisis de la evolución de la planificación temporal y de costes a lo largo de la evolución del proyecto.

Destacar que, en lo referente a los costes, el proyecto ha respetado el margen superior inicialmente propuesto, manteniéndose incluso por debajo de este.

11.1. Diagramas de Gantt

En este apartado se presentan los diagramas de Gantt que inicialmente se plantearon para el proyecto.

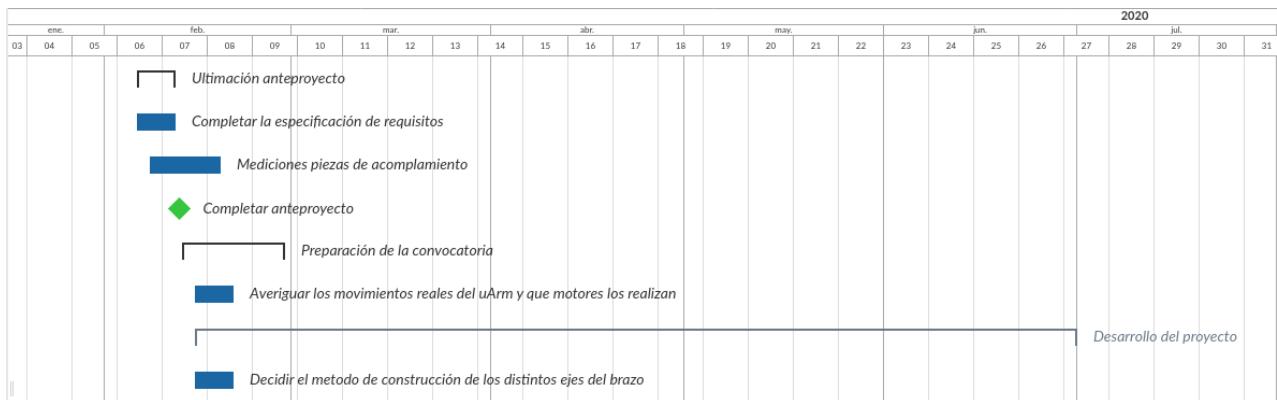


Figura 11.1: Diagrama de Gantt general.

En el diagrama 11.1 se observa la planificación inicial del proyecto el cual tenía como fecha prevista de finalización el 30 de junio.

Se pueden observar que, a nivel general, la planificación constaba de tres partes:

- El anteproyecto.

- La preparación para la convocatoria de soporte para el desarrollo de proyectos HW.
- El desarrollo efectivo del proyecto.

El anteproyecto se desglosaba en las siguientes partes (figura 11.2):

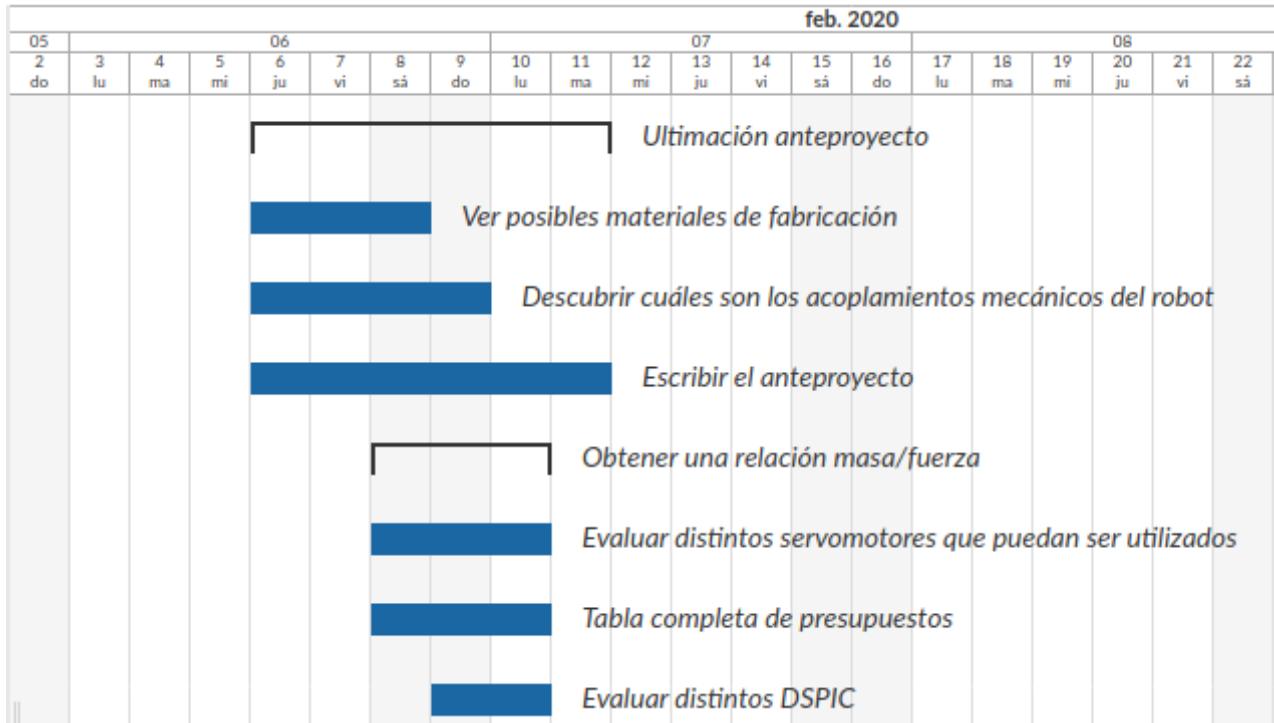


Figura 11.2: Diagrama de Gantt del anteproyecto.

Para realizar el anteproyecto el equipo de desarrollo tuvo que investigar posibles materiales con los que podría ser construido el brazo, además de los acoplamientos mecánicos entre las distintas piezas. Esto ultimo era de especial importancia ya que escapaba al ámbito de conocimiento de los integrantes del equipo.

Por otro lado, se tenía que obtener una relación entre la masa que levantaría el brazo y la fuerza que han de ejercer los motores. Esto era de interés ya que se necesitaba elegir unos motores que fuesen capaces de, al menos, mover la estructura del brazo sin carga adicional. Por otro lado, y debido a que el proyecto fue propuesto para una ayuda económica para trabajos de fin de grado, era necesario una tabla detallada de presupuestos, y conocer el modelo de los motores a comprar y del microcontrolador a utilizar sería obligatorio a la hora de hacer dicha tabla.

mientras que en la figura 11.3 se muestran las distintas labores que se realizaron para cumplimentar el documento requerido:

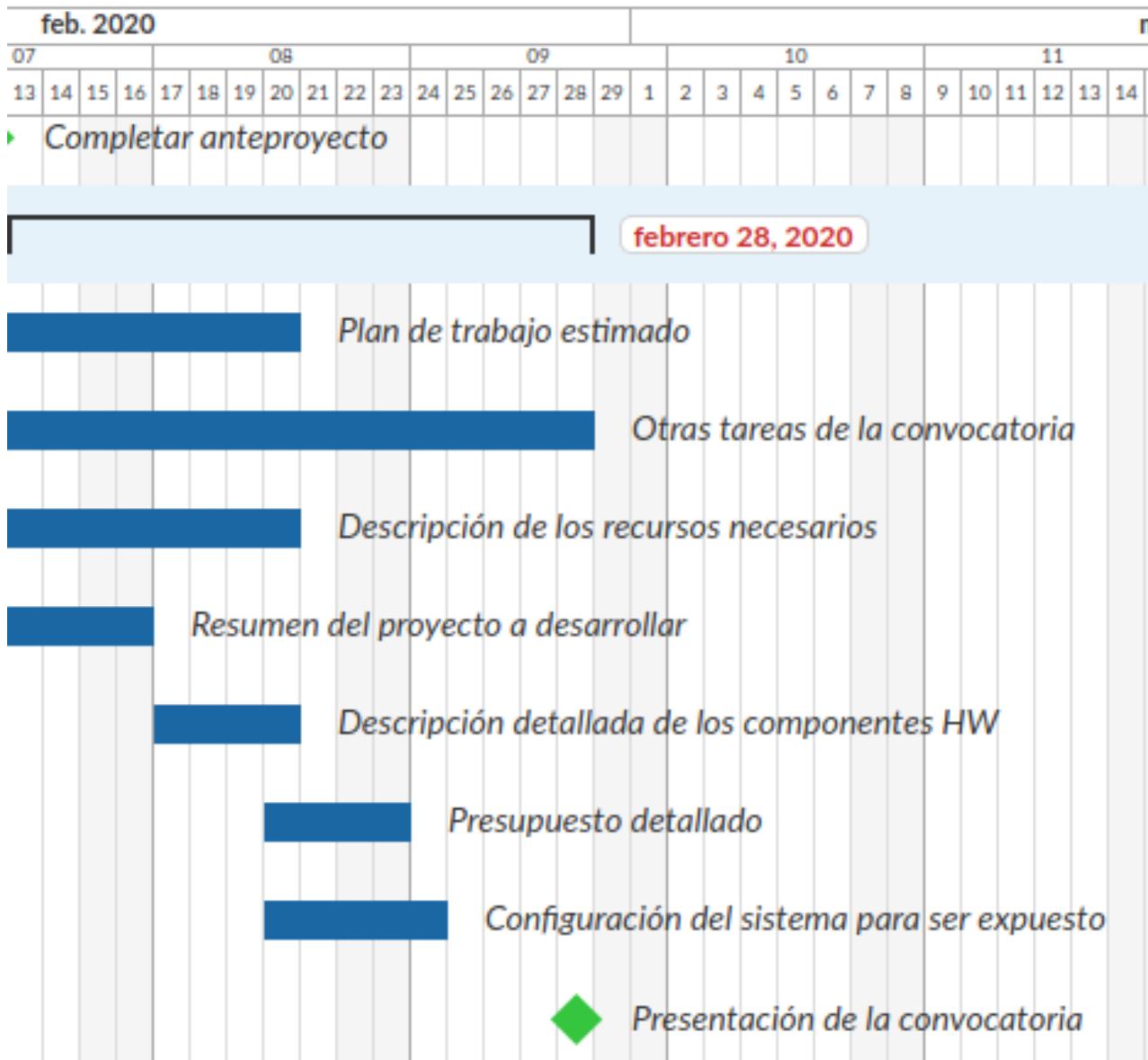


Figura 11.3: Diagrama de Gantt de la preparación de la convocatoria

Para poder completar el documento que se requería presentar para beneficiar de la ayuda económica, el equipo de desarrollo describió un plan de trabajo estimado y empleando la tabla de presupuestos del anteproyecto hizo constar la necesidad de cada uno de los elementos que aparecían en esta.

Por otro lado, hizo una breve descripción del proyecto y dejó clara la relevancia de la parte hardware dentro de este. Finalmente, detalló la configuración que debería tener el sistema para poder hacer una demostración en caso de ser este expuesto.

Por su parte, la planificación técnica del proyecto se desglosa en el diagrama 11.4:

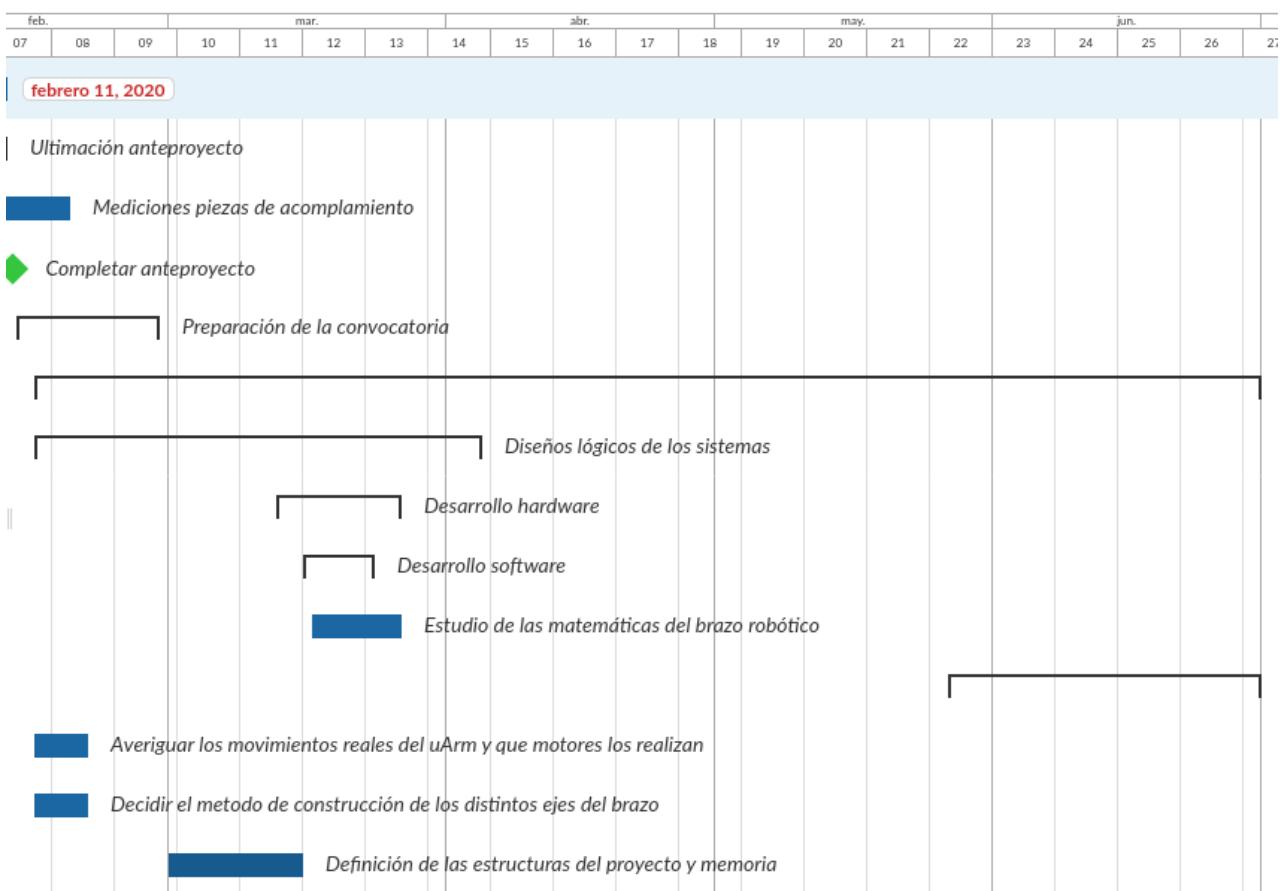


Figura 11.4: Diagrama de Gantt del desarrollo del proyecto

En la planificación que el equipo había propuesto para el desarrollo del proyecto, se observa una división del software y del hardware en períodos de desarrollo individuales, y concurrentes entre si y con respecto al estudio del fundamento matemático del proyecto.

Por otro lado era necesario realizar un estudio de la manera en la que el movimiento conjunto de dos o más motores afectaba al movimiento del *end-effector*. En relación al estudio de los acoplamientos mecánicos del brazo, el equipo de desarrollo debía decidir el método de construcción de los distintos ejes del brazo.

Finalmente se definiría la estructura del proyecto y la memoria ya que era intención del equipo de desarrollo realizar la documentación del proyecto en paralelo al desarrollo técnico de este.

La planificación temporal del proyecto se desglosa en múltiples apartados que no son representados en las imágenes anteriores, pero se deja un enlace por si se quieren consultar así como el detalle de cada una de las tareas propuestas:

<https://s.javinator9889.com/pArm-gantt>



Además, el diagrama al completo se encuentra también en el anexo F.

11.2. Sueldos propuestos y costes obtenidos

Las principales labores desempeñadas por los integrantes del equipo de desarrollo están relacionadas con el ámbito de la ingeniería de computadores, estas labores incluyen, pero no están limitadas a:

- Capacidad para sintetizar y proponer una solución a un problema basada en un sistema HW.
- Desarrollo de diagramas que abstraigan la construcción física de un sistema empotrado.
- Desarrollo de diagramas que abstraigan el SW de un sistema empotrado.
- Intervención directa en las labores de fabricación del sistema empotrado
- Comprobación del sistema empotrado mediante pruebas SW y HW.
- Documentación del proceso de diseño y fabricación.
- Desarrollo software de bajo nivel.
- Documentación del SW

Debido a lo anteriormente mencionado, el sueldo propuesto para cada uno de los integrantes es el sueldo medio de un ingeniero de computadores en España, es decir: 30.798 €/año.¹

Debido a que los integrantes del equipo de desarrollo han realizado labores ajena al ámbito de conocimiento de la ingeniería de computadores como son el diseño y la impresión 3D, el desarrollo de UI/UX, y el desarrollo de un SW de alto nivel el equipo ha considerado oportuno el incremento de la supuesta remuneración en 10.000 € al año, dejando así el salario final en 40.798 €/año.

¹https://www.glassdoor.es/Sueldos/spain-computer-engineer-sueldo-SRCH_IL.0,5_IN219_K06,23.htm?countryRedirect=true

11.3. Coste de los materiales inicial - coste de los materiales final

Elemento	Descripción	Cantidad	Precio	Código RS
Servomotor Parallax Inc.	Los servomotores permitirán controlar los distintos movimientos que puede realizar el robot. Será necesario disponer de varios para poder controlar los tres grados de libertad de los que dispondrá el <i>pArm</i> : uno para la base, otro para el primer segmento y un último para el segundo segmento.	4	16,01 €	781-3058
Rodamiento de bolas NMB, Radial.	La unión entre ejes del brazo robótico se realiza mediante rodamientos, permitiendo así un movimiento fluido y duradero del brazo.	16	5,035 €	612-6013
Pasador cilíndrico de 20 mm. de largo y 4 mm. de diámetro	El pasador que permitirá unir dos ejes separados, pasando por medio de los rodamientos anteriores.	1 bolsa	7,05 €	270-439
Microinterruptor de hasta 125V@5A	Fin de carrera para controlar la posición del brazo robótico.	4	1,709 €	682-0866
Resistencia de 240Ω	Resistencia necesaria para la construcción de la placa de control del brazo robótico.	10	0,116 €	148-354
Resistencia de 5,1KΩ	Resistencia necesaria para la construcción de la placa de control del brazo robótico.	100	0,033 €	199-7887
Total:			162,949 €	

Cuadro 11.1: Tabla completa de presupuestos.

La tabla 11.1 representa la estimación inicial de los materiales necesarios y el precio necesario para adquirirlos. Llegado el final del proyecto se confirma que no se han tenido que hacer gastos adicionales.

Cabe destacar que el gasto de las bobinas de material plástico necesarias para la impresión 3D ha sido soportado por la universidad y por lo tanto no aparece en la tabla 11.1.

11.4. Evolución del tiempo empleado, contratiempos y tiempo de desarrollo final

Debido a la crisis sanitaria mundial del COVID-19, al periodo de confinamiento nacional y a muchos otros contratiempos derivados de esta situación, el proyecto no ha podido respetar la planificación temporal inicial.

El día 15 de marzo se declaró el comienzo del confinamiento y como se observa en el diagrama 11.4 ya se había dado comienzo a la fase de desarrollo. A partir de ese día, se dejó de tener constancia documentada del tiempo empleado en las distintas fases del proyecto.

Se observa que el diseño lógico del sistema, el desarrollo HW y el desarrollo SW fueron interrumpidos. El equipo de desarrollo pudo volver a la universidad el día 1 de julio con un permiso especial para continuar con el proyecto. Entre el 15 de marzo y el 1 de julio transcurrieron 108 días.

En este periodo se completaron tareas que no requiriesen el uso de las instalaciones o los materiales que la universidad ponía a su disposición, a saber:

- Los diagramas de clases que abstraían S1.
- Los diagramas de bloques y de estados que abstraían S2.
- Los diagramas lógicos y físicos de la placa de control de S2.
- La creación del proyecto L^AT_EX y la estructura general de la memoria.
- La redacción de ciertos apartados de la memoria.
- La evaluación de distintos *frameworks* que facilitaran el desarrollo de la interfaz gráfica.

La intención del equipo de desarrollo en aquel momento era intentar terminar la máxima cantidad de trabajo que no requiriese ni de las instalaciones ni del material de la universidad, con el objetivo de poder centrarse en las labores de impresión del brazo y construcción de la PCB una vez fuese posible volver a ella.

El día 1 de julio se pudo volver la universidad en un horario de 9:00 a 14:00, donde se realizaron las siguientes labores:

CUADRO 11.2 Planificación de los meses de julio y agosto.

- 01/07 ↑ Estos días fueron empleados para organizar trabajo y planificar cursos de acción. Además, se empezó a preparar el laboratorio I1 para trabajar en él y se buscó información sobre la impresora para realizar una primera configuración. Se realizaron las primeras impresiones de prueba (ver imagen 11.5).

- 06/07 • Se refinaron los diseños lógicos y físicos de la PCB y, tras dos fracasos, se consiguieron obtener las pistas que unen los componentes de manera satisfactoria.
- 13/07 • Se hicieron distintas comprobaciones sobre la placa, se solucionaron distintos problemas referentes a las pistas y se soldaron los componentes.
- 20/07 • En esta semana se hizo la primera puesta en marcha de la placa de control y se comprobó el correcto funcionamiento de ciertos módulos mientras que otros presentaban fallos en su funcionamiento. Con la placa completamente construida, se observó que esta no podía caber dentro de la caja del diseño original de *pArm* por lo que el equipo de desarrollo se empezó a formar en diseño 3D dado que sería necesario crear partes nuevas para el brazo.
- 27/07 • Se siguió experimentando con el diseño 3D. Se escribieron apartados relacionados con el proceso de fabricación de la placa en la memoria del proyecto y se empezó a desarrollar código para S2. En relación a lo ocurrido la semana anterior, se intentaron corregir los errores en la PCB de los módulos que fallaban.
- 03/08 • Se tuvo que arreglar la impresora 3D ya que una impresión fallida provocó la obstrucción de uno de los extrusores (ver imagen 11.6). Además, varios integrantes del equipo de desarrollo junto con el tutor del proyecto se desplazaron hasta una tienda especializada para comprar materiales de impresión, ya que los que habían sido pedidos previamente no llegaron a lo largo de las dos semanas anteriores, provocando que múltiples piezas no pudieran ser impresas.
- 10/08 • Durante estos días la universidad estuvo cerrada hasta el 23/08 y se continuó escribiendo apartados de la memoria.
- 24/08 • En esta semana, debido a la compra del material de impresión necesario, se empieza la impresión diaria de piezas para el brazo. Los esfuerzos también se empiezan a centrar en el desarrollo de los códigos de S1 y S2.

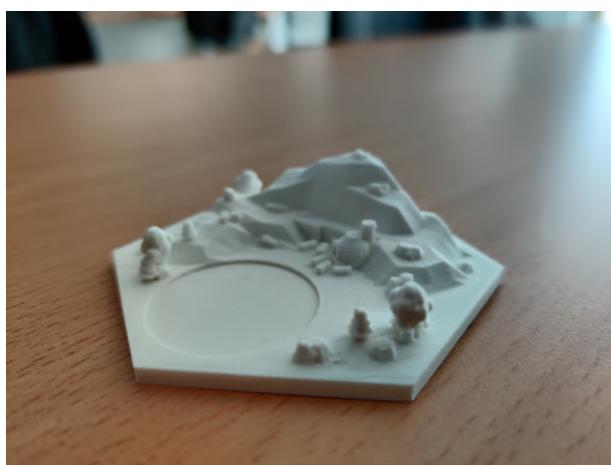


Figura 11.5: Figura de prueba impresa en 3D.

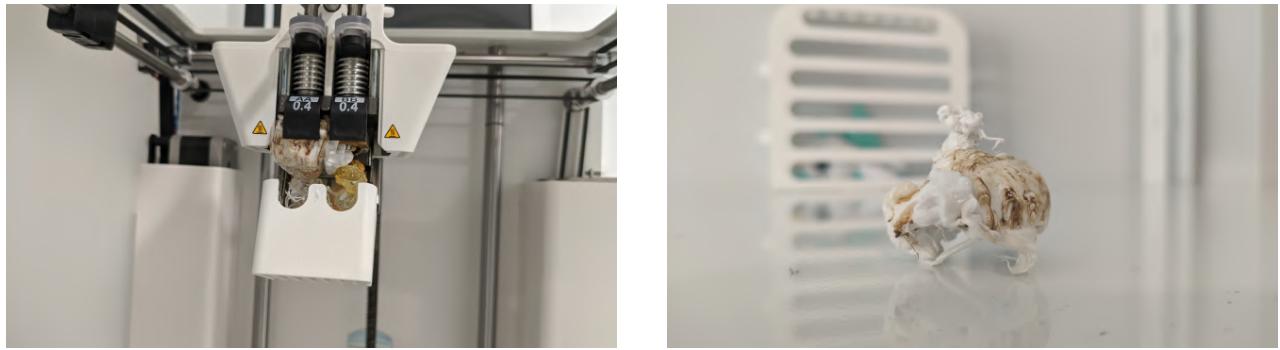


Figura 11.6: Extrusores bloqueados y parcialmente dañados por una bola de plástico.

A partir de esta semana y hasta el momento de la entrega de este documento, el equipo de desarrollo se ha centrado en trabajar de manera paralela en el código de los dos sistemas, imprimir y refinar las distintas piezas del brazo y escribir apartados de la memoria.

Cabe destacar que el hecho de que se haya empleado tanto tiempo en estas labores durante la etapa final del proyecto es principalmente por la multitud de problemas que han ido apareciendo a medida que este ha ido avanzando, entre ellas se pueden nombrar:

- Problemas con el material hidrosoluble. Este absorbe humedad del ambiente y se deteriora con el tiempo.
- Editar todas las piezas para adaptarlas a las métricas de los tornillos. El equipo de desarrollo llegó a la conclusión de que todas las piezas del brazo debían de ser editadas para adaptar sus medidas a los tornillo, ejes y rodamientos ya existentes.
- Varios problemas con la placa de control: desde el primer momento se produjeron múltiples problemas con la placa, pero empeoraron con el paso del tiempo. Los módulos de la placa de control dejaron de funcionar hasta el punto en el que la placa mostraba un funcionamiento anómalo e impredecible.
- El SW de S1 tuvo que ser adaptado a los problemas que iban apareciendo en relación con la interconexión de la interfaz de usuario y con la lógica, esta a su vez con S2.

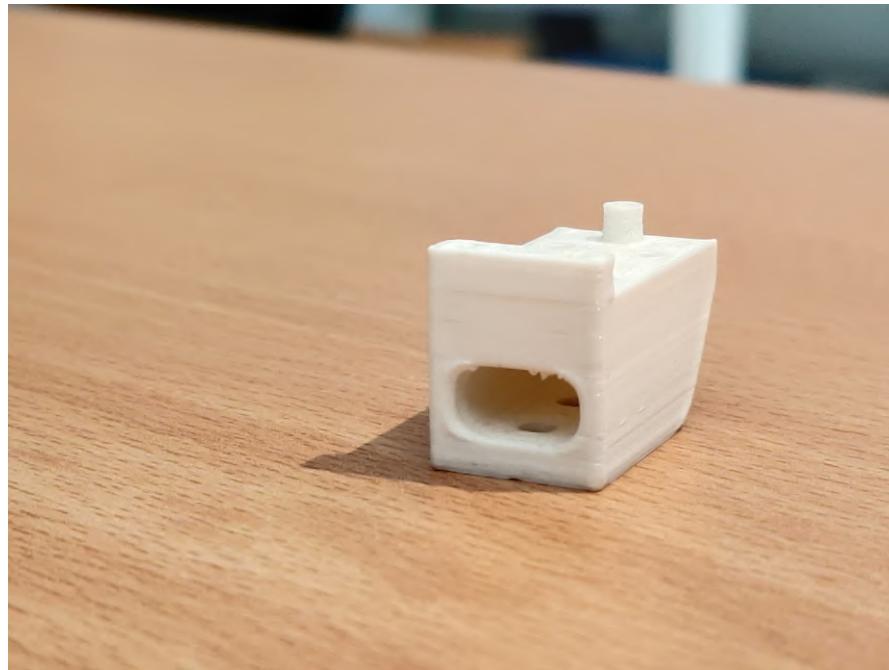
Sabiendo que la fecha de inicio oficial del proyecto es el 6 de febrero de 2020 y la fecha de entrega de la memoria del proyecto es el 16 de octubre de 2020, se calcula que el periodo de desarrollo de este proyecto es de 250 días.

11.4.1. Contratiempos de la impresión 3D

Semana del 1 de julio

En el momento en que se recibió la impresora 3D, se hicieron varias pruebas de impresión (figura 11.5) y se comenzaron a estudiar las características de la impresora. Se vio entonces

el sistema de doble extrusor que utilizaba para generar el material de impresión junto con el material de soporte soluble (PVA).



Pieza de prueba componente actual del *pArm*.

Sin embargo, cuando se realizaron impresiones utilizando dicho material de soporte, se vio que la calidad del mismo o bien no era la mejor o no bien se correspondía con cómo debería quedar, según la web de Ultimaker (figura 11.9):



Figura 11.7: Una figura de prueba que necesita PVA.



Figura 11.8: Cómo debería quedar una impresión con PVA según Ultimaker [60].

Figura 11.9: Comparación de la pieza de prueba con PVA frente a cómo deberían quedar según la web de Ultimaker.

Semana del 6 de julio

Después de hacer múltiples pruebas con el material de soporte se observó que salían granos de color negro en lugar de un hilo de plástico, por lo que se dedujo que el material de soporte no estaba en buen estado o que el extrusor estaba dañado. Ante esa situación, se desmontó el *nozzle* y se comprobó que en efecto estaba completamente lleno de PVA seco y podrido (figura 11.10):

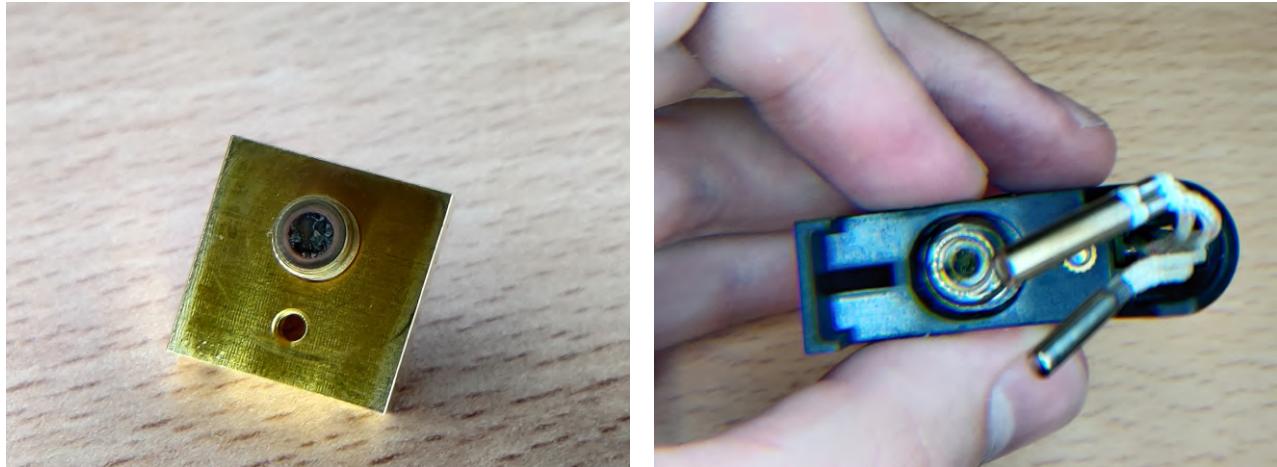


Figura 11.10: *Nozzle* completamente obstruido con PVA.

Como estaba lleno de PVA, se optó por dejar el *nozzle* en remojo para intentar que el material de soporte se disuelva. Tras varios días, una gran parte pudo ser quitada pero todavía quedaba bastante en su interior, que tuvo que ser retirada manualmente.

Igualmente, tras poder utilizar nuevamente el *nozzle* para generar material de soporte, este quedó nuevamente bloqueado cuando se imprimió con él múltiples veces. Se dedujo entonces que el material PVA estaba dañado posiblemente por el usuario anterior. Como se comentó en el apartado de impresión 3D, es un material que absorbe rápidamente la humedad y que si, además, se encuentra expuesto a la luz se agrava el daño recibido. Cuando se recibió la impresora, el material estaba completamente expuesto al ambiente además de no estar protegido siquiera contra la luz.

Semana del 13 de julio

Tras realizar una investigación sobre este problema, se descubrió que era algo común a los usuarios que utilizaban PVA como material de soporte y que se solucionaba guardando los plásticos de impresión en una caja aislada del exterior junto con un purificador de aire que absorbe humedad².

En la figura 11.11 se puede ver el resultado final:

²Se puede encontrar la caja empleada en Thingiverse: <https://www.thingiverse.com/thing:2756012>



Figura 11.11: Caja para guardar los plásticos de impresión y mantenerlos protegidos de la humedad.

Semana del 20 de julio

Tras intentar solucionar el problema anterior, la impresión con PVA seguía sin salir como se buscaba, por lo que fue necesario comprar un nuevo hilo de dicho material.

Se empezó además a trabajar con material de impresión CPE, pero ocurrió un nuevo imprevisto al imprimir cuando el cabezal de impresión se chocó contra una de las piezas que estaban siendo impresas, provocando que se generase una bola de plástico alrededor de los extrusores, dejándolos bastante dañados (figura 11.12).

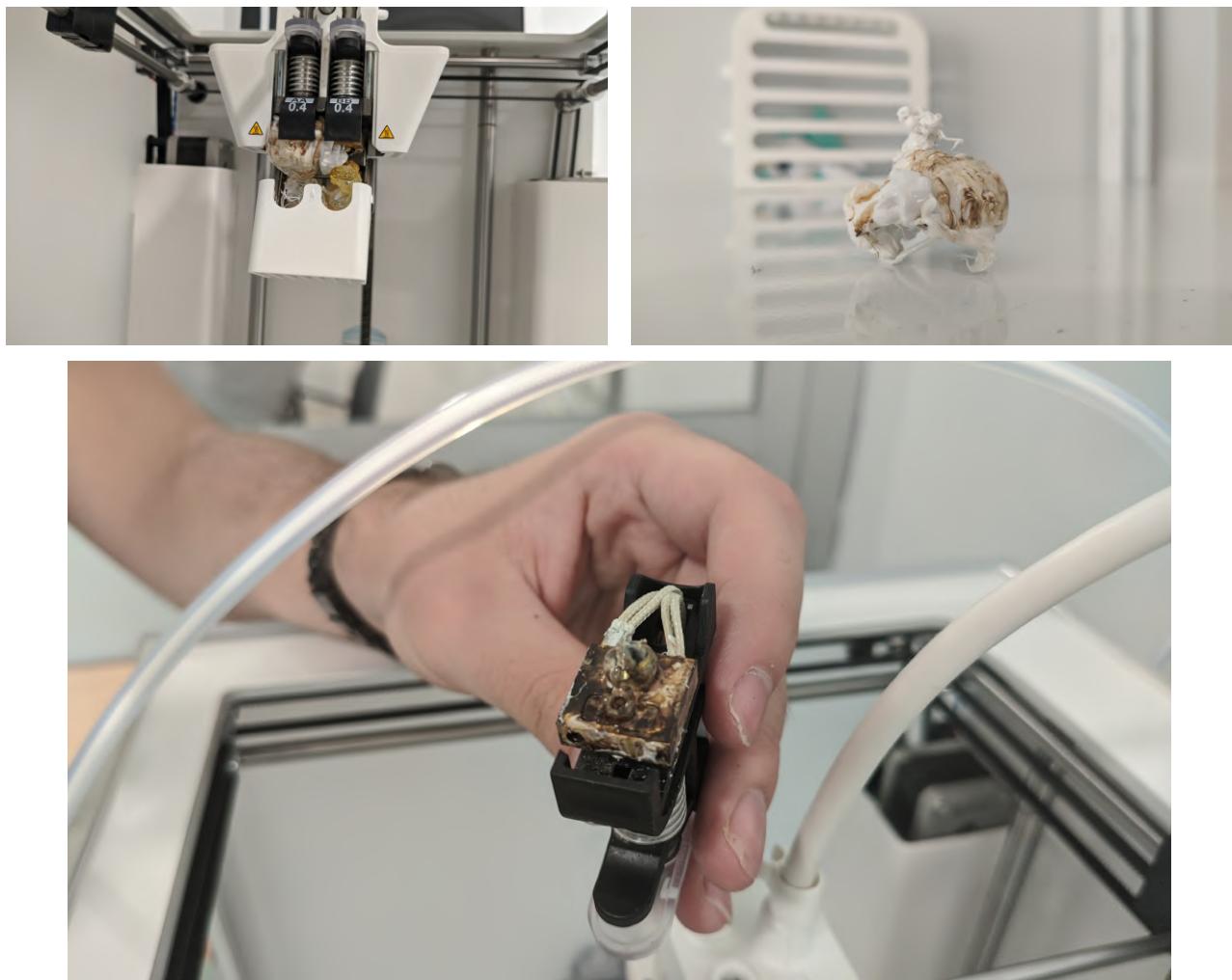


Figura 11.12: Los extrusores bloqueados y dañados tras una colisión con una pieza.

Semana del 3 de agosto

Se consiguieron arreglar los extrusores después de la obstrucción anterior, se pudieron empezar a imprimir diversas piezas que requerían de soporte con PVA de forma exitosa, obteniendo los resultados deseados (ver figura 11.13):

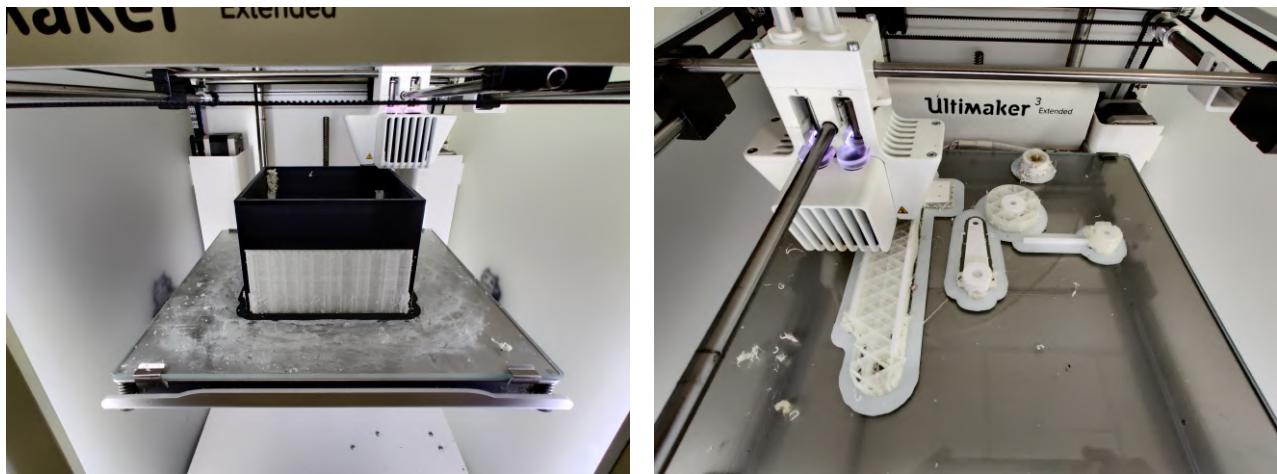


Figura 11.13: Figuras siendo correctamente impresas tras reparar los extrusores y usando material nuevo.

Semana del 14 de septiembre

Finalmente, pese a que se consiguió reparar parcialmente los cabezales de impresión, tras el uso continuado, el extrusor del material de soporte de PVA se acabó rompiendo por lo que las impresiones tuvieron que detenerse. Por suerte, se contactó con unos alumnos de la Escuela Técnica Superior de Telecomunicaciones y se pudo hacer un uso provisional de sus impresoras 3D (ya que cuentan de un laboratorio de fabricación) para continuar el desarrollo, usando además una Ultimaker 3, mismo modelo con el que se contaba en la universidad (figura 11.14):

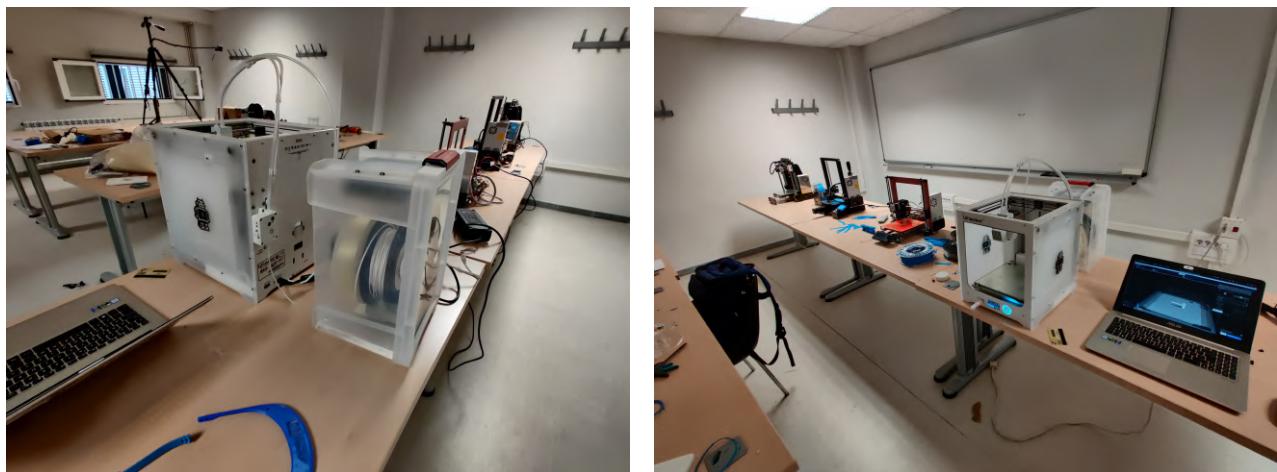


Figura 11.14: Laboratorio de fabricación de la escuela de telecomunicaciones prestado temporalmente para continuar con el proyecto.

Una vez se recibió el extrusor nuevo, se pudo continuar con las impresiones sin más inconvenientes. Añadir que, después de cada impresión utilizando PVA, era necesario realizar un mantenimiento de los cabezales para evitar su obstrucción.

Capítulo 12

Conclusiones

En este apartado procedemos a exponer los conocimientos adquiridos por el equipo de desarrollo así como las experiencias personales que cada uno de los integrantes consideran lo suficientemente relevantes como para plasmarlas en este documento.

12.1. Conclusiones técnicas

En base al trabajo realizado los integrantes del equipo de desarrollo han tomado las siguientes conclusiones técnicas

- En lo referente al *framework* de desarrollo de interfaces gráficas Qt, se han implementado todas las funcionalidades que han sido requeridas, además se han creado funcionalidades propias y se han modificado funcionalidades ya existentes. Por tanto, se recomienda su uso para futuros proyectos.
- En cuanto al diseño 3D el CAD FreeCAD se presenta como una alternativa libre, gratuita y multiplataforma frente a SW de pago. Por otro lado, destacar que su curva de aprendizaje es acusada y que en ocasiones, ocurren errores inesperados los cuales provocan, en el peor de los casos, perder trabajo no guardado y en otros casos que el trabajo ya existente se corrompa.
- En lo referente a la impresión 3D, la Ultimaker 3 Extended es una opción profesional y de alta calidad la cual ofrece un flujo de trabajo intuitivo, permitiendo al usuario hacer configuraciones rápidas, así como editar los parámetros de impresión casi en su totalidad. Sin embargo, en piezas de reducido tamaño y con características y formas complicadas, la impresión 3D no es la alternativa más precisa. Se recomienda, en caso de hacer un proyecto similar, trabajar con piezas de dimensiones superiores (en comparación con las utilizadas en este proyecto).
- Python es una alternativa libre y multiplataforma que nos ha permitido desarrollar la interfaz gráfica y la lógica del S1 con relativa sencillez y sin plantear demasiadas complicaciones. Además, la comunicación a través de la UART se realiza de forma sencilla

gracias a librerías ya existente que facilitan esta labor. Se recomienda su uso para proyectos similares.

- MPLAB X IDE simplifica la programación de los microcontroladores de la familia Microchip ahorrando mucho tiempo de configuración, añadiendo *plugins* para realizar operaciones automáticamente (cálculo de valores, inicialización de puerto, etc.). Sin embargo, no es un entorno amigable con el desarrollador ya que ciertas funcionalidades como auto-completado mientras se escribe, sugerencia de tipos, detección de errores mientras se escribe (falta de ;, tipos de datos erróneos, funciones inexistentes, ...), no se encuentran implementadas. Además, el compilador está limitado en la versión gratuita, no habilitando ciertas opciones que son de pago. Al no haber otras alternativas viables, no se puede usar otro entorno de desarrollo para crear código para microcontroladores Microchip.
- Dada la envergadura del proyecto a nivel HW, en las etapas previas al proceso de fabricación se recomienda realizar una extensa verificación de los diagramas lógicos y físicos para evitar contratiempos en etapas posteriores, ya que estos pueden suponer soluciones sobre la marcha y, en el peor de los casos, modificar el diseño y reconstruir la placa.
- A pesar de la experiencia previa de parte del equipo de desarrollo con el manipulador μ Arm, el fundamento matemático del proyecto tuvo que ser replanteado ya que se vio que el modelo cinemático no se ajustaba al comportamiento de la estructura física del brazo. Tras dicho estudio se obtuvo un modelo fiel del comportamiento del p Arm.

12.2. Conocimientos adquiridos y nuevas competencias

Para realizar este proyecto el equipo de desarrollo ha tenido que trabajar en áreas de conocimiento que no son propias de la ingeniería de computadores.

Las competencias adquiridas y perfeccionadas se pueden resumir en, pero no se limitan a:

- Conocimientos en diseño 3D.
- Proceso de fabricación mediante impresión 3D.
- Labores de mantenimiento de impresoras 3D.
- Proceso de mecanizado de piezas.
- Diseño de interfaces de usuario.
- Perfeccionamiento en el desarrollo de código de bajo y alto nivel.
- Desarrollo de habilidades de trabajo en equipo.
- Perseverancia ante las adversidades.
- Desarrollo de la resiliencia de los integrantes del equipo.
- Toma de decisiones

- Perfeccionamiento del diseño electrónico, lógico y físico de placas de circuito impreso.
- Perfeccionamiento de los procesos de fabricación y montaje de placas de circuito impreso.
- Mejora de la capacidad de sintetización de modelos matemáticos.

12.3. Reflexión final

Desde el inicio del proyecto, los integrantes del equipo han mostrado un gran interés y ganas de desarrollar este proyecto. La idea inicial se planteó como un robot impreso en 3D que pudiera resultar accesible para cualquiera, y surgió a raíz de lo estudiado y visto en la asignatura de Robótica del grado de Ingeniería de Computadores.

El proyecto se planteó como un desarrollo integral de ingeniería, lo cual implica, entre otros, los siguientes aspectos:

- Desarrollo de una especificación de requisitos que describieran el proyecto.
- Desarrollo de distintos diagramas que modelasen tanto el apartado SW como HW.
- Planificación temporal y de costes del proyecto.
- Gestión de recursos humanos (trabajo en equipo, gestión de tareas, etc.).
- Diseño y construcción de componentes HW, así como desarrollo del SW que las controla.
- Verificación del HW construido y desarrollo de distintas pruebas tanto para el HW como el SW.
- Documentación de los pasos seguidos así como de los resultados obtenidos y de los imprevistos sucedidos.
- Construcción y diseño 3D de las distintas piezas que componen el brazo robótico.

Cuando se comenzaron las primeras fases del desarrollo, la estimación de tiempo se consideraba realista por parte de los integrantes del equipo, pero a medida que avanzaba el tiempo, se vio que no cumplía con los plazos reales obtenidos.

Por una parte, la crisis mundial del COVID–19 fue decisiva a la hora de tener que postergar distintas fases críticas del proyecto:

- Construcción del HW.
- Impresión de las piezas que componen el *pArm*.
- Ensamblado del brazo robótico.
- Pruebas del SW desarrollado en S2.
- Integración de S1 con S2.

Durante los meses de confinamiento sin embargo se prosiguió con la modelización de los sistemas mediante distintos diagramas con la intención de poder trabajar directamente sobre los componentes restantes cuando se volviera a la Universidad.

Dado que el diseño del *pArm* se basa originalmente en el desarrollado por UFACTORY para el μ Arm, se asumió que el proceso de impresión 3D sería rápido y no ocasionaría problemas, nada más lejos de la realidad. Esto se debe principalmente a que los integrantes del equipo no contaban con experiencia previa en este campo y a que se contaba con que el diseño original sería igualmente válido para una impresora 3D (teniendo en cuenta que este se había concebido para ser fabricado en aluminio). Por ende, fue necesario aprender a trabajar con una impresora de este estilo junto con cómo modelar y diseñar en 3D, para poder adaptar las piezas a los nuevos diseños y componentes.

Por otra parte, la placa desarrollada para albergar al microcontrolador y a los componentes necesarios requirió de gran parte de los esfuerzos del equipo de desarrollo para intentar obtener una solución que no fuese excesivamente compleja (debido a que la fabricación de las mismas es artesanal y se cuentan con ciertas limitaciones) y que aún así permitiese hacer todo lo que se propuso. Dada la magnitud del proyecto, la PCB obtenida finalmente ha tenido que ser revisada en múltiples ocasiones hasta que se ha dado con una versión completamente válida.

En lo referente al desarrollo de los sistemas SW, ya se contaba con experiencia previa a la hora tanto de desarrollar aplicaciones de alto nivel basadas en Python como de aplicaciones de bajo nivel para manejar microcontroladores. Aún así, los distintos objetivos planteados para el proyecto implicaron seguir aprendiendo en lo referente a técnicas de programación como a nuevas soluciones para implementar dichos objetivos, como el *framework* de Qt para desarrollo de interfaces gráficas, una implementación adaptada al microcontrolador del algoritmo RSA, etc.

Finalmente, el fundamento matemático que define el comportamiento del *pArm* no pudo ser validado hasta que múltiples componentes (como el brazo físico en sí o la interfaz de usuario) fueron completados. Esto es debido a que, por ejemplo, gracias a la interfaz gráfica se pudo comprobar cómo reaccionaba el brazo ante unos valores de entrada y, con el brazo construido en sí, se pudo posteriormente verificar con el diseño físico.

Teniendo en cuenta lo anterior y lo mencionado a lo largo de este documento, cuando se pudo volver a la Universidad y comenzar el trabajo tanto físico como lógico (con las limitaciones de tiempo anteriormente mencionadas), se descubrió que las estimaciones temporales, sobre todo en lo que respecta al diseño e impresión 3D, resultaban bastante optimistas: se tuvieron que fabricar distintas placas de control ya que algunas salieron defectuosas o con errores, se tuvo que investigar sobre los distintos parámetros de impresión 3D para buscar que las piezas salieran con un resultado óptimo, se tuvo que trabajar en arreglar y preparar la impresora para poder imprimir a un ritmo casi continuo, se tuvieron que postergar los desarrollos de los sistemas SW para centrar los esfuerzos en las tareas “críticas” de ese momento (como la verificación, fabricación y construcción de la placa HW y adaptación de las piezas del brazo a los nuevos componentes, principalmente), etc.

De esta manera, el desarrollo del proyecto se puede dividir en:

- Durante los meses de febrero a junio se centraron los esfuerzos en la especificación de requisitos, modelado de los sistemas tanto HW como SW mediante distintos diagra-

mas, diseño lógico de la PCB así como el diseño físico de la misma, teniendo en cuenta las características electrónicas de los componentes que la conforman. Se realizaron varias revisiones de los requisitos así como múltiples diagramas que pretendían definir las capacidades que habrían de tener los sistemas individualmente. Por otra parte, los diagramas físicos también fueron validados en numerosas ocasiones para reducir la cantidad de errores posibles en producción. También se realizó una primera aproximación al modelo matemático.

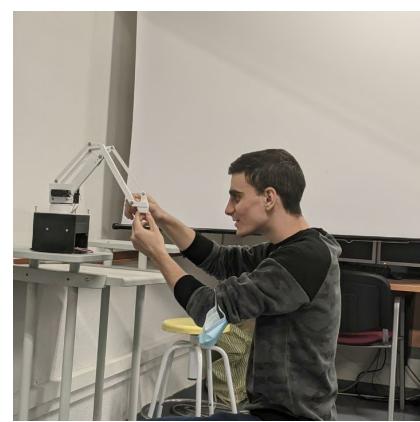
- Durante los meses de julio a octubre se centraron los esfuerzos en la fabricación de las placas de control del *pArm*, la impresión de todas las piezas que componen al brazo, reparaciones en la impresora para poder continuar con la fabricación de piezas, el montaje final del brazo robótico, el diseño y adaptación de las piezas originales a las necesidades del *pArm*, el desarrollo del SW que controla los distintos componentes que permiten el movimiento del manipulador junto con la comunicación con el exterior, el desarrollo del SW que se ejecuta en S1 ofreciendo una interfaz gráfica para manejar el *pArm*, así como ir desarrollando y escribiendo la memoria del proyecto. Con diversos componentes finalizados, se pudo mejorar el modelo matemático hasta que este describió fielmente el comportamiento del brazo.

Tras todo lo anterior, se saca en claro lo siguiente:

- Las estimaciones temporales fueron bastante optimistas, ya que se pretendía presentar en julio pero se ha tenido que postergar hasta el mes de octubre.
- Junto con lo anterior, los diversos problemas que han aparecido se consideran gratamente superados, ya que el equipo siguió adelante y se acabó por culminar el proyecto.
- La impresión 3D es todavía un campo en crecimiento y, para según qué se necesite, puede no contar con precisión suficiente. Para este proyecto, si bien ha supuesto una gran ayuda, ha sido necesario prestarle una atención elevada, teniendo que dedicarle una gran parte del tiempo hasta que se han obtenido resultados adecuados.
- El desarrollo de los distintos sistemas por separado permite un avance rápido pero limita el alcance de las pruebas que se pueden hacer, ya que hasta el momento en que no se tienen los dos funcionales individualmente no se pueden probar en conjunto.
- El proceso de fabricación de PCB requiere de especial atención y tiempo, ya que pueden aparecer fallos cuando ya se tiene ensamblada, que implican volver a fabricarla.

Como conclusión, se considera abiertamente que el proyecto ha sido superado y que se ha avanzado bastante en este campo de desarrollo, permitiendo que otros usuarios y/o estudiantes continúen el mismo y añadan nuevas características. Tras haber realizado las modificaciones pertinentes en el diseño 3D así como desarrollado los modelos físicos de la placa de control y el SW que maneja los sistemas, se podría replicar el proyecto como se buscaba inicialmente, con las siguientes limitaciones:

- Se puede encargar la fabricación de la placa de control a PCBway, llegando lista para ser utilizada¹.
- Es necesario adquirir una sonda de grabación de Microchip PICKit3 para poder cargar el SW de S2.
- El proceso de fabricación del brazo se recomienda que se haga o bien a mayor escala o bien que sea encargado a alguna empresa de fabricación, para evitar tener que lidiar con ciertos problemas propios de trabajar con la tecnología de impresión 3D.



¹<https://s.javinator9889.com/pArm-PCB>



Capítulo 13

Futuras mejoras

El objetivo de este apartado es el de mostrar algunas de las mejoras que podrían realizarse en futuras versiones o implementaciones del proyecto y que además, el equipo de desarrollo de este TFG, considera como ideas o mejoras factibles que aumentarían el valor del proyecto en términos generales.

Este conjunto de ideas y mejoras, ha ido creciendo a lo largo del proyecto y ha sido fruto del crecimiento en términos de conocimiento técnico de los miembros del equipo de desarrollo. Las mejoras que se plantean en este apartado afectan a distintos apartados del proyecto y es por ello que contienen propuestas multidisciplinares.

A continuación se presentan las principales ideas de mejora, es decir, aquellas que se consideran factibles y que son, en términos objetivos, realizables en un futuro si se dispusiese de la financiación, herramientas, materiales y tiempo necesario:

- En primer lugar, una de las mejoras que se considera más relevante y que cambiaría por completo el resultado final del proyecto, sería el emplear materiales de construcción más resistentes, por ejemplo, el aluminio.

Esta mejora plantearía la construcción de la estructura física del brazo robótico en aluminio, dotándolo de una mayor integridad estructural, resistencia y rigidez. El hecho de que la estructura estuviese construida en aluminio en lugar de en material plástico aumentaría sin ninguna duda las prestaciones del brazo robótico, haciéndolo mucho más resistente al desgaste de las partes móviles producido por el funcionamiento habitual, así como reduciendo el riesgo de rotura de la estructura al levantar objetos de peso considerable.

Actualmente y puesto que el brazo está construido en material plástico, se ha observado que el brazo muestra poca tenacidad frente al levantamiento de objetos, así como problemas de sobreesfuerzo de la estructura física en ciertas posiciones del rango de movilidad del brazo robótico, los cuales, ponen en peligro la integridad estructural del mismo.

En relación a esta mejora y dispuestos a realizar la construcción del brazo robótico en aluminio, se presenta también la posibilidad de incrementar el tamaño del mismo, así como de emplear nuevos motores eléctricos con mayor potencia, para dotar al brazo de una mayor capacidad de transporte de objetos.

Esta mejora acarrearía la modificación de algunos de los elementos del brazo robótico, como por ejemplo, la PCB que orquesta el movimiento de los motores, la tornillería, juntas, ejes y demás, con el fin de adaptarlos al nuevo tamaño y material de la estructura del brazo robótico. A pesar de ello y puesto que el diseño de la estructura ya se ha realizado, esta mejora consistiría en realizar adaptaciones, con lo cual, se considera viable.

- En segundo lugar y relacionada con la mejora anterior, se plantea el estudio del modelo dinámico del brazo robótico. Esta mejora se considera viable y permitiría una definición más precisa de los movimientos del manipulador así como la posibilidad de usarlo para levantar cargas más pesadas que las actuales.

El modelo dinámico del brazo robótico establece las relaciones matemáticas entre el movimiento del manipulador y las fuerzas que afectan al mismo. En este modelo, se contemplan aspectos tales como la relación existente entre las coordenadas angulares de las articulaciones y torques aplicados en ellas, con respecto a las coordenadas cartesianas, velocidad y aceleración del *end-effector*, así como con algunos parámetros de la estructura del brazo robótico, como la masa de los diferentes segmentos, inercias y demás [61].

Este modelo adquiere suma importancia cuando la estructura del brazo robótico se define por unas dimensiones y una masa no despreciables, puesto que las fuerzas descritas por el modelo dinámico comienzan a tener un efecto innegable en el comportamiento del mismo al realizar movimientos.

Se considera que este modelo aportaría información muy útil a la hora de realizar el control de los movimientos del manipulador, sobre todo, si este fuera replicado en un tamaño mayor y con materiales más resistentes, los cuales probablemente afecten a la masa del mismo.

- En tercer lugar, otra de las mejoras que se considera viable y que no ha podido ser implementada durante el desarrollo del proyecto, es la implementación de un modo de descripción de trayectorias.

Esta mejora plantearía la posibilidad de que el sistema ofreciese al usuario un modo de descripción de trayectorias, en el cual el usuario podría generar una trayectoria formada por diversos puntos espaciales o por una función matemática, para que posteriormente el brazo realizase de forma automática dicha trayectoria.

Esta mejora se considera una de las más viables y para su implementación, se tendrían que modificar principalmente los elementos SW del sistema, es decir, el código de S1 y S2, así como optimizar y depurar el protocolo de comunicación para soportar el aumento del tráfico de mensajes entre S1 y S2.

- En cuarto lugar, se plantea la posibilidad de poder controlar el movimiento del brazo robótico en tiempo real utilizando un controlador como el ratón o un *joystick*. Esta funcionalidad se planteó desde un inicio en el proyecto, sin embargo, debido a su complejidad, se ha clasificado como futura mejora.

La implementación de la misma conllevaría la realización de modificaciones en el código de S1 y S2 y el protocolo de comunicación, así como alguna posible modificación HW para el uso del nuevo dispositivo de control, en el caso por ejemplo de tratarse de un *joystick* o similar.

- En quinto lugar, se plantea una mejora en relación a las comunicaciones entre S1 y S2, la cual aumentaría la comodidad a la hora de poner en funcionamiento el brazo robótico. Esta mejora consiste en incluir un método de conexión inalámbrica entre S1 y S2, la cual se podría llevar a cabo mediante la tecnología *WiFi* o *Bluetooth*, por ejemplo.

La implementación de esta mejora conllevaría la realización de modificaciones en el código de S1 y S2 y el protocolo de comunicación, así como modificaciones HW en S2 para dotar a la PCB de un chip que la permitiese establecer comunicaciones inalámbricas.

- En sexto lugar, se plantea una de las mejoras que podría generar más impacto en cuanto a las funcionalidades del prototipo final. Esta mejora consiste en el diseño y construcción de diferentes adaptadores para el *end-effector* del brazo robótico, los cuales podrían dotarlo de capacidades muy variadas.

Existen innumerables tipos de *end-effector* así como herramientas que pueden ser usadas e incrustadas en ellos. Sin embargo, se considera que los siguientes son los factibles a incluir en este proyecto:

- Pinza que fuese capaz de agarrar y soltar objetos.
- Ventosa con la capacidad de succionar objetos para sujetarlos.
- Adaptador para bolígrafo, lápiz o similar.
- Cabezal de impresora 3D.

Además, a cada uno del *end-effector* anteriores se les podría incluir una cámara que permitiese al brazo robótico realizar procesado de imagen y visión por computador.

Esta mejora plantea retos interesantes y de mediana dificultad, los cuales podrían añadir numerosas capacidades nuevas al brazo robótico.

- En séptimo lugar, se plantea una mejora técnica que afectaría a S2 y que consiste en incluir un μ Kernel en el microcontrolador dsPIC de la PCB.

Mediante la implementación de dicho μ Kernel, se conseguiría una gestión más eficiente de los recursos y componentes de S2, así como la posibilidad de ejecución concurrente de varios procesos en este sistema.

Esta mejora presenta gran complejidad y requeriría una gran inversión de tiempo.

- En octavo y último lugar, se plantea la mejora de la interfaz gráfica de usuario, para hacerla aún más amigable y estéticamente atractiva.

La interfaz gráfica de usuario actual cumple su función y permite al usuario operar el brazo robótico. Sin embargo, por simplicidad se decidió que esta no fuese *responsive* y que por lo tanto, su apariencia y estructura tuviesen tamaños fijos y estáticos.

Esta mejora pretende la reconversión de la interfaz gráfica de usuario para que esta presentase una estructura dinámica y adaptativa a diferentes dimensiones, dispositivos, tamaños de monitor, etc. Es evidente que, a pesar de que esta mejora no representa un gran aporte en términos funcionales, sin duda alguna añade valor a la aplicación de S1, puesto que la hace más atractiva y amigable frente al usuario.

Existen numerosas mejoras más que podrían aplicarse a este proyecto pero se considera que las descritas anteriormente son la intersección perfecta entre valor añadido al proyecto, dificultad de implementación y posibilidad de realización en función de los conocimientos técnicos de los integrantes del equipo.

13.1. Impactos sociales y medioambientales

Si bien es cierto que el *μArm*, brazo en el que está basado el *pArm*, ya es un sistema avanzado y capaz, como se explicó en la Introducción (3.1), el sistema desarrollado busca que pueda servir para ayudar y facilitar la entrada a este tipo de tecnologías a otras personas, haciéndolo comprensible y, aprovechando la tecnología de la impresión en 3D, fabricable por uno mismo.

Además, en relación a los Objetivos de Desarrollo Sostenible (ODS), con el desarrollo de este sistema se pretende cumplir con los siguientes puntos:

4 - Educación de Calidad¹.

7 - Energía Asequible y No Contaminante².

10 - Reducción de las desigualdades³.

Para el primero, se tiene en cuenta que el producto se desarrolla siguiendo las iniciativas OS y OH, las cuales facilitan el acceso a la información a cualquiera que la requiera.

Para el segundo, el *pArm* usa la electricidad como fuente de energía, evitando así otros más contaminantes como las producidas por combustibles fósiles. En añadido, se ha trabajado para que el consumo de energía sea el menor posible mediante el estudio detallado del HW que conforma la placa, permitiendo así un mayor tiempo de uso con la misma fuente de alimentación y no abusando de los recursos de los que se disponen.

Además, se ha buscado que el *pArm* tenga un coste bajo, permitiendo así el acceso a los recursos y a los procesos de fabricación a todo el mundo que pudiera estar interesado.

El apartado de la impresión 3D, como se ha visto anteriormente, puede suponer un gran gasto a nivel tanto de recursos como de material debido a las impresiones fallidas. Por suerte, el material fabricado por Ultimaker para sus impresoras suele aprovechar materiales plásticos reciclados⁴ y también existe una técnica que permite aprovechar las piezas fallidas y generar nuevamente filamento que puede ser aprovechado para una nueva impresión. Dicha técnica se conoce como “filastruder”, y el funcionamiento es sencillo (más detalles en la imagen 13.1):

- Se cogen las piezas inservibles y se “trituran” para hacerlas pequeñas.
- Se utiliza una cámara de calor para fundir el plástico.

¹<https://www.un.org/sustainabledevelopment/es/education/>

²<https://www.un.org/sustainabledevelopment/es/energy/>

³<https://www.un.org/sustainabledevelopment/es/inequality/>

⁴<https://ultimaker.com/es/learn/100-recycled-filament-from-perpetual-plastic-project>

- Mediante unas ruedas a presión, se extruye material renovado con el diámetro adecuado y se puede volver a utilizar para imprimir.

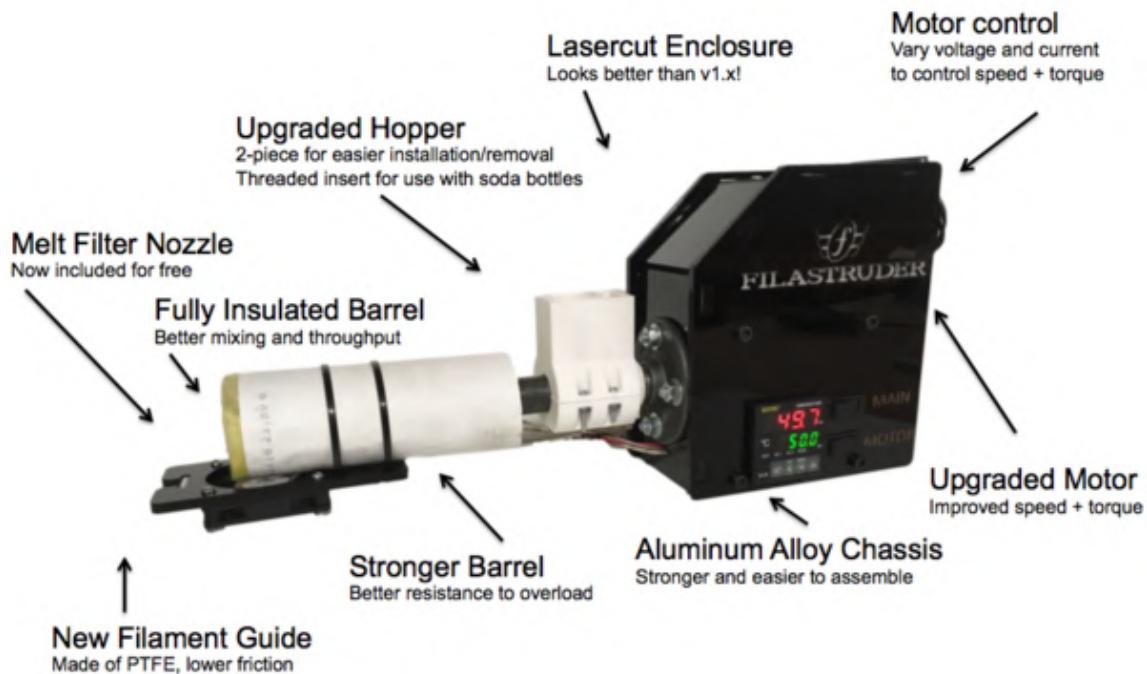


Figura 13.1: Modo de funcionamiento del “filastruder” [62].

Durante el tiempo de desarrollo, no se contaban con estas herramientas, pero sería perfectamente viable emplearlas para hacer un brazo robótico 100 % impreso con materiales reciclados.

Bibliografía

- [1] M. E. Moran, «Evolution of robotic arms», *Journal of Robotic Surgery*, vol. 1, n.º 2, págs. 103-111, jul. de 2007, ISSN: 1863-2491. DOI: 10.1007/s11701-006-0002-x.
- [2] J. de Vaucanson, *Le mécanisme du flûteur automate: présenté à Messieurs de l'Académie royale des Sciences : avec la description d'un canard artificiel et aussi celle d'une autre figure également merveilleuse, jouant du tambourin et de la flûte*. chez Jacques Guerin, imprimeur-libraire, 1738, Google-Books-ID: UNw6AAAAcAAJ.
- [3] Chapuis, Alfred and Droz, Edmond, *Automata: A Historical and Technological Study*. L'Editions du Griffon, 1958.
- [4] Standage, Tom, *The Turk: The Life and Times of the Famous Eighteenth-Century Chess-Playing Machine*. Walker company, 2002.
- [5] Belarmino, J and Moran, ME and Firoozi, F and Capello, S and Kolios, E and Perrotti, M, «Tesla's Robot and the Dawn of the Current Era. Society of Urology and Engineering», 7.^a ép., vol. 19, A214, J Endourol 2005.
- [6] ——, «An Oriental Culture of Robotics—the Coming Maelstrom. Society of Urology and Engineering», 7.^a ép., vol. 19, A119, J Endourol 2005.
- [7] *Mobile Servicing System*. feb. de 2020, Page Version ID: 942154747
Publication Title: Wikipedia. dirección: https://en.wikipedia.org/w/index.php?title=Mobile_Servicing_System&oldid=942154747 (visitado 14-06-2020).
- [8] *Spirit (Rover)*. jun. de 2020, Page Version ID: 960186651
Publication Title: Wikipedia. dirección: [https://en.wikipedia.org/w/index.php?title=Spirit_\(rover\)&oldid=960186651](https://en.wikipedia.org/w/index.php?title=Spirit_(rover)&oldid=960186651) (visitado 15-06-2020).
- [9] Bill Marshall. (12 de sep. de 2018). «Give your Robot the Mobility Control of a real Mars Rover: Part 4», dirección: <https://www.rs-online.com/designspark/give-your-robot-the-mobility-control-of-a-real-mars-rover-part-4> (visitado 22-09-2020).
- [10] *Opportunity (Rover)*. mayo de 2020, Page Version ID: 960035335
Publication Title: Wikipedia. dirección: [https://en.wikipedia.org/w/index.php?title=Opportunity_\(rover\)&oldid=960035335](https://en.wikipedia.org/w/index.php?title=Opportunity_(rover)&oldid=960035335) (visitado 15-06-2020).
- [11] *The Mars Rovers: Spirit and Opportunity* | NASA Space Place – NASA Science for Kids. dirección: <https://spaceplace.nasa.gov/mars-spirit-opportunity/en/> (visitado 15-06-2020).
- [12] *History of Robots*. jun. de 2020, Page Version ID: 961981460

- Publication Title: Wikipedia. dirección: https://en.wikipedia.org/w/index.php?title=History_of_robots&oldid=961981460 (visitado 15-06-2020).
- [13] R. Baldwin, *Tesla's Video Shows What Its Autopilot System Sees*. feb. de 2020, Library Catalog: www.caranddriver.com Section: News
Publication Title: Car and Driver. dirección: <https://www.caranddriver.com/news/a30733506/tesla-autopilot-full-self-driving-video/> (visitado 15-06-2020).
- [14] *Tesla's Full Self-Driving Computer Is Now in All New Cars and a next-Gen Chip Is Already 'Halfway Done'*, Library Catalog: techcrunch.com
Publication Title: TechCrunch. dirección: <https://social.techcrunch.com/2019/04/22/teslas-computer-is-now-in-all-new-cars-and-a-next-gen-chip-is-already-halfway-done/> (visitado 15-06-2020).
- [15] *Boston Dynamics*. mar. de 2020, Page Version ID: 123964032
Publication Title: Wikipedia, la enciclopedia libre. dirección: https://es.wikipedia.org/w/index.php?title=Boston_Dynamics&oldid=123964032 (visitado 15-06-2020).
- [16] *KR 1000 titan*, Library Catalog: www.kuka.com
Publication Title: KUKA AG. dirección: <https://www.kuka.com/es-es/productos-servicios/sistemas-de-robot/robot-industrial/kr-1000-titan> (visitado 15-06-2020).
- [17] *UFACTORY Official Website*, Library Catalog: store.ufactory.cc
Publication Title: store.ufactory.cc. dirección: <https://store.ufactory.cc/> (visitado 15-06-2020).
- [18] J. A. Silva, Roberto Álvarez Garrido y José Alejandro Moya Blanco, *UPM-Robotics/Uarm*, Robotics UPM, 19 de nov. de 2019. dirección: <https://github.com/UPM-Robotics/uarm> (visitado 28-06-2020).
- [19] UFACTORY, *UFACTORY xArm/uArm*. dirección: <https://www.ufactory.cc/#/en/uarmswift> (visitado 28-01-2020).
- [20] Arduino. (). «Arduino Mega 2560 Rev3 | Arduino Official Store», dirección: <https://store.arduino.cc/arduino-mega-2560-rev3> (visitado 28-06-2020).
- [21] S. Sharma y C. Scheurer, «Generalized Unified Closed Form Inverse Kinematics for Mobile Manipulators With Reusable Redundancy Parameters», 6 de ago. de 2017. DOI: [10.1115/DETC2017-68104](https://doi.org/10.1115/DETC2017-68104).
- [22] Roboy.org. (). «Roboy 2.0 - Inverse Kinematics», dirección: <https://ik-test.readthedocs.io/en/latest/> (visitado 28-06-2020).
- [23] Ultimaker. (). «CPE Family – Ultimaker Support», dirección: <https://support.ultimaker.com/hc/en-us/sections/360003511860> (visitado 08-10-2020).
- [24] UFACTORY, *uArm Swift Pro - Developer Guide*, oct. de 2017. dirección: <http://download.ufactory.cc/docs/en/uArm-Swift-Pro-Developer-Guide-171013.pdf>.
- [25] *Denavit–Hartenberg parameters*, en Wikipedia, 20 de jun. de 2020. dirección: https://en.wikipedia.org/w/index.php?title=Denavit%20%93Hartenberg_parameters&oldid=963594857 (visitado 10-09-2020).
- [26] *Common normal (robotics)*, en Wikipedia, 2 de jun. de 2017. dirección: [https://en.wikipedia.org/w/index.php?title=Common_normal_\(robotics\)&oldid=783528894](https://en.wikipedia.org/w/index.php?title=Common_normal_(robotics)&oldid=783528894) (visitado 10-09-2020).

- [27] Javinator9889, *pArm-TFG/pArm-Configurator*, pArm - TFG, 17 de sep. de 2020. dirección: <https://github.com/pArm-TFG/pArm-configurator> (visitado 17-09-2020).
- [28] Microchip, *dsPIC33E/PIC24E FRM Section 2. CPU*, 2010. dirección: <http://ww1.microchip.com/downloads/en/devicedoc/s2.pdf>.
- [29] UFACTORY. (). «uArm-Developer/SwiftProForArduino», dirección: <https://github.com/uArm-Developer/SwiftProForArduino> (visitado 22-09-2020).
- [30] travisdewolf. (3 de sep. de 2013). «Robot control part 2: Jacobians, velocity, and force», dirección: <https://studywolf.wordpress.com/2013/09/02/robot-control-jacobians-velocity-and-force/> (visitado 24-09-2020).
- [31] *Potencia (física)*, en *Wikipedia, la enciclopedia libre*, 21 de sep. de 2020. dirección: [https://es.wikipedia.org/w/index.php?title=Potencia_\(f%C3%ADsica\)&oldid=129438450](https://es.wikipedia.org/w/index.php?title=Potencia_(f%C3%ADsica)&oldid=129438450) (visitado 24-09-2020).
- [32] *Invertible matrix*, en *Wikipedia, la enciclopedia libre*, 11 de sep. de 2020. dirección: https://en.wikipedia.org/w/index.php?title=Invertible_matrix&oldid=977818984 (visitado 25-09-2020).
- [33] *Pseudoinversa de Moore-Penrose*, en *Wikipedia, la enciclopedia libre*, 24 de ago. de 2020. dirección: https://es.wikipedia.org/w/index.php?title=Pseudoinversa_de_Moore-Penrose&oldid=128716323 (visitado 25-09-2020).
- [34] Circuits Root. (). «The Pantograph in Context», dirección: <https://www.circuitousroot.com/artifice/letters/press/typemaking/making-matrices/pantograph-in-context/index.html#:~:text=A%20pantograph%20is%20a%20system,an%20increased%20or%20decreased%20size.&text=The%20so%2Dcalled%20%22parallel%20motion,transmission%20of%20power%20and%20motion> (visitado 23-06-2020).
- [35] ST, *Regulador LM317 Datasheet*. dirección: <https://www.st.com/resource/en/datasheet/lm217.pdf>.
- [36] ——, *Regulador L7805CV Datasheet*. dirección: <https://www.st.com/resource/en/datasheet/178.pdf>.
- [37] Diodes Inc., *Regulador AZ1117H Datasheet*. dirección: https://www.diodes.com/assets/Datasheets/products_inactive_data/AZ1117.pdf.
- [38] *dsPIC33EP512GM604 - 16-Bit - Microcontrollers and Digital Signal Controllers*. dirección: <https://www.microchip.com/wwwproducts/en/dsPIC33EP512GM604> (visitado 16-06-2020).
- [39] Universidad de Granada, *Construcción de Una PCB*. dirección: http://www.ugr.es/~amroldan/cursos/pcb_uhu_98/construccionpcb.html (visitado 04-09-2020).
- [40] Interesting Engineering, *10 of the Best Engineering Quotes Ever*. dirección: [These%2010%20great%20quotes%20enhance%20the%20value%20we%20should%20place%20on%20engineers%20and%20their%20innovations..](https://www.interestingengineering.com/10-best-engineering-quotes-ever)
- [41] Datasheet.ex. (). «775-9009F-C-CC Datasheet (Hoja de Datos) - DC Motor», dirección: <http://www.datasheet.es/PDF/917192/775-9009F-C-CC-pdf.html> (visitado 28-06-2020).
- [42] *Motor de corriente continua*, en *Wikipedia, la enciclopedia libre*, 26 de jun. de 2020. dirección: https://es.wikipedia.org/w/index.php?title=Motor_de_corriente_continua&oldid=127266375 (visitado 28-06-2020).

- [43] Banggood.com. (). «3pcs 42mm 12V Nema 17 Stepper de dos fases motor», dirección: <https://www.banggood.com/3pcs-42mm-12V-Nema-17-Two-Phase-Stepper-Motor-For-3D-Printer-p-1556469.html> (visitado 28-06-2020).
- [44] monolithicpower. (). «Stepper Motors: Types, Uses and Working Principle | Article | MPS», dirección: <https://www.monolithicpower.com/en/stepper-motors-basics-types-uses> (visitado 28-06-2020).
- [45] Ebotics. (). «Mini Servomotor - Ebotics», dirección: <https://www.ebotics.com/es/producto/mini-servomotor/> (visitado 28-06-2020).
- [46] (20 de oct. de 2019). «How Servo Motor Works & Interface It With Arduino», dirección: <https://lastminuteengineers.com/servo-motor-arduino-tutorial/> (visitado 28-06-2020).
- [47] Zona Maker - Servo-Motores. dirección: <https://www.zonamaker.com/electronica/intro-electronica/componentes/motores/servo-motores> (visitado 16-06-2020).
- [48] RS-Online. (). «900-00005 | Servomotor Parallax Inc 140 mA, 4 → 6 V | RS Components», dirección: <https://es.rs-online.com/web/p/servomotores/7813058/> (visitado 28-06-2020).
- [49] Programación estructurada, en Wikipedia, la enciclopedia libre, 12 de oct. de 2020. dirección: https://es.wikipedia.org/w/index.php?title=Programaci%C3%B3n_structurada&oldid=130000309 (visitado 12-10-2020).
- [50] M. Paland, *Mpaland/Printf*, 12 de oct. de 2020. dirección: <https://github.com/mpaland/printf> (visitado 12-10-2020).
- [51] Microchip, *dsPIC33E/PIC24E FRM Section 7. Oscillator - Microchip*, mar. de 2012. dirección: <http://ww1.microchip.com/downloads/en/devicedoc/70580c.pdf>.
- [52] ——, *dsPIC33E/PIC24E Family Reference Manual Section 14. High-Speed PWM*, dic. de 2011. dirección: <http://ww1.microchip.com/downloads/en/devicedoc/70645c.pdf>.
- [53] ——, *dsPIC33E/PIC24E Family Ref. Manual, Sect. 10 /I/O Ports*, mar. de 2013. dirección: <http://ww1.microchip.com/downloads/en/devicedoc/70000598c.pdf>.
- [54] G-code, en Wikipedia, la enciclopedia libre, 24 de abr. de 2020. dirección: <https://es.wikipedia.org/w/index.php?title=G-code&oldid=125469865> (visitado 13-10-2020).
- [55] (). «G-Code/Es - RepRap», dirección: <https://reprap.org/wiki/G-code/es> (visitado 13-10-2020).
- [56] Ácido poliláctico. mayo de 2020, Page Version ID: 126434778. dirección: https://es.wikipedia.org/w/index.php?title=%C3%81cido_polil%C3%A1ctico&oldid=126434778 (visitado 16-06-2020).
- [57] Acrilonitrilo butadieno estireno. jun. de 2020, Page Version ID: 126716603. dirección: https://es.wikipedia.org/w/index.php?title=Acrilonitrilo_butadieno_estireno&oldid=126716603 (visitado 16-06-2020).
- [58] Cura (software), en Wikipedia, la enciclopedia libre, 10 de jul. de 2019. dirección: [https://es.wikipedia.org/w/index.php?title=Cura_\(software\)&oldid=117310854](https://es.wikipedia.org/w/index.php?title=Cura_(software)&oldid=117310854) (visitado 14-10-2020).

- [59] 3D Universe. (). «Ultimaker 3», dirección: <https://shop3duniverse.com/products/ultimaker-3> (visitado 14-10-2020).
- [60] Ultimaker. (). «Material Ultimaker PVA de acetato de polivinilo para impresora 3D: soporte soluble en agua para impresiones complejas | Ultimaker», dirección: <https://ultimaker.com/es/materials/pva> (visitado 14-10-2020).
- [61] U. M. Hernández. (). «Libro «Prácticas de Robótica Utilizando Matlab» | Grupo de Investigación NBIO», dirección: <https://nbio.umh.es/es/libro-practicas-de-robotica-utilizando-matlab/> (visitado 21-10-2020).
- [62] Filastruder. (). «Filastruder Kit», dirección: <https://www.filastruder.com/products/filastruder-kit> (visitado 20-10-2020).

Anexo A

Código fuente “*pArm configurator*”

```

1 #                         manipulator
2 #
3 # This program is free software: you can redistribute it and/or modify
4 # it under the terms of the GNU General Public License as published by
5 # the Free Software Foundation, either version 3 of the License, or
6 # (at your option) any later version.
7 #
8 # This program is distributed in the hope that it will be useful,
9 # but WITHOUT ANY WARRANTY; without even the implied warranty of
10 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
11 # GNU General Public License for more details.
12 #
13 # You should have received a copy of the GNU General Public License
14 # along with this program. If not, see <http://www.gnu.org/licenses/>.
15 from typing import List
16 from typing import Union
17 from .symbols import Symbol
18
19
20 class DHTable:
21     """
22     Container class for the Denavit-Hartenberg table.
23     Creates a data structure containing the necessary data for constructing
24     the required matrix.
25     """
26
27     def __init__(self, table: List[dict] = None, check: bool = True):
28         """
29         Creates a new instance for the class. If any argument is provided, then
30         it checks if the dict is OK.
31         :param table: a dictionary containing the DH table. It must have the following
32         structure:
33         {
34             1: {
35                 'a': length,
36                 'd': distance,
37                 'alpha': angle with i + 1,
38                 'theta': arm angle (symbol)
39             }, ...

```

```
40 }
41 :param check: skip the checking of the structure of the dictionary.
42 """
43     if table is None:
44         table = list()
45         check = False
46     if check:
47         for value in table:
48             assert len(value.keys()) == 4
49     self.__table = table
50     self.symbols = list()
51 """
52 List of symbols used in DH-Table
53 """
54 self.max = 0
55 """
56 How many items does the class have
57 """
58 self.Tx: float = 0.
59 """
60 Translation in 'X' axis
61 """
62 self.Ty: float = 0.
63 """
64 Translation in 'Y' axis
65 """
66 self.Tz: float = 0.
67 """
68 Translation in 'Z' axis
69 """
70 self.__lengths = [set() for _ in range(4)]
71
72 @staticmethod
73 def _check_errors(theta: Union[Symbol, float],
74                     d: Union[Symbol, float],
75                     a: Union[Symbol, float],
76                     alpha: Union[Symbol, float]) -> bool:
77     return (type(theta) is Symbol and (type(d) is Symbol or type(a) is Symbol or
78                                         type(alpha) is Symbol)
79             or type(d) is Symbol and (
80                 type(theta) is Symbol or type(a) is Symbol or
81                 type(alpha) is Symbol)
82             or type(a) is Symbol and (type(d) is Symbol or type(theta) is
83                                         Symbol or
84                                         type(alpha) is Symbol)
85             or type(alpha) is Symbol and (
86                 type(d) is Symbol or type(a) is Symbol or
87                 type(theta) is Symbol))
88
89 def add(self,
90         theta: Union[Symbol, float],
91         d: Union[Symbol, float],
92         a: Union[Symbol, float],
93         alpha: Union[Symbol, float],
94         checkAttrs: bool = True) -> 'DHTable':
95     """
```

```
96     Add new params to the Denavit-Hartenberg table, in order. This method can safely
97     be called by using the "Builder" structure (.add(...).add(...)...).
98     By default, all params can be both "Symbol" or "float" but, if "checkAttrs" is
99     changed, only one can be a "Symbol". If not, it raises "AttributeError".
100
101    :param theta: the parameter theta.
102    :param d: the distance (elevation) between axes.
103    :param a: the length of the segment.
104    :param alpha: the angle between Zi and Zi+1 (radians).
105    :param checkAttrs: whether to perform a check or not - default: True
106    :return: the class itself.
107    :raises AttributeError when there is two or more params whose type is Symbol.
108    Disable "checkAttrs" for not throwing any exception.
109    """
110
111    if checkAttrs:
112        if self._check_errors(theta, d, a, alpha):
113            raise AttributeError("Only one param can be a Symbol")
114
115    self.__table.append({
116        'a': a,
117        'd': d,
118        "alpha": alpha,
119        "theta": theta
120    })
121    self.max += 1
122    if type(theta) is Symbol:
123        self.symbols.append(theta)
124    if type(d) is Symbol:
125        self.symbols.append(d)
126    if type(a) is Symbol:
127        self.symbols.append(a)
128    if type(alpha) is Symbol:
129        self.symbols.append(alpha)
130    self.__lengths[0].add(len(str(theta)))
131    self.__lengths[1].add(len(str(d)))
132    self.__lengths[2].add(len(str(a)))
133    self.__lengths[3].add(len(str(alpha)))
134
135    def change(self, i: int, **kwargs):
136        """
137        Changes an existing param in the Denavit-Hartenber tables. 'i' (index) must
138        exist.
139
140        :param i: the table index (from 1 to n).
141        :param kwargs: the keys to modify - [theta, d, a, alpha]
142        :raises IndexError when the 'i' does not exist.
143        """
144        i -= 1
145        for key, value in kwargs.items():
146            if key not in self.__table[i].keys():
147                raise KeyError(f"The key '{key}' is not a valid entry - it must be: "
148                               f"[theta, d, a, alpha]")
149            old_value = self.__table[i][key]
150            self.__table[i][key] = value
151            if type(value) is Symbol:
```

```

152         if type(old_value) is Symbol:
153             self.symbols.insert(self.symbols.index(old_value), value)
154
155     def remove(self, i: int) -> dict:
156         """
157             Removes an item from the Denavit-Hartenberg table.
158             :param i: the item to remove (from 1 to n).
159             :return: the removed item.
160             :raises IndexError when the item does not exists.
161         """
162         i -= 1
163         item = self.__table.pop(i)
164         for key, value in item.items():
165             if type(item[key]) is Symbol:
166                 self.symbols.remove(value)
167         return item
168
169     def get(self) -> List[dict]:
170         """
171             Obtains the table itself.
172             :return: the Denavit-Hartenberg table.
173         """
174         return self.__table
175
176     def __getitem__(self, item):
177         assert isinstance(item, int)
178         return self.__table[item]
179
180     def __iter__(self):
181         i = 0
182         for element in self.__table:
183             i += 1
184             yield i, element["theta"], element['d'], element['a'], element["alpha"]
185
186     def __str__(self):
187         row_format = "{}{}{}{}{}{}".format("{:>4}",
188                                             "{:>" + str(4 + max(self.__lengths[0])) + "}",
189                                             "{:>" + str(4 + max(self.__lengths[1])) + "}",
190                                             "{:>" + str(4 + max(self.__lengths[2])) + "}",
191                                             "{:>" + str(4 + max(self.__lengths[3])) + "}")
192         result = row_format.format('i', "t", "d", "a", "alpha") + "\n"
193         i = 1
194         for values in self.__table:
195             result += row_format.format(str(i),
196                                         str(values["theta"]),
197                                         str(values['d']),
198                                         str(values['a']),
199                                         str(values["alpha"])) + "\n"
200             i += 1
201         return result

```

Listing A.1: pArm–configurator/src/manipulator/dh_table.py

```

1 #                         manipulator
2 #
3 # This program is free software: you can redistribute it and/or modify

```

```
4 # it under the terms of the GNU General Public License as published by
5 # the Free Software Foundation, either version 3 of the License, or
6 # (at your option) any later version.
7 #
8 # This program is distributed in the hope that it will be useful,
9 # but WITHOUT ANY WARRANTY; without even the implied warranty of
10 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
11 # GNU General Public License for more details.
12 #
13 # You should have received a copy of the GNU General Public License
14 # along with this program. If not, see <http://www.gnu.org/licenses/>.
15 from typing import Union
16 from typing import Tuple
17 from typing import Dict
18 from typing import Any
19
20 from sympy import Matrix
21 from sympy import symbols
22 from sympy import simplify
23
24 from numbers import Number
25
26 from . import sin
27 from . import cos
28 from . import sqrt
29 from . import atan2
30 from . import Symbol
31 from . import DHTable
32 from . import to_latrix
33
34
35 class ForwardKinematics:
36     """
37     Container for the Forward Kinematics (FK) for an arbitrary manipulator.
38     By using the Denavit-Hartenberg table, generates the required matrices
39     in order to use them later.
40     Refer to: https://en.wikipedia.org/wiki/Denavit%20%93Hartenberg\_parameters
41     for more information.
42     The accessible params are:
43         - params: DHTable.
44         - transformation_matrices: dict with the forward transformation matrices.
45         - phi_e: expression for phi_e.
46     Matrices are accessible by using square brackets: fk["A03"].
47     """
48
49     def __init__(self, params: DHTable, optimize: bool = True):
50         """
51             Generates a new instance for the class. It calculates the forward
52             transformation matrices (symbolically) in order to use them later
53             and not calculating them every time they are needed.
54             :param params: the Denavit-Hartenberg params.
55             :param optimize: whether to optimize or not the matrices - requires more
56             computation time - default: True
57         """
58         self.params = params
59         self.transformation_matrices: Dict[str, Matrix] = {}
```

```
60     self._calc_matrices(optimize)
61     self.phi_e = None
62
63     def _calc_matrices(self, optimize: bool):
64         """
65             Internal function which iteratively calculates the required transformation
66             matrices.
67             :param optimize: whether to optimize or not the matrices.
68             """
69             for i, theta, d, a, alpha in self.params:
70                 self.transformation_matrices[f"A{i - 1}{i}"] = \
71                     self._matrix(theta, d, a, alpha)
72             for i in range(2, self.params.max + 1):
73                 self.transformation_matrices[f"A0{i}"] = \
74                     self.transformation_matrices[f"A0{i - 1}"] * \
75                     self.transformation_matrices[f"A{i - 1}{i}"]
76             if optimize:
77                 self.transformation_matrices[f"A0{i}"].simplify()
78             self.transformation_matrices[f"A0{self.params.max}"][0, 3] += self.params.Tx
79             self.transformation_matrices[f"A0{self.params.max}"][1, 3] += self.params.Ty
80             self.transformation_matrices[f"A0{self.params.max}"][2, 3] += self.params.Tz
81
82     def set_phi(self, expression: Union[Symbol, Number]):
83         """
84             Sets the phi_e expression.
85             :param expression: expression - can be a Symbol or a number.
86             """
87             self.phi_e = expression
88
89     def point(self,
90               subs: Dict[Symbol, Any],
91               matrix_index: str = None) -> Tuple[Number, Number, Number, Any]:
92         """
93             Obtain the (X, Y, Z, Phi) coordinates by changing the articulations.
94             :param subs: the articulations' values.
95             :param matrix_index: the transformation matrix in which apply the values.
96             By default, it is the forward transformation matrix.
97             :return: (X, Y, Z, Phi) as a tuple.
98             """
99             if matrix_index is None:
100                 matrix_index = f"A0{self.params.max}"
101             return self.transformation_matrices[matrix_index].subs(subs)[0, 3], \
102                   self.transformation_matrices[matrix_index].subs(subs)[1, 3], \
103                   self.transformation_matrices[matrix_index].subs(subs)[2, 3], \
104                   self.phi_e.subs(subs) if self.phi_e is not None else None
105
106     def __getitem__(self, item):
107         return self.transformation_matrices.get(item)
108
109     @staticmethod
110     def _matrix(theta: Union[Symbol, float],
111                 d: Union[Symbol, float],
112                 a: Union[Symbol, float],
113                 alpha: Union[Symbol, float]) -> Matrix:
114         """
115             Forward transformation matrix template.
```

```

116     :param theta: "theta" param.
117     :param d: 'd' param.
118     :param a: 'a' param.
119     :param alpha: "alpha" param.
120     :return: the forward transformation matrix.
121     """
122
123     return Matrix(
124         [[cos(theta), - cos(alpha) * sin(theta), sin(alpha) * sin(theta),
125           a * cos(theta)],
126          [sin(theta), cos(alpha) * cos(theta), - sin(alpha) * cos(theta),
127           a * sin(theta)],
128          [0, sin(alpha), cos(alpha), d],
129          [0, 0, 0, 1]])
130
131 class InverseKinematics:
132     """
133     Container for the Inverse Kinematics (IK) for an arbitrary manipulator.
134     By using the Forward Kinematics for that manipulator, generates and
135     calculates the Jacobian matrix that can be used for both direct
136     manipulation and inverse manipulation, relating linear speed (end-effector)
137     and angular speed (joints).
138
139     The accessible params are:
140     - params: the DHTable params.
141     - Xe: expression for X.
142     - Ye: expression for Y.
143     - Ze: expression for Z.
144     - det: the determinant of the Jacobian.
145     - upper_jacobian: the first part of the Jacobian matrix (linear velocity).
146     - lower_jacobian: the lower part of the Jacobian matrix (angular velocity).
147     - m_jacobian: Jacobian matrix.
148     - i_jacobian: inverse Jacobian.
149     - pinv_jacobian: pseudo-inverse Jacobian.
150
151     For accessing the inverse matrix, it is better to use the "inverse" property,
152     as it will return the pseudo-inverse or the inverse, in case the latest one
153     does not exists.
154     """
155
156     def __init__(self, forward_kinematics: ForwardKinematics, phi_e: dict = None):
157         """
158             Generates a new instance for the inverse kinematics class.
159             :param forward_kinematics: the forward kinematics for the manipulator.
160             :param phi_e: the Phi_e dict which relates the 'x', 'y' and 'z' expressions.
161         """
162         self._end_effector_matrix = forward_kinematics[
163             f"A0{forward_kinematics.params.max}"]
164         self._phi_e = phi_e if phi_e is not None else dict()
165         self.params = forward_kinematics.params
166         self.Xe = self._end_effector_matrix[0, 3]
167         self.Ye = self._end_effector_matrix[1, 3]
168         self.Ze = self._end_effector_matrix[2, 3]
169         self.det = None
170         self.upper_jacobian = None
171         self.lower_jacobian = None

```

```
172     self.m_jacobian = None
173     self.i_jacobian = None
174     self.pinv_jacobian = None
175
176 def set_phi(self, xyz: str, expression: Union[Symbol, Number]):
177     """
178     Sets the Phi_e expression, which relates the angle to an axis.
179     :param xyz: the axis in which update the expression - must be: {'x', 'y', 'z'}.
180     :param expression: the expression for the Phi - can be a number or an expression.
181     :raises AttributeError when xyz not in 'x', 'y', 'z'.
182     """
183     if xyz.lower() not in ['x', 'y', 'z']:
184         raise AttributeError("xyz attribute must be ['x', 'y', 'z']")
185     self._phi_e[xyz.lower()] = expression
186
187 def jacobian(self, subs: list = None) -> Matrix:
188     """
189     Calculates the Jacobian matrix. If the determinant is '0', then it
190     calculates the pseudo-inverse.
191     :param subs: list of symbols that will be used for calculating the
192     difference for the Jacobian.
193     :return: the Jacobian matrix.
194     """
195     smatrix = Matrix([self.Xe,
196                      self.Ye,
197                      self.Ze,
198                      self._phi_e['x'],
199                      self._phi_e['y'],
200                      self._phi_e['z']])
201     if subs is None:
202         subs = self.params.symbols
203     self.m_jacobian = smatrix.jacobian(subs)
204     self.upper_jacobian = self.m_jacobian[:3, :]
205     self.lower_jacobian = self.m_jacobian[3:, :]
206     self.det = self.upper_jacobian.det().simplify()
207     if self.det != 0:
208         self.i_jacobian = simplify(self.upper_jacobian ** -1)
209     else:
210         self.pinv_jacobian = self.upper_jacobian.pinv()
211     return self.m_jacobian
212
213 @property
214 def inverse(self):
215     """
216     :return: the inverse Jacobian.
217     """
218     return self.pinv_jacobian if self.i_jacobian is None else self.i_jacobian
219
220
221 class UArmInverseKinematics:
222     """
223     uArm Inverse Kinematics class wrapper for the uArm robotic arm.
224     Accessible values are:
225     - X_e: X value.
226     - Y_e: Y value.
227     - Z_e: Z value.
```

```

228     - phi: phi value.
229     - theta_1: expression for theta_1.
230     - theta_2: expression for theta_2.
231     - theta_3: expression for theta_3.
232 """
233
234 def __init__(self, params: DHTable):
235     """
236     Generates a new instance for the uArmInverseKinematics class.
237     :param params: the Denavit-Hartenberg params.
238     """
239     self.X_e, self.Y_e, self.Z_e, self.phi = symbols("X_e Y_e Z_e phi_e")
240     cos_t3 = (
241         (self.X_e ** 2) + (self.Z_e ** 2) -
242         (params[1]['a'] ** 2) - (params[2]['a'] ** 2)
243         /
244         2 * params[1]['a'] * params[2]['a']
245     )
246     sin_t3 = (
247         sqrt(1 - (cos_t3 ** 2))
248     )
249     self.theta_1 = atan2(self.Y_e, self.X_e + params.Tx + params[0]['d'])
250     self.theta_3 = atan2(sin_t3, cos_t3)
251     self.theta_2 = self.phi + self.theta_3
252
253 def eval(self,
254         Xe: Union[Symbol, Number],
255         Ye: Union[Symbol, Number],
256         Ze: Union[Symbol, Number],
257         phi: Union[Symbol, Number]) -> Tuple[Union[Symbol, Number],
258                                             Union[Symbol, Number],
259                                             Union[Symbol, Number]]:
260 """
261     With a given point, returns the joints at which the robotic arm achieves
262     that position.
263     :param Xe: X position.
264     :param Ye: Y position.
265     :param Ze: Z position.
266     :param phi: phi value.
267     :return: (theta_1, theta_2, theta_3) as a tuple.
268 """
269     subs = {self.X_e: Xe, self.Y_e: Ye, self.Z_e: Ze, self.phi: phi}
270     theta_1 = self.theta_1.subs(subs).evalf(chop=True)
271     theta_3 = self.theta_3.subs(subs).evalf(chop=True)
272     theta_2 = self.theta_2.subs(subs).evalf(chop=True)
273     return theta_1, theta_2, theta_3
274
275
276 class Manipulator:
277 """
278     Wrapper class for working with the manipulator.
279     Accessible params are:
280     - direct_kinematics: the direct kinematics for the DHTable.
281     - inverse_kinematics: the inverse kinematics for the DHTable.
282     - uarm_ik: the uArm inverse kinematics.
283 """

```

```
284
285     def __init__(self, params: DHTable, optimize: bool = True):
286         self.params = params
287         self.direct_kinematics = ForwardKinematics(params, optimize)
288         self.inverse_kinematics = InverseKinematics(self.direct_kinematics)
289         self.uarm_ik = UArmInverseKinematics(params)
290
291     def point(self, subs: Dict[Symbol, Any],
292              matrix_index: str = None) -> Tuple[Number, Number, Number, Any]:
293         """
294             Obtain the (X, Y, Z, Phi) coordinates by changing the articulations.
295             :param subs: the articulations' values.
296             :param matrix_index: the transformation matrix in which apply the values.
297             By default, it is the forward transformation matrix.
298             :return: (X, Y, Z, Phi) as a tuple.
299         """
300         return self.direct_kinematics.point(subs, matrix_index)
301
302     def set_phi(self, xyz: str, expression: Union[Symbol, Number]):
303         """
304             Sets the Phi_e expression, which relates the angle to an axis.
305             :param xyz: the axis in which update the expression - must be: {'x', 'y', 'z'}.
306             :param expression: the expression for the Phi - can be a number or an expression.
307             :raises AttributeError when xyz not in 'x', 'y', 'z'.
308         """
309         self.inverse_kinematics.set_phi(xyz, expression)
310
311     def jacobian(self, subs: list = None) -> Matrix:
312         """
313             Calculates the Jacobian matrix. If the determinant is '0', then it
314             calculates the pseudo-inverse.
315             :param subs: list of symbols that will be used for calculating the
316             difference for the Jacobian.
317             :return: the Jacobian matrix.
318         """
319         return self.inverse_kinematics.jacobian(subs)
320
321     @property
322     def inverse(self):
323         """
324             :return: the inverse Jacobian.
325         """
326         return self.inverse_kinematics.inverse
327
328     def eval(self,
329             Xe: Union[Symbol, Number],
330             Ye: Union[Symbol, Number],
331             Ze: Union[Symbol, Number],
332             phi: Union[Symbol, Number]) -> Tuple[Union[Symbol, Number],
333                                                 Union[Symbol, Number],
334                                                 Union[Symbol, Number]]:
335         """
336             With a given point, returns the joints at which the robotic arm achieves
337             that position.
338             :param Xe: X position.
339             :param Ye: Y position.
```

```
340     :param Ze: Z position.
341     :param phi: phi value.
342     :return: (theta_1, theta_2, theta_3) as a tuple.
343     """
344     return self.uarm_ik.eval(Xe, Ye, Ze, phi)
345
346     def to_latrix(self, matrix_type: str, matrix_index: str) -> str:
347         """
348             With a given Matrix, obtain its representation as a LaTeX matrix.
349             :param matrix_type: the type of the matrix. Possible values can be:
350             ['b', 'p', 'v', 'V', '']. View
351             https://en.wikibooks.org/wiki/LaTeX/Mathematics#Matrices_and_arrays
352             for more information.
353             :param matrix: the Matrix from which obtain the representation.
354             :return: the LaTeX string representation.
355             """
356
357     return to_latrix(matrix_type, self.direct_kinematics[matrix_index])
```

Listing A.2: pArm–configurator/src/manipulator/manipulator.py

A.1. Enlace a *Jupyter Notebook* para configurar el *pArm*

<https://s.javinator9889.com/pArm-config>



Anexo B

Enlaces útiles

Nombre	Enlace	Código QR
Proyecto en GitHub	https://github.com/pArm-TFG	
Estudio del μ Arm	https://github.com/UPM-Robotics/uarm	
Código fuente pArm-S1	https://github.com/pArm-TFG/pArm-S1	
Código fuente pArm-S2	https://github.com/pArm-TFG/pArm-S2	

Anexo C

Código fuente S2

C.1. *Header files*

```

1  /*
2   * 2020 | pArm-S2 by Javinator9889
3   *
4   * This program is free software: you can redistribute it and/or modify
5   * it under the terms of the GNU General Public License as published by
6   * the Free Software Foundation, either version 3 of the License, or
7   * (at your option) any later version.
8   *
9   * This program is distributed in the hope that it will be useful,
10  * but WITHOUT ANY WARRANTY; without even the implied warranty of
11  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12  * GNU General Public License for more details.
13  *
14  * You should have received a copy of the GNU General Public License
15  * along with this program. If not, see https://www.gnu.org/licenses/.
16  *
17  * Created by Javinator9889 on 2020 - pArm-S1.
18  */
19
20 /*
21  * File: planner.h
22  * Author: Javinator9889
23  * Comments: The planner that controls how the arm moves
24  * Revision history: v1.0
25  */
26
27 // This is a guard condition so that contents of this file are not included
28 // more than once.
29 #ifndef PLANNER_H
30 #define PLANNER_H
31
32 #include <stdint.h>
33 #include "../motor/motor.h"
34 #include "../utils/types.h"
35 #include "../sync/barrier.h"

```

```
36
37 typedef struct {
38     /**
39      * pArm base motor, which moves along Y axis.
40      */
41     motor_t *base_motor;
42
43     /**
44      * pArm lower arm motor, which moves along X axis.
45      */
46     motor_t *lower_arm;
47
48     /**
49      * pArm upper motor, which moves along Z axis.
50      */
51     motor_t *upper_arm;
52
53     /**
54      * pArm end-effector motor.
55      */
56     motor_t *end_effector_arm;
57 }
58 /**
59  * Simple struct containing pointers to every pArm motor.
60  */
61 motors_t;
62
63 extern motors_t motors;
64
65 #ifdef LIMIT_SWITCH_ENABLED
66 /**
67  * Initialize the planner so movements can be done.
68  *
69  * @param barrier - the planner synchronization barrier.
70  * @param switch_map - an array shared with {@link interrupts.h#_CNInterrupt}.
71  */
72 void PLANNER_init(barrier_t *barrier, uint_fast8_t switch_map[4]);
73 #else
74 /**
75  * Initialize the planner so movements can be done.
76  *
77  * @param barrier - the planner synchronization barrier.
78  */
79 void PLANNER_init(volatile barrier_t *barrier);
80 #endif
81
82 /**
83  * Moves every motor to its home position.
84  *
85  * @return the time, in us, that will pass until the movement is done.
86  */
87 double64_t PLANNER_go_home(void);
88
89 /**
90  * Moves to the specified position at point (x, y, z).
91  *
```

```

92 * @param xyz - the final position.
93 * @return the time, in us, that will pass until the movement is done.
94 */
95 double64_t PLANNER_move_xyz(point_t xyz);
96
97 /**
98 * Moves each motor to the specified angle (t0, t1, t2).
99 *
100 * @param angle - the motors' angles.
101 * @return the time, in us, that will pass until the movement is done.
102 */
103 double64_t PLANNER_move_angle(angle_t angle);
104
105 /**
106 * Moves the specified angle and waits until the movement is completed.
107 *
108 * @param angle - the motors' angles.
109 */
110 void PLANNER_move_waiting(angle_t angle);
111
112 /**
113 * Immediately stops any possible movement of the motors.
114 *
115 * @return EXIT_SUCCESS if there was any movement or EXIT_FAILURE if none.
116 */
117 uint8_t PLANNER_stop_moving(void);
118
119 /**
120 * Obtains the current end-effector position, in terms of (x, y, z).
121 *
122 * @return a pointer to the position.
123 */
124 point_t *PLANNER_get_position(void);
125
126 /**
127 * Obtains the current motor angles, in terms of (t0, t1, t2).
128 *
129 * @return a pointer to the angles.
130 */
131 angle_t *PLANNER_get_angles(void);
132
133 #endif /* PLANNER_H */

```

Listing C.1: pArm-S2/pArm.X/arm/planner.h

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

```

```

12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: gcode.h
22 * Author: Javinator9889
23 * Comments: GCode interpreter
24 * Revision history: v1.0
25 */
26
27 #ifndef GCODE_H
28 #define GCODE_H
29
30 #include <stdint.h>
31 #include "../utils/types.h"
32 #include "../utils/uart.h"
33
34 /**
35 * With the given order, process the command sent to the device.
36 *
37 * @param order the order to be parsed.
38 * @return GCODE_ret_t containing the parsed command.
39 */
40 GCODE_ret_t GCODE_process_command(volatile order_t *order);
41
42 #endif /* GCODE_H */

```

Listing C.2: pArm-S2/pArm.X/gcode/gcode.h

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: servo.h
22 * Author: Javinator9889

```

```
23 * Comments: Servo controller header file
24 * Revision history: 1.0
25 */
26
27 #ifndef SERVO_H
28 #define SERVO_H
29
30 #include <stdint.h>
31 #include "../utils/defs.h"
32 #include "../utils/types.h"
33
34 #define usToTicks(_us)      ( (clockCyclesPerMicrosecond() * _us) / PRESCALE )
35 #define ticksToUs(_ticks)   ( ((unsigned)_ticks * PRESCALE) / clockCyclesPerMicrosecond() )
36 #define MIN_PULSE_WIDTH    usToTicks(((unsigned long) MIN_PULSE_MS * 1000))
37 #define MAX_PULSE_WIDTH    usToTicks(((unsigned long) MAX_PULSE_MS * 1000))
38
39 // Servo definition
40 typedef struct {
41     /**
42     * A reference to the register in which the duty cycle is stored. This
43     * way, it is possible to define multiple servo_t handling different motors.
44     */
45     volatile uint16_t *dutyCycleRegister;
46
47     /**
48     * A reference to the limit switch map used to indicate whether the
49     * servomotor has ended its movement by touching a physical limit switch.
50     * For using this field, LIMIT_SWITCH_ENABLED macro should be defined.
51     */
52     volatile uint_fast8_t *limit_switch_value;
53
54     /**
55     * The servo home position.
56     */
57     double64_t home;
58
59     /**
60     * The servo physical minimum angle.
61     */
62     double64_t min_angle;
63
64     /**
65     * The servo physical maximum angle.
66     */
67     double64_t max_angle;
68 } servo_t;
69
70 /**
71 * Writes an angle directly into the servo register.
72 *
73 * @param servo - the servo to move.
74 * @param angle_rad - the angle to move.
75 */
76 void SERVO_write_angle(const servo_t *servo, double64_t angle_rad);
77
78 /**
```

```

79 * Writes a value of milliseconds as the angle to move the servo. This
80 * milliseconds equals the duty cycle period.
81 *
82 * @param servo - the servo to move.
83 * @param ms - the milliseconds representing the angle.
84 */
85 void SERVO_write_milliseconds(const servo_t *servo, double64_t ms);
86
87 /**
88 * Writes directly a value into the servo's register. Use this method
89 * with caution.
90 *
91 * @param servo - the servo to move.
92 * @param dtc_value - the duty cycle register value.
93 */
94 void SERVO_write_value(const servo_t *servo, uint16_t dtc_value);
95
96 /**
97 * Obtains the equivalent milliseconds stamp representing the given angle.
98 *
99 * @param angle_rad - the angle to transform.
100 * @return double64_t representing the milliseconds.
101 */
102 double64_t SERVO_from_angle_to_ms(double64_t angle_rad);
103
104 #endif /* SERVO_H */

```

Listing C.3: pArm-S2/pArm.X/motor/servo.h

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: motor.h
22 * Author: Javinator9889
23 * Comments: The motor handler header file definition
24 * Revision history: 1.0
25 */
26
27 #ifndef MOTOR_H

```

```
28 #define MOTOR_H
29
30 #include <stdint.h>
31 #include <stdbool.h>
32 #include "servo.h"
33 #include "../utils/types.h"
34 #include "../utils/utils.h"
35 #include "../utils/defs.h"
36
37 #define MAX_MOTORS 4U
38 #define US_PER_DEGREE 5245.275704F
39 #define MOTOR_elapsed_time_us(rad) (US_PER_DEGREE * rad * MATH_TRANS)
40
41 typedef struct {
42     /**
43      * Pointer storing the driver that manages the servomotor.
44      */
45     servo_t *servoHandler;
46
47     /**
48      * Motor unique identifier.
49      */
50     const uint8_t id;
51
52     /**
53      * The current movement duration, in us.
54      */
55     volatile double64_t movement_duration;
56
57     /**
58      * The current motor angle in us.
59      */
60     volatile double64_t angle_us;
61
62     /**
63      * Flag indicating that the motor has finished its movement.
64      */
65     volatile bool movement_finished;
66
67     /**
68      * Flag indicating whether the movement is clockwise or anticlockwise.
69      */
70     int8_t clockwise;
71
72     /**
73      * Volatile counter indicating the elapsed time since the movement started.
74      */
75     volatile time_t current_movement_count;
76
77     /**
78      * Function that initializes the timer attached to the motor.
79      */
80     TMR_func TMR_Start;
81
82     /**
83      * Function that finishes the timer attached to the motor.
```

```

84     */
85     TMR_func TMR_Stop;
86 } motor_t;
87
88 /**
89 * Moves the motor to the specified angle in radians.
90 *
91 * @param motor a pointer to the motor to move.
92 * @param angle_rad the radians to move.
93 */
94 void MOTOR_move(motor_t *motor, double64_t angle_rad);
95
96 /**
97 * Freezes the motor at the current position.
98 *
99 * @param motor a pointer to the motor to freeze.
100 */
101 void MOTOR_freeze(motor_t *motor);
102
103 /**
104 * Performs the motor calibration.
105 *
106 * @param motor a pointer to the motor to calibrate.
107 * @return EXIT_SUCCESS if calibration is OK or EXIT_FAILURE in other case.
108 */
109 char MOTOR_calibrate(motor_t *motor);
110
111 /**
112 * Gets the motor position as us.
113 *
114 * @param motor a pointer to the motor.
115 * @return the position in microseconds.
116 */
117 double64_t MOTOR_position_us(motor_t *motor);
118
119 /**
120 * Gets the motor actual position in radians.
121 *
122 * @param motor a pointer to the motor.
123 * @return the position in radians.
124 */
125 double64_t MOTOR_position_rad(motor_t *motor);
126
127 /**
128 * Gets the motor actual position in degrees.
129 *
130 * @param motor a pointer to the motor.
131 * @return the position in degrees.
132 */
133 double64_t MOTOR_position_deg(motor_t *motor);
134
135 #endif /* MOTOR_H */

```

Listing C.4: pArm-S2/pArm.X/motor/motor.h

```
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20/*
21 * File: kinematics.h
22 * Author: Javinator9889
23 * Comments: Contains the definitions for both forward and inverse kinematics
24 * Revision history: v1.0
25 */
26
27// This is a guard condition so that contents of this file are not included
28// more than once.
29#ifndef KINEMATICS_H
30#define KINEMATICS_H
31
32#include <stdbool.h>
33#include "../utils/types.h"
34
35/**
36 * Utility function used to check if the given angle and point are OK.
37 *
38 * @param angle the input angle. Can be NULL.
39 * @param point the input point. Can be NULL.
40 * @return true or false indicating if the input params are OK.
41 */
42bool check_constraints_ok(angle_t* angle, point_t* point);
43
44/**
45 * Performs the inverse kinematics for the pArm.
46 *
47 * @param in_cartesian the input point.
48 * @param angle a pointer in which the obtained angle will be stored.
49 * @return EXIT_SUCCESS if everything went OK or EXIT_FAILURE in other case.
50 */
51char inverse_kinematics(point_t in_cartesian, angle_t *angle);
52
53/**
54 * Performs the forward kinematics for the pArm.
55 *
56 * @param in_angle the input angle.
57 * @param position a pointer in which the obtained position will be stored.
```

```

58 * @return EXIT_SUCCESS if everything went OK or EXIT_FAILURE in other case.
59 */
60 char forward_kinematics(angle_t in_angle, point_t *position);
61
62
63 #endif /* KINEMATICS_H */

```

Listing C.5: pArm-S2/pArm.X/motor/kinematics.h

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: io.h
22 * Author: Javinator9889
23 * Comments: I/O wrapper for printf lib
24 * Revision history: v1.0
25 */
26
27 // This is a guard condition so that contents of this file are not included
28 // more than once.
29 #ifndef IO_H
30 #define IO_H
31
32 /**
33 * This is a simple wrapper for switching between custom printf implementation
34 * and stdio.h one, alongside with all the other tools included in both
35 * packages
36 */
37 #ifdef USE_CUSTOM_PRINTF
38 #include "printf.h"
39 #else
40 #include <stdio.h>
41 #endif
42
43 #endif /* IO_H */

```

Listing C.6: pArm-S2/pArm.X/printf/io.h

```

1 /*

```

```
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20/*
21 * File: printf_config.h
22 * Author: Javinator9889
23 * Comments: Configuration file for the printf.h lib - https://github.com/mpaland/printf
24 * Revision history: 1.0
25 */
26
27// Configuration file for the printf.h lib
28#ifndef PRINTF_CONFIG_H
29#define PRINTF_CONFIG_H
30
31/**
32 * ntoa (integer) conversion buffer size.
33 * This must be big enough to hold one converted numeric
34 * number including leading zeros, normally 32 is a
35 * sufficient value. Created on the stack
36 */
37#define PRINTF_NTOA_BUFFER_SIZE          32
38
39/**
40 * ftoa (float) conversion buffer size.
41 * This must be big enough to hold one converted float number
42 * including leading zeros, normally 32 is a sufficient value.
43 * Created on the stack
44 */
45#define PRINTF_FTOA_BUFFER_SIZE         32
46
47/**
48 * Define the default floating point precision
49 */
50#define PRINTF_DEFAULT_FLOAT_PRECISION 6
51
52/**
53 * Define the largest suitable value to be printed with %f,
54 * before using exponential representation
55 */
56#define PRINTF_MAX_FLOAT              1E10
57
```

```

58 /**
59 * Define this to disable floating point (%f) support
60 */
61 // #define PRINTF_DISABLE_SUPPORT_FLOAT
62
63 #ifndef CONFIG_DEBUG
64 /**
65 * Define this to disable exponential floating point (%e) support
66 */
67 #define PRINTF_DISABLE_SUPPORT_EXPONENTIAL
68
69 /**
70 * Define this to disable long long (%ll) support
71 */
72 // #define PRINTF_DISABLE_SUPPORT_LONG_LONG
73
74 /**
75 * Define this to disable ptrdiff_t (%t) support
76 */
77 #define PRINTF_DISABLE_SUPPORT_PTRDIFF_T
78 #endif
79
80 #endif /* PRINTF_CONFIG_H */

```

Listing C.7: pArm-S2/pArm.X/printf/printf_config.h

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: rsa.h
22 * Author: Javinator9889
23 * Comments: RSA header file which common types and functions
24 * Revision history: v1.1
25 */
26
27 ****
28 RSA Algorithm
29 Reminder: ((m*e)**d) % n = m
30 e: encryption, d: decryption

```

```
31 Encryption: ciphertext = message**e % n
32 Decryption: (c**d == (m**e)**d == m )    % n
33
34 RSA Key Generation
35 p and q, two distinct prime numbers
36 n = pq
37 fi is Euler's Totient Function
38 fi(n) = fi(p) * fi(q) = (p - 1) * (q - 1) = n - (p + q - 1)
39
40 chose e, the public key:
41     - 1 < e < fi(n)
42     - gcd(e, fi(n)) == 1 (i.e. e and fi(n) are coprime)
43 chose d, the private key:
44     - d == e**-1 (mod fi(n))
45     -> d is the modular multiplicative inverse of e (modulo(fi(n)))
46 ****
47 #ifndef RSA_H
48 #define RSA_H
49
50 // Custom type definitions
51 #include <stdint.h>
52
53 /**
54 * Prime number limits
55 */
56 #define MIN_PRIME_NUMBER 12049ULL
57 #define MAX_PRIME_NUMBER 1299827ULL
58
59 typedef struct {
60     /**
61     * In RSA, 'n' stands for the module.
62     */
63     int_fast64_t n;
64
65     /**
66     * In RSA, 'phi' equals: (p - 1) * (q - 1)
67     */
68     int_fast64_t phi;
69
70     /**
71     * In RSA, 'e' stands for "exponent".
72     */
73     int_fast64_t e;
74
75     /**
76     * In RSA, 'd' is the private key.
77     */
78     int_fast64_t d;
79 }
80 /**
81 * Custom structure containing all the RSA required information.
82 */
83 rsa_t;
84
85 /**
86 * Generates new pseudo-random RSA keys. Those keys may be considered safe
```

```

87 * as they are composed upon prime numbers but their length is very small
88 * (32 bits).
89 *
90 * @return rsa_t containing the new keys.
91 */
92 rsa_t RSA_keygen(void);
93
94 /**
95 * Encrypts the given message by using an already existing key.
96 *
97 * @param msg - the message to be encrypted.
98 * @param key - the key used to encrypt the message.
99 * @return int_fast64_t representing the encrypted message.
100 */
101 int_fast64_t RSA_encrypt(int_fast64_t msg, rsa_t *key);
102
103 /**
104 * Decrypts the given message by using an already existing key.
105 *
106 * @param text - the text to be decrypted.
107 * @param key - the key used to decrypt the message.
108 * @return int_fast64_t representing the decrypted message.
109 */
110 int_fast64_t RSA_decrypt(int_fast64_t text, rsa_t *key);
111
112 /**
113 * Digitally signs a given message by using the private key.
114 *
115 * @param msg - the message to be signed.
116 * @param key - the key used to sign the message.
117 * @return int_fast64_t representing the signed message.
118 */
119 int_fast64_t RSA_sign(int_fast64_t msg, rsa_t *key);
120
121 #endif /* RSA_H */

```

Listing C.8: pArm-S2/pArm.X/rsa/rsa.h

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */

```

```
19/*
20 * File: rand.h
21 * Author: Javinator9889
22 * Comments: Generate random numbers using the elapsed time in ns.
23 * Revision history: v1.0
24 */
25
26
27// This is a guard condition so that contents of this file are not included
28// more than once.
29#ifndef RAND_H
30#define RAND_H
31
32#include <stdlib.h>
33#include <stdint.h>
34
35#ifndef time_t
36typedef uint64_t time_t;
37#define time_t time_t
38#endif
39
40/**
41 * Initializes the pseudo-random number generator machine. Must be called
42 * before any other function in this class.
43 */
44void RAND_init(void);
45
46/**
47 * Initializes a new random seed. Useful for generating non-deterministic
48 * pseudo-random numbers.
49 */
50void RAND_init_seed(void);
51
52/**
53 * Stops the pseudo-random machine.
54 */
55void RAND_stop(void);
56
57/**
58 * Generates a pseudo-random number in between the given range.
59 *
60 * @param min - the minimum number allowed to be generated.
61 * @param max - the maximum number allowed to be generated.
62 * @return the pseudo-random value.
63 */
64int_fast64_t RAND(int_fast64_t min, int_fast64_t max);
65
66/**
67 * Generates a simple pseudo-random number.
68 *
69 * @return the pseudo-random value.
70 */
71int RAND_random(void);
72
73/**
74 * Pseudo-random machine custom generator.
```

```
75 */  
76 void __attribute__((interrupt, no_auto_psv)) _T6Interrupt(void);  
77  
78 #endif /* RAND_H */
```

Listing C.9: pArm-S2/pArm.X/rsa/rand.h

```
1 /*  
2 * 2020 | pArm-S2 by Javinator9889  
3 *  
4 * This program is free software: you can redistribute it and/or modify  
5 * it under the terms of the GNU General Public License as published by  
6 * the Free Software Foundation, either version 3 of the License, or  
7 * (at your option) any later version.  
8 *  
9 * This program is distributed in the hope that it will be useful,  
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of  
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  
12 * GNU General Public License for more details.  
13 *  
14 * You should have received a copy of the GNU General Public License  
15 * along with this program. If not, see https://www.gnu.org/licenses/.  
16 *  
17 * Created by Javinator9889 on 2020 - pArm-S1.  
18 */  
19  
20 /*  
21 * File: zeros.h  
22 * Author: Javinator9889  
23 * Comments: Utils for counting both trailing and leading zeros.  
24 * Revision history: v1.0  
25 */  
26  
27 // This is a guard condition so that contents of this file are not included  
28 // more than once.  
29 #ifndef ZEROS_H  
30 #define ZEROS_H  
31  
32 #include <stdint.h>  
33  
34 #define REPEAT_2x(X) (X), (X)  
35 #define REPEAT_4x(X) REPEAT_2x(X), REPEAT_2x(X)  
36 #define REPEAT_8x(X) REPEAT_4x(X), REPEAT_4x(X)  
37 #define REPEAT_16x(X) REPEAT_8x(X), REPEAT_8x(X)  
38 #define REPEAT_32x(X) REPEAT_16x(X), REPEAT_16x(X)  
39 #define REPEAT_64x(X) REPEAT_32x(X), REPEAT_32x(X)  
40 #define REPEAT_128x(X) REPEAT_64x(X), REPEAT_64x(X)  
41  
42 extern const unsigned char cls_8b[256];  
43  
44 /**  
45 * Counts the amount of leading zeros of the given value.  
46 *  
47 * @param x - the value to check.  
48 * @return the amount of leading zeros.  
49 */
```

```
50 int_fast64_t cls(int_fast64_t x);
51
52 /**
53 * Counts the amount of trailing zeros of the given value.
54 *
55 * @param x - the value to check.
56 * @return the amount of trailing zeros.
57 */
58 int_fast64_t ctz(int_fast64_t x);
59
60 #endif /* ZEROS_H */
```

Listing C.10: pArm-S2/pArm.X/rsa/zeros.h

```
1  /*
2   * 2020 | pArm-S2 by Javinator9889
3   *
4   * This program is free software: you can redistribute it and/or modify
5   * it under the terms of the GNU General Public License as published by
6   * the Free Software Foundation, either version 3 of the License, or
7   * (at your option) any later version.
8   *
9   * This program is distributed in the hope that it will be useful,
10  * but WITHOUT ANY WARRANTY; without even the implied warranty of
11  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12  * GNU General Public License for more details.
13  *
14  * You should have received a copy of the GNU General Public License
15  * along with this program. If not, see https://www.gnu.org/licenses/.
16  *
17  * Created by Javinator9889 on 2020 - pArm-S1.
18  */
19
20 /*
21  * File: primes.h
22  * Author: Javinator9889
23  * Comments: Helpful operations for working with prime numbers.
24  * Revision history: v1.0
25  */
26
27 // This is a guard condition so that contents of this file are not included
28 // more than once.
29 #ifndef PRIMES_H
30 #define PRIMES_H
31
32 #include <stdint.h>
33 #include <stdbool.h>
34
35 /**
36  * Checks if the given number is prime or not. More trials improves accuracy.
37  * Uses the Miller-Rabin primality test.
38  *
39  * @param p - the number to be checked.
40  * @param trials - the trials until a number is decided to be prime or
41  *                 check again.
42  * @return true if prime, else false.
43  */
```

```

43 */
44 bool check_prime(int_fast64_t p, uint_fast16_t trials);
45
46 #endif /* PRIMES_H */

```

Listing C.11: pArm-S2/pArm.X/rsa/primes.h

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: mutex.h
22 * Author: Javinator9889
23 * Comments: Ensures mutual exclusion when accessing a critical section.
24 * Revision history: v1.0
25 */
26
27 // This is a guard condition so that contents of this file are not included
28 // more than once.
29 #ifndef MUTEX_H
30 #define MUTEX_H
31
32 #include <stdbool.h>
33 #define LOCKED 1
34 #define UNLOCKED 0
35
36 #ifndef mut_t
37 typedef volatile unsigned char mut_t;
38 #define mut_t mut_t
39 #endif
40
41 /**
42 * With the given lock, enter the mutual exclusion zone.
43 *
44 * @param lock - the mutex lock.
45 */
46 void mutex_acquire(mut_t *lock);
47
48 /**
49 * With the given lock, exit the mutual exclusion zone.

```

```

50  /*
51   * @param lock - the mutex lock.
52   */
53 void mutex_release(mutex_t *lock);
54
55 #endif /* MUTEX_H */

```

Listing C.12: pArm-S2/pArm.X/sync/mutex.h

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: barrier.h
22 * Author: Javinator9889
23 * Comments: Synchronization mechanism using a barrier and mutex.
24 * Revision history: v1.0
25 */
26
27 // This is a guard condition so that contents of this file are not included
28 // more than once.
29 #ifndef BARRIER_H
30 #define BARRIER_H
31
32 #include <stdint.h>
33 #include <stdbool.h>
34 #include "mutex.h"
35
36 #ifndef barrier_t
37 typedef struct {
38     /**
39      * Barrier internal counter - contains the amount of process that has
40      * reached the barrier.
41      */
42     uint16_t counter;
43
44     /**
45      * Barrier internal value for knowing how many processes have to reach
46      * the barrier before setting the {@code flag} to {@code true}.
47      */

```

```
48     uint16_t total;
49
50     /**
51      * Barrier flag indicating whether all expected processes have reached
52      * the barrier. True when {@code counter} equals {@code total}.
53      */
54     bool flag;
55
56     /**
57      * Barrier internal lock for ensuring mutual exclusion during barrier
58      * value set.
59      */
60     mut_t lock;
61 }
62 /**
63 * Custom structure containing all the required information for handling
64 * barriers. Must be used with specific methods defined in {@code barrier.h}.
65 */
66 barrier_t;
67 #define barrier_t barrier_t
68 #endif
69
70 /**
71 * Creates a new barrier ready to be used.
72 *
73 * @param total how many processes will define the barrier. Sets the amount to
74 * wait for.
75 * @return a pointer to the new created barrier.
76 */
77 barrier_t *BARRIER_create(uint16_t total);
78
79 /**
80 * Method for indicating that a new process has reached the barrier.
81 *
82 * @param barrier the reached barrier.
83 */
84 void BARRIER_arrive(volatile barrier_t *barrier);
85
86 /**
87 * Update the amount of process that the barrier will have.
88 *
89 * @param barrier the barrier to be updated.
90 * @param p the new amount of processes.
91 */
92 void BARRIER_set_total(volatile barrier_t *barrier, uint16_t p);
93
94 /**
95 * Safely clears the barrier structure.
96 *
97 * @param barrier the barrier to be cleared.
98 */
99 void BARRIER_clr(volatile barrier_t *barrier);
100
101 /**
102 * Forcely updates the barrier setting its {@code flag} to true.
103 *
```

```

104 * @param barrier the barrier to be updated.
105 */
106 void BARRIER_set_done(volatile barrier_t *barrier);
107
108 /**
109 * Checks if all the processes have reached the barrier.
110 *
111 * @param barrier the barrier in which to check.
112 * @return the flag value.
113 */
114 bool BARRIER_all_done(volatile barrier_t *barrier);
115
116 #endif /* BARRIER_H */

```

Listing C.13: pArm-S2/pArm.Xsync/barrier.h

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File:
22 * Author:
23 * Comments:
24 * Revision history:
25 */
26
27 // This is a guard condition so that contents of this file are not included
28 // more than once.
29 #ifndef TIMER3_H
30 #define TIMER3_H
31
32 #include "../motor/motor.h"
33 #include "../sync/barrier.h"
34
35 /**
36 * Motor managed by TMR3.
37 */
38 extern motor_t *TMR3_motor;
39
40 /**

```

```

41 * Barrier used for synchronizing motors.
42 */
43 extern volatile barrier_t *TMR3_barrier;
44
45 /**
46 * Initializes the TMR3 to manage specified motor position in the given
47 * barrier.
48 *
49 * @param motor - the motor to manage.
50 * @param barrier - the barrier that handles all the motors.
51 */
52 void TMR3_Initialize(motor_t *motor, volatile barrier_t *barrier);
53
54 /**
55 * TMR3 interrupt handler for managing the motor status.
56 */
57 void __attribute__((interrupt, no_auto_psv)) _T3Interrupt(void);
58
59 /**
60 * Starts the TMR3 that handles the specified motor.
61 */
62 void TMR3_Start(void);
63
64 /**
65 * Stops the TMR3 that handles the specified motor.
66 */
67 void TMR3_Stop(void);
68
69 #endif /* TIMER3_H */

```

Listing C.14: pArm-S2/pArm.X/timers/tmr3.h

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File:
22 * Author:
23 * Comments:
24 * Revision history:

```

```

25 */
26
27 // This is a guard condition so that contents of this file are not included
28 // more than once.
29 #ifndef TIMER4_H
30 #define TIMER4_H
31
32 #include "../motor/motor.h"
33 #include "../sync/barrier.h"
34
35 /**
36 * Motor managed by TMR4.
37 */
38 extern motor_t *TMR4_motor;
39
40 /**
41 * Barrier used for synchronizing motors.
42 */
43 extern volatile barrier_t *TMR4_barrier;
44
45 /**
46 * Initializes the TMR4 to manage specified motor position in the given
47 * barrier.
48 *
49 * @param motor - the motor to manage.
50 * @param barrier - the barrier that handles all the motors.
51 */
52 void TMR4_Initialize(motor_t *motor, volatile barrier_t *barrier);
53
54 /**
55 * TMR4 interrupt handler for managing the motor status.
56 */
57 void __attribute__((interrupt, no_auto_psv)) _T4Interrupt(void);
58
59 /**
60 * Starts the TMR4 that handles the specified motor.
61 */
62 void TMR4_Start(void);
63
64 /**
65 * Stops the TMR4 that handles the specified motor.
66 */
67 void TMR4_Stop(void);
68
69#endif /* TIMER4_H */

```

Listing C.15: pArm-S2/pArm.X/timers/tmr4.h

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *

```

```
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20/*
21 * File:
22 * Author:
23 * Comments:
24 * Revision history:
25 */
26
27// This is a guard condition so that contents of this file are not included
28// more than once.
29#ifndef TIMER5_H
30#define TIMER5_H
31
32#include "../motor/motor.h"
33#include "../sync/barrier.h"
34
35/**
36 * Motor managed by TMR5.
37 */
38extern motor_t *TMR5_motor;
39
40/**
41 * Barrier used for synchronizing motors.
42 */
43extern volatile barrier_t *TMR5_barrier;
44
45/**
46 * Initializes the TMR5 to manage specified motor position in the given
47 * barrier.
48 *
49 * @param motor - the motor to manage.
50 * @param barrier - the barrier that handles all the motors.
51 */
52void TMR5_Initialize(motor_t *motor, volatile barrier_t *barrier);
53
54/**
55 * TMR5 interrupt handler for managing the motor status.
56 */
57void __attribute__((interrupt, no_auto_psv)) _T5Interrupt(void);
58
59/**
60 * Starts the TMR5 that handles the specified motor.
61 */
62void TMR5_Start(void);
63
64/**
```

```

65 * Stops the TMR5 that handles the specified motor.
66 */
67 void TMR5_Stop(void);
68
69 #endif /* TIMER3_H */

```

Listing C.16: pArm-S2/pArm.X/timers/tmr5.h

```

1 /*
2  * 2020 | pArm-S2 by Javinator9889
3  *
4  * This program is free software: you can redistribute it and/or modify
5  * it under the terms of the GNU General Public License as published by
6  * the Free Software Foundation, either version 3 of the License, or
7  * (at your option) any later version.
8  *
9  * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: buffer.h
22 * Author: Javinator9889
23 * Comments: Buffer handler and shortcuts for handling buffer_t
24 * Revision history: v1.0
25 */
26
27 // This is a guard condition so that contents of this file are not included
28 // more than once.
29 #ifndef BUFFER_H
30 #define BUFFER_H
31
32 #include <stdlib.h>
33 #include "types.h"
34
35 /**
36 * Creates a buffer of the given size, and initializes it.
37 *
38 * @param size - the buffer size.
39 * @return a pointer to the new created buffer.
40 */
41 buffer_t *BUFFER_create(size_t size);
42
43 /**
44 * Updates the given buffer size. If the new size is lower than the one
45 * that was before, any stored value is destroyed.
46 *
47 * @param buffer - the buffer whose size will be updated.
48 * @param size - the new buffer size. Set to '0' to destroy the buffer.

```

```

49 */
50 void BUFFER_update_size(buffer_t *buffer, size_t size);
51
52 /**
53 * Safely frees the buffer data from the memory. Caution: this method
54 * does not free the space used by the buffer_t itself. Use {@link free(void*)}
55 * instead.
56 *
57 * @param buffer - the buffer to be cleared.
58 */
59 void BUFFER_free(buffer_t *buffer);
60
61 #endif /* BUFFER_H */

```

Listing C.17: pArm-S2/pArm.X/utils/buffer.h

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: defs.h
22 * Author: Javinator9889
23 * Comments: Includes common definitions that may be used in the entire project
24 * Revision history: 1.0
25 */
26
27 #ifndef DEFS_H
28 #define DEFS_H
29 #include <stdint.h>
30
31 // CLK definitions
32 #define FOSC 119808000UL
33 #define FCY 59904000UL
34 #define CLK_SPEED 7.3728
35 #define FCLK_SPEED 73728000UL
36 #define PRESCALE ((uint8_t) FOSC / FCLK_SPEED)
37
38 // Servo definitions
39 #define MIN_PULSE_MS 0.75F
40 #define MAX_PULSE_MS 2.25F

```

```
41 // UART delay
42 #define DELAY_105uS asm volatile ("REPEAT, #4201"); Nop(); // 105uS delay
43
44 // Mathematical constants
45 #define MATH_PI          3.141592653589793238463F
46 #define MATH_TRANS         57.29577951308232087679F
47 #define MATH_TRANS_I       0.017453292519943295769F
48 #define DEG_151            2.356194490192344928847F
49 #define DEG_135            2.356194490192344928847F
50 #define DEG_120            2.094395102393195492308F
51 #define DEG_60             1.047197551196597746154F
52 #define DEG_55             0.959931088596881267308F
53 #define DEG_45             0.785398163397448309616F
54
55 // Possible missing functions
56 #ifndef max
57 #define max(a, b) ((a) > (b)) ? (a) : (b)
58 #endif
59
60 #ifndef min
61 #define min(a, b) ((a) < (b)) ? (a) : (b)
62 #endif
63
64
65 /* A union which permits us to convert between a double and two 32 bit ints. */
66 typedef union {
67     double value;
68
69     struct {
70         uint32_t lsw;
71         uint32_t msw;
72     } parts;
73     uint64_t word;
74 } ieee_double_shape_type;
75
76 /* Get all in one, efficient on 64-bit machines. */
77 #ifndef EXTRACT_WORDS64
78 #define EXTRACT_WORDS64(i,d)
79 do {
80     ieee_double_shape_type gh_u;
81     gh_u.value = (d);
82     (i) = gh_u.word;
83 } while (0)
84 #endif
85
86 /* Unsigned. */
87 #define UINT8_C(c)          c
88 #define UINT16_C(c)         c
89 #define UINT32_C(c)         c ## U
90 #if __WORDSIZE == 64
91 #define UINT64_C(c)        c ## UL
92 #else
93 #define UINT64_C(c)        c ## ULL
94 #endif
95
96 /* Define ALIASNAME as a weak alias for NAME.
```

```

97  If weak aliases are not available, this defines a strong alias. */
98 #define weak_alias(name, aliasname) __weak_alias (name, aliasname)
99 #define __weak_alias(name, aliasname) \
100   extern __typeof (name) aliasname __attribute__ ((weak, alias (#name)));
101
102#endif /* DEFS_H */

```

Listing C.18: pArm-S2/pArm.X/utils/defs.h

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: time.h
22 * Author: Javinator9889
23 * Comments: the time management library
24 * Revision history: 1.0
25 */
26
27 // This is a guard condition so that contents of this file are not included
28 // more than once.
29 #ifndef TIME_H
30 #define TIME_H
31
32 #include "types.h"
33
34 /**
35 * Initializes the time counting machine.
36 */
37 void TIME_init(void);
38
39 /**
40 * Updates the milliseconds timestamp using the microseconds one.
41 */
42 void TIME_updateMs(void);
43
44 /**
45 * Obtains current time in milliseconds.
46 *
47 * @return time_t - current time in milliseconds.

```

```

48 */
49 time_t TIME_now(void);
50
51 /**
52 * Obtains the current time in microseconds.
53 *
54 * @return time_t - current time in microseconds.
55 */
56 time_t TIME_now_us(void);
57
58 /**
59 * Sets the current time with the given value, in microseconds.
60 *
61 * @param value_us - the new timestamp.
62 */
63 void TIME_set_time(time_t value_us);
64
65 /**
66 * TMR1 interrupt handler - counts-up microseconds.
67 */
68 void __attribute__((__interrupt__, no_auto_psv)) _T1Interrupt(void);
69
70 /**
71 * TMR2 interrupt handler - counts-up milliseconds.
72 */
73 void __attribute__((__interrupt__, no_auto_psv)) _T2Interrupt(void);
74
75 #endif /* TIME_H */

```

Listing C.19: pArm-S2/pArm.X/utils/time.h

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: types.h
22 * Author: Javinator9889
23 * Comments: A header file that contains all custom data types used in this project.
24 * Revision history: 1.0
25 */

```

```
26
27 #ifndef TYPES_H
28 #define TYPES_H
29
30 #include <stdint.h>
31 #include <stdbool.h>
32 #include <float.h>
33 #include <stddef.h>
34
35 // Time definitions
36 #ifndef time_t
37 typedef uint64_t time_t;
38 #define time_t time_t
39 #endif
40
41 // Double precision double type
42 #ifndef double64_t
43 #if DBL_MANT_DIG < LDBL_MANT_DIG
44 typedef long double double64_t;
45 #else
46 typedef double double64_t;
47 #endif
48 #define double64_t double64_t
49 #endif
50
51 // Position definitions
52 #ifndef point_t
53
54 typedef struct {
55     double64_t x;
56     double64_t y;
57     double64_t z;
58 } point_t;
59 #define point_t point_t
60 #endif
61
62 // Angle definitions
63 #ifndef angle_t
64
65 typedef struct {
66     double64_t theta0;
67     double64_t theta1;
68     double64_t theta2;
69 } angle_t;
70 #define angle_t angle_t
71 #endif
72
73 // Pointer to function definitions
74 #ifndef TMR_func
75
76 typedef void (*TMR_func)(void) ;
77 #define TMR_func TMR_func
78 #endif
79
80 // GCODE custom return type
81 #ifndef GCODE_ret_t
```

```
82
83 typedef struct {
84     bool is_err;
85     int_fast16_t code;
86     void *gcode_ret_val;
87 } GCODE_ret;
88 #define GCODE_ret_t GCODE_ret
89 #endif
90
91 // order_t defs
92 #ifndef MAX_ORDER_LENGTH
93 #define MAX_ORDER_LENGTH 1024U
94 #endif
95
96
97 // buffer_t definition for arbitrary buffers
98 #ifndef buffer_t
99
100 typedef struct {
101     /**
102      * Current buffer size. Not variable by default. {@see utils/buffer.h}
103      */
104     size_t size;
105
106     /**
107      * Current buffer size, in bytes. Not variable by default. {@see utils/buffer.h}
108      */
109     size_t bsize;
110
111     /**
112      * The buffer contents itself.
113      */
114     char *buffer;
115 }
116 /**
117 * Custom buffer definition for better handling arbitrary size strings.
118 */
119 buffer_t;
120 #define buffer_t buffer_t
121 #endif
122
123 #ifndef order_t
124
125 typedef struct {
126     /**
127      * Flag active when a new message is received through UART port.
128      * It is updated at <pre>interrupts.h#_U1RXInterrupt</pre>.
129      *
130      * @type bool
131      * @see interrupts.h#_U1RXInterrupt
132      */
133     bool message_received;
134
135     /**
136      * Buffer which contains the order received by the UART. It has fixed
137      * size so no extra space is used. This variable is updated at
```

```

138     * <pre>interrupts.h#_U1RXInterrupt</pre>.
139     *
140     * @type buffer_t
141     * @see interrupts.h#_U1RXInterrupt
142     */
143     buffer_t *order_buffer;
144 }
145 /**
146 * Order container with all the required information for managing
147 * the UART messages.
148 */
149 order_t;
150 #define order_t order_t
151 #endif
152
153 #endif /* TYPES_H */

```

Listing C.20: pArm-S2/pArm.X/utils/types.h

```

1 /*
2  * 2020 | pArm-S2 by Javinator9889
3  *
4  * This program is free software: you can redistribute it and/or modify
5  * it under the terms of the GNU General Public License as published by
6  * the Free Software Foundation, either version 3 of the License, or
7  * (at your option) any later version.
8  *
9  * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: uart.h
22 * Author: Javinator9889
23 * Comments: UART general I/O file handler
24 * Revision history: 1.0
25 */
26
27 // This is a guard condition so that contents of this file are not included
28 // more than once.
29 #ifndef UART_H
30 #define UART_H
31
32 #include <stdint.h>
33 #include <stdbool.h>
34
35 /**
36 * Writes the given character through the UART1.
37 * @param character - the char to be written.

```

```

38 */
39 void putch(char character);
40
41 #ifdef USE_CUSTOM_PRINTF
42 /**
43 * Wrapper for {@link putch}.
44 */
45 void _putchar(char character);
46#endif
47
48#endif /* UART_H */

```

Listing C.21: pArm-S2/pArm.X/utils/uart.h

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: utils.h
22 * Author: Javinator9889
23 * Comments: Standard utils for using them along the project
24 * Revision history: 1.0
25 */
26
27 // This is a guard condition so that contents of this file are not included
28 // more than once.
29#ifndef UTILS_H
30#define UTILS_H
31
32#include "types.h"
33#include "defs.h"
34
35// Gets the size of an array
36#define arrsize(array) (sizeof (array) / sizeof *(array))
37
38// Iterates through an array
39#define foreach(idxtyp, item, array) \
40    idxtyp* item; \
41    size_t size = arrsize(array); \
42    for (item = array; item < (array + size); ++item)

```

```
43 #define clockCyclesPerMicrosecond() ( FCY / 1000000UL )
44 #define constraint(amt,low,high) ((amt)<(low)?(low):((amt)>(high)?(high):(amt)))
45
46 /**
47 * Maps a given value in between the constraints.
48 *
49 * @param x - the given value.
50 * @param in_min - minimum possible input value.
51 * @param in_max - maximum possible input value.
52 * @param out_min - minimum possible output value.
53 * @param out_max - maximum possible output value.
54 * @return the mapped value.
55 */
56 long map(long x, long in_min, long in_max, long out_min, long out_max);
57
58 /**
59 * Rounds the given value with better precision as by default.
60 *
61 * @param value - the value to be rounded.
62 * @return the rounded value.
63 */
64 double64_t roundp(double64_t value);
65
66 /**
67 * Maps a given value in between the constraints.
68 *
69 * @param value - the given value.
70 * @param in_min - minimum possible input value.
71 * @param in_max - maximum possible input value.
72 * @param out_min - minimum possible output value.
73 * @param out_max - maximum possible output value.
74 * @return the mapped value.
75 */
76 double64_t preciseMap(
77     double64_t value,
78     double64_t in_min,
79     double64_t in_max,
80     double64_t out_min,
81     double64_t out_max);
82
83
84 /**
85 * Maps a given value in between the constraints.
86 *
87 * @param x - the given value.
88 * @param in_min - minimum possible input value.
89 * @param in_max - maximum possible input value.
90 * @param out_min - minimum possible output value.
91 * @param out_max - maximum possible output value.
92 * @return the mapped value.
93 */
94 double64_t mapf(double64_t x, double64_t in_min, double64_t in_max, double64_t out_min,
95                 double64_t out_max);
96
97 /**
98 * Checks if the given value is NaN (Not a Number).
```

```

98  /*
99   * @param x - the input value.
100  * @return true if is NaN, else false.
101  */
102 bool __isnan(double64_t x);
103
104 /**
105  * Delays the specified amount of time.
106  * @param ms - the time to delay.
107  */
108 void delay_ms(time_t ms);
109
110 /**
111  * Delays the specified amount of time.
112  * @param us - the time to delay.
113  */
114 void delay_us(time_t us);
115
116 #endif /* UTILS_H */

```

Listing C.22: pArm-S2/pArm.X/utils/utils.h

```

1 /*
2  * 2020 | pArm-S2 by Javinator9889
3  *
4  * This program is free software: you can redistribute it and/or modify
5  * it under the terms of the GNU General Public License as published by
6  * the Free Software Foundation, either version 3 of the License, or
7  * (at your option) any later version.
8  *
9  * This program is distributed in the hope that it will be useful,
10  * but WITHOUT ANY WARRANTY; without even the implied warranty of
11  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12  * GNU General Public License for more details.
13  *
14  * You should have received a copy of the GNU General Public License
15  * along with this program. If not, see https://www.gnu.org/licenses/.
16  *
17  * Created by Javinator9889 on 2020 - pArm-S1.
18  */
19
20 /*
21  * File: arm_config.h
22  * Author: Javinator9889
23  * Comments: The arm configuration that may be used across the application
24  * Revision history: v1.0
25  */
26
27 // This is a guard condition so that contents of this file are not included
28 // more than once.
29 /**
30  * Different configurations are defined in this document for the pArm
31  * manipulator. As intended, multiple end-effectors can be attached,
32  * so here are the definitions for all of them (got from both GitHub
33  * UFACTORY project and custom pieces definition)
34 */

```

```

35 #ifndef ARM_CONFIG_H
36 #define ARM_CONFIG_H
37
38 #define DEFAULT_NORMAL_HEIGHT    74.55
39 #define DEFAULT_NORMAL_FRONT     56.65
40
41 #define DEFAULT_LASER_HEIGHT     51.04
42 #define DEFAULT_LASER_FRONT      64.4
43
44 #define DEFAULT_3DPRINT_HEIGHT   74.43
45 #define DEFAULT_3DPRINT_FRONT    56.5
46
47 #define DEFAULT_PEN_HEIGHT       74.43
48 #define DEFAULT_PEN_FRONT        69.5
49
50 #define ARM_BASE_HEIGHT          133.2
51 #define ARM_BASE_DEVIATION       13.2
52 #define ARM_LOWER_ARM            142.07
53 #define ARM_UPPER_ARM            158.81
54 #define ARM_UPPER_LOWER           (ARM_UPPER_ARM / ARM_LOWER_ARM)
55
56 #ifdef NORMAL_MODE
57 #define height_offset DEFAULT_NORMAL_HEIGHT
58 #define front_end_offset DEFAULT_NORMAL_FRONT
59 #elif defined(PEN_MODE)
60 #define height_offset DEFAULT_PEN_HEIGHT
61 #define front_end_offset DEFAULT_PEN_FRONT
62 #elif defined(LASER_MODE)
63 #define height_offset DEFAULT_LASER_HEIGHT
64 #define front_end_offset DEFAULT_LASER_FRONT
65 #else
66 #define height_offset DEFAULT_NORMAL_HEIGHT
67 #define front_end_offset DEFAULT_NORMAL_FRONT
68#endif
69
70 #define LOWER_ARM_MAX_ANGLE      135.6
71 #define LOWER_ARM_MIN_ANGLE       0.0
72 #define UPPER_ARM_MAX_ANGLE      119.9
73 #define UPPER_ARM_MIN_ANGLE       0.0
74 #define LOWER_UPPER_MAX_ANGLE     151.0
75 #define LOWER_UPPER_MIN_ANGLE     10.0
76
77 // https://github.com/UPM-Robotics/uarm/blob/master/docs/robot-information/
78 #define ARM_MAX_X_LENGTH         346F
79 #define ARM_MAX_Z_HEIGHT          360.6F
80 #define ARM_MIN_X_LENGTH          87F
81 #define ARM_MIN_Z_HEIGHT          ARM_BASE_HEIGHT
82
83#endif /* ARM_CONFIG_H */

```

Listing C.23: pArm-S2/pArm.X/arm_config.h

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify

```

```
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20/*
21 * File:
22 * Author:
23 * Comments:
24 * Revision history:
25 */
26
27// This is a guard condition so that contents of this file are not included
28// more than once.
29#ifndef INIT_H
30#define INIT_H
31
32#include <xc.h>
33#include "system_types.h"
34
35/**
36 * Initializes the PWM module with the required values at the specific ports.
37 */
38void initPWM(void);
39
40/**
41 * Initializes TMR1, which counts up every microsecond.
42 */
43void TMR1_Initialize(void);
44
45/**
46 * Initializes TMR2, which counts up every millisecond.
47 */
48void TMR2_Initialize(void);
49
50/**
51 * Initializes the peripheral ports.
52 */
53void init_ports(void);
54
55/**
56 * Initializes the UART communications.
57 */
58void initUART(void);
59
60/**
```

```
61 * Initializes the system clock.  
62 */  
63 void init_clock(void);  
64  
65 /**  
66  * @Summary  
67  * Enables global interrupts of the dsPIC33EP512GM604  
68  
69  * @Description  
70  * This routine enables the global interrupt bit for the dsPIC33EP512GM604  
71  
72  * @Preconditions  
73  * None.  
74  
75  * @Returns  
76  * None.  
77  
78  * @Param  
79  * None.  
80  
81  * @Example  
82  * <code>  
83  * void SYSTEM_Initialize(void)  
84  * {  
85  *     // Other initializers are called from this function  
86  *     INTERRUPT_GlobalEnable ();  
87  * }  
88  * </code>  
89  
90 */  
91 inline static void INTERRUPT_GlobalEnable(void)  
92 {  
93     __builtin_enable_interrupts();  
94 }  
95  
96 inline static void INTERRUPT_GlobalDisable(void) {  
97     __builtin_disable_interrupts();  
98 }  
99  
100 /**  
101 * Sets the CPU core control register operating mode to a value that is decided by the  
102 * SYSTEM_CORCON_MODES argument.  
103 * @param modeValue SYSTEM_CORCON_MODES initialization mode specifier  
104 * @example  
105 * <code>  
106 * SYSTEM_CORCONModeOperatingSet(CORCON_MODE_ENABLEALLSATNORMAL_ROUNDUNBIASED);  
107 * </code>  
108 */  
109 inline static void SYSTEM_CORCONModeOperatingSet(SYSTEM_CORCON_MODES modeValue)  
110 {  
111     CORCON = (CORCON & 0x00F2) | modeValue;  
112 }  
113  
114 /**  
115 * Initializes the entire system. This function is a wrapper of all the other  
116 * functions declared in {@code init.h}.
```

```
117 */  
118 void system_initialize(void);  
119  
120 #endif /* INIT_H */
```

Listing C.24: pArm-S2/pArm.X/init.h

```
1 /*  
2 * 2020 | pArm-S2 by Javinator9889  
3 *  
4 * This program is free software: you can redistribute it and/or modify  
5 * it under the terms of the GNU General Public License as published by  
6 * the Free Software Foundation, either version 3 of the License, or  
7 * (at your option) any later version.  
8 *  
9 * This program is distributed in the hope that it will be useful,  
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of  
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the  
12 * GNU General Public License for more details.  
13 *  
14 * You should have received a copy of the GNU General Public License  
15 * along with this program. If not, see https://www.gnu.org/licenses/.  
16 *  
17 * Created by Javinator9889 on 2020 - pArm-S1.  
18 */  
19  
20 /*  
21 * File: interrupts.h  
22 * Author: Javinator9889  
23 * Comments: base interrupts handler with different handlers  
24 * Revision history: 1.0  
25 */  
26  
27 // This is a guard condition so that contents of this file are not included  
28 // more than once.  
29 #ifndef INTERRUPTS_H  
30 #define INTERRUPTS_H  
31  
32 #include <xc.h>  
33 #include <stdint.h>  
34 #include "utils/types.h"  
35  
36 // UART RX initializer  
37 void U1RX_Init(volatile order_t* order);  
38 #ifdef LIMIT_SWITCH_ENABLED  
39 // Limit switch interrupts initializer  
40 void CN_Init(volatile uint_fast8_t *switch_map);  
41 #endif  
42  
43 // Define Timer interrupts  
44 /**  
45 * UART1 Transmission interrupt  
46 */  
47 void __attribute__((__interrupt__, no_auto_psv)) _U1TXInterrupt(void);  
48  
49 /**
```

```

50 * UART1 Reception interrupt
51 */
52 void __attribute__((__interrupt__, no_auto_psv)) _U1RXInterrupt(void);
53
54 /**
55 * UART1 Error interrupt
56 */
57 void __attribute__((__interrupt__, no_auto_psv)) _U1ErrInterrupt(void);
58
59 /**
60 * Peripheral change interrupt
61 */
62 void __attribute__((__interrupt__, no_auto_psv)) _CNInterrupt(void);
63
64 #endif /* INTERRUPTS_H */

```

Listing C.25: pArm-S2/pArm.X/interrupts.h

```

1 /*
2  * 2020 | pArm-S2 by Javinator9889
3  *
4  * This program is free software: you can redistribute it and/or modify
5  * it under the terms of the GNU General Public License as published by
6  * the Free Software Foundation, either version 3 of the License, or
7  * (at your option) any later version.
8  *
9  * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: pragmas.h
22 * Author: Javinator9889
23 * Comments: A collection of configurations used in dsPIC33EP
24 * Revision history: 1.0
25 */
26
27 // This is a guard condition so that contents of this file are not included
28 // more than once.
29 #ifndef PRAGMAS_H
30 #define PRAGMAS_H
31
32 #ifndef CONFIG_SIMULATOR
33
34 // FICD
35 #pragma config ICS = PGD1      //ICD Communication Channel Select bits->Communicate on PGEC1 and
36 // PGED1
36 #pragma config JTAGEN = OFF    //JTAG Enable bit->JTAG is disabled
37

```

```

38 // FPOR
39 #pragma config BOREN = ON      // -> BOR is enabled
40 #pragma config ALTI2C1 = OFF    // Alternate I2C1 pins -> I2C1 mapped to SDA1/SCL1 pins
41 #pragma config ALTI2C2 = OFF    // Alternate I2C2 pins -> I2C2 mapped to SDA2/SCL2 pins
42 #pragma config WDTWIN = WIN25   // Watchdog Window Select bits -> WDT Window is 25% of WDT period
43
44 // FWDT
45 #pragma config WDTPS = PS32768  // Watchdog Timer Postscaler bits -> 1:32768
46 #pragma config WDTPRE = PR128   // Watchdog Timer Prescaler bit -> 1:128
47 #pragma config PLLKEN = ON     // PLL Lock Enable bit -> Clock switch to PLL source will wait until
        the PLL lock signal is valid.
48 #pragma config WINDIS = OFF    // Watchdog Timer Window Enable bit -> Watchdog Timer in Non-Window
        mode
49 #pragma config FWDTEN = OFF    // Watchdog Timer Enable bit -> Watchdog timer enabled/disabled by
        user software
50
51 // FOSC
52 #pragma config POSCMD = NONE   // Primary Oscillator Mode Select bits -> Primary Oscillator
        disabled
53 #pragma config OSCIOFNC = OFF   // OSC2 Pin Function bit -> OSC2 is clock output
54 #pragma config IOL1WAY = OFF    // Peripheral pin select configuration -> Allow only one reconfig
        uration
55 #pragma config FCKSM = CSECME  // Clock Switching Mode bits -> Both Clock switching and Fail-
        safe Clock Monitor are enabled
56
57 // FOSCSEL
58 #pragma config FNOSC = FRC     // Oscillator Source Selection -> FRC
59 #pragma config PWMLOCK = OFF   // PWM Lock Enable bit -> Certain PWM registers may only be
        written after key sequence
60 #pragma config IESO = OFF      // Two-speed Oscillator Start-up Enable bit -> Start up with user-
        selected oscillator source
61
62 // FGS
63 #pragma config GWRP = OFF     // General Segment Write-Protect bit -> General Segment may be
        written
64 #pragma config GCP = OFF      // General Segment Code-Protect bit -> General Segment Code protect is
        Disabled
65
66 #endif /* CONFIG_SIMULATOR */
67
68 #endif /* PRAGMAS_H */

```

Listing C.26: pArm-S2/pArm.X/pragmas.h

```

1 /**
2  * @Generated PIC24 / dsPIC33 / PIC32MM MCUs Source File
3
4  * @Company:
5  *   Microchip Technology Inc.
6
7  * @File Name:
8  *   system_types.h
9
10 * @Summary:
11 *   This is the system_types.h file generated using PIC24 / dsPIC33 / PIC32MM MCUs
12

```



```
normal                                     * ACCA/ACCB saturation mode and
67 set                                         * rounding to Biased (
68 conventional)                           * mode. Rest of CORCON settings
69 are                                         * set to the default POR values.
70
71 CORCON_MODE_ENABLEALLSATNORMAL_ROUNDUNBIASED = 0x00E0, /* Enable saturation for ACCA,
72 ACCB                                         * and Dataspace write, enable
73 normal                                     * ACCA/ACCB saturation mode and
74 set                                         * rounding to Unbiased (
75 convergent)                            * mode. Rest of CORCON settings
76 are                                         * set to the default POR values.
77
78 CORCON_MODE_DISABLEALLSAT_ROUNDUNBIASED = 0x0022, /* Disable saturation for ACCA,
79 ACCB                                         * and Dataspace write and set
80 conventional)                           * rounding to Biased (
81 are                                         * mode. Rest of CORCON settings
82
83 CORCON_MODE_DISABLEALLSAT_ROUNDUNBIASED = 0x0020, /* Disable saturation for ACCA,
84 ACCB                                         * and Dataspace write and set
85 conventional)                           * rounding to Unbiased (
86 are                                         * mode. Rest of CORCON settings
87
88 CORCON_MODE_ENABLEALLSATSUPER_ROUNDUNBIASED = 0x00F2, /* Enable saturation for ACCA, ACCB
89 super                                         * and Dataspace write, enable
90 set                                         * ACCA/ACCB saturation mode and
91 conventional)                           * rounding to Biased (
92 are                                         * mode. Rest of CORCON settings
93
94 CORCON_MODE_ENABLEALLSATSUPER_ROUNDUNBIASED = 0x00F0, /* Enable saturation for ACCA, ACCB
95 super                                         * and Dataspace write, enable
96 set                                         * ACCA/ACCB saturation mode and
97 conventional)                           * rounding to Unbiased (
98
99
100 CORCON_MODE_ENABLEALLSATSUPER_ROUNDUNBIASED = 0x00F0, /* Enable saturation for ACCA, ACCB
101 super                                         * and Dataspace write, enable
      * ACCA/ACCB saturation mode and
      * rounding to Unbiased (
```

```

102     convergent)
103         are
104
105 } SYSTEM_CORCON_MODES;
106
107 #endif /* SYSTEM_TYPES_H */
108 /**
109 End of File
110 */

```

Listing C.27: pArm-S2/pArm.X/system_types.h

C.2. *Source files*

```

1 /*
2 * 2020 - present | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: planner.h
22 * Author: Javinator9889
23 * Comments: The planner that controls how the arm moves
24 * Revision history: v1.0
25 */
26
27 #include <xc.h>
28 #include <float.h>
29 #include <math.h>
30 #include <stdlib.h>
31 #include <stdint.h>
32 #include "planner.h"
33 #include "../motor/motor.h"
34 #include "../utils/types.h"
35 #include "../utils/utils.h"
36 #include "../motor/kinematics.h"
37 #include "../arm_config.h"

```

```
38 #include "../timers/tmr3.h"
39 #include "../timers/tmr4.h"
40 #include "../timers/tmr5.h"
41 #include "../utils/defs.h"
42 #include "../sync/barrier.h"
43 #ifdef DEBUG_ENABLED
44 #include "../printf/io.h"
45 #endif
46
47 servo_t base_servo = {&SDC1, NULL, MATH_PI / 2, LOWER_UPPER_MIN_ANGLE, LOWER_UPPER_MAX_ANGLE};
48 servo_t lower_arm_servo = {&SDC2, NULL, MATH_PI, LOWER_ARM_MIN_ANGLE, LOWER_ARM_MAX_ANGLE};
49 servo_t upper_arm_servo = {&SDC3, NULL, MATH_PI, UPPER_ARM_MIN_ANGLE, UPPER_ARM_MAX_ANGLE};
50 servo_t end_effector_servo = {&SDC4, NULL, .0, .0, 180.};
51
52 motor_t base_motor = {&base_servo, 0ULL, .0F, .0F, false, 1, 0ULL, TMR3_Start, TMR3_Stop};
53 motor_t lower_arm_motor = {&lower_arm_servo, 1ULL, .0F, .0F, false, 1, 0ULL, TMR4_Start,
     TMR4_Stop};
54 motor_t upper_arm_motor = {&upper_arm_servo, 2ULL, .0F, .0F, false, 1, 0ULL, TMR5_Start,
     TMR5_Stop};
55 motor_t end_effector_motor = {&end_effector_servo, 3ULL, .0F, .0F, false, 1, 0ULL, NULL, NULL};
56
57 motors_t motors = {&base_motor, &lower_arm_motor, &upper_arm_motor, &end_effector_motor};
58 volatile barrier_t *PLANNER_barrier;
59
60 static inline double64_t expected_duration(angle_t angle) {
61     double64_t max_angle = LDBL_MIN;
62     double64_t diff_base = fabsl(angle.theta0 - MOTOR_position_rad(motors.base_motor));
63     double64_t diff_lower_arm = fabsl(angle.theta1 - MOTOR_position_rad(motors.lower_arm));
64     double64_t diff_upper_arm = fabsl(angle.theta2 - MOTOR_position_rad(motors.upper_arm));
65     max_angle = max(diff_base, max_angle);
66     max_angle = max(diff_lower_arm, max_angle);
67     max_angle = max(diff_upper_arm, max_angle);
68
69     return MOTOR_elapsed_time_us(max_angle);
70 }
71
72 static inline void map_angle(angle_t *angle) {
73     angle->theta0 = mapf(angle->theta0, -(MATH_PI / 2), (MATH_PI / 2), 0, MATH_PI);
74     angle->theta1 = mapf(angle->theta1, .0F, DEG_135, MATH_PI, DEG_45);
75     angle->theta2 = mapf(angle->theta2, .0F, DEG_60, MATH_PI, DEG_135);
76 }
77
78 static inline void unmap_angle(angle_t *angle) {
79     angle->theta0 = mapf(angle->theta0, 0, MATH_PI, -(MATH_PI / 2), (MATH_PI / 2));
80     angle->theta1 = mapf(angle->theta1, MATH_PI, DEG_45, .0F, DEG_135);
81     angle->theta2 = mapf(angle->theta2, MATH_PI, DEG_135, .0F, DEG_60);
82 }
83
84 #ifdef LIMIT_SWITCH_ENABLED
85 void PLANNER_init(volatile barrier_t *barrier, uint_fast8_t switch_map[4]) {
86     base_servo.limit_switch_value = &switch_map[0];
87     lower_arm_servo.limit_switch_value = &switch_map[1];
88     upper_arm_servo.limit_switch_value = &switch_map[2];
89     end_effector_servo.limit_switch_value = &switch_map[3];
90 } else
91 void PLANNER_init(volatile barrier_t *barrier) {
```

```
92 #endif
93     PLANNER_barrier = barrier;
94     TMR3_Initialize(motors.base_motor, PLANNER_barrier);
95     TMR4_Initialize(motors.lower_arm, PLANNER_barrier);
96     TMR5_Initialize(motors.upper_arm, PLANNER_barrier);
97 }
98
99 double64_t PLANNER_go_home(void) {
100 #ifdef DEBUG_ENABLED
101     printf("[DEBUG]\tMoving motors to home...\n");
102 #endif
103     BARRIER_clr(PLANNER_barrier);
104     angle_t home_angles = {
105         motors.base_motor->servoHandler->home,
106         motors.lower_arm->servoHandler->home,
107         motors.upper_arm->servoHandler->home
108     };
109     MOTOR_move(motors.base_motor, motors.base_motor->servoHandler->home);
110     MOTOR_move(motors.lower_arm, motors.lower_arm->servoHandler->home);
111     MOTOR_move(motors.upper_arm, motors.upper_arm->servoHandler->home);
112     return expected_duration(home_angles);
113 }
114
115 double64_t PLANNER_move_xyz(point_t xyz) {
116     BARRIER_clr(PLANNER_barrier);
117     angle_t *angle = (angle_t *) malloc(sizeof(angle_t));
118     char ret = inverse_kinematics(xyz, angle);
119     if (ret != EXIT_SUCCESS)
120         return -1.0F;
121     map_angle(angle);
122     double64_t duration = expected_duration(*angle);
123     MOTOR_move(motors.base_motor, angle->theta0);
124     MOTOR_move(motors.lower_arm, angle->theta1);
125     MOTOR_move(motors.upper_arm, angle->theta2);
126     free(angle);
127     return duration;
128 }
129
130 double64_t PLANNER_move_angle(angle_t angle) {
131     BARRIER_clr(PLANNER_barrier);
132     point_t *position = (point_t *) malloc(sizeof(point_t));
133     forward_kinematics(angle, position);
134     if (!check_constraints_ok(&angle, position))
135         return -1.0F;
136     map_angle(&angle);
137     MOTOR_move(motors.base_motor, angle.theta0);
138     MOTOR_move(motors.lower_arm, angle.theta1);
139     MOTOR_move(motors.upper_arm, angle.theta2);
140     free(position);
141     return expected_duration(angle);
142 }
143
144 void PLANNER_move_waiting(angle_t angle) {
145     double64_t expected_time = PLANNER_move_angle(angle);
146     delay_us(expected_time);
147 }
```

```

148
149 uint8_t PLANNER_stop_moving(void) {
150     if (PLANNER_barrier->flag)
151         return EXIT_FAILURE;
152     MOTOR_freeze(motors.base_motor);
153     MOTOR_freeze(motors.lower_arm);
154     MOTOR_freeze(motors.upper_arm);
155     BARRIER_set_done(PLANNER_barrier);
156     return EXIT_SUCCESS;
157 }
158
159 point_t *PLANNER_get_position(void) {
160     point_t *position = (point_t *) malloc(sizeof (point_t));
161     angle_t *angles = PLANNER_get_angles();
162     forward_kinematics(*angles, position);
163     free(angles);
164     return position;
165 }
166
167 angle_t *PLANNER_get_angles(void) {
168     angle_t *angles = (angle_t *) malloc(sizeof (angle_t));
169     angles->theta0 = MOTOR_position_rad(motors.base_motor);
170     angles->theta1 = MOTOR_position_rad(motors.lower_arm);
171     angles->theta2 = MOTOR_position_rad(motors.upper_arm);
172
173     return angles;
174 }
```

Listing C.28: pArm-S2/pArm.X/arm/planner.c

```

1 /*
2  * 2020 | pArm-S2 by Javinator9889
3  *
4  * This program is free software: you can redistribute it and/or modify
5  * it under the terms of the GNU General Public License as published by
6  * the Free Software Foundation, either version 3 of the License, or
7  * (at your option) any later version.
8  *
9  * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: gcode.c
22 * Author: javinator9889
23 *
24 * Created on 24 de agosto, 2020
25 */
26
```

```
27 #include <float.h>
28 #include <stdlib.h>
29 #include <string.h>
30 #include "gcode.h"
31 #include "../arm/planner.h"
32 #include "../utils/utils.h"
33 #include "../utils/types.h"
34 #include "../utils/buffer.h"
35 #include "../printf/io.h"
36
37 static buffer_t *gcode_buffer = NULL;
38
39 /**
40 * With the given input code and the buffer, iterates through the latest one
41 * until the code is found or returns the default value.
42 *
43 * @param code - the code to search.
44 * @param ret - return value if not found.
45 * @return the number after the given code or the given return value if not found.
46 */
47 double64_t GCODE_parse_number(char code, double64_t ret) {
48     char *token;
49     char *ptr;
50     char *copy = (char *) malloc(gcode_buffer->bsize);
51     double64_t value_if_missing = ret;
52     strcpy(copy, gcode_buffer->buffer);
53     for (ptr = copy; ret == value_if_missing; ptr = NULL) {
54         token = strtok(ptr, " ");
55         if (token == NULL) break;
56         if (token[0] == code) ret = atof(token + 1);
57     }
58     free(copy);
59     return ret;
60 }
61
62 /**
63 * Safely clears the GCODE structures.
64 *
65 * @param ret - the return code to be returned after finishing.
66 * @return ret
67 */
68 static inline GCODE_ret_t GCODE_finish(GCODE_ret_t ret) {
69     BUFFER_free(gcode_buffer);
70     return ret;
71 }
72
73 GCODE_ret_t GCODE_process_command(volatile order_t *order) {
74 #ifdef DEBUG_ENABLED
75     printf("[DEBUG]\tParsing order '%s'\n", order->order_buffer->buffer);
76 #endif
77     GCODE_ret_t ret; // = {false, -1, NULL};
78
79     if (gcode_buffer == NULL) {
80         gcode_buffer = BUFFER_create(order->order_buffer->size);
81     }
82 }
```

```
83 if (gcode_buffer->size != order->order_buffer->size) {
84     BUFFER_update_size(gcode_buffer, order->order_buffer->size);
85 }
86 strcpy(gcode_buffer->buffer, order->order_buffer->buffer);
87 int_fast16_t cmd = (int_fast16_t) GCODE_parse_number('G', -1.0F);
88 switch (cmd) {
89     // G0 X1.234 Y1.234 Z1.234
90     // receives a position by the given
91     // coordinates in XYZ.
92     // If coordinates are missing,
93     // throw an error
94     case 0:
95     {
96         point_t *position = (point_t *) malloc(sizeof(point_t));
97         position->x = GCODE_parse_number('X', LDBL_MIN);
98         position->y = GCODE_parse_number('Y', LDBL_MIN);
99         position->z = GCODE_parse_number('Z', LDBL_MIN);
100        if (position->x == LDBL_MIN ||
101            position->y == LDBL_MIN ||
102            position->z == LDBL_MIN) {
103 #ifdef DEBUG_ENABLED
104             printf("[ERROR]\tCoordinates missing for GCODE G0!\n");
105 #endif
106             ret = (GCODE_ret_t){
107                 true, // is_err
108                 cmd,
109                 NULL
110             };
111         } else {
112             ret = (GCODE_ret_t){
113                 false, // is_err
114                 cmd, // code
115                 position // the return value itself
116             };
117         }
118         break;
119     }
120     // G1 X1.234 Y1.234 Z1.234
121     // receives a set of angles by the given
122     // values in XYZ.
123     // Some angles can be missing, so keep the
124     // current angle in the other motors.
125     // If three angles are missing, throw
126     // an error
127     case 1:
128     {
129         angle_t *angles = (angle_t *) malloc(sizeof(angle_t));
130         angles->theta0 = GCODE_parse_number('X', LDBL_MIN);
131         angles->theta1 = GCODE_parse_number('Y', LDBL_MIN);
132         angles->theta2 = GCODE_parse_number('Z', LDBL_MIN);
133         if (angles->theta0 == LDBL_MIN &&
134             angles->theta1 == LDBL_MIN &&
135             angles->theta2 == LDBL_MIN) {
136 #ifdef DEBUG_ENABLED
137             printf("[ERROR]\tAngles missing for GCODE G1!\n");
138 #endif
```

```
139         ret = (GCODE_ret_t){
140             true, // is_err
141             cmd,
142             NULL
143         };
144     } else {
145         ret = (GCODE_ret_t){
146             false, // is_err
147             cmd, // code
148             angles // the return value itself
149         };
150     }
151     break;
152 }
153 // GCODE G28 - move to home
154 // Leave this to main controller as it
155 // knows the home of each motor and
156 // sets the timer interrupt for them
157 case 28:
158 {
159     ret = (GCODE_ret_t){
160         false, // is_err
161         cmd, // code
162         NULL // the return value itself
163     };
164     break;
165 }
166 // The input GCODE is not recognized
167 // (not implemented)
168 default:
169 {
170 #ifdef DEBUG_ENABLED
171     if (cmd == -1)
172         printf("[INFO]\tGCODE type 'G' not found\n");
173     else
174         printf("[ERROR]\tUnknown GCODE G%d\n", cmd);
175 #endif
176     ret = (GCODE_ret_t){
177         true, // is_err
178         cmd, // the code itself
179         NULL // the error message
180     };
181     break;
182 }
183 }
184
185 // GCODE found so quit and return value
186 if (cmd != -1)
187     return GCODE_finish(ret);
188
189 cmd = (int_fast16_t) GCODE_parse_number('M', -1.0F);
190 switch (cmd) {
191     // GCODE M1 - unconditional stop
192     // When this command is received, the arm
193     // must stop moving and keep current position
194     case 1:
```

```
195     // GCODE M114 - get current position in XYZ
196     // By default, obtains the position in angular
197     // coordinates and leaves the conversion to
198     // main orchestrator
199 case 114:
200     // GCODE M280 - get current position in angles
201 case 280:
202 {
203     ret = (GCODE_ret_t){
204         false, // is_err
205         cmd * 10, // code
206         NULL // the return value itself
207     };
208     break;
209 }
210 // The input GCODE is not recognized
211 // (not implemented)
212 default:
213 {
214 #ifdef DEBUG_ENABLED
215     if (cmd == -1)
216         printf("[INFO]\tGCODE type 'M' not found\n");
217     else
218         printf("[ERROR]\tUnknown GCODE M%d\n", cmd);
219#endif
220     ret = (GCODE_ret_t){
221         true, // is_err
222         cmd * 10, // the code
223         NULL // the error message
224     };
225     break;
226 }
227 }
228
229 // GCODE found so quit and return value
230 if (cmd != -1)
231     return GCODE_finish(ret);
232
233 cmd = (int_fast16_t) GCODE_parse_number('I', -1.0F);
234 switch (cmd) {
235     // GCODE I1 - custom command for sending RSA public key
236     // When received, the main orchestrator must send
237     // both modulus (n) and public exponent (e)
238     // so the other system can decrypt our messages.
239 case 1:
240 {
241     ret = (GCODE_ret_t){
242         false, // is_err
243         cmd * 100, // the code
244         NULL // the return value itself
245     };
246     break;
247 }
248 // All the following must have the random message encrypted.
249 // GCODE I5 - received the unsigned message
250 // when verifying
```

```

251     case 5:
252         // GCODE I6 - generate new RSA keys
253     case 6:
254         // GCODE I7 - received heartbeat so we know
255         // the trusted device is still alive
256     case 7:
257     {
258         size_t size = (size_t) ((gcode_buffer->size - 3) * sizeof (char));
259         char *msg = (char *) malloc(size);
260         strncpy(msg, gcode_buffer->buffer + 3, size);
261         ret = (GCODE_ret_t){
262             false, // is_err
263             cmd * 100, // the code
264             msg // the msg
265         };
266         break;
267     }
268     // The input GCODE is not recognized
269     // (not implemented)
270     default:
271     {
272 #ifdef DEBUG_ENABLED
273         if (cmd == -1)
274             printf("[INFO]\tGCODE type 'I' not found\n");
275         else
276             printf("[ERROR]\tUnknown GCODE I%d\n", cmd);
277 #endif
278         ret = (GCODE_ret_t){
279             true, // is_err
280             cmd * 100, // the code
281             NULL // the msg
282         };
283         break;
284     }
285 }
286 return GCODE_finish(ret);
287 }
```

Listing C.29: pArm-S2/pArm.X/gcode/gcode.c

```

1 /*
2  * 2020 | pArm-S2 by Javinator9889
3  *
4  * This program is free software: you can redistribute it and/or modify
5  * it under the terms of the GNU General Public License as published by
6  * the Free Software Foundation, either version 3 of the License, or
7  * (at your option) any later version.
8  *
9  * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 */
```

```

17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 #include "servo.h"
21 #include "../utils/defs.h"
22 #include "../utils/utils.h"
23
24 //volatile uint_fast8_t limit_switch_map[4] = {0U};
25
26 void SERVO_write_angle(const servo_t *servo, double64_t angle_rad) {
27     double64_t time = SERVO_from_angle_to_ms(angle_rad);
28     SERVO_write_milliseconds(servo, time);
29 }
30
31 inline void SERVO_write_milliseconds(const servo_t *servo, double64_t ms) {
32     *servo->dutyCycleRegister = (uint16_t) (FOSC / ((1 / ms) * 1000 * 64));
33 }
34
35 inline void SERVO_write_value(const servo_t *servo, uint16_t dutyCycleValue) {
36     *servo->dutyCycleRegister = dutyCycleValue;
37 }
38
39 inline double64_t SERVO_from_angle_to_ms(double64_t angle_rad) {
40     return mapf(angle_rad, .0F, MATH_PI, MIN_PULSE_MS, MAX_PULSE_MS);
41 }

```

Listing C.30: pArm-S2/pArm.X/motor/servo.c

```

1 /*
2  * 2020 | pArm-S2 by Javinator9889
3  *
4  * This program is free software: you can redistribute it and/or modify
5  * it under the terms of the GNU General Public License as published by
6  * the Free Software Foundation, either version 3 of the License, or
7  * (at your option) any later version.
8  *
9  * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 #include <stdbool.h>
21 #include <float.h>
22 #include <stdlib.h>
23 #include <math.h>
24 #include "motor.h"
25 #ifdef DEBUG_ENABLED
26 #include "../printf/io.h"
27 #endif
28 #include "../utils/utils.h"

```

```
29 #include "../utils/defs.h"
30 #include "../utils/types.h"
31 #include "../utils/time.h"
32
33 static inline double64_t us_to_deg(double64_t us) {
34     return (us / US_PER_DEGREE);
35 }
36
37 static inline double64_t us_to_rad(double64_t us) {
38     return ((us * (MATH_PI / 180.0F)) / US_PER_DEGREE);
39 }
40
41 static inline double64_t rad_to_us(double64_t rad) {
42     return (rad * MATH_TRANS * US_PER_DEGREE);
43 }
44
45 static inline double64_t deg_to_us(double64_t deg) {
46     return (deg * US_PER_DEGREE);
47 }
48
49 void MOTOR_move(motor_t *motor, double64_t angle_rad) {
50     double64_t current_angle = us_to_rad(motor->angle_us);
51     double64_t expected_time_us = MOTOR_elapsed_time_us(fabsl(angle_rad - current_angle));
52     motor->clockwise = (angle_rad > current_angle)
53         ? 1
54         : -1;
55     motor->movement_duration = expected_time_us;
56     motor->movement_finished = false;
57     motor->current_movement_count = 0ULL;
58     SERVO_write_angle(motor->servoHandler, angle_rad);
59 #ifdef USE_MOTOR_TMRS
60     if (motor->TMR_Start != NULL)
61         motor->TMR_Start();
62 #endif
63 #ifdef DEBUG_ENABLED
64     printf("[DEBUG]\tMotor # %d is moving\n", motor->id);
65 #endif
66 }
67
68 void MOTOR_freeze(motor_t *motor) {
69 #ifdef USE_MOTOR_TMRS
70     // Disable motor interrupts so stop counting
71     motor->TMR_Stop();
72 #endif
73     motor->angle_us = motor->current_movement_count;
74     // Get current position and fix the angle to its value
75     SERVO_write_milliseconds(motor->servoHandler, (motor->angle_us * 1000.0F));
76     motor->current_movement_count = 0ULL;
77     motor->movement_finished = true;
78 }
79
80 inline double64_t MOTOR_position_us(motor_t *motor) {
81     return motor->angle_us;
82 }
83
84 inline double64_t MOTOR_position_rad(motor_t *motor) {
```

```
85     return us_to_rad(motor->angle_us);
86 }
87
88 inline double64_t MOTOR_position_deg(motor_t *motor) {
89     return us_to_deg(motor->angle_us);
90 }
91
92 inline bool check_motor_finished(motor_t *motor, time_t max_waiting_time) {
93 #ifdef LIMIT_SWITCH_ENABLED
94     if (*motor->servoHandler->limit_switch_value == 1)
95         return true;
96 #endif
97     return ((TIME_now_us() >= max_waiting_time))
98         ? true
99         : motor->movement_finished;
100}
101
102 char MOTOR_calibrate(motor_t *motor) {
103 #ifdef DEBUG_ENABLED
104     printf("[SETUP]\tCalibrating motor %d\n", motor->id);
105 #endif
106     // Init the angle to minimum long double value
107     motor->angle_us = LDBL_MIN;
108     // Move the motor to 0 radians
109 #ifdef DEBUG_ENABLED
110     printf("[SETUP]\tMoving motor %d to position 0\n", motor->id);
111 #endif
112     SERVO_write_angle(motor->servoHandler, .0F);
113     // and wait until the interruptor is pressed.
114     // As maybe the interruptor can be not pressed, wait
115     // a maximum amount of time equals to a 180 degrees spin
116     // or the "movement_finished" flag to be true
117     const time_t max_waiting_time =
118         (time_t) (TIME_now_us() + (US_PER_DEGREE * 180.0F));
119 #ifdef DEBUG_ENABLED
120     printf("[SETUP]\tWaiting at most %f s\n", (max_waiting_time / 1E6));
121 #endif
122     while (!check_motor_finished(motor, max_waiting_time));
123     const bool timeout_happened = ((TIME_now_us() >= max_waiting_time) == 1);
124 #ifdef DEBUG_ENABLED
125     printf("[SETUP]\tTimeout? %d\n", timeout_happened);
126 #endif
127     *motor->servoHandler->limit_switch_value = 0;
128     // Disable the PWM signal, so the motor stops moving
129 #ifdef DEBUG_ENABLED
130     printf("[SETUP]\tDisabling PWM signal\n");
131 #endif
132     SERVO_write_value(motor->servoHandler, 0U);
133     // Timeout happened, so return error
134     if (timeout_happened)
135         return EXIT_FAILURE;
136 #ifdef DEBUG_ENABLED
137     printf("[SETUP]\tMoving to 30 degrees\n");
138 #endif
139     // and move it to an arbitrary position at 30 degrees
140     SERVO_write_angle(motor->servoHandler, (MATH_PI / 6));
```

```

141     double64_t duration_us = rad_to_us(MATH_PI / 6);
142 #ifdef DEBUG_ENABLED
143     printf("[SETUP]\tExpected duration: %Lf us\n", duration_us);
144 #endif
145     // waiting until the movement should finish
146     delay_us(rad_to_us(duration_us));
147     // Finally, plan a movement again to 0 radians
148 #ifdef DEBUG_ENABLED
149     printf("[SETUP]\tFinishing calibration... Moving to 0 again\n");
150 #endif
151     motor->movement_duration = duration_us;
152     motor->movement_finished = false;
153     SERVO_write_angle(motor->servoHandler, .0F);
154     motor->TMR_Start();
155     // This time, wait until the interruptor is pressed
156     // or the movement flag finished is set to true
157     while ((*motor->servoHandler->limit_switch_value != 1) ||
158            motor->movement_finished);
159 #ifdef DEBUG_ENABLED
160     printf("[SETUP]\tMovement for motor %d finished!\n", motor->id);
161 #endif
162     // When done, finish the counter and update the minimum angle
163     // with the difference in us obtained
164     motor->TMR_Stop();
165     *motor->servoHandler->limit_switch_value = 0;
166     motor->movement_finished = true;
167     // If the motor is correctly calibrated, this difference may be zero
168     // or approximately zero. In other case, the difference will be
169     // the new minimum angle the motor can reach
170     double64_t min_angle_us = fabsl(motor->angle_us - motor->movement_duration);
171     motor->servoHandler->min_angle = us_to_rad(min_angle_us);
172 #ifdef DEBUG_ENABLED
173     printf("[SETUP]\tMinimum angle for motor %d is: %Lf rad\n", motor->id, motor->servoHandler
174             ->min_angle);
175 #endif
176     // Return OK
177     return EXIT_SUCCESS;
178 }
```

Listing C.31: pArm-S2/pArm.X/motor/motor.c

```

1 /*
2  * 2020 | pArm-S2 by Javinator9889
3  *
4  * This program is free software: you can redistribute it and/or modify
5  * it under the terms of the GNU General Public License as published by
6  * the Free Software Foundation, either version 3 of the License, or
7  * (at your option) any later version.
8  *
9  * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
```

```
16 *  
17 * Created by Javinator9889 on 2020 - pArm-S1.  
18 */  
19  
20 /*  
21 * File: kinematics.c  
22 * Author: Javinator9889  
23 * Comments: Contains the definitions for both forward and inverse kinematics  
24 * Revision history: v1.0  
25 */  
26  
27 #include <math.h>  
28 #include <stdbool.h>  
29 #include <stdlib.h>  
30 #include "kinematics.h"  
31 #include "../utils/defs.h"  
32 #include "../utils/types.h"  
33 #include "../utils/utils.h"  
34 #include "../arm_config.h"  
35 #ifdef DEBUG_ENABLED  
36 #include "../printf/io.h"  
37 #endif  
38  
39 void do_forward_kinematics(  
40     const angle_t angle,  
41     point_t *res,  
42     const double64_t a1,  
43     const double64_t a2,  
44     const double64_t a3,  
45     const double64_t d1,  
46     const double64_t Tx,  
47     const double64_t Tz  
48 ) {  
49     res->x = (a2 * cosl(angle.theta1) + a3 * cosl(angle.theta1 - angle.theta2) + d1)  
50         * cosl(angle.theta0) + Tx;  
51     res->y = (a2 * cosl(angle.theta1) + a3 * cosl(angle.theta1 - angle.theta2) + d1)  
52         * sinl(angle.theta0);  
53     res->z = a1 + (a2 * sinl(angle.theta1)) + (a3 * sinl(angle.theta1 - angle.theta2))  
54         - Tz;  
55 }  
56  
57 bool check_constraints_ok(angle_t *angle, point_t *point) {  
58     bool res = true;  
59     if (__isnan(angle->theta0) || __isnan(angle->theta1) || __isnan(angle->theta2)  
60         || __isnan(point->x) || __isnan(point->y) || __isnan(point->z))  
61         return false;  
62  
63     if (angle->theta0 > DEG_151) {  
64 #ifdef DEBUG_ENABLED  
65         printf("[ERROR]\tTheta 0 value bigger than 151D\n");  
66 #endif  
67         res = false;  
68     }  
69     if (angle->theta1 > DEG_135) {  
70 #ifdef DEBUG_ENABLED  
71         printf("[ERROR]\tTheta 1 value bigger than 135D\n");
```

```
72 #endif
73     res = false;
74 }
75     if (angle->theta2 > DEG_120) {
76 #ifdef DEBUG_ENABLED
77     printf("[ERROR]\tTheta 2 value bigger than 120D\n");
78 #endif
79     res = false;
80 }
81     if (sqrtl(powl(point->x, 2) + powl(point->y, 2)) > 261.0F) {
82 #ifdef DEBUG_ENABLED
83     printf("[ERROR]\tLength sqrt(x2 + y2) > 261!\n");
84 #endif
85     res = false;
86 }
87     if (angle->theta2 > (angle->theta1 + DEG_55)) {
88 #ifdef DEBUG_ENABLED
89     printf("[ERROR]\tPhysical structure limit\n");
90 #endif
91     res = false;
92 }
93     if ((point->x > -53.0F && point->x < 53.0F) &&
94         (point->z < 0.0F) &&
95         (point->y > -53.0F && point->y < 53.0F)) {
96 #ifdef DEBUG_ENABLED
97     printf("[ERROR]\tEnd-effector colliding with arm base!\n");
98 #endif
99     res = false;
100 }
101
102     return res;
103 }
104
105 char inverse_kinematics(point_t in_cartesian, angle_t* angle) {
106 #define AL2      (ARM_LOWER_ARM * ARM_LOWER_ARM)
107 #define AU2      (ARM_UPPER_ARM * ARM_UPPER_ARM)
108
109     double64_t theta_0 = atan2l(in_cartesian.x, in_cartesian.y);
110     double64_t xyz = powl(in_cartesian.x, 2) +
111         powl(in_cartesian.y, 2) +
112         powl(in_cartesian.z, 2);
113     double64_t lxyz = sqrtl(xyz);
114     double64_t theta_1 = acosl((-AL2 - xyz + AU2) / (-2 * ARM_LOWER_ARM * lxyz));
115     double64_t theta_2 = acosl((-AL2 - AU2 + xyz) / (-2 * ARM_LOWER_ARM * ARM_UPPER_ARM));
116     double64_t phi = atan2l(in_cartesian.z, sqrtl(powl(in_cartesian.x, 2) + powl(in_cartesian.y,
117 , 2)));
118
119     theta_1 += phi;
120     theta_1 = DEG_135 - theta_1;
121
122     angle->theta0 = theta_0;
123     angle->theta1 = theta_1;
124     angle->theta2 = theta_2;
125
126     if (check_constraints_ok(angle, &in_cartesian) == true)
127         return EXIT_SUCCESS;
```

```

127     return EXIT_FAILURE;
128 }
129
130 char forward_kinematics(angle_t in_angle, point_t *position) {
131     do_forward_kinematics(
132         in_angle,
133         position,
134         ARM_BASE_HEIGHT,
135         ARM_LOWER_ARM,
136         ARM_UPPER_ARM,
137         ARM_BASE_DEVIATION,
138         front_end_offset,
139         -ARM_BASE_DEVIATION
140     );
141     return check_constraints_ok(&in_angle, position)
142         ? EXIT_SUCCESS
143         : EXIT_FAILURE;
144 }
```

Listing C.32: pArm-S2/pArm.X/motor/kinematics.c

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: rsa.c
22 * Author: Javinator9889
23 * Comments: RSA file with necessary functions
24 * Revision history: v1.1
25 */
26
27 ****
28 RSA Algorithm
29 Reminder: ((m**e)**d) % n = m
30 e: encryption, d: decryption
31 Encryption: ciphertext = message**e % n
32 Decryption: (c**d == (m**e)**d == m ) % n
33
34 RSA Key Generation
35 p and q, two distinct prime numbers
```

```
36 n = pq
37 fi is Euler's Totient Function
38 fi(n) = fi(p) * fi(q) = (p - 1) * (q - 1) = n - (p + q - 1)
39
40 chose e, the public key:
41   - 1 < e < fi(n)
42   - gcd(e, fi(n)) == 1 (i.e. e and fi(n) are coprime)
43 chose d, the private key:
44   - d == e**-1 (mod fi(n))
45   -> d is the modular multiplicative inverse of e (modulo(fi(n)))
46 ****
47
48 #include <stdint.h>
49 #include <stdbool.h>
50 #include <math.h>
51 #include "rand.h"
52 #include "zeros.h"
53 #include "rsa.h"
54 #include "primes.h"
55
56 static int_fast64_t gcd(int_fast64_t u, int_fast64_t v) {
57     int_fast64_t shift;
58     if (u == 0ULL)
59         return v;
60     if (v == 0ULL)
61         return u;
62
63     shift = ctz(u | v);
64     u >>= ctz(u);
65
66     do {
67         v >>= ctz(v);
68         if (u > v) {
69             int_fast64_t temp = v;
70             v = u;
71             u = temp;
72         }
73         v -= u;
74     } while (v != 0ULL);
75
76     return u << shift;
77 }
78
79 // Returns modulo inverse of a with respect
80 // to m using extended Euclid Algorithm
81 // Assumption: a and m are coprimes, i.e.,
82 // gcd(a, m) = 1
83 static int_fast64_t mod_inverse(int_fast64_t a, int_fast64_t b) {
84     int_fast64_t m0 = b;
85     int_fast64_t x = 1;
86     int_fast64_t y = 0;
87
88     if (b == 1)
89         return 0ULL;
90
91     while (a > 1) {
```

```
92     int_fast64_t quotient = (int_fast64_t) (a / b);
93     int_fast64_t temp = b;
94
95     b = a % b;
96     a = temp;
97     temp = y;
98
99     y = x - quotient * y;
100    x = temp;
101 }
102
103 if (x < 0)
104     x += m0;
105
106 return x;
107 }
108
109 static int_fast64_t right_to_left(
110     int_fast64_t value,
111     int_fast64_t exp,
112     int_fast64_t mod) {
113     int_fast64_t ret = 1ULL;
114
115     if (mod == 1ULL)
116         return 0ULL;
117
118     value %= mod;
119
120     while (exp > 0ULL) {
121         if (exp % 2 == 1) {
122             ret = (ret * value) % mod;
123         }
124         exp >>= 1;
125         value = (value * value) % mod;
126     }
127
128     return ret;
129 }
130
131 static bool is_valid_key(rsa_t *key) {
132     if (key->d <= 0LL) {
133         return false;
134     }
135
136     static const char *test_msg = "RSATEST";
137     static const int_fast8_t length = 8;
138     bool keys_ok = true;
139     int_fast64_t enc_msg[8] = {0LL};
140     for (int_fast8_t i = 0; i < length; ++i) {
141         enc_msg[i] = RSA_encrypt(test_msg[i], key);
142     }
143
144     for (int_fast8_t i = 0; (i < length) && (keys_ok == true); ++i) {
145         char dec_char = RSA_decrypt(enc_msg[i], key);
146         keys_ok = (dec_char == test_msg[i]);
147     }
```

```

148     return keys_ok;
149 }
150 }
151 }
152 rsa_t RSA_keygen() {
153     rsa_t ret = {0LL};
154     ret.e = 65537LL;
155     int_fast64_t p;
156     int_fast64_t q;
157     int_fast64_t n;
158     int_fast64_t phi;
159
160     do {
161         do {
162             do {
163                 do {
164                     p = RAND(MIN_PRIME_NUMBER, MAX_PRIME_NUMBER);
165                     } while (check_prime(p, 5) == false);
166                 do {
167                     q = RAND(MIN_PRIME_NUMBER, MAX_PRIME_NUMBER);
168                     } while (check_prime(q, 5) == false);
169                 } while (gcd(p, q) != 1ULL);
170                 n = p * q;
171                 phi = (p - 1ULL) * (q - 1ULL);
172             } while (gcd(ret.e, phi) != 1ULL);
173             ret.n = n;
174             ret.phi = phi;
175             ret.d = mod_inverse(ret.e, ret.phi);
176         } while (!is_valid_key(&ret));
177
178     return ret;
179 }
180
181 inline int_fast64_t RSA_encrypt(int_fast64_t msg, rsa_t *key) {
182     return right_to_left(msg, key->e, key->n);
183 }
184
185 inline int_fast64_t RSA_sign(int_fast64_t msg, rsa_t *key) {
186     return RSA_decrypt(msg, key);
187 }
188
189 inline int_fast64_t RSA_decrypt(int_fast64_t text, rsa_t *key) {
190     return right_to_left(text, key->d, key->n);
191 }

```

Listing C.33: pArm-S2/pArm.X/rsa/rsa.c

```

1 /*
2  * 2020 | pArm-S2 by Javinator9889
3  *
4  * This program is free software: you can redistribute it and/or modify
5  * it under the terms of the GNU General Public License as published by
6  * the Free Software Foundation, either version 3 of the License, or
7  * (at your option) any later version.
8  *
9  * This program is distributed in the hope that it will be useful,

```

```
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: rand.c
22 * Author: Javinator9889
23 * Comments: Generate random numbers using the elapsed time in ns.
24 * Revision history: v1.0
25 */
26
27 #include <xc.h>
28 #include <stdint.h>
29 #include <stdbool.h>
30 #include <p33EP512GM604.h>
31 #include "rand.h"
32
33 static volatile time_t _ns;
34 // Package utils/time.h
35 extern time_t TIME_now_us(void);
36 static bool _rand_init = false;
37
38 void RAND_init(void) {
39     _ns = 0ULL;
40     //TMR6 0;
41     TMR6 = 0x00;
42     //Period = 0.00000217 s; Frequency = 59904000 Hz; PR6 12;
43     PR6 = 0x0C;
44     //TCKPS 1:1; T32 16 Bit; TON enabled; TSDL disabled; TCS FOSC/2; TGATE disabled;
45     T6CON = 0x8000;
46     // Enable interrupt
47     IFS2bits.T6IF = 0;
48     IEC2bits.T6IE = 1;
49 }
50
51 void RAND_init_seed(void) {
52     srand(_ns);
53     _rand_init = true;
54 }
55
56 void RAND_stop(void) {
57     T6CONbits.TON = 0;
58     _ns = 0ULL;
59     _rand_init = false;
60 }
61
62 int_fast64_t RAND(int_fast64_t min, int_fast64_t max) {
63     if (! _rand_init)
64         return 0ULL;
65     return (int_fast64_t) (min + ((int_fast64_t) rand()) /
```

```

66         (((int_fast64_t) RAND_MAX) / (max - min + 1LL) + 1LL));
67 }
68
69 inline int RAND_random(void) {
70     if (!_rand_init)
71         return 0;
72     return rand();
73 }
74
75 void __attribute__((interrupt, no_auto_psv)) _T6Interrupt(void) {
76     _ns += 217ULL;
77     IFS2bits.T6IF = 0;
78 }
```

Listing C.34: pArm-S2/pArm.X/rsa/rand.c

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: zeros.c
22 * Author: Javinator9889
23 * Comments: Utils for counting both trailing and leading zeros.
24 * Revision history: v1.0
25 */
26 #include <stdint.h>
27 #include "zeros.h"
28
29 const unsigned char clz_8b[256] = {
30     8,
31     7,
32     REPEAT_2x(6),
33     REPEAT_4x(5),
34     REPEAT_8x(4),
35     REPEAT_16x(3),
36     REPEAT_32x(2),
37     REPEAT_64x(1),
38     REPEAT_128x(0)
39 };
40
```

```

41 int_fast64_t __clz(int_fast64_t x) {
42     int_fast64_t b, ms_oct;
43
44     if (x & 0xFFFFFFFF00000000U)
45         if (x & 0xFFFF000000000000U)
46             if (x & 0xFF00000000000000U)
47                 b = 0, ms_oct = x >> 56;
48             else
49                 b = 8, ms_oct = x >> 48;
50         else
51             if (x & 0x0000FF0000000000U)
52                 b = 16, ms_oct = x >> 40;
53             else
54                 b = 24, ms_oct = x >> 32;
55         else
56             if (x & 0x00000000FFFF0000U)
57                 if (x & 0x00000000FF000000U)
58                     b = 32, ms_oct = x >> 24;
59                 else
60                     b = 40, ms_oct = x >> 16;
61             else
62                 if (x & 0x000000000000FF00U)
63                     b = 48, ms_oct = x >> 8;
64             else
65                 b = 56, ms_oct = x >> 0;
66
67     return b + __clz_8b[ms_oct];
68 }
69
70 int_fast64_t __ctz(int_fast64_t x) {
71     x &= -x;
72     return 63LL - __clz(x);
73 }
```

Listing C.35: pArm-S2/pArm.X/rsa/zeros.c

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 #include <stdint.h>
```

```
21 #include <stdbool.h>
22 #include "primes.h"
23 #include "rand.h"
24
25 /*
26 * calculates (a * b) % c taking into account that a * b might overflow
27 */
28 static int_fast64_t mulmod(int_fast64_t a, int_fast64_t b, int_fast64_t mod) {
29     int_fast64_t x = 0;
30     int_fast64_t y = a % mod;
31
32     while (b > 0) {
33         if (b % 2 == 1) {
34             x = (x + y) % mod;
35         }
36         y = (y * 2) % mod;
37         b /= 2;
38     }
39     return x % mod;
40 }
41
42 /*
43 * modular exponentiation
44 */
45 static int_fast64_t modulus(int_fast64_t b, int_fast64_t exp, int_fast64_t mod) {
46     int_fast64_t x = 1;
47     int_fast64_t y = b;
48     while (exp > 0) {
49         if (exp % 2 == 1)
50             x = (x * y) % mod;
51         y = (y * y) % mod;
52         exp /= 2;
53     }
54     return x % mod;
55 }
56
57 /*
58 * Miller-Rabin Primality test, trials signifies the accuracy
59 */
60 bool check_prime(int_fast64_t p, uint_fast16_t trials) {
61     uint_fast16_t i;
62     int_fast64_t s;
63     if (p < 2) {
64         return false;
65     }
66     if (p != 2 && p % 2 == 0) {
67         return false;
68     }
69     s = p - 1;
70     while (s % 2 == 0) {
71         s /= 2;
72     }
73     for (i = 0UL; i < trials; i += 1UL) {
74         int_fast64_t a = rand() % (p - 1) + 1, temp = s;
75         int_fast64_t mod = modulus(a, temp, p);
76         while (temp != p - 1 && mod != 1 && mod != p - 1) {
```

```

77         mod = mulmod(mod, mod, p);
78         temp *= 2;
79     }
80     if ((mod != p - 1) && temp % 2 == 0) {
81         return false;
82     }
83 }
84 return true;
85 }
```

Listing C.36: pArm-S2/pArm.X/rsa/primes.c

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: mutex.c
22 * Author: Javinator9889
23 * Comments: Ensures mutual exclusion when accessing a critical section.
24 * Revision history: v1.0
25 */
26 #include <stdbool.h>
27 #include <xc.h>
28 #include "mutex.h"
29
30 static inline void mutex_switch(bool state) {
31     IEC0bits.T3IE = state;
32     IEC1bits.T4IE = state;
33     IEC1bits.T5IE = state;
34 }
35
36 static unsigned char test_and_set(mutex_t *lock_ptr) {
37     mut_t old_value;
38     // Disable interrupts
39     mutex_switch(false);
40     old_value = *lock_ptr;
41     *lock_ptr = LOCKED;
42     // Enable interrupts
43     mutex_switch(true);
44     return old_value;
```

```

45 }
46
47 void mutex_acquire(mutex_t *lock) {
48     while (test_and_set(lock) == LOCKED);
49 }
50
51 void mutex_release(mutex_t *lock) {
52     if (test_and_set(lock) == LOCKED)
53         *lock = UNLOCKED;
54 }
```

Listing C.37: pArm-S2/pArm.X/sync/mutex.c

```

1 /*
2  * 2020 | pArm-S2 by Javinator9889
3  *
4  * This program is free software: you can redistribute it and/or modify
5  * it under the terms of the GNU General Public License as published by
6  * the Free Software Foundation, either version 3 of the License, or
7  * (at your option) any later version.
8  *
9  * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: barrier.c
22 * Author: Javinator9889
23 * Comments: Synchronization mechanism using a barrier and mutex.
24 * Revision history: v1.0
25 */
26
27 #include <stdint.h>
28 #include <stdbool.h>
29 #include <stdlib.h>
30 #include "mutex.h"
31 #include "barrier.h"
32
33 barrier_t *BARRIER_create(uint16_t total) {
34     barrier_t *created_barrier = (barrier_t *) malloc(sizeof(barrier_t));
35     BARRIER_clr(created_barrier);
36     created_barrier->total = total;
37     return created_barrier;
38 }
39
40 void BARRIER_arrive(volatile barrier_t *barrier) {
41     mutex_acquire(&barrier->lock);
42     if (++barrier->counter >= barrier->total)
43         barrier->flag = true;
```

```

44     mutex_release(&barrier->lock);
45 }
46
47 void BARRIER_set_total(volatile barrier_t *barrier, uint16_t p) {
48     mutex_acquire(&barrier->lock);
49     barrier->total = p;
50     mutex_release(&barrier->lock);
51 }
52
53 void BARRIER_clr(volatile barrier_t *barrier) {
54     barrier->counter = 0;
55     barrier->flag = false;
56     barrier->lock = UNLOCKED;
57 }
58
59 void BARRIER_set_done(volatile barrier_t *barrier) {
60     mutex_acquire(&barrier->lock);
61     barrier->flag = true;
62     mutex_release(&barrier->lock);
63 }
64
65 bool BARRIER_all_done(volatile barrier_t *barrier) {
66     return barrier->flag;
67 }

```

Listing C.38: pArm-S2/pArm.X/sync/barrier.c

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: tmr3.c
22 * Author: javinator9889
23 *
24 * Created on 25 de agosto, 2020
25 */
26
27 #include <p33EP512GM604.h>
28 #include "tmr3.h"
29 #include "../utils/types.h"

```

```
30 #include "../motor/motor.h"
31 #include "../sync/barrier.h"
32
33 motor_t *TMR3_motor;
34 volatile barrier_t *TMR3_barrier;
35 static volatile int_fast32_t TMR3_count;
36 static volatile int_fast32_t duration;
37
38 void TMR3_Initialize(motor_t *motor, volatile barrier_t *barrier) {
39     TMR3_motor = motor;
40     TMR3_barrier = barrier;
41     TMR3_count = .0F;
42
43     //TMR3 0;
44     TMR3 = 0x00;
45     //Period = 1 us; Frequency = 59904000 Hz; PR3 59903;
46     PR3 = 0x3B;
47     //TCKPS 1:1; TON disabled; TSDL disabled; TCS FOSC/2; TGATE disabled;
48     T3CON = 0x0;
49
50     IFS0bits.T3IF = 0;
51
52     IEC0bits.T3IE = 0;
53 }
54
55 void __attribute__((interrupt, no_auto_psv)) _T3Interrupt(void) {
56     TMR3_count += 1L;
57
58     if (TMR3_count >= duration)
59         TMR3_Stop();
60
61     IFS0bits.T3IF = 0;
62 }
63
64 void TMR3_Start(void) {
65     /* Clear old value*/
66     TMR3_count = .0F;
67     duration = (int_fast32_t) TMR3_motor->movement_duration;
68
69     /*Enable the interrupt*/
70     IEC0bits.T3IE = 1;
71
72     IFS0bits.T3IF = 0;
73
74     /* Start the Timer */
75     T3CONbits.TON = 1;
76 }
77
78 void TMR3_Stop(void) {
79     TMR3_motor->movement_finished = true;
80     BARRIER_arrive(TMR3_barrier);
81     // If movement is clockwise then add the count to current angle_us
82     // else, the count must be substracted
83     TMR3_motor->angle_us += (TMR3_motor->clockwise * TMR3_count);
84     /* Stop the Timer */
85     T3CONbits.TON = 0;
```

```
86
87     /*Disable the interrupt*/
88     IEC0bits.T3IE = 0;
89 }
```

Listing C.39: pArm-S2/pArm.X/timers/tmr3.c

```
1 /*
2  * 2020 | pArm-S2 by Javinator9889
3  *
4  * This program is free software: you can redistribute it and/or modify
5  * it under the terms of the GNU General Public License as published by
6  * the Free Software Foundation, either version 3 of the License, or
7  * (at your option) any later version.
8  *
9  * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: tmr4.c
22 * Author: javinator9889
23 *
24 * Created on 25 de agosto, 2020
25 */
26
27 #include <p33EP512GM604.h>
28 #include "tmr4.h"
29 #include "../utils/types.h"
30 #include "../motor/motor.h"
31 #include "../sync/barrier.h"
32
33 motor_t *TMR4_motor;
34 volatile barrier_t *TMR4_barrier;
35 static volatile int_fast32_t TMR4_count;
36 static volatile int_fast32_t duration;
37
38 void TMR4_Initialize(motor_t *motor, volatile barrier_t *barrier) {
39     TMR4_motor = motor;
40     TMR4_barrier = barrier;
41     TMR4_count = .0F;
42
43     //TMR4 0;
44     TMR4 = 0x00;
45     //Period = 1 us; Frequency = 59904000 Hz; PR4 59903;
46     PR4 = 0x3B;
47     //TCKPS 1:1; T32 16 Bit; TON enabled; TSIDL disabled; TCS FOSC/2; TGATE disabled;
48     T4CON = 0x0;
49 }
```

```

50     IFS1bits.T4IF = 0;
51     IEC1bits.T4IE = 0;
52 }
53
54 void __attribute__((interrupt, no_auto_psv)) _T4Interrupt(void) {
55     TMR4_count += 1L;
56
57     if (TMR4_count >= duration)
58         TMR4_Stop();
59
60     IFS1bits.T4IF = 0;
61 }
62
63 void TMR4_Start(void) {
64     /* Clear old value*/
65     TMR4_count = .0F;
66
67     /*Enable the interrupt*/
68     IEC1bits.T4IE = 1;
69
70     /* Start the Timer */
71     T4CONbits.TON = 1;
72 }
73
74 void TMR4_Stop(void) {
75     TMR4_motor->movement_finished = true;
76     BARRIER_arrive(TMR4_barrier);
77     // If movement is clockwise then add the count to current angle_us
78     // else, the count must be substracted
79     TMR4_motor->angle_us += (TMR4_motor->clockwise * TMR4_count);
80     /* Stop the Timer */
81     T4CONbits.TON = 0;
82
83     /*Disable the interrupt*/
84     IEC1bits.T4IE = 0;
85 }

```

Listing C.40: pArm-S2/pArm.X/timers/tmr4.c

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.

```

```
18 */
19 /*
20 * File: tmr5.c
21 * Author: javinator9889
22 *
23 * Created on 25 de agosto, 2020
24 */
25
26
27 #include <p33EP512GM604.h>
28 #include "tmr5.h"
29 #include "../utils/types.h"
30 #include "../motor/motor.h"
31 #include "../sync/barrier.h"
32
33 motor_t *TMR5_motor;
34 volatile barrier_t *TMR5_barrier;
35 static volatile int_fast32_t TMR5_count;
36 static volatile int_fast32_t duration;
37
38 void TMR5_Initialize(motor_t *motor, volatile barrier_t *barrier) {
39     TMR5_motor = motor;
40     TMR5_barrier = barrier;
41     TMR5_count = .0F;
42
43     //TMR5 0;
44     TMR5 = 0x00;
45     //Period = 1 us; Frequency = 59904000 Hz; PR5 59903;
46     PR5 = 0x3B;
47     //TCKPS 1:1; TON enabled; TSIDL disabled; TCS FOSC/2; TGATE disabled;
48     T5CON = 0x0;
49
50     IFS1bits.T5IF = 0;
51     IEC1bits.T5IE = 0;
52 }
53
54 void __attribute__((interrupt, no_auto_psv)) _T5Interrupt(void) {
55     TMR5_count += 1L;
56
57     if (TMR5_count >= duration)
58         TMR5_Stop();
59
60     IFS1bits.T5IF = 0;
61 }
62
63 void TMR5_Start(void) {
64     /* Clear old value*/
65     TMR5_count = .0F;
66
67     /*Enable the interrupt*/
68     IEC1bits.T5IE = 1;
69
70     /* Start the Timer */
71     T5CONbits.TON = 1;
72 }
73 }
```

```

74 void TMR5_Stop(void) {
75     TMR5_motor->movement_finished = true;
76     BARRIER_arrive(TMR5_barrier);
77     // If movement is clockwise then add the count to current angle_us
78     // else, the count must be substracted
79     TMR5_motor->angle_us += (TMR5_motor->clockwise * TMR5_count);
80     /* Stop the Timer */
81     T5CONbits.TON = 0;
82
83     /*Disable the interrupt*/
84     IEC1bits.T5IE = 0;
85 }
```

Listing C.41: pArm-S2/pArm.X/timers/tmr5.c

```

1 /*
2  * 2020 | pArm-S2 by Javinator9889
3  *
4  * This program is free software: you can redistribute it and/or modify
5  * it under the terms of the GNU General Public License as published by
6  * the Free Software Foundation, either version 3 of the License, or
7  * (at your option) any later version.
8  *
9  * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: buffer.c
22 * Author: Javinator9889
23 * Comments: Buffer handler and shortcuts for handling buffer_t
24 * Revision history: v1.0
25 */
26 #include <stdlib.h>
27 #ifdef DEBUG_ENABLED
28 #include "../printf/io.h"
29 #endif
30 #include "buffer.h"
31 #include "types.h"
32
33 buffer_t *BUFFER_create(size_t size) {
34     buffer_t *ptr = (buffer_t *) malloc(sizeof(buffer_t));
35     if (ptr == NULL) {
36 #ifdef DEBUG_ENABLED
37         printf("[ERROR]\tFailed to allocate struct buffer_t\n");
38 #endif
39         return NULL;
40     }
41     ptr->buffer = NULL;
```

```

42     BUFFER_update_size(ptr, size);
43     return ptr;
44 }
45
46 void BUFFER_update_size(buffer_t *buffer, size_t size) {
47     if (buffer->size != size) {
48         buffer->size = size;
49         buffer->bsize = (size * sizeof (char));
50         buffer->buffer = (char *) realloc(buffer->buffer, buffer->bsize);
51 #ifdef DEBUG_ENABLED
52     if (buffer->buffer == NULL && size != 0U) {
53         printf("[ERROR]\tFailed to re-allocate %dB for new buffer!\n", buffer->bsize);
54     }
55 #endif
56 }
57 }
58
59 inline void BUFFER_free(buffer_t *buffer) {
60     return BUFFER_update_size(buffer, 0U);
61 }
```

Listing C.42: pArm-S2/pArm.X/utils/buffer.c

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 #include "time.h"
21 #include "types.h"
22 #include "../init.h"
23
24 static time_t _now_us = 0ULL;
25 static time_t _now_ms = 0ULL;
26
27 inline void TIME_updateMs(void) {
28     _now_ms = (time_t) (_now_us / 1000ULL);
29 }
30
31 void TIME_init(void) {
32     TIME_set_time(0ULL);
33     TMR1_Initialize();
```

```

34     TMR2_Initialize();
35 }
36
37 time_t TIME_now(void) {
38     return _now_ms;
39 }
40
41 time_t TIME_now_us(void) {
42     return _now_us;
43 }
44
45 void TIME_set_time(time_t value_us) {
46     _now_us = value_us;
47     TIME_updateMs();
48 }
49
50 void __attribute__((__interrupt__, no_auto_psv)) _T1Interrupt(void) {
51     _now_us += 1ULL;
52     // Clear Timer1 interrupt
53     IFS0bits.T1IF = 0;
54 }
55
56 void __attribute__((__interrupt__, no_auto_psv)) _T2Interrupt(void) {
57     _now_ms += 1ULL;
58     // Clear Timer2 interrupt
59     IFS0bits.T2IF = 0;
60 }
```

Listing C.43: pArm-S2/pArm.X/utils/time.c

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 #include <xc.h>
21 #include "uart.h"
22
23 void putch(char character) {
24     while (!IFS0bits.U1TXIF);
25     U1TXREG = character;
26 }
```

```

27
28 void _putchar(char character) {
29     putch(character);
30 }
```

Listing C.44: pArm-S2/pArm.X/utils/uart.c

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: utils.c
22 * Author: Javinator9889
23 * Comments: Standard utils for using them along the project
24 * Revision history: 1.0
25 */
26 #include <math.h>
27 #include <stddef.h>
28 #include <stdint.h>
29 #include <libpic30.h>
30 #include <stdbool.h>
31 #include "utils.h"
32 #include "types.h"
33 #include "defs.h"
34
35 /**
36 * Maps a value in between the output range by the given input range
37 * @param x the value to map.
38 * @param in_min the minimum acceptable value.
39 * @param in_max the maximum acceptable value.
40 * @param out_min the minimum output value.
41 * @param out_max the maximum output value.
42 * @return the mapped 'x' value in between [out_min, out_max]
43 * @see https://www.arduino.cc/reference/en/language/functions/math/map/
44 */
45 inline long map(long x, long in_min, long in_max, long out_min, long out_max) {
46     return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
47 }
48
49 inline double64_t roundp(double64_t value) {
```

```

50     return floorl(value + .5F);
51 }
52
53 inline double64_t preciseMap(
54     double64_t value,
55     double64_t in_min,
56     double64_t in_max,
57     double64_t out_min,
58     double64_t out_max) {
59     double64_t slope = 1.0F * (out_max - out_min) / (in_max - in_min);
60     return out_min + roundp(slope * (value - in_min));
61 }
62
63 inline double64_t mapf(double64_t x, double64_t in_min, double64_t in_max, double64_t out_min,
64     double64_t out_max) {
65     return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
66 }
67
68 //##ifndef __isnan
69 bool __isnan(double64_t x) {
70     int_fast64_t hx;
71     EXTRACT_WORDS64(hx, x);
72     hx &= UINT64_C(0xFFFFFFFFFFFFFFFF);
73     hx = UINT64_C(0x7ff000000000000) - hx;
74     return (bool) (((uint64_t) hx) >> 63);
75 }
76 //##define isnan __isnan
77 //##endif
78
79 inline void delay_ms(time_t ms) {
80     return delay_us(ms * 1000ULL);
81 }
82
83 inline void delay_us(time_t us) {
84     __delay32(us * (FCY / 1000000ULL));
85 }
86 void cstrncpy(char source[], char dest[], uint16_t size) {
87     for (int i = (size - 1); i >= 0; i--) dest[i] = source[i];
88 }

```

Listing C.45: pArm-S2/pArm.X/utils/utils.c

```

1 /*
2  * 2020 | pArm-S2 by Javinator9889
3  *
4  * This program is free software: you can redistribute it and/or modify
5  * it under the terms of the GNU General Public License as published by
6  * the Free Software Foundation, either version 3 of the License, or
7  * (at your option) any later version.
8  *
9  * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 */

```

```
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 /*
21 * File: init.c
22 * Author: javinator9889
23 *
24 * Created on 3 de julio de 2020, 13:07
25 */
26
27 #include <p33EP512GM604.h>
28 #include "init.h"
29 #include "utils/utils.h"
30 #include "utils/defs.h"
31 #include "system_types.h"
32
33
34 void init_clock(void) {
35 #ifndef CONFIG_SIMULATOR
36     // FRCDIV FRC/1; PLLPRE 2; DOZE 1:8; PLLPOST 1:2; DOZEN disabled; ROI disabled;
37     CLKDIV = 0x3000;
38     // TUN Center frequency;
39     OSCTUN = 0x00;
40     // ROON disabled; ROSEL FOSC; RODIV 0; ROSSLP disabled;
41     REFOCON = 0x00;
42     // Setup de PLL for reaching 40 MHz with a 7.3728 clock.
43     // Maximum speed is of 140 MHz as the maximum temperature
44     // of 85 °C implies 70 MIPS.
45     //
46     // For working at ~120 MHz:
47     // F_osc = F_in * M / (N1 * N2)
48     // F_cy = F_osc / 2
49     // F_osc ~= 120 MHz -> F_osc = 7.3728 * 65 / (2 * 2) = 119.808 MHz
50     // F_cy = F_osc / 2 = 59.904 MHz
51     //
52     // Then, setup the PLL's prescaler, postcaler and divisor
53     PLLFBD = 0x3F;
54     // AD1MD enabled; PWMMMD enabled; T3MD enabled; T4MD enabled; T1MD enabled; U2MD enabled;
55     // T2MD enabled; U1MD enabled; QEI1MD enabled; SPI2MD enabled; SPI1MD enabled; C2MD enabled;
56     // C1MD enabled; DCIMD enabled; T5MD enabled; I2C1MD enabled;
57     PMD1 = 0x00;
58     // OC5MD enabled; OC6MD enabled; OC7MD enabled; OC8MD enabled; OC1MD enabled; IC2MD enabled;
59     // OC2MD enabled; IC1MD enabled; OC3MD enabled; OC4MD enabled; IC6MD enabled; IC7MD enabled;
60     // IC5MD enabled; IC8MD enabled; IC4MD enabled; IC3MD enabled;
61     PMD2 = 0x00;
62     // ADC2MD enabled; PMPMD enabled; U3MD enabled; QEI2MD enabled; RTCCMD enabled; CMPMD
63     enabled; T9MD enabled; T8MD enabled; CRCMD enabled; T7MD enabled; I2C2MD enabled; T6MD
64     enabled;
65     PMD3 = 0x00;
66     // U4MD enabled; CTMUMD enabled; REFOMD enabled;
67     PMD4 = 0x00;
68     // PWM2MD enabled; PWM1MD enabled; PWM4MD enabled; SPI3MD enabled; PWM3MD enabled; PWM6MD
69     enabled; PWM5MD enabled;
```

```
63 PMD6 = 0x00;
64 // PTGMD enabled; DMA0MD enabled;
65 PMD7 = 0x00;
66 // CF no clock failure; NOSC FRCPLL; CLKLOCK unlocked; OSWEN Switch is Complete; IOLOCK not
- active;
67 __builtin_write_OSCCONH((uint8_t) (0x01));
68 __builtin_write_OSCCONL((uint8_t) (0x01));
69
70 // Wait for Clock switch to occur
71 while (OSCCONbits.OSWEN != 0);
72 // And wait for clock switching to happen
73 // First, wait for clock switch to occur
74 // and then wait the PLL to lock
75 while (OSCCONbits.LOCK != 1);
76 #endif
77 }
78
79 void initUART(void) {
80 // Unlock the Peripheral Pin Selector (PPS)
81 // for allowing changes on TRIS ports without
82 // affecting expected device behavior.
83 // 0xBF is a shortcut for ~(1 << 6) == 191
84 #ifndef CONFIG_SIMULATOR
85 __builtin_write_OSCCONL(OSCCON & 0xBF); // unlock PPS
86 #endif
87 TRISCbits.TRISC7 = 1; // RC7 set as input
88 TRISCbits.TRISC6 = 0; // RC6 set as output
89
90 RPOR6bits.RP54R = 0b00001; // RC6->UART1:U1TX
91 RPINR18bits.U1RXR = 55;
92 // Lock again the PPS as we are done
93 // configuring the remappable ports.
94 // 0x40 is a shortcut for (1 << 6) == 64
95 #ifndef CONFIG_SIMULATOR
96 __builtin_write_OSCCONL(OSCCON | 0x40); // lock PPS
97 #endif
98
99 IEC0bits.U1TXIE = 0;
100 IEC0bits.U1RXIE = 0;
101
102 // Setup UART
103 // Stop on idle
104 U1MODEbits.USIDL = 1;
105 // Disable IrDA
106 U1MODEbits.IREN = 0;
107 // Use only TX and RX pins
108 // ignoring CTS, RTS and BCLK
109 U1MODEbits.UEN = 0b00;
110 // Do not wake-up with UART
111 U1MODEbits.WAKE = 0;
112 // Disable loopback mode
113 U1MODEbits.LPBACK = 0;
114 // Do not use automatic baudrate when receiving
115 U1MODEbits.ABAUD = 0;
116 // Disable polarity inversion. Idle state is '1'
117 U1MODEbits.URXINV = 0;
```

```
118 // Do not use high speed baudrate
119 U1MODEbits.BRGH = 0;
120 // 8 data bits without parity
121 U1MODEbits.PDSEL = 0b00;
122 // One stop bit
123 U1MODEbits.STSEL = 0;
124
125 // Calculate the baudrate using the following equation
126 // UxBRG = ((FCY / Desired Baud rate) / 16) - 1
127 // For 9600 bauds and FCY = 59.904E6, the obtained BRG is
128 // -> 389, and the obtained baudrate is: 9600, with an error
129 // of 0%
130 U1BRG = 0x185;
131
132 // Interrupt after one RX character is received;
133 // UTXISEL0 TX_ONE_CHAR; UTXINV disabled; OERR NO_ERROR_cleared; URXISEL RX_ONE_CHAR;
134 UTXBRK COMPLETED; UTXEN enabled; ADDEN disabled;
135 U1STA = 0x400;
136
137 // Enable UART TX Interrupt
138 IEC0bits.U1TXIE = 1;
139 IEC0bits.U1RXIE = 1;
140 IEC4bits.U1EIF = 1;
141 IFS0bits.U1RXIF = 0;
142 IFS0bits.U1TXIF = 0;
143 IFS4bits.U1EIF = 0;
144 IPC2bits.U1RXIP = 0b110;
145
146 //Make sure to set LAT bit corresponding to TxPin as high before UART initialization
147 LATCbits.LATC7 = 1;
148 LATCbits.LATC6 = 1;
149 U1MODEbits.UARTEN = 1; // enabling UART ON bit
150 U1STAbits.UTXEN = 1;
151
152 // Wait 105 uS (when baudrate is 9600) for a first
153 // transmission bit to be sent and detected, so then
154 // the UART can be used
155 DELAY_105uS;
156 }
157 void initPWM(void) {
158 TRISBbits.TRISB11 = 0; // PWM3L
159 TRISBbits.TRISB13 = 0; // PWM2L
160 TRISBbits.TRISB15 = 0; // PWM1L
161 TRISAbits.TRISA7 = 0; // PWM4L
162
163 PTCON2bits.PCLKDIV = 0b110; // Prescaler 1:32
164
165 // Setup PWM period - the motors have a
166 // minimum time in between pulses of 20ms,
167 // so the frequency must be of 50 Hz.
168 //
169 // F_osc = 119.808 MHz
170 // F_PWM = 50 Hz
171 // PWM_Prescaler = 64
172 // PTPER = F_osc / (F_PWM * PWM_Prescaler) --> PTPER = 119.808 MHz / (50 Hz * 32)
```

```
173 // = 37440 = PTPER --> F_PWM = 50.000 Hz
174 PTPER = 37440;
175
176 // Initialize independent time base to zero.
177 // As we are using PWMxL, we only use
178 // SPHASEx ports. If using PWMxH, just change
179 // SPHASEx to PHASEx ones.
180 SPHASE3 = 0;
181 SPHASE2 = 0;
182 SPHASE1 = 0;
183 SPHASE4 = 0;
184
185 // By default, set no duty cycle of programmed signals
186 SDC3 = 0;
187 SDC2 = 0;
188 SDC1 = 0;
189 SDC4 = 0;
190
191 // Disable Dead Time values
192 ALTDTR4 = 0;
193 ALTDTR3 = 0;
194 ALTDTR2 = 0;
195 ALTDTR1 = 0;
196
197 DTR4 = 0;
198 DTR3 = 0;
199 DTR2 = 0;
200 DTR1 = 0;
201
202 // True independent work mode, so then both PWMxH and
203 // PWMxL can be used independently
204 IOCON4bits.PMOD = 0b11;
205 IOCON3bits.PMOD = 0b11;
206 IOCON2bits.PMOD = 0b11;
207 IOCON1bits.PMOD = 0b11;
208
209 // Disable PWM fault input
210 FCLCON4bits.FLTMOD = 0b11;
211 FCLCON3bits.FLTMOD = 0b11;
212 FCLCON2bits.FLTMOD = 0b11;
213 FCLCON1bits.FLTMOD = 0b11;
214
215 // Do not swap LOW/HIGH values
216 IOCON4bits.SWAP = 0;
217 IOCON3bits.SWAP = 0;
218 IOCON2bits.SWAP = 0;
219 IOCON1bits.SWAP = 0;
220
221 // Set pins as PWM ones
222 IOCON4bits.PENL = 1;
223 IOCON3bits.PENL = 1;
224 IOCON2bits.PENL = 1;
225 IOCON1bits.PENL = 1;
226 // Disable high output as we are not using it
227 IOCON4bits.PENH = 0;
228 IOCON3bits.PENH = 0;
```

```
229 IOCON2bits.PENH = 0;
230 IOCON6bits.PENH = 0;
231 IOCON6bits.PENL = 0;
232 IOCON6bits.OVRENH = 1;
233 IOCON6bits.OVRENH = 1;
234
235 // Set PWM configurations to zero by default
236 PWMCON6 = 0;
237 PWMCON5 = 0;
238 PWMCON4 = 0;
239 PWMCON3 = 0;
240 PWMCON2 = 0;
241 PWMCON1 = 0;
242
243 // Disable dead time in-between output switches
244 PWMCON4bits.DTC = 0b10;
245 PWMCON3bits.DTC = 0b10;
246 PWMCON2bits.DTC = 0b10;
247 PWMCON1bits.DTC = 0b10;
248
249 // and enable the PWM module
250 PTCONbits.PTEN = 1;
251 }
252
253 void TMR1_Initialize(void) {
254     TMR1 = 0x00;
255     // Period = 1 us;
256     // Frequency = 59904000 Hz;
257     // PR1 = 59 == an interrupt ~= 1.0016 us
258     PR1 = 0x3B;
259
260     // TON enabled; ->      1
261     // Empty bit ->        0
262     // TSIDL disabled; ->  0
263
264     // Empty bit ->        0
265     // Empty bit ->        0
266     // Empty bit ->        0
267     // Empty bit ->        0
268
269     // Empty bit ->        0
270     // Empty bit ->        0
271     // TGATE disabled; ->  0
272     // TCKPS 1:1; ->       0
273
274     // Empty bit ->        0
275     // TSYNC enabled ->    0
276     // TCS FOSC/2; ->     0
277     // Empty bit ->        0
278     // 1000 0000 0000 0000 == 0x8000
279     T1CON = 0x8000;
280
281     // Clear interrupt flag,
282     IFS0bits.T1IF = 0;
283     // set high priority
284     IPC0bits.T1IP = 5;
```

```
285 // and enable interrupts
286 IEC0bits.T1IE = 1;
287 }
288
289 void TMR2_Initialize(void) {
290 // Reset TMR2 to zero
291 TMR2 = 0x00;
292
293 // Period = 1 ms;
294 // Frequency = 59904000 Hz;
295 // PR2 59903 == an interrupt each millisecond.
296 PR2 = 0xE9FF;
297 // TON enabled; -> 1
298 // Empty bit -> 0
299 // TSIDL disabled; -> 0
300
301 // Empty bit -> 0
302 // Empty bit -> 0
303 // Empty bit -> 0
304 // Empty bit -> 0
305
306 // Empty bit -> 0
307 // Empty bit -> 0
308 // TGATE disabled; -> 0
309 // TCKPS 1:1; -> 0
310
311 // T32 16 Bit; -> 0
312 // Empty bit -> 0
313 // TCS FOSC/2; -> 0
314 // Empty bit -> 0
315 // 1000 0000 0000 0000 == 0x8000
316 T2CON = 0x8000;
317
318 // Clear interrupt flag...
319 IFS0bits.T2IF = 0;
320 // set high priority
321 IPC1bits.T2IP = 5;
322 // and enable TMR2 interruptions
323 IEC0bits.T2IE = 1;
324 }
325
326 void init_ports(void)
327 {
328 //Digital Ports for micro-interruptors, set as input
329 TRISAbits.TRISA0 = 1;
330 TRISAbits.TRISA1 = 1;
331 TRISBbits.TRISB0 = 1;
332 TRISBbits.TRISB1 = 1;
333
334 //Input Change Notification Interrupt configuration
335 _CNIP = 5; // priority (7 = highest)
336 _CNIE = 1; // Enable CN interrupts
337 _CNIF = 0; // Interrupt flag cleared
338 CNENBbits.CNIEB0 = 1;
339 CNENBbits.CNIEB1 = 1;
340 CNENAbits.CNIEA0 = 1;
```

```

341     CNENAbits.CNIEA1 = 1;
342
343
344     //Digital Ports for LED lights, set as output.
345     TRISBbits.TRISB5 = 0;
346     TRISBbits.TRISB6 = 0;
347     TRISBbits.TRISB7 = 0;
348     TRISBbits.TRISB8 = 0;
349
350     //Set I/O ports to digital, clear the analogic enable bit.
351     ANSELAbits.ANSA0 = 0;
352     ANSELAbits.ANSA1 = 0;
353     ANSELBbits.ANSB0 = 0;
354     ANSELBbits.ANSB1 = 0;
355     ANSELBbits.ANSB7 = 0;
356     ANSELBbits.ANSB8 = 0;
357 }
358
359 inline void system_initialize(void) {
360     INTERRUPT_GlobalDisable();
361     init_clock();
362     init_ports();
363     initPWM();
364     initUART();
365     SYSTEM_CORCONModeOperatingSet(CORCON_MODE_PORVALUES);
366     INTERRUPT_GlobalEnable();
367 }
```

Listing C.46: pArm-S2/pArm.X/init.c

```

1 /*
2 * 2020 | pArm-S2 by Javinator9889
3 *
4 * This program is free software: you can redistribute it and/or modify
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20 #include <stdbool.h>
21 #include <stdlib.h>
22 #include <string.h>
23 #include <stddef.h>
24 #include <stdint.h>
25 #include "printf/io.h"
26 #include "interrupts.h"
```

```
27 #include "motor/servo.h"
28 #include "utils/utils.h"
29 #include "utils/time.h"
30 #include "utils/uart.h"
31 #include "utils/types.h"
32 #include "utils/buffer.h"
33
34 volatile time_t _ns = 0ULL;
35 static char uart_buffer[1024] = {0};
36 static uint16_t uart_chars = 0U;
37 static volatile order_t *urx_order = NULL;
38 #ifdef LIMIT_SWITCH_ENABLED
39 static volatile uint_fast8_t *limit_switch_map = NULL;
40#endif
41
42 void U1RX_Init(volatile order_t *order) {
43     urx_order = order;
44 }
45
46 #ifdef LIMIT_SWITCH_ENABLED
47 void CN_Init(volatile uint_fast8_t *switch_map) {
48     limit_switch_map = switch_map;
49 }
50#endif
51
52 void __attribute__((__interrupt__, no_auto_psv)) _U1TXInterrupt(void) {
53     PORTBbits.RB6 = U1TXREG;
54     IFS0bits.U1TXIF = 0; // Clear TX Interrupt flag
55 }
56
57 void __attribute__((__interrupt__, no_auto_psv)) _U1RXInterrupt(void) {
58     IFS0bits.U1RXIF = 0;
59     if (U1STAbits.FERR == 1)
60         return;
61     if (U1STAbits.URXDA == 1) {
62         char received_val = U1RXREG;
63         PORTBbits.RB5 = received_val;
64 #ifdef CLI_MODE
65         printf("%c", received_val);
66#endif
67         if (received_val == '\n' || received_val == '\r') {
68             if (urx_order == NULL) {
69 #ifdef DEBUG_ENABLED
70                 printf("[DEBUG]\tU1RX not initialized!\n");
71#endif
72             }
73         }
74 #ifdef CLI_MODE
75         printf("\n");
76#endif
77         uart_buffer[uart_chars++] = '\0';
78 #ifdef DEBUG_ENABLED
79         printf("[DEBUG]\tNew order received: %s\n", uart_buffer);
80#endif
81         if (urx_order->order_buffer == NULL) {
82             urx_order->order_buffer = BUFFER_create(uart_chars);
```

```

83 }
84     if (urx_order->order_buffer->size != uart_chars) {
85         BUFFER_update_size(urx_order->order_buffer, uart_chars);
86     }
87     if (urx_order->order_buffer->buffer == NULL) {
88 #ifdef DEBUG_ENABLED
89         printf("[ERROR]\tFailed to allocate %dB for order!\n",
90                (uart_chars * sizeof (char)));
91 #endif
92         BUFFER_free(urx_order->order_buffer);
93         uart_chars = 0U;
94         return;
95     }
96     strncpy(urx_order->order_buffer->buffer,
97             uart_buffer,
98             urx_order->order_buffer->size);
99     uart_chars = 0U;
100    urx_order->message_received = true;
101 } else {
102     uart_buffer[uart_chars++] = received_val;
103     if (uart_chars >= 1024) {
104         // UART buffer overflow...
105         // Release memory and ignore instruction
106         uart_chars = 0;
107         printf("J11\n");
108     }
109 }
110 }
111 }
112
113 void __attribute__((__interrupt__, no_auto_psv)) _U1ErrInterrupt(void) {
114     if ((U1STAbits.OERR == 1)) {
115         U1STAbits.OERR = 0;
116     }
117
118     IFS4bits.U1EIF = 0;
119 }
120
121 void __attribute__((__interrupt__, no_auto_psv)) _CNInterrupt(void) {
122 #ifdef LIMIT_SWITCH_ENABLED
123     if (limit_switch_map != NULL) {
124         limit_switch_map[0] = PORTAbits.RA0;
125         limit_switch_map[1] = PORTAbits.RA1;
126         limit_switch_map[2] = PORTBbits.RB0;
127         limit_switch_map[3] = PORTBbits.RB1;
128     }
129 #endif
130     _CNIF = 0; // Clear the interruption flag
131 }
```

Listing C.47: pArm-S2/pArm.X/interrupts.c

```

1 /*
2  * 2020 | pArm-S2 by Javinator9889
3  *
4  * This program is free software: you can redistribute it and/or modify
```

```
5 * it under the terms of the GNU General Public License as published by
6 * the Free Software Foundation, either version 3 of the License, or
7 * (at your option) any later version.
8 *
9 * This program is distributed in the hope that it will be useful,
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of
11 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
12 * GNU General Public License for more details.
13 *
14 * You should have received a copy of the GNU General Public License
15 * along with this program. If not, see https://www.gnu.org/licenses/.
16 *
17 * Created by Javinator9889 on 2020 - pArm-S1.
18 */
19
20/*
21 * File: main.c
22 * Author: javinator9889
23 *
24 * Created on 11 de junio de 2020, 11:16
25 */
26
27#include "utils/defs.h"
28#include "pragmas.h"
29#include <xc.h>
30#include <stdbool.h>
31#include <stddef.h>
32#include <stdlib.h>
33#include <math.h>
34#include <float.h>
35#include <p33EP512GM604.h>
36#include <libpic30.h>
37#include "printf/io.h"
38#include "utils/types.h"
39#include "utils/uart.h"
40#include "init.h"
41#include "interrupts.h"
42#include "utils/time.h"
43#include "rsa/rsa.h"
44#include "motor/motor.h"
45#include "arm/planner.h"
46#include "gcode/gcode.h"
47#include "rsa/rand.h"
48#include "sync/barrier.h"
49#include "utils/buffer.h"
50
51 /**
52 * Global program RSA key that will be used for securing communications.
53 * @type rsa_t
54 * @see rsa_t
55 */
56rsa_t *RSA_key = NULL;
57
58 /**
59 * Global order container for managing the received orders
60 * from UART.
```

```
61  *
62  * @see order_t
63  */
64 volatile order_t *order = NULL;
65
66 /**
67 * Custom handler defined to know when all the motors have finished their
68 * movement.
69 */
70 static double64_t motor_movement_finished_time = LDBL_MAX;
71
72 /**
73 * Custom handler to know when all the motors started moving.
74 */
75 static time_t movement_start_time = UINT64_MAX;
76
77 /**
78 * Global defined barrier using accross the entire program.
79 */
80 volatile barrier_t *barrier;
81
82 /**
83 * Own flag defined to know if there is any movement.
84 */
85 static volatile bool is_moving = false;
86
87 #ifdef LIMIT_SWITCH_ENABLED
88 /**
89 * Limit switch map used accross the entire application.
90 */
91 volatile uint_fast8_t limit_switch_map[4] = {0};
92 #endif
93
94 #ifdef CLI_MODE
95 /**
96 * Custom flag (used only in CLI_MODE) to show the cursor when a command ends.
97 */
98 static bool show_cursor = true;
99 #else
100
101 /**
102 * Last beat time - used to check if S1 is still alive.
103 */
104 static time_t last_beat = 0ULL;
105
106 /**
107 * Application's random message used for authoring the remote
108 * device. When the host changes this message is destroyed.
109 */
110 static int_fast64_t rnd_message;
111
112 /**
113 * Flag active when the handshake has been successful.
114 * If there is any kind of error during the message exchange
115 * the device will be marked as untrusted.
116 */
```

```
117 volatile bool trusted_device = false;
118 #endif
119
120 /**
121 * Does the device setup. Setups motors, UART and other configuration needed
122 * to allow the usage of the pArm. Can fail if any of the motors is not
123 * correctly detected. In addition, this method calibrates the motors.
124 */
125 void setup(void);
126
127 /**
128 * Heart of the application. The looping logic that is executed always.
129 */
130 void loop(void);
131
132 /**
133 * Checks the attached motors status.
134 *
135 * @return EXIT_SUCCESS if all motors are OK, else EXIT_FAILURE.
136 */
137 char check_motor_status(void);
138
139 #ifndef USE_MOTOR_TMRS
140 /**
141 * When not using motors timers, updates current motor position if moving.
142 *
143 * @param motor - the motor to be updated.
144 */
145 void update_motor_time(motor_t *motor);
146#endif
147
148 #ifndef CLI_MODE
149
150 /**
151 * Performs the handshake with S1. Called only if not in CLI_MODE and no
152 * trusted device.
153 */
154 void do_handshake(void);
155
156 /**
157 * Algorithm to update S1 status. If 5 missing beats (i.e.: 1 second passed)
158 * deauthenticates the device.
159 *
160 * @param encrypted_msg - the random message used when authenticating.
161 */
162 void beat(int_fast64_t encrypted_msg);
163#endif
164
165 int main(void) {
166     setup();
167     while (true) {
168         loop();
169     }
170     return 0;
171 }
```

```
173 inline void setup(void) {
174     // Initialize different system modules
175     system_initialize();
176 #ifdef DEBUG_ENABLED
177     printf("[SETUP]\tStarting system setup\n");
178 #endif
179     PORTBbits.RB5 = 1;
180     PORTBbits.RB6 = 1;
181     PORTBbits.RB7 = 1;
182     TIME_init();
183 #ifdef DEBUG_ENABLED
184     printf("[SETUP]\tTime set to 0. Starting count...\n");
185     printf("[SETUP]\tAllocating pointer to order\n");
186 #endif
187     order = (order_t *) malloc(sizeof(order_t));
188     if (order == NULL) {
189 #ifdef DEBUG_ENABLED
190         printf("[ERROR]\tFailed to initialize order_t!\n");
191 #endif
192         order_t ptr = {
193             false, NULL
194         };
195         order = &ptr;
196     }
197     order->message_received = false;
198     order->order_buffer = NULL;
199 #ifdef DEBUG_ENABLED
200     printf("[SETUP]\tInitializing UART RX\n");
201 #endif
202     U1RX_Init(order);
203 #ifdef LIMIT_SWITCH_ENABLED
204     CN_Init(limit_switch_map);
205 #endif
206
207 #ifdef DEBUG_ENABLED
208     printf("[SETUP]\tInitializing RAND seed\n");
209 #endif
210     // Initialize RAND seed before generating the new keys
211     RAND_init();
212     RAND_init_seed();
213
214 #ifdef DEBUG_ENABLED
215     printf("[SETUP]\tCreating barrier for motors\n");
216 #endif
217     barrier = BARRIER_create(MAX_MOTORS - 1);
218
219 #ifdef DEBUG_ENABLED
220     printf("[SETUP]\tChecking motor status...\n");
221 #endif
222     // Init the planner so the motors are available
223 #ifdef LIMIT_SWITCH_ENABLED
224     PLANNER_init(barrier, limit_switch_map);
225
226     // Calibrate the motors. If someone returns
227     // not OK, stop execution until rebooted
228     // and notify turning on an LED
```

```
229     if (check_motor_status() == EXIT_FAILURE) {  
230 #ifdef DEBUG_ENABLED  
231     printf("[SETUP]\tMotor failure!\n");  
232 #endif  
233     bool led_on = true;  
234     while (true) {  
235         // Switch on LEDs  
236         PORTBbits.RB5 = led_on;  
237         PORTBbits.RB6 = led_on;  
238         PORTBbits.RB7 = led_on;  
239         led_on = !led_on;  
240         // J2 stands for motor failure  
241         printf("J2\n");  
242         delay_ms(500);  
243     }  
244 }  
245 #else  
246     PLANNER_init(barrier);  
247 #endif  
248     // Move the motors to home position  
249     PLANNER_go_home();  
250  
251     PORTBbits.RB5 = 0;  
252     PORTBbits.RB6 = 0;  
253     PORTBbits.RB7 = 0;  
254 #ifdef DEBUG_ENABLED  
255     printf("[DEBUG]\tFinished setup!\n");  
256 #endif  
257 }  
258  
259 inline void loop(void) {  
260 #ifndef CLI_MODE  
261     if (!trusted_device) {  
262 #ifdef DEBUG_ENABLED  
263         printf("[DEBUG]\tDevice not trusted... Waiting I1\n");  
264 #endif  
265         do_handshake();  
266     }  
267     PORTBbits.RB7 = trusted_device;  
268 #else  
269     if (show_cursor) {  
270         printf("$> ");  
271         show_cursor = false;  
272     }  
273 #endif  
274     if (order->message_received) {  
275         order->message_received = false;  
276         GCODE_ret_t ret = GCODE_process_command(order);  
277         switch (ret.code) {  
278             // G0  
279             case 0:  
280             {  
281                 if (ret.is_err) {  
282                     // Coordinates missing for G0  
283                     printf("J8\n");  
284                 } else {  
285                 }  
286             }  
287         }  
288     }  
289 }
```

```
285         point_t *position = (point_t *) ret.gcode_ret_val;
286         double64_t expected_time = PLANNER_move_xyz(*position);
287 #ifdef DEBUG_ENABLED
288         printf("[DEBUG]\tMoving motors to x: %Lf, y: %Lf, z: %Lf\n",
289                 position->x, position->y, position->z);
290 #endif
291         free(position);
292         if (expected_time == -1.0F) {
293             // Out-of-range
294             printf("J4\n");
295         } else {
296             is_moving = true;
297             printf("J1 %lf\n", (expected_time / 1000.0F));
298             movement_start_time = TIME_now_us();
299             motor_movement_finished_time =
300                     movement_start_time + expected_time;
301         }
302     }
303     break;
304 }
305 // G1
306 case 1:
307 {
308     if (ret.is_err) {
309         // Coordinates missing for G1
310         printf("J9\n");
311     } else {
312         angle_t *angles = (angle_t *) ret.gcode_ret_val;
313         double64_t expected_time = PLANNER_move_angle(*angles);
314 #ifdef DEBUG_ENABLED
315         printf("[DEBUG]\tMoving motors to t0: %Lf, t1: %Lf, t2: %Lf\n",
316                 angles->theta0, angles->theta1, angles->theta2);
317 #endif
318         free(angles);
319         if (expected_time == -1.0F) {
320             // Out-of-range
321             printf("J4\n");
322         } else {
323             is_moving = true;
324             printf("J1 %lf\n", (expected_time / 1000.0F));
325             movement_start_time = TIME_now_us();
326             motor_movement_finished_time =
327                     movement_start_time + expected_time;
328         }
329     }
330     break;
331 }
332 // G28
333 case 28:
334 {
335     double64_t expected_time = PLANNER_go_home();
336     is_moving = true;
337     printf("J1 %lf\n", (expected_time / 1000.0F));
338     movement_start_time = TIME_now_us();
339     motor_movement_finished_time =
340         movement_start_time + expected_time;
```

```
341 #ifdef DEBUG_ENABLED
342     printf("[DEBUG]\tMoving motors to home position...\n");
343 #endif
344     break;
345 }
346 // M1
347 case 10:
348 {
349     if (!is_moving)
350         printf("J5\n");
351     else {
352         uint8_t ret = PLANNER_stop_moving();
353         if (ret == EXIT_SUCCESS)
354             printf("M1\n");
355         else // Motors not moving
356             printf("J5\n");
357         is_moving = false;
358     }
359     break;
360 }
361 // M114
362 case 1140:
363 {
364     point_t *position = PLANNER_get_position();
365     printf("G0 X%Lf Y%Lf Z%Lf\n",
366            position->x,
367            position->y,
368            position->z);
369     free(position);
370     break;
371 }
372 // M280
373 case 2800:
374 {
375     angle_t *position = PLANNER_get_angles();
376     printf("G1 X%Lf Y%Lf Z%Lf\n",
377            position->theta0,
378            position->theta1,
379            position->theta2);
380     free(position);
381     break;
382 }
383 #ifndef CLI_MODE
384     // I6
385     case 600:
386     {
387         char *ord_msg = (char *) ret.gcode_ret_val;
388         int_fast64_t encrypted_msg = atol(ord_msg);
389         int_fast64_t dec_msg = RSA_decrypt(encrypted_msg, RSA_key);
390         if (dec_msg == rnd_message) {
391             trusted_device = false;
392             *RSA_key = RSA_keygen();
393             rnd_message = 0LL;
394         } else {
395             printf("J10\n");
396         }
```

```
397         free(ord_msg);
398         break;
399     }
400     // I7
401     case 700:
402     {
403         char *ord_msg = (char *) ret.gcode_ret_val;
404         int_fast64_t encrypted_msg = atol(ord_msg);
405         beat(encrypted_msg);
406         free(ord_msg);
407         break;
408     }
409 #endif
410     // Invalid GCODE found
411     default:
412     {
413         printf("J3\n");
414         break;
415     }
416 }
417 BUFFER_free(order->order_buffer);
418 #ifdef CLI_MODE
419     show_cursor = true;
420 #endif
421 }
422 #ifndef USE_MOTOR_TMRS
423 if (is_moving) {
424     update_motor_time(motors.base_motor);
425     update_motor_time(motors.lower_arm);
426     update_motor_time(motors.upper_arm);
427     if (TIME_now_us() >= motor_movement_finished_time) {
428         printf("J21\n");
429         is_moving = false;
430     #ifdef CLI_MODE
431         show_cursor = true;
432     #endif
433     }
434 } else
435 if (is_moving && BARRIER_all_done(barrier)) {
436     // Notify all motors have finished their movement
437     printf("J21\n");
438     // and clear barrier interrupt flag
439     BARRIER_clr(barrier);
440     is_moving = false;
441 #ifdef CLI_MODE
442     show_cursor = true;
443 #endif
444 } #endif
445 #endif
446 #ifndef CLI_MODE
447 if (trusted_device) {
448     // If last beat happened at least 1 second ago
449     // untrust the device and send 'J6' for informing
450     if ((TIME_now() - last_beat) >= 1000ULL) {
451         printf("J10\n");
452         trusted_device = false;
```

```
453     *RSA_key = RSA_keygen();
454     PORTBbits.RB7 = 0;
455     rnd_message = 0LL;
456 }
457 }
458 #endif
459 }
460
461 inline char check_motor_status(void) {
462     char motor_status = MOTOR_calibrate(motors.base_motor);
463     if (motor_status == EXIT_FAILURE)
464         return motor_status;
465     motor_status = MOTOR_calibrate(motors.lower_arm);
466     if (motor_status == EXIT_FAILURE)
467         return motor_status;
468     return MOTOR_calibrate(motors.upper_arm);
469 }
470
471 #ifndef CLI_MODE
472
473 inline void do_handshake(void) {
474 #ifdef DEBUG_ENABLED
475     printf("[DEBUG]\tWaiting for handshake message...\n");
476 #endif
477     while (!order->message_received);
478     order->message_received = false;
479     GCODE_ret_t ret = GCODE_process_command(order);
480     switch (ret.code) {
481         // I1
482         case 100:
483         {
484             // Initialize the seed every time this function is called
485             RAND_init_seed();
486             if (RSA_key == NULL) {
487 #ifdef DEBUG_ENABLED
488                 printf("[DEBUG]\tGenerating new RSA keys...\n");
489 #endif
490                 rsa_t key = RSA_keygen();
491                 RSA_key = &key;
492             }
493             printf("I2 %lld\n", RSA_key->n);
494             printf("I3 %lld\n", RSA_key->e);
495             rnd_message = RAND(10007LL, 104729LL);
496             int_fast64_t signed_message = RSA_sign(rnd_message, RSA_key);
497             printf("I4 %lld\n", signed_message);
498             break;
499         }
500         // I5 with encrypted msg
501         case 500:
502         {
503             char *ord_msg = (char *) ret.gcode_ret_val;
504             int_fast64_t encrypted_msg = atol(ord_msg);
505             int_fast64_t msg = RSA_decrypt(encrypted_msg, RSA_key);
506             if (msg == rnd_message) {
507                 trusted_device = true;
508                 last_beat = TIME_now();
```

```
509         printf("I5\n");
510     } else {
511         printf("J6\n");
512         trusted_device = false;
513     }
514     free(ord_msg);
515     break;
516 }
517 default:
518 {
519     // Untrusted device
520     printf("J10\n");
521     break;
522 }
523 }
524 }
525 #endif
526
527 #ifndef USE_MOTOR_TMRS
528
529 void update_motor_time(motor_t *motor) {
530     if (!motor->movement_finished) {
531         motor->current_movement_count =
532             (TIME_now_us() - movement_start_time);
533         if (motor->current_movement_count >= motor->movement_duration) {
534             motor->movement_finished = true;
535             motor->angle_us = motor->current_movement_count;
536             motor->current_movement_count = 0ULL;
537         }
538     }
539 }
540 #endif
541
542 #ifndef CLI_MODE
543
544 inline void beat(int_fast64_t encrypted_msg) {
545     printf("BEAT!\n");
546     int_fast64_t msg = RSA_decrypt(encrypted_msg, RSA_key);
547     if (msg == rnd_message) {
548         last_beat = TIME_now();
549         PORTBbits.RB5 = 1;
550         PORTBbits.RB6 = 1;
551         PORTBbits.RB7 = 1;
552     }
553 }
554 #endif
```

Listing C.48: pArm-S2/pArm.X/main.c

Anexo D

Matriz pseudo-inversa cuando $|J(\dot{q})| = 0$

La matriz pseudo-inversa se puede ver desde el siguiente enlace: <https://raw.githubusercontent.com/pArm-TFG/Memoria/master/pictures/equation.svg>



Anexo E

Código fuente S1

```

1 #                                     pArm-S1
2 #
3 # This program is free software: you can redistribute it and/or modify
4 # it under the terms of the GNU General Public License as published by
5 # the Free Software Foundation, either version 3 of the License, or
6 # (at your option) any later version.
7 #
8 # This program is distributed in the hope that it will be useful,
9 # but WITHOUT ANY WARRANTY; without even the implied warranty of
10 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
11 # GNU General Public License for more details.
12 #
13 # You should have received a copy of the GNU General Public License
14 # along with this program. If not, see <http://www.gnu.org/licenses/>.
15 import serial
16 from typing import Optional
17 from threading import Lock
18 import logging
19
20 log = logging.getLogger("Roger")
21
22
23 class Connection:
24     __instance = None
25
26     def __new__(cls, *args, **kwargs):
27         """
28             Creates a singleton instance that handles all the connections
29             with the UART device.
30             :param args: arbitrary arguments used while creating the class.
31             :param kwargs: arbitrary keyword arguments used while creating the
32             class.
33         """
34
35         if not Connection.__instance:
36             Connection.__instance = object.__new__(cls)
37             Connection.__instance.__must_init__ = True
38         else:
39             Connection.__instance.__must_init__ = False

```

```
40         return Connection.__instance
41
42     def __init__(self,
43                  port: str = "/dev/ttyUSB0",
44                  baudrate: int = 9600,
45                  should_open: bool = False):
46         """
47         Sets port and a baudrate for a serial connection. Also opens the port
48         with the current configuration.
49
50         :param port: the port used for serial communication
51         :param baudrate: the baudrate of the serial connection
52         :param should_open: if true, the port is opened
53         """
54         if self.__must_init:
55             self.ser = serial.Serial()
56             self.ser.port = port
57             self.ser.baudrate = baudrate
58             self.ser.bytesize = 8
59             self.ser.parity = serial.PARITY_NONE
60             self.ser.stopbits = 1
61             self.ser.timeout = 3
62             self.lock = Lock()
63             self._port = port
64             if should_open:
65                 self.ser.open()
66
67     def __enter__(self):
68         """
69         If not open, it opens the serial port.
70         """
71         if not self.ser.is_open:
72             self.ser.open()
73         return self
74
75     def __exit__(self, exc_type, exc_val, exc_tb):
76         """
77         Closes current port
78         """
79         self.ser.close()
80
81     def write(self, data: bytes) -> Optional[int]:
82         """
83         Writes data of a specified size to serial port.
84
85         :param data: data to be written to the port.
86         :return: the length of the written data.
87         """
88
89         with self.lock:
90             length = self.ser.write(data)
91
92         return length
93
94     def swrite(self, data: str, encoding: str = 'utf-8') -> Optional[int]:
95         """
```

```
96     Writes specified data string to serial port.
97     :param data: the data to write to serial port, as string.
98     :param encoding: in which encoding the string is written.
99     :return: the length of the written data.
100    """
101    return self.write(data.encode(encoding))
102
103    def read(self, size: int = 1) -> bytes:
104        """
105            Reads data from of a specified size to serial port.
106
107            :param size: the size to be read.
108            :return: the read value in bytes.
109        """
110
111        with self.lock:
112            byte = self.ser.read(size)
113
114        return byte
115
116    def sread(self, size: int = 1, encoding: str = 'utf-8') -> str:
117        """
118            Reads from serial port the specified size.
119            :param size: the size to be read.
120            :param encoding: the encoding in which the bytes is expected to be.
121            :return: the read value as string.
122        """
123
124        return self.read(size).decode(encoding)
125
126    def readline(self) -> bytes:
127        """
128            Reads a line from the serial buffer
129            :return: the read line in bytes.
130        """
131
132        with self.lock:
133            if not self.ser.is_open:
134                self.ser.open()
135            return self.ser.readline()
136
137    def sreadline(self, encoding: str = 'utf-8') -> str:
138        """
139            Reads a line from the serial buffer.
140            :param encoding: the encoding in which the bytes is expected to be.
141            :return: the read line as string.
142        """
143
144        return self.readline().decode(encoding)
145
146    def readall(self) -> bytes:
147        """
148            Reads all the serial buffer.
149            :return: the entire buffer in bytes.
150        """
151
152        with self.lock:
153            whole_buffer = self.ser.readall()
```

```

152         return whole_buffer
153
154     def sreadall(self, encoding: str = 'utf-8') -> str:
155         """
156             Reads all the serial buffer.
157
158             :param encoding: the encoding in which the bytes are expected to be.
159             :return: the entire buffer as string.
160             """
161
162         return self.readall().decode(encoding)
163
164     @property
165     def is_closed(self) -> bool:
166         return not self.ser.is_open
167
168     @property
169     def is_open(self) -> bool:
170         return self.ser.is_open
171
172     @property
173     def port(self):
174         ser_port = self.ser.port
175         if ser_port != self._port:
176             self._port = ser_port
177         return ser_port
178
179     @port.setter
180     def port(self, port: str):
181         self._port = port
182         self.ser.port = self._port

```

Listing E.1: pArm-S1/pArm/communications/connection.py

```

1 from ..gcode import generator
2 from ..gcode import interpreter
3 from ..security import RSA
4 from .. import Connection
5 from serial import SerialException
6 from logging import getLogger
7 from .control_interface import ControlInterface
8 from typing import Callable, Optional, Union
9 from pArm.control import control_management
10 from concurrent.futures import ThreadPoolExecutor, Future
11 from ..utils import AtomicFloat, ErrorData
12 from .heart_beat import Heart
13 from math import pi as pi
14
15 LOWEST_X_VALUE = 0
16 HIGHEST_X_VALUE = 300
17 LOWEST_Y_VALUE = 0
18 HIGHEST_Y_VALUE = 300
19 LOWEST_Z_VALUE = 0
20 HIGHEST_Z_VALUE = 300
21
22 log = getLogger("Roger")

```

```
23
24
25 class Control(ControlInterface):
26
27     def __init__(self, executor: ThreadPoolExecutor, x=0, y=0, z=0, theta1=0,
28                  theta2=0, theta3=0, port=''):
29         super(Control, self).__init__(executor, x, y, z, theta1, theta2, theta3,
30                                       port)
31
32         self._err_fn = None
33
34         self.connection = Connection()
35         self.heart = Heart(conn=self.connection)
36
37     @property
38     def x(self):
39         return self._x
40
41     @property
42     def y(self):
43         return self._y
44
45     @property
46     def z(self):
47         return self._z
48
49     @property
50     def port(self):
51         return self.connection.port
52
53     @property
54     def err_fn(self) -> Callable[[int, str], None]:
55         return self._err_fn
56
57     @err_fn.setter
58     def err_fn(self, fn: Callable[[int, str], None]):
59         self._err_fn = fn
60
61     @x.setter
62     def x(self, x):
63         if LOWEST_X_VALUE <= x <= HIGHEST_X_VALUE:
64             self._x = x
65         else:
66             print("X value out of bounds")
67
68     @y.setter
69     def y(self, y):
70         if LOWEST_Y_VALUE <= y <= HIGHEST_Y_VALUE:
71             self._y = y
72         else:
73             print("Y value out of bounds")
74
75     @z.setter
76     def z(self, z):
77         if LOWEST_Z_VALUE <= z <= HIGHEST_Z_VALUE:
78             self._z = z
```

```
79     else:
80         print("Z value out of bounds")
81
82     @port.setter
83     def port(self, port):
84         self.connection.port = port
85
86     def move_to_xyz(self, x, y, z,
87                     time_object: Optional[AtomicFloat] = None) -> \
88         Future:
89     """
90     Triggers the needed procedures to move the arm to the cartesian position
91     that is indicated in its parameters.
92     :param x: x position to which the end effector shall move
93     :param y: y position to which the end effector shall move
94     :param z: z position to which the end effector shall move
95     :param time_object: the atomic float holder value.
96     :return: the future object.
97     """
98
99     def fn():
100        byte_stream = generator.generate_xyz_movement(x, y, z)
101        try:
102            with self.connection as conn:
103                conn.write(byte_stream)
104        except SerialException:
105            log.warning("There is no suitable connection with the device", exc_info=True)
106            err = control_management.verify_movement_completed(time_object)
107            if err:
108                return err
109            self.read_cartesian_positions()
110            self.read_angular_positions()
111
112        return self
113
114    return self.executor.submit(fn)
115
116    def move_to_thetas(self,
117                      theta1,
118                      theta2,
119                      theta3,
120                      time_object: Optional[AtomicFloat] = None) -> \
121        Future:
122    """
123    Triggers the needed procedures to move the arm to the angular position
124    that is indicated in its parameters.
125    :param theta1: theta1 angle to which the base motor shall move
126    :param theta2: theta2 angle to which the shoulder motor shall move
127    :param theta3: theta3 angle to which the elbow motor shall move
128    :param time_object: the atomic float holder value.
129    :return: the future object.
130    """
131
132    theta1_in_radians = theta1 * (pi/180)
133    theta2_in_radians = theta2 * (pi/180)
134    theta3_in_radians = theta3 * (pi/180)
```

```
135     byte_stream = generator.generate_theta_movement(theta1_in_radians,
136                                                     theta2_in_radians,
137                                                     theta3_in_radians)
138
139
140     def fn():
141         try:
142             with self.connection as conn:
143                 conn.write(byte_stream)
144         except SerialException:
145             log.warning("There is no suitable connection with the device", exc_info=True)
146         else:
147             log.debug(
148                 "theta1, theta2, theta3 values successfully sent to device")
149             err = control_management.verify_movement_completed(time_object)
150             if err:
151                 return err
152             self.read_cartesian_positions()
153             self.read_angular_positions()
154         return self
155
156     return self.executor.submit(fn)
157
158     def send_to_origin(self, time_object: Optional[AtomicFloat] = None) -> \
159         Future:
160         """
161         This function send the arm to its initial position.
162         :param time_object: the atomic float holder value.
163         :return: the future object.
164         """
165
166     byte_stream = generator.generate_send_to_origin()
167
168     def fn():
169         try:
170             with self.connection as conn:
171                 conn.write(byte_stream)
172         except SerialException:
173             log.warning("There is no suitable connection with the device", exc_info=True)
174         else:
175             log.debug(f"Device sent to origin")
176             err = control_management.verify_movement_completed(time_object)
177             if err:
178                 return err
179             self.read_cartesian_positions()
180             self.read_angular_positions()
181
182         return self
183
184     return self.executor.submit(fn)
185
186     def read_cartesian_positions(self):
187         """
188         This function request the real physical cartesian position from the arm
189         controller and then save them in the class variables
190         :return: no return.
```

```
191     """
192     control_management.request_cartesian_position()
193
194     try:
195         found, missed_instructions, line = interpreter.wait_for('G0')
196         if found:
197             cartesian_positions = interpreter.parse_line(line)
198             self.x = cartesian_positions.x
199             self.y = cartesian_positions.y
200             self.z = cartesian_positions.z
201     except SerialException:
202         log.warning("There is no suitable connection with the device", exc_info=True)
203
204     def read_angular_positions(self):
205         """
206             This function request the real physical angular values from the arm
207             controller and then save them in the class variables
208             :return: no return.
209         """
210         control_management.request_angular_position()
211
212         try:
213             found, missed_instructions, line = interpreter.wait_for('G1')
214             if found:
215                 angular_positions = interpreter.parse_line(line)
216                 self.theta1 = angular_positions.t1
217                 self.theta2 = angular_positions.t2
218                 self.theta3 = angular_positions.t3
219         except SerialException:
220             log.warning("There is no suitable connection with the device", exc_info=True)
221
222     def cancel_movement(self) -> Future:
223         """
224             This function cancels the current movement. If the controller confirms
225             that the movement has been canceled, this function also updates the
226             class position variables with the real physical ones.
227             :return: ErrorData if the cancellation could not complete successfully.
228         """
229         control_management.request_cancel_movement()
230
231     def fn():
232         gcode = ["J{}".format(x) for x in range(2, 21)]
233         gcode.append('M1')
234
235         try:
236             found, missed_instructions, line = interpreter.wait_for(gcode)
237             if found and line is True:
238                 self.read_angular_positions()
239                 self.read_cartesian_positions()
240                 return self
241             else:
242                 err_code = int(line[1:])
243                 return interpreter.errors[err_code]
244         except SerialException:
245             log.warning("There is no suitable connection with the device", exc_info=True)
246
```

```
247     return self.executor.submit(fn)
248
249 def read_handshake_values(self, order: str):
250     """
251     This function reads through the buffer searching for a line that contains
252     the order specified in it parameter. If found it return the whole line.
253     :param order: the order that the function shall look for
254     :return: the complete line where that instruction has been found.
255     """
256     try:
257         found, missed_instructions, line = interpreter.wait_for(order)
258         if found:
259             return interpreter.parse_line(line)
260     except SerialException as e:
261         log.warning(str(e), exc_info=True)
262
263     def quit(self):
264         self.cancel_movement()
265         self.heart.is_beating = False
266
267     def do_handshake(self):
268         """
269         Starts the handshake procedure.
270         The procedure is as follows:
271         1. The control application (this software) requests the procedure to start
272
273         2. The arm controller sends I2 {n} where n is the module needed to "un-sign"
274         a string that will follow.
275
276         3. The arm controller sends I3 {e} where e is the exponent needed to "un-sign"
277         a string that will follow.
278
279         4. The control application (this software) will proceed to create an instance
280         of the RSA class with n and e.
281
282         5. The arm controller sends I4 {signed integer} where the signed integer
283         its a random integer that the arm controller has signed.
284
285         6. Using the RSA object the control application (this software) first
286         "un-sign" the integer. Then, using the same n and e we encrypt it.
287
288         7. Then we use this new encrypted integer to generate the heartbeat
289         and send it back to the arm controller.
290
291         8. The arm controller verifies the integer, and if it succeeds,
292         it send an I5 to confirm the handshake has been done correctly.
293
294     The control application (this software) can also receive an error code
295     with the format Jx, where x its an integer between 2 and 21. This can happen
296     if at any step, the arm controller receives an unexpected value. This event
297     would finish the handshaking procedure and the pairing would fail.
298     :return: The returns from the else statements are possible error codes
299     that the arm controller could return.
300     """
301     gcode = {"J{}".format(x) for x in range(2, 21)}
302     control_management.request_handshake()
```

```

303     try:
304         gcode.add('I2')
305         found, missed_instructions, line = interpreter.wait_for(gcode, timeout=20)
306         n = interpreter.parse_line(line)
307         log.debug("n: "+n)
308         if found and isinstance(n, str):
309             gcode.add('I3')
310             found, missed_instructions, line = interpreter.wait_for(gcode)
311             e = interpreter.parse_line(line)
312             log.debug("e: "+e)
313             if found and isinstance(e, str):
314                 n, e = int(n), int(e)
315                 rsa = RSA(n=n, e=e)
316                 gcode.add('I4')
317                 found, missed_instructions, line = \
318                     interpreter.wait_for(gcode)
319                 signed_value = interpreter.parse_line(line)
320                 print(signed_value)
321                 if found and isinstance(signed_value, str):
322                     signed_value = int(signed_value)
323                     verified_value = rsa.verify(signed_value)
324                     encrypted_value = rsa.encrypt(verified_value)
325                     log.debug("Se ha iniciado el corazon")
326                     self.connection.write(generator
327                         .generate_unsigned_string(encrypted_value))
328                     gcode.add('I5')
329                     log.debug("He escrito el valor encriptado")
330                     found, missed_instructions, line = \
331                         interpreter.wait_for('I5')
332                     if found:
333                         log.info("Handshake done.")
334                         self.heart.beat = encrypted_value
335                         self.heart.is_beating = True
336                     else:
337                         self.connection.ser.close()
338                         return signed_value
339                 else:
340                     return e
341             else:
342                 return n
343
344     except SerialException:
345         log.error("Error in handshake procedure", exc_info=True)

```

Listing E.2: pArm-S1/pArm/control/control.py

```

1 from abc import ABC, abstractmethod
2 from typing import Callable, Optional
3 from concurrent.futures import ThreadPoolExecutor
4 from ..utils import AtomicFloat
5
6
7 class ControlInterface(ABC):
8
9     @abstractmethod
10    def __init__(self, executor: ThreadPoolExecutor, x=0, y=0, z=0, theta1=0, theta2=0, theta3

```

```
11     =0, port='')):
12         self.executor = executor
13         self.x = x
14         self.y = y
15         self.z = z
16
17         self.theta1 = theta1
18         self.theta2 = theta2
19         self.theta3 = theta3
20
21     @abstractmethod
22     def move_to_xyz(self, x, y, z, time_object: Optional[AtomicFloat] = None):
23         """
24             Triggers the needed procedures to move the arm to the cartesian position
25             that is indicated in its parameters.
26             :param x: x position to which the end effector shall move
27             :param y: y position to which the end effector shall move
28             :param z: z position to which the end effector shall move
29             :param time_object: the atomic float holder value.
30             :return: the future object.
31         """
32
33     pass
34
35     @abstractmethod
36     def move_to_thetas(self, theta1, theta2, theta3, time_object: Optional[AtomicFloat] = None):
37         """
38             Triggers the needed procedures to move the arm to the angular position
39             that is indicated in its parameters.
40             :param theta1: theta1 angle to which the base motor shall move
41             :param theta2: theta2 angle to which the shoulder motor shall move
42             :param theta3: theta3 angle to which the elbow motor shall move
43             :param time_object: the atomic float holder value.
44             :return: the future object.
45         """
46
47     pass
48
49     @abstractmethod
50     def cancel_movement(self):
51         """
52             This function sends a request to the arm controller telling it to stop
53             the movement that is currently being made. If the controller confirms
54             that the movement has been canceled, this function also updates the
55             class position variables with the real physical ones.
56             :return: no return.
57         """
58
59     pass
60
61     @abstractmethod
62     def do_handshake(self):
63         """
64             Starts the handshake procedure.
65             The procedure is as follows:
66             1. The control application (this software) requests the procedure to start
67
68             2. The arm controller sends I2 {n} where n is the module needed to "un-sign"
```

```
65     a string that will follow.  
66  
67     3. The arm controller sends I3 {e} where e is the exponent needed to "un-sign"  
68     a string that will follow.  
69  
70     4. The control application (this software) will proceed to create an instance  
71     of the RSA class with n and e.  
72  
73     5. The arm controller sends I4 {signed integer} where the signed integer  
74     its a random integer that the arm controller has signed.  
75  
76     6. Using the RSA object the control application (this software) first  
77     "un-sign" the integer. Then, using the same n and e we encrypt it.  
78  
79     7. Then we use this new encrypted integer to generate the heartbeat  
80     and send it back to the arm controller.  
81  
82     8. The arm controller verifies the integer, and if it succeeds,  
83     it send an I5 to confirm the handshake has been done correctly.  
84  
85     The control application (this software) can also receive an error code  
86     with the format Jx, where x its an integer between 2 and 21. This can happen  
87     if at any step, the arm controller receives an unexpected value. This event  
88     would finish the handshaking procedure and the pairing would fail.  
89     :return: The returns from the else statements are possible error codes  
90     that the arm controller could return.  
91     """  
92         pass  
93  
94     @abstractmethod  
95     def quit(self):  
96         pass  
97  
98     @property  
99     @abstractmethod  
100    def err_fn(self) -> Callable[[int, str], None]:  
101        pass  
102  
103    @property  
104    @abstractmethod  
105    def x(self):  
106        pass  
107  
108    @property  
109    @abstractmethod  
110    def y(self):  
111        pass  
112  
113    @property  
114    @abstractmethod  
115    def z(self):  
116        pass  
117  
118    @property  
119    @abstractmethod  
120    def port(self):
```

```

121     pass
122
123     @x.setter
124     @abstractmethod
125     def x(self, value):
126         pass
127
128     @y.setter
129     @abstractmethod
130     def y(self, value):
131         pass
132
133     @z.setter
134     @abstractmethod
135     def z(self, value):
136         pass
137
138     @port.setter
139     @abstractmethod
140     def port(self, value):
141         pass
142
143     @err_fn.setter
144     @abstractmethod
145     def err_fn(self, fn: Callable[[int, str], None]):
146         pass

```

Listing E.3: pArm-S1/pArm/control/control_interface.py

```

1 from serial import SerialException
2 from pArm.communications.connection import Connection
3 from pArm.gcode import interpreter
4 from logging import getLogger
5 from pArm.gcode import generator
6 from pArm.utils.error_data import ErrorData
7 import logging
8 from ..utils import AtomicFloat
9 from typing import Optional
10
11 log = getLogger("Roger")
12 connection = Connection()
13
14
15 def verify_movement_completed(time_object: Optional[AtomicFloat] = None):
16     """
17     Verifies that, after a movement order has been issued, it completes correctly.
18     If not, this function takes care of the errors.
19     :param time_object: the atomic float holder value.
20     :return: An ErrorData named tuple, in case of an error.
21     """
22     try:
23         gcode = ["J{}".format(x) for x in range(1, 21)]
24         found, missed_instructions, line = interpreter.wait_for(gcode)
25         line_meaning = interpreter.parse_line(line)
26
27         if found and isinstance(line_meaning, float):

```

```
28         if time_object:
29             time_object.value = line_meaning
30             found, missed_instructions, line = interpreter.wait_for('J21')
31             if found:
32                 log.info(line)
33             else:
34                 return ErrorData(logging.ERROR,
35                                 'El brazo no ha podido llegar al destino')
36             else:
37                 return line_meaning
38
39     except SerialException:
40         log.warning("There is no suitable connection with the device", exc_info=True)
41
42
43 def request_cartesian_position():
44     """
45     This function sends a Gcode requesting the cartesian positions of the
46     arm
47     :return: no return
48     """
49     byte_stream = generator.generate_request_cartesian_position()
50
51     try:
52         with connection as conn:
53             conn.write(byte_stream)
54     except SerialException:
55         log.warning("There is no suitable connection with the device", exc_info=True)
56     else:
57         log.debug(f"Cartesian position requested")
58
59
60 def request_angular_position():
61     """
62     This function sends a Gcode requesting the angular positions of the
63     arm
64     :return: no return
65     """
66     byte_stream = generator.generate_request_angular_position()
67
68     try:
69         with connection as conn:
70             conn.write(byte_stream)
71     except SerialException:
72         log.warning("There is no suitable connection with the device", exc_info=True)
73     else:
74         log.debug(f"Angular position requested")
75
76
77 def request_recalculate_keys():
78     """
79     This function sends a Gcode requesting the arm controller to re-calculate
80     the keys used for the hand-shake. In the current state of the code and project
81     this function is not being used.
82     :return: no return
83     """
```

```

84     byte_stream = generator.generate_recalculate_keys()
85
86     try:
87         with connection as conn:
88             conn.write(byte_stream)
89     except SerialException:
90         log.warning("There is no suitable connection with the device", exc_info=True)
91     else:
92         log.debug(f"Key recalculation requested")
93
94
95 def request_cancel_movement():
96     """
97     This function sends a Gcode requesting the arm controller to cancel the
98     current movement being made.
99     :return: no return
100    """
101   byte_stream = generator.generate_cancel_movement()
102   try:
103       with connection as conn:
104           conn.write(byte_stream)
105   except SerialException:
106       log.warning("There is no suitable connection with the device", exc_info=True)
107   else:
108       log.debug(f"Requested cancel movement")
109
110
111 def request_handshake():
112     """
113     This function sends a Gcode requesting to begin the handshake procedure.
114     If received correctly, the arm controller proceeds to also start the
115     handshaking procedure on its end.
116     :return: no return
117     """
118   byte_stream = generator.generate_request_n_e()
119   try:
120       if not connection.is_open:
121           connection.ser.open()
122       connection.write(byte_stream)
123   except SerialException:
124       log.warning("There is no suitable connection with the device", exc_info=True)
125   else:
126       log.debug(f"Requested handshake start")

```

Listing E.4: pArm-S1/pArm/control/control_management.py

```

1 import time
2 from typing import Optional
3 from threading import Thread
4 from logging import getLogger
5 from serial import SerialException
6 from .. import Connection
7
8 log = getLogger("Roger")
9
10

```

```
11 class Heart:
12     def __init__(self,
13         beat: int = 0,
14         start_beating: bool = False,
15         conn: Optional[Connection] = None):
16         self._beat = f"I7 {beat}".encode('utf-8')
17         self._is_beating = start_beating
18         self.connection = conn if conn else Connection()
19         self._t = Thread(target=lambda: self.background_repeated_heartbeat())
20         if start_beating:
21             self._t.start()
22
23     @property
24     def is_beating(self):
25         return self._is_beating
26
27     @is_beating.setter
28     def is_beating(self, val: bool):
29         self._is_beating = val
30         if val and not self._t.is_alive():
31             self._t.start()
32         elif not val and self._t.is_alive():
33             self._t.join()
34
35     @property
36     def beat(self):
37         return self._beat
38
39     @beat.setter
40     def beat(self, beat):
41         self._beat = f"I7 {beat}".encode('utf-8')
42
43     def heartbeat_tick(self):
44         """
45             This function generate a heartbeat. The heartbeat its an int passed as
46             parameter to the class
47             :return: no return.
48         """
49
50         try:
51             log.debug('Tick')
52             self.connection.write(self.beat)
53         except SerialException:
54             log.warning("There is no suitable connection with the device",
55                         exc_info=True)
56
57     def background_repeated_heartbeat(self):
58         """
59             If is_beating is true, this function repeats the beat at an interval of
60             .2 seconds. If is_beating is false the heart does not beat.
61             :return: no return.
62         """
63
64         while self.is_beating:
65             self.heartbeat_tick()
66             time.sleep(.195)
```

Listing E.5: pArm-S1/pArm/control/heart_beat.py

```
1 def generate_xyz_movement(x, y, z):
2     """
3         Generate a Gcode such that the arm controller moves to the cartesian position
4         that are passed as parameters
5
6         :param x: x position to there the arm end effector should move
7         :param y: y position to there the arm end effector should move
8         :param z: z position to there the arm end effector should move
9         :return: the actual Gcode, ready to be sent.
10    """
11
12    return f"G0 X{x} Y{y} Z{z}\n".encode('utf-8')
13
14
15 def generate_theta_movement(theta1, theta2, theta3):
16     """
17         Generate a Gcode such that the arm controller moves the motors to the angles
18         specified as parameters
19
20         :param theta1: angle to which the base motor shall move.
21         :param theta2: angle to which the shoulder motor shall move.
22         :param theta3: angle to which the elbow motor shall move.
23         :return: the actual Gcode, ready to be sent.
24    """
25
26    return f"G1 X{theta1} Y{theta2} Z{theta3}\n".encode('utf-8')
27
28
29 def generate_send_to_origin():
30     """
31         Generates a Gcode such that the arm moves to the original positions.
32         This process is called zeroing
33         :return: the actual Gcode, ready to be sent.
34    """
35
36
37 def generate_cancel_movement():
38     """
39         Generates a Gcode such that the arm controller shall stop the movement that
40         the arm is actually doing.
41         :return: the actual Gcode, ready to be sent.
42    """
43
44
45
46 def generate_request_cartesian_position():
47     """
48         Generates a Gcode to request the physical cartesian positions at which the
49         arm currently is.
50         :return: the actual Gcode, ready to be sent.
51    """
52
53    return 'M114\n'.encode('utf-8')
```

```

54
55 def generate_request_angular_position():
56     """
57     Generates a Gcode to request the physical angular positions at which the
58     arm currently is.
59     :return: the actual Gcode, ready to be sent.
60     """
61     return 'M280\n'.encode('utf-8')
62
63
64 def generate_request_n_e():
65     """
66     Generates a Gcode to request the arm controller to send "n" and "e", values
67     needed to calculate the public key of the device.
68     :return: the actual Gcode, ready to be sent.
69     """
70     return 'I1\n'.encode('utf-8')
71
72
73 def generate_unsigned_string(unsigned_string):
74     """
75     Generates a Gcode with the unsigned string attached.
76     :param unsigned_string: unsigned string to be sent to the device
77     :return: the actual Gcode, ready to be sent.
78     """
79     return f'I5 {unsigned_string}\n'.encode('utf-8')
80
81
82 def generate_recalculate_keys(encrypted_string):
83     """
84     Generates a Gcode to request that the device calculate new keys for the
85     authentication process.
86     :return: the actual Gcode, ready to be sent.
87     """
88     return f'I6 {encrypted_string}\n'.encode('utf-8')
89
90
91 def generate_heart_beat(beat):
92     """
93     Generates a Gcode to create a heartbeat.
94     :param beat: the message that goes into the heartbeat
95     :return: the actual GCode, ready to be sent.
96     """
97     return f'I7 {beat}\n'.encode('utf-8')

```

Listing E.6: pArm-S1/pArm/gcode/generator.py

```

1 from ..communications import Connection
2 from serial import SerialException
3 from logging import getLogger
4 from typing import Tuple, Union
5 from collections import namedtuple
6 from typing import Optional
7 from typing import List
8 from typing import Iterable
9 from ..utils.error_data import ErrorData

```

```
10| import logging
11| import time
12|
13| log = getLogger("Roger")
14|
15| connection = Connection()
16|
17| XYZ = namedtuple('XYZ', 'x y z')
18| Theta = namedtuple('Theta', 't1 t2 t3')
19|
20| errors = {
21|     2: ErrorData(logging.ERROR, 'Error en la calibración'),
22|     3: ErrorData(logging.ERROR, 'GCode desconocido'),
23|     4: ErrorData(logging.ERROR, 'Posición fuera del rango'),
24|     5: ErrorData(logging.ERROR, 'El brazo no puede cancelar un movimiento inexistente'),
25|     6: ErrorData(logging.ERROR, 'Error en el handshake'),
26|     7: ErrorData(logging.ERROR, 'El brazo ya se está moviendo'),
27|     8: ErrorData(logging.ERROR, 'No se han especificado coordenadas para el movimiento
28|         cartesiano'),
29|     9: ErrorData(logging.ERROR, 'No se han especificado coordenadas para el movimiento angular.
29|         '),
30|    10:ErrorData(logging.ERROR, 'Dispositivo no identificado'),
31|    11:ErrorData(logging.ERROR, 'Desbordamiento del buffers')
32| }
33|
34| def read_buffer_line():
35|     """
36|     Reads a line from the UART.
37|     :return: returns the first line read.
38|     """
39|     try:
40|         with connection as conn:
41|             line = conn.readline()
42|     except SerialException:
43|         log.warning("There is no suitable connection with the device", exc_info=True)
44|     else:
45|         log.debug("Line read successfully")
46|
47|     return line
48|
49|
50| def parse_line(line: Optional[Union[str, bytes]] = None) -> Union[bool, XYZ, Theta, str]:
51|     """
52|     Parses the line passed as parameter looking for the kind of order that it is
53|     If no line is passed as parameter, it reads the first line of the buffer.
54|
55|     Parsing means that this function will decide what kind of order it is and
56|     will call the corresponding function to react accordingly.
57|     :param line: The line that needs to be parsed
58|     :return: calls the corresponding function.
59|     """
60|     if not line:
61|         line = read_buffer_line()
62|
63|     if isinstance(line, bytes):
```

```
64     line = line.decode("utf-8")
65
66     if line[0] == "I":
67         return parse_i_order(line)
68     elif line[0] == "G":
69         return parse_g_order(line)
70     elif line[0] == "M":
71         return parse_m_order(line)
72     elif line[0] == "J":
73         return parse_j_order(line)
74
75
76 def parse_i_order(i_order):
77     """
78     This function is called when the order is an I order. It continues to parse
79     it to the number of the order and acts accordingly.
80     :param i_order: the I order that has to be parsed.
81     :return: returns the parameter of the order.
82     """
83     split_order = i_order.split(' ')
84     order_number = int(split_order[0][1:])
85
86     if order_number == 2:
87         return split_order[1]
88     elif order_number == 3:
89         return split_order[1]
90     elif order_number == 4:
91         return split_order[1]
92     elif order_number == 5:
93         return True
94
95
96 def parse_g_order(g_order) -> Tuple[float, float, float]:
97     """
98     This function is called when the order is an G order. It continue to parse
99     it to the number of the order and acts accordingly.
100    :param g_order: the G order that has to be parsed.
101    :return: a namedTuple that contains either the angular values or the
102    cartesian ones
103    """
104    split_order = g_order.split(' ')
105    order_number = int(split_order[0][1:])
106
107    if order_number == 0:
108        return XYZ(x=float(split_order[1][1:]),
109                   y=float(split_order[2][1:]),
110                   z=float(split_order[3][1:]))
111
112    elif order_number == 1:
113        return Theta(t1=float(split_order[1][1:]),
114                     t2=float(split_order[2][1:]),
115                     t3=float(split_order[3][1:])))
116
117
118 def parse_m_order(m_order):
119     """
```

```
120 This function is called when the order is an M order. It continue to parse
121 it to the number of the order and acts accordingly.
122 :param m_order: the M order that has to be parsed.
123 :return: returns True if the order is type M1
124 """
125 split_order = m_order.split(' ')
126 order_number = int(split_order[0][1:])
127
128 if order_number == 1:
129     return True
130
131
132 def parse_j_order(j_order):
133 """
134 This function is called when the order is an J order. It continue to parse
135 it to the number of the order and acts accordingly.
136 :param j_order: the J order that has to be parsed.
137 :return: Either confirmation messages (For J1 and J21) or error codes
138 (from J2 to J20)
139 """
140 order_number = int(j_order[1:])
141
142 if order_number == 1:
143     return float(j_order.split()[1])
144 if 2 <= order_number <= 20:
145     return errors[order_number]
146 if order_number == 21:
147     return 'Arrived to position'
148
149
150 def wait_for(gcode: Union[str, Iterable[str]], timeout: int = 5) -> \
151     Tuple[bool, List[str], str]:
152 """
153 This function keeps reading the buffer until it finds the GCode that its
154 passed as parameter. It is also capable to wait for an order from within an
155 interval of order.
156
157 :param gcode: The order or interval of orders that the function has to
158 look for
159 :param timeout: The time that has to elapse until the function reaches timeout
160 and stops looking for the specified order
161 :return: Boolean, to know if the function finished because it found the
162 order or because it reached timeout.
163 List, containing other orders that have been read that were not the one that
164 the function was specifically looking for.
165 String, contains the whole line where the order has been found.
166 """
167 missed_inst = []
168 timeout += time.time()
169
170 line = connection.sreadline()
171 log.debug(f"Read line: {line}")
172
173 def check_valid(c_line, gcode) -> bool:
174     if len(c_line) == 0:
175         return False
```

```

176     code = c_line.split()[0]
177     return code in gcode if isinstance(gcode, Iterable) else code != gcode
178
179     while not check_valid(line, gcode) and time.time() <= timeout:
180         if line != '':
181             missed_inst.append(line)
182         time.sleep(0.1)
183         line = connection.sreadline()
184
185     return timeout > time.time(), missed_inst, line

```

Listing E.7: pArm-S1/pArm/gcode/interpreter.py

```

1 import logging
2 import math
3 import os
4 import pyqtgraph
5 import serial.tools.list_ports
6 import webbrowser
7 from PyQt5 import QtCore, QtWidgets, uic, QtGui
8 from PyQt5.QtWidgets import QMessageBox, QMenu, QAction
9 from pyqtgraph import PlotWidget
10 from concurrent.futures import Future
11 from ..utils import AtomicFloat
12 from ..utils.error_data import ErrorData
13 from ..control.control_interface import ControlInterface
14 from .progress_widget import ProgressWidget
15 from .rect_item import RectItem
16 from ..logger import add_handler, QTextEditLogger
17
18
19 def inverse_kinematics(x_coord, y_coord, z_coord):
20     """
21     This function performs the calculations related to inverse
22     kinematic model, which are used to graphically draw the
23     position of the arm on the GUI.
24
25     :params x, y, z coord: Cartesian coordinates of the point to which
26                           inverse kinematics is applied.
27     """
28     try:
29         from math import acos, atan, atan2, pi, sqrt, sin, cos
30
31         x_coord = 11.5 if x_coord < 11.5 else x_coord
32
33         al = 142.07
34         au = 158.08
35
36         theta_0 = atan2(x_coord, y_coord)
37         xz = (x_coord ** 2) + (y_coord**2) + (z_coord ** 2)
38         lxz = sqrt(xz)
39         theta_1 = acos((-al ** 2) - xz + (au ** 2)) / (-2 * al * lxz))
40         theta_2 = acos((-al ** 2) - (au ** 2) + xz) / (-2 * al * au))
41         phi = atan2(z_coord, sqrt(x_coord ** 2 + y_coord ** 2))
42         theta_1 += phi
43

```

```
44     theta_0 *= (180/pi)
45     theta_1 *= (180/pi)
46     theta_2 *= (180/pi)
47     theta_1 = 135 - theta_1
48     return theta_0, theta_1, theta_2
49 except ValueError:
50     return None
51
52
53 class Ui(QtWidgets.QMainWindow):
54     """
55     Ui class contains all the code related to the graphic user interface, including
56     initializations, configurations, Qt signals handling, etc. This class inherits from
57     the QMainWindow class, which represent the main window graphical component of Qt framework
58     """
59     def __init__(self, control: ControlInterface):
60         """
61             This method is in charge of loading the .ui XML file, which contains all the graphic
62             layout of the
63             GUI, and then matching all the graphical components into the code by using 'findchild()'
64             . Most attributes
65             of the class are declared in this method.
66
67             :param control: object instance of control.py class that is used to interact with the
68             logic communication code
69             """
70
71         #Class is initialized by calling to its parent class init.
72         super(Ui, self).__init__()
73
74         #.ui XML file is loaded
75         uic.loadUi(os.path.join(os.path.dirname(os.path.realpath(__file__)), 'InProgressGUI.ui'))
76         self
77
78         #instance of control.py class which is used to communicate with the logic
79         communications code
80         self.handler = control
81
82         #Serial Port used to send data to PCB
83         self.port = None
84
85         #Auxiliar counter
86         self.counter = 200
87
88         #mouse flag
89         self.mouse_enabler = False
90
91         #GUI window left section set up
92         self.menu_port = self.findChild(QtWidgets.QMenu, 'menuPort_Selection')
93
94         self.menu_info = self.findChild(QtWidgets.QMenu, 'menu_info')
95
96         self.slider_1 = self.findChild(QtWidgets.QSlider, 'Slider1')
97         self.slider_2 = self.findChild(QtWidgets.QSlider, 'Slider2')
98         self.slider_3 = self.findChild(QtWidgets.QSlider, 'Slider3')
99
100        self.spin_box_1 = self.findChild(QtWidgets.QDoubleSpinBox, 'SpinBox1')
```

```
95      self.spin_box_2 = self.findChild(QtWidgets.QDoubleSpinBox, 'SpinBox2')
96      self.spin_box_3 = self.findChild(QtWidgets.QDoubleSpinBox, 'SpinBox3')
97
98      self.execute_button = self.findChild(QtWidgets.QPushButton, 'ExecuteButton')
99      self.origin_button = self.findChild(QtWidgets.QPushButton, 'origin_button')
100
101     self.combo_box_coordinates = self.findChild(QtWidgets.QComboBox, 'ComboBoxCoordinates')
102
103     self.logger_box = self.findChild(QtWidgets.QPlainTextEdit, 'LoggerBox')
104
105     self.slider_1_label = self.findChild(QtWidgets.QLabel, 'SliderLabel1')
106     self.slider_2_label = self.findChild(QtWidgets.QLabel, 'SliderLabel2')
107     self.slider_3_label = self.findChild(QtWidgets.QLabel, 'SliderLabel3')
108     self.slider_1_left_label = self.findChild(QtWidgets.QLabel, 'LeftLabelSlider1')
109     self.slider_2_left_label = self.findChild(QtWidgets.QLabel, 'LeftLabelSlider2')
110     self.slider_3_left_label = self.findChild(QtWidgets.QLabel, 'LeftLabelSlider3')
111     self.slider_1_right_label = self.findChild(QtWidgets.QLabel, 'RightLabelSlider1')
112     self.slider_2_right_label = self.findChild(QtWidgets.QLabel, 'RightLabelSlider2')
113     self.slider_3_right_label = self.findChild(QtWidgets.QLabel, 'RightLabelSlider3')
114     self.slider_2_mid_label = self.findChild(QtWidgets.QLabel, 'MidLabelSlider2')
115
116     self.progress_bar = ProgressWidget.from_bar(self.findChild(QtWidgets.QProgressBar, 'ProgressBar'))
117     self.progress_bar.hide()
118
119 #Right Window Section
120     self.logger_box = self.findChild(QtWidgets.QPlainTextEdit, 'LoggerBox')
121     qt_logger = QTextEditLogger(edit_text=self.logger_box)
122     add_handler(qt_logger,
123                 logger_name="Roger",
124                 level=logging.INFO,
125                 log_format="%(asctime)s | [%(levelname)s]: %(message)s")
126     self.log = logging.getLogger("Roger")
127
128     self.top_view = self.findChild(PlotWidget, 'TopView')
129     self.side_view = self.findChild(PlotWidget, 'SideView')
130
131 def setupGUI(self):
132     """
133     Every graphic component/widget of the GUI is initialized and configured within this
134     method.
135     """
136
137     # Grouped widgets in order to ease parameter passing
138     sliders = [self.slider_1, self.slider_2, self.slider_3]
139     spin_boxes = [self.spin_box_1, self.spin_box_2, self.spin_box_3]
140     sliders_labels = [self.slider_1_label, self.slider_2_label, self.slider_3_label,
141                       self.slider_1_left_label, self.slider_1_right_label, self.
142                       slider_2_left_label,
143                           self.slider_2_right_label, self.slider_3_left_label, self.
144                       slider_3_right_label,
145                           self.slider_2_mid_label]
146     graphics = [self.top_view, self.side_view]
147
148     # Extra setting initialization
149     if getattr(self.slider_1, "id", None) is None:
150         setattr(self.slider_1, "id", 1)
```

```
148
149     if getattr(self.slider_2, "id", None) is None:
150         setattr(self.slider_2, "id", 2)
151
152     if getattr(self.slider_3, "id", None) is None:
153         setattr(self.slider_3, "id", 3)
154
155     self.slider_1.setMaximum(1510)
156     self.slider_1.setMinimum(0)
157     self.slider_1.setTickInterval(377)
158     self.slider_1.setTickPosition(3)
159     self.slider_1.setSliderPosition(900)
160
161     self.slider_2.setMaximum(1350)
162     self.slider_2.setMinimum(0)
163     self.slider_2.setTickInterval(337)
164     self.slider_2.setTickPosition(3)
165
166     self.slider_3.setMaximum(1200)
167     self.slider_3.setMinimum(100)
168     self.slider_3.setTickInterval(275)
169     self.slider_3.setTickPosition(3)
170
171     self.slider_1.valueChanged.connect(lambda: self.adjust_widget_value("slider", sliders,
172                                         spin_boxes, graphics,
173                                         self.combo_box_coordinates.currentIndex(), 1))
173     self.slider_2.valueChanged.connect(lambda: self.adjust_widget_value("slider", sliders,
174                                         spin_boxes, graphics,
175                                         self.combo_box_coordinates.currentIndex(), 2))
175     self.slider_3.valueChanged.connect(lambda: self.adjust_widget_value("slider", sliders,
176                                         spin_boxes, graphics,
177                                         self.combo_box_coordinates.currentIndex(), 3))
177
178     if getattr(self.spin_box_1, "id", None) is None:
179         setattr(self.spin_box_1, "id", 1)
180
181     if getattr(self.spin_box_2, "id", None) is None:
182         setattr(self.spin_box_2, "id", 2)
183
184     if getattr(self.spin_box_3, "id", None) is None:
185         setattr(self.spin_box_3, "id", 3)
186
187     self.spin_box_1.setRange(0,151.0)
188     self.spin_box_1.setSingleStep(0.1)
189     self.spin_box_1.setValue(90.0)
190     self.spin_box_2.setRange(0,135.0)
191     self.spin_box_2.setSingleStep(0.1)
192     self.spin_box_3.setRange(10.0, 120.0)
193     self.spin_box_3.setSingleStep(0.1)
194
195     self.spin_box_1.valueChanged.connect(lambda: self.adjust_widget_value("spinBox",
196                                         sliders, spin_boxes, graphics,
197                                         self.combo_box_coordinates.currentIndex(), 1))
196     self.spin_box_2.valueChanged.connect(lambda: self.adjust_widget_value("spinBox",
197                                         sliders, spin_boxes, graphics,
198                                         self.combo_box_coordinates.currentIndex(), 2))
```

```

199     self.spin_box_3.valueChanged.connect(lambda: self.adjust_widget_value("spinBox",
200                                         sliders, spin_boxes, graphics,
201                                         self.combo_box_coordinates.currentIndex(), 3))
202
203     self.slider_2_mid_label.hide()
204
205     self.combo_box_coordinates.activated.connect(lambda index: self.switch_coordinate_menu(
206                                         self.combo_box_coordinates,
207                                         sliders_labels, sliders, spin_boxes, index)
208 )
209
210     self.menu_port.triggered.connect(lambda port_id: self.set_serial_port(port_id))
211
212     self.menu_info.triggered.connect(lambda action: self.open_browser_info(action))
213
214     if getattr(self.execute_button, "State", None) is None:
215         setattr(self.execute_button, "State", True)
216     self.execute_button.clicked.connect(lambda: self.execute_movement(self.execute_button,
217                                         self.logger_box, spin_boxes,
218                                         self.combo_box_coordinates.currentIndex()))
219     self.origin_button.clicked.connect(lambda: self.move_to_origin(self.origin_button,
220                                         sliders, spin_boxes,
221                                         self.combo_box_coordinates.currentIndex()))
222
223     self.log.info("Welcome to the p-Arm GUI")
224     self.log.info("The arm is now being initialized...")
225
226     self.top_view.setBackground("w")
227     self.side_view.setBackground("w")
228
229     self.top_view.setXRange(-400, 400, padding = 0)
230     self.top_view.invertX(True)
231     self.top_view.setYRange(280,-120, padding = 0)
232     pen = pyqtgraph.mkPen(color=(0, 255, 0), width=8, style = QtCore.Qt.SolidLine)
233     self.draw_view_from_angle(graphics, spin_boxes,1)
234     self.side_view.setXRange(-420, 420, padding = 0)
235     self.side_view.setYRange(261, -133.2, padding = 0)
236     self.draw_view_from_angle(graphics, spin_boxes,3)
237
238     self.top_view.mousePressEvent = self.enable_mouse_control
239     self.top_view.mouseMoveEvent = self.top_view_mouse_control
240     self.top_view.mouseReleaseEvent = self.disable_mouse_control
241
242     self.side_view.mousePressEvent = self.enable_mouse_control
243     self.side_view.mouseMoveEvent = self.side_view_mouse_control
244     self.side_view.mouseReleaseEvent = self.disable_mouse_control
245
246     self.scan_serial_ports(self.menu_port)
247
248 def closeEvent(self, event):
249     """
250     This method handles the closing of the GUI app.
251
252     :param event: Event generated when the user attempts to close the GUI app.
253                 This event may be accepted or declined.
254     """

```

```
250     ft = self.handler.cancel_movement()
251     msg = QMessageBox()
252     msg.setWindowTitle("Application Shut down")
253     msg.setText("The arm application will be closed and communications with the pArm will
be stopped.")
254     msg.setIcon(QMessageBox.Information)
255     msg.setStandardButtons(QMessageBox.Close)
256     msg.exec_()
257     ft.add_done_callback(lambda _: event.accept())
258
259 def enable_mouse_control(self, event):
260     """
261         This method handles the event generated when the user clicks on the graphic_view widget
262
263         When this event happens, the mouse enabler is set to True.
264
265         :param event: Event generated when the user clicks on the graphical representation of
the arm
266         """
267         self.mouse_enabler = True
268
269 def top_view_mouse_control(self, event):
270     """
271         This method handles the event generated when the moves the mouse within the
top_view graphical widget. This method performs the calculations needed to
272         update the graphical representation of the arm, as well as the sliders's and
273         spinboxes's value.
274
275         :param event: Event generated when the user moves the mouse within
the top view widget.
276         """
277
278     if self.mouse_enabler:
279         y_coord = event.x()
280         x_coord = event.y()
281
282         x_coord = 106.75 - x_coord
283         x_coord *= (450/170)
284
285         y_coord = y_coord - 197
286         y_coord *= (930/350)
287
288         if self.combo_box_coordinates.currentIndex() == 1:
289             self.spin_box_1.setValue(x_coord)
290             self.spin_box_2.setValue(-y_coord) #negative y coord due to inverted Y axis on
top view
291         elif self.combo_box_coordinates.currentIndex() == 0:
292             angles = inverse_kinematics(x_coord, -y_coord, self.spin_box_3.value())
293             if angles:
294                 thetas_0, theta_1, theta_2 = angles
295                 self.spin_box_1.setValue(theta_0)
296             else:
297                 pass
298
299 def side_view_mouse_control(self, event):
300     """
301         This method handles the event generated when the moves the mouse within the
```

```
302     side view graphical widget. This method performs the calculations needed to
303     update the graphical representation of the arm, as well as the sliders's and
304     spinboxes's value.
305
306     :param event: Event generated when the user moves the mouse within
307     the side view widget.
308     """
309     if self.mouse_enabler:
310         x2_coord = event.x()
311         z_coord = event.y()
312
313         z_coord = 109.75 - z_coord
314         z_coord *= (450/170)
315
316         x2_coord = x2_coord - 193
317         x2_coord *= (930/350)
318
319         if self.combo_box_coordinates.currentIndex() == 1:
320             self.spin_box_1.setValue(x2_coord)
321             self.spin_box_3.setValue(z_coord)
322         elif self.combo_box_coordinates.currentIndex() == 0:
323             angles = inverse_kinematics(x2_coord,
324                                         self.spin_box_2.value(),
325                                         z_coord)
326             if angles:
327                 thetas_0, theta_1, theta_2 = angles
328                 self.spin_box_2.setValue(theta_1)
329                 self.spin_box_3.setValue(theta_2)
330             else:
331                 pass
332
333     def disable_mouse_control(self, _):
334         """
335         This method handles the event generated when the users stop moving the mouse within
336         the graphic view widgets. When this happens, the mouse enabler flag is set to false.
337
338         :param event: Event generated when the user stop controlling the arm by using the mouse
339         """
340         self.mouse_enabler = False
341
342     def adjust_widget_value(self, type, sliders: QtWidgets.QSlider, spin_boxes: QtWidgets.QDoubleSpinBox,
343                           graphics: QtWidgets.QGraphicsView, index: int, id):
344         """
345         This method adjust the value of the spinboxes when the sliders are moved and viceversa.
346         When this happens, the graphical representation of the arm is uptaded. This function is
347         called when a signal is emitted.
348
349         :param type: Indicates if the widget that has been changed is a slider or a spinbox.
350         :param sliders: List of all the slider widgets.
351         :param spin_boxes: List of all the spinboxes widgets.
352         :param graphics: List of all the graphic view widgets.
353         :param index: Indicates if the coordinates systems are angular or cartesian.
354         :param id: Indicates the indentifier of the widget that changes its value.
355         """
```

```
356     if type == "slider":
357         if index == 0:
358             self.draw_view_from_angle(graphics, spin_boxes, id)
359         elif index == 1:
360             self.draw_view_from_cartesian(graphics, spin_boxes, id)
361             spin_boxes[id-1].setValue(sliders[id-1].value()/10)
362         elif type == "spinBox":
363             sliders[id-1].setSliderPosition(spin_boxes[id-1].value()*10)
364
365 def label_color_change(self, label: QtWidgets.QLabel, r, g, b):
366     """
367     This method is used to change the color of a given label.
368
369     :param label: label who's color is going to be changed
370     :param (r,g,b): rgb code of the new color
371     """
372
373     palette = QtGui.QPalette()
374     brush = QtGui.QBrush(QtGui.QColor(r, g, b))
375     brush.setStyle(QtCore.Qt.SolidPattern)
376     palette.setBrush(QtGui.QPalette.Active, QtGui.QPalette.WindowText, brush)
377     brush = QtGui.QBrush(QtGui.QColor(0, 85, 120))
378     brush.setStyle(QtCore.Qt.SolidPattern)
379     palette.setBrush(QtGui.QPalette.Inactive, QtGui.QPalette.WindowText, brush)
380     brush = QtGui.QBrush(QtGui.QColor(120, 120, 120))
381     brush.setStyle(QtCore.Qt.SolidPattern)
382     palette.setBrush(QtGui.QPalette.Disabled, QtGui.QPalette.WindowText, brush)
383     label.setPalette(palette)
384
385 def set_angular_highlight(self, sliders_labels: QtWidgets.QLabel, sliders: QtWidgets.QSlider,
386                         spin_boxes: QtWidgets.QDoubleSpinBox):
387     """
388     This method highlights the sliders's labels when the coordinates combobox selection
389     is changed to angular. This functionality has not been used in the final version of the
390     GUI.
391
392     :param sliders_labels: List of all the labels that are located close to the sliders.
393     :param sliders: List of all the slider widgets.
394     :param spin_boxes: List of all the spinboxes widgets.
395     """
396
397     sliders_labels[0].setText("Base Servo Angle")
398     sliders_labels[1].setText("Shoulder Servo Angle")
399     sliders_labels[2].setText("Elbow Servo Angle")
400     self.label_color_change(sliders_labels[0], 245, 110, 110)
401     self.label_color_change(sliders_labels[1], 245, 110, 110)
402     self.label_color_change(sliders_labels[2], 245, 110, 110)
403
404 def set_cartesian_highlight(self, sliders_labels: QtWidgets.QLabel, sliders: QtWidgets.QSlider,
405                           spin_boxes: QtWidgets.QDoubleSpinBox):
406     """
407     This method highlights the sliders's labels when the coordinates combobox selection
408     is changed to cartesian. This functionality has not been used in the final version of
409     the GUI.
410
411     :param sliders_labels: List of all the labels that are located close to the sliders.
```

```
408     :param sliders: List of all the slider widgets.
409     :param spin_boxes: List of all the spinboxes widgets.
410     """
411     sliders_labels[0].setText("X Coordinate")
412     sliders_labels[1].setText("Y Coordinate")
413     sliders_labels[2].setText("Z Coordinate")
414     self.label_color_change(sliders_labels[0],245,110,110)
415     self.label_color_change(sliders_labels[1],245,110,110)
416     self.label_color_change(sliders_labels[2],245,110,110)
417
418     def set_angular_menu(self,sliders_labels: QtWidgets.QLabel, sliders: QtWidgets.QSlider,
419                         spin_boxes: QtWidgets.QDoubleSpinBox):
420     """
421         This method changes the GUI coordinate system to angular, this means that sliders and
422         spinboxes
423         represent the theta_i angles of the arm.
424
425         :param sliders_label: List of all the labels that are located close to the sliders.
426         :param sliders: List of all the slider widgets.
427         :param spin_boxes: List of all the spinboxes widgets.
428         """
429         self.label_color_change(sliders_labels[0],212,0,0)
430         self.label_color_change(sliders_labels[1],212,0,0)
431         self.label_color_change(sliders_labels[2],212,0,0)
432
433         self.top_view.clear()
434         self.side_view.clear()
435
436         sliders_labels[0].setText("Base Servo Angle")
437         sliders_labels[1].setText("Shoulder Servo Angle")
438         sliders_labels[2].setText("Elbow Servo Angle")
439         sliders_labels[3].setText("0°")
440         sliders_labels[4].setText("151°")
441         sliders_labels[5].setText("0°")
442         sliders_labels[6].setText("135°")
443         sliders_labels[7].setText("10°")
444         sliders_labels[8].setText("120°")
445         sliders_labels[9].hide()
446
447         sliders[0].setMaximum(1510)
448         sliders[0].setMinimum(0)
449         sliders[0].setTickInterval(377)
450         sliders[0].setSliderPosition(0)
451         spin_boxes[0].setRange(0,151.0)
452         spin_boxes[0].setValue(0.0)
453
454         sliders[1].setMaximum(1350)
455         sliders[1].setMinimum(0)
456         sliders[1].setTickInterval(337)
457         sliders[1].setSliderPosition(0)
458         spin_boxes[1].setRange(0,135.0)
459         spin_boxes[1].setValue(0.0)
460
461         sliders[2].setMaximum(1200)
462         sliders[2].setMinimum(100)
463         sliders[2].setTickInterval(275)
```

```
463     sliders[2].setSliderPosition(0)
464     spin_boxes[2].setRange(10.,110.0)
465     spin_boxes[2].setValue(0.0)
466
467 def set_cartesian_menu(self,sliders_labels: QtWidgets.QLabel, sliders: QtWidgets.QSlider,
468                         spin_boxes: QtWidgets.QDoubleSpinBox):
469     """
470         This method changes the GUI coordinate system to angular, this means that sliders and
471         spinboxes
472         represent the cartesian position of the arm's end effector.
473
474     :param sliders_label: List of all the labels that are located close to the sliders.
475     :param sliders: List of all the slider widgets.
476     :param spin_boxes: List of all the spinboxes widgets.
477     """
478     self.label_color_change(sliders_labels[0],212,0,0)
479     self.label_color_change(sliders_labels[1],212,0,0)
480     self.label_color_change(sliders_labels[2],212,0,0)
481
482     self.top_view.clear()
483     self.side_view.clear()
484
485     sliders_labels[0].setText("X Coordinate")
486     sliders_labels[1].setText("Y Coordinate")
487     sliders_labels[2].setText("Z Coordinate")
488     sliders_labels[3].setText("0.0mm")
489     sliders_labels[4].setText("346.0mm")
490     sliders_labels[5].setText("-346.0mm")
491     sliders_labels[6].setText("346.0mm")
492     sliders_labels[7].setText("-106.1mm")
493     sliders_labels[8].setText("360.6mm")
494     sliders_labels[9].show()
495
496     sliders[0].setMaximum(3460)
497     sliders[0].setMinimum(0)
498     sliders[0].setTickInterval(865)
499     sliders[0].setSliderPosition(0)
500     spin_boxes[0].setRange(0,346.0)
501     spin_boxes[0].setValue(0)
502
503     sliders[1].setMaximum(3460)
504     sliders[1].setMinimum(-3460)
505     sliders[1].setTickInterval(1730)
506     sliders[1].setSliderPosition(0)
507     spin_boxes[1].setRange(-346.0,346.0)
508     spin_boxes[1].setValue(0)
509
510     sliders[2].setMaximum(3606)
511     sliders[2].setMinimum(-1061)
512     sliders[2].setTickInterval(901)
513     sliders[2].setSliderPosition(0)
514     spin_boxes[2].setRange(-106.1,360.6)
515     spin_boxes[2].setValue(0)
516
517 def coordinates_highlight(self, sliders_labels: QtWidgets.QLabel, sliders: QtWidgets.QSlider
518 ,
```

```
517             spin_boxes: QtWidgets.QDoubleSpinBox, index):
518         """
519             This method highlights the sliders's labels when the coordinate system is changed from
520             angular
521                 to cartesian and viceversa. This functionality has not been used in the final version
522                 of the GUI.
523
524             :param sliders_label: List of all the labels that are located close to the sliders.
525             :param sliders: List of all the slider widgets.
526             :param spin_boxes: List of all the spinboxes widgets.
527             :param index: Indicates whether the coordinates system is angular or cartesian
528             """
529             if index == 1:
530                 self.set_cartesian_highlight(sliders_labels, sliders, spin_boxes)
531             elif index == 0:
532                 self.set_angular_highlight(sliders_labels, sliders, spin_boxes)
533
534     def switch_coordinate_menu(self, combo_box: QtWidgets.QComboBox, sliders_labels: QtWidgets.QLabel,
535                               sliders: QtWidgets.QSlider, spin_boxes: QtWidgets.QDoubleSpinBox, index):
536         """
537             This method switch the coordinate from angular to cartesian and viceversa when the
538             combobox selection
539             changes.
540
541             :param combo_box: Combo box coordinate selection.
542             :param sliders_label: List of all the labels that are located close to the sliders.
543             :param sliders: List of all the slider widgets.
544             :param spin_boxes: List of all the spinboxes widgets.
545             :param index: Indicates whether the coordinates system is angular or cartesian
546             """
547             if index == 1:
548                 self.set_cartesian_menu(sliders_labels, sliders, spin_boxes)
549             elif index == 0:
550                 self.set_angular_menu(sliders_labels, sliders, spin_boxes)
551
552     def show_popup(self, message: str):
553         """
554             This method generates a warning pop that shows a given prompt.
555
556             :param message: string that is going to be showed up in the pop up.
557             """
558             msg = QMessageBox()
559             msg.setWindowTitle("Warning")
560             msg.setText(message)
561             msg.setIcon(2)
562             msg.exec_()
563
564     def execute_movement(self,
565                          button: QtWidgets.QPushButton,
566                          logger: QtWidgets.QPlainTextEdit,
567                          spin_boxes: QtWidgets.QDoubleSpinBox,
568                          index: int):
569         """
570             This method is in charge of executing/canceling a movement by communicating it to
571             the communications code through the handler object. This method is connected to a
```

```
569     button widget, which triggers the execution of this function.  
570  
571     When this function is executed, a future object is generated in orden to not freeze  
572     the GUI execution thread. When S2 completes the movement, the future object is trully  
573     returned, indicating if the movement was completed succesfully or if there was an error  
574  
575     :param button: button widget.  
576     :param logger: logger object instance, that is used to generate logs.  
577     :param spin_boxes: spinboxes object, used to get the values selected by the user.  
578     :param index: indicates whether the GUI coordinates are angular or cartesian.  
579     """  
580  
581     if button.State:  
582         self.progress_bar.show()  
583         button.setText("Cancel movement")  
584         button.State = False  
585         time_holder_val = AtomicFloat(initial_value=-1)  
586         self.progress_bar.run_worker(time_holder_val)  
587         ft = None  
588         if index == 0:  
589             ft = self.handler.move_to_thetas(spin_boxes[0].value(),  
590                                             spin_boxes[1].value(),  
591                                             spin_boxes[2].value(),  
592                                             time_holder_val)  
593             self.log.info(f'Sending joints to pArm: {{'  
594             f't0: {spin_boxes[0].value()}, '  
595             f't1: {spin_boxes[1].value()}, '  
596             f't2: {spin_boxes[2].value()}}}')  
597         elif index == 1:  
598             ft = self.handler.move_to_xyz(spin_boxes[0].value(),  
599                                         spin_boxes[1].value(),  
600                                         spin_boxes[2].value(),  
601                                         time_holder_val)  
602             self.log.info(f'Sending joints to pArm: {{'  
603             f'x: {spin_boxes[0].value()}, '  
604             f'y: {spin_boxes[1].value()}, '  
605             f'z: {spin_boxes[2].value()}}}')  
606         if ft:  
607             ft.add_done_callback(lambda fut: self.future_callback(fut))  
608     else:  
609         self.progress_bar.hide()  
610         self.handler.cancel_movement()  
611         self.show_popup("Movement Cancelled")  
612         self.log.warning('Movement cancelled!')  
613         button.setText("Execute Movement")  
614         button.State = True  
615  
616     def draw_view_from_angle(self,  
617                             graphics: QtWidgets.QGraphicsView,  
618                             spin_boxes: QtWidgets.QDoubleSpinBox, _):  
619     """  
620         This method perform the calculations need to draw the arm preview. This  
621         calculations are made from the angles selected by the user and by using  
622         the shortened direct kinematics model.  
623     """
```

```
624     :param graphics: List of graphic views widgets.
625     :param spin_boxes: List of spinboxes widgets.
626     """
627
628     t0, t1, t2 = spin_boxes[0].value(), \
629                  spin_boxes[1].value(), \
630                  spin_boxes[2].value()
631
632     math_trans = math.pi / 180
633     x_coord1 = 142.07 * math.cos((135 - t1) * math_trans)
634     x_coord2 = x_coord1 + 158.81 * \
635                 math.cos((180 - (135 - t1) - t2) * math_trans)
636     z_coord1 = 142.07 * math.sin((135 - t1) * math_trans)
637     z_coord2 = z_coord1 - 158.81 * \
638                 math.sin((180 - (135 - t1) - t2) * math_trans)
639
640     y_coord = x_coord2 * math.cos(t0 * math_trans)
641     x_coord = x_coord2 * math.sin(t0 * math_trans)
642     y1_coord = x_coord1 * math.cos(t0 * math_trans)
643     x1_coord = x_coord1 * math.sin(t0 * math_trans)
644
645     graphics[0].clear()
646     rect_item = RectItem(QtCore.QRectF(-60, -60, 120, 120))
647     graphics[0].addItem(rect_item)
648
649     if self.check_list(t0, t1, t2, x_coord, y_coord, z_coord2):
650         pen1 = pyqtgraph.mkPen(color=(0, 240, 0),
651                                width=8,
652                                style=QtCore.Qt.SolidLine)
653         pen2 = pyqtgraph.mkPen(color=(0, 220, 215),
654                                width=8,
655                                style=QtCore.Qt.SolidLine)
656         self.disable_execute_button(True)
657     else:
658         pen1 = pyqtgraph.mkPen(color=(255, 0, 0),
659                                width=8,
660                                style=QtCore.Qt.SolidLine)
661         pen2 = pyqtgraph.mkPen(color=(255, 0, 0),
662                                width=8,
663                                style=QtCore.Qt.SolidLine)
664         self.disable_execute_button(False)
665
666     # Upper arm above Lower Arm
667     if z_coord2 > z_coord1 and x_coord2 > x_coord1:
668         graphics[0].plot((0, y1_coord), (0, x1_coord),
669                           pen=pen1, symbol='o',
670                           symbolSize=15, symbolBrush='b')
671         graphics[0].plot((y1_coord, y_coord), (x1_coord, x_coord),
672                           pen=pen2, symbol='o',
673                           symbolSize=15, symbolBrush='b')
674     # Lowe arm above Upper Arm
675     elif z_coord2 < z_coord1 and x_coord2 < x_coord1:
676         graphics[0].plot((y1_coord, y_coord), (x1_coord, x_coord),
677                           pen=pen2, symbol='o',
678                           symbolSize=15, symbolBrush='b')
679         graphics[0].plot((0, y1_coord), (0, x1_coord),
```



```
736         self.disable_execute_button(False)
737     else:
738         pen1 = pyqtgraph.mkPen(color=(0, 240, 0),
739                                width=8,
740                                style=QtCore.Qt.SolidLine)
741         pen2 = pyqtgraph.mkPen(color=(0, 220, 215),
742                                width=8,
743                                style=QtCore.Qt.SolidLine)
744         self.disable_execute_button(True)
745
746     math_trans = math.pi / 180
747     x_coord1 = 142.07 * math.cos((135 - t1) * math_trans)
748     x_coord2 = x_coord1 + 158.08 * \
749                 math.cos((180 - (135 - t1) - t2) * math_trans)
750     z_coord1 = 142.07 * math.sin((135 - t1) * math_trans)
751     z_coord2 = z_coord1 - 158.08 * \
752                 math.sin((180 - (135 - t1) - t2) * math_trans)
753
754     mid_x = x_coord1 * math.sin(t0 * math_trans)
755     mid_y = x_coord1 * math.cos(t0 * math_trans)
756
757     graphics[0].clear()
758     rect_item = RectItem(QtCore.QRectF(-53.05, -53.05, 106.1, 106.1))
759     graphics[0].addItem(rect_item)
760
761     # Upper arm above Lower arm
762     if z_coord2 > z_coord1 and x_coord2 > x_coord1:
763         graphics[0].plot((0, mid_y), (0, mid_x),
764                           pen=pen1, symbol='o',
765                           symbolSize=15, symbolBrush='b')
766         graphics[0].plot((mid_y, y_coord), (mid_x, x_coord),
767                           pen=pen2, symbol='o',
768                           symbolSize=15, symbolBrush='b')
769     # Lowe arm above Upper arm
770     elif z_coord2 < z_coord1 and x_coord2 < x_coord1:
771         graphics[0].plot((mid_y, y_coord), (mid_x, x_coord),
772                           pen=pen1, symbol='o',
773                           symbolSize=15, symbolBrush='b')
774         graphics[0].plot((0, mid_y), (0, mid_x),
775                           pen=pen1, symbol='o',
776                           symbolSize=15, symbolBrush='b')
777     # neutral position
778     else:
779         graphics[0].plot((0, mid_y), (0, mid_x),
780                           pen=pen1, symbol='o',
781                           symbolSize=15, symbolBrush='b')
782         graphics[0].plot((mid_y, y_coord), (mid_x, x_coord),
783                           pen=pen2, symbol='o',
784                           symbolSize=15, symbolBrush='b')
785
786     graphics[1].clear()
787     rect_item2 = RectItem(QtCore.QRectF(-60, -133.2, 120, 113.2))
788     rect_item3 = RectItem(QtCore.QRectF(-36, -20, 72, 20))
789     graphics[1].addItem(rect_item2)
790     graphics[1].addItem(rect_item3)
791     graphics[1].plot((0, x_coord1),
```

```
792             (0, z_coord1),
793             pen=pen1,
794             symbol='o',
795             symbolSize=15,
796             symbolBrush='b')
797         graphics[1].plot((x_coord1, x_coord2),
798                           (z_coord1, z_coord2),
799                           pen=pen2,
800                           symbol='o',
801                           symbolSize=15,
802                           symbolBrush='b')
803
804     def scan_serial_ports(self, menu: QMenu):
805         """
806             This method performs a scan of the available serial ports of the PC
807             that is executing the GUI app, and then, add these available ports to
808             the port selection menu of the GUI.
809
810             :param menu: port selection menu object of the GUI app.
811             """
812             port_list = serial.tools.list_ports.comports()
813             if len(port_list) == 0:
814                 menu.addAction('No ports available')
815                 self.log.warning('No ports available - Check out the connections')
816
817             for port in port_list:
818                 menu.addAction(port.device)
819                 self.log.info(f'Port {port.device} detected & ready')
820
821     def set_serial_port(self, port_id: QAction):
822         """
823             This method is used to set the serial port that is going to
824             be used to communicate the GUI app with the PCB. When the user
825             select a port in the GUI, this function is executed and
826             the port selected is passed to the logic communication code
827             using the handler object.
828
829
830             :param port_id: indicates the port that was selected by the user.
831             """
832             self.port = port_id.iconText()
833             self.handler.port = self.port
834             if not (self.port == 'No ports available'):
835                 self.log.info(f'Port {self.port} selected as communication bay')
836                 self.handler.do_handshake()
837             else:
838                 self.log.warning('No ports available - Check out the connections')
839
840     def future_callback(self, ft: Future):
841         """
842             This method is executed when the future object created in the
843             execute movement process is finally returned, which means that
844             this method is a call back.
845
846             When the future object is finally returned, the GUI receives it
847             and decides whether the movement was completed successfully or
```

```
848     if there was an error.
849
850     If the future object is an error type object, the GUI shows a pop
851     up that notifies the user.
852
853     If the future object is an control type object, the GUI updates the
854     value of the sliders and spin box with the data provided by the PCB
855     and notifies the user that the movement was completed successfully.
856
857     :param ft: Future object instance that was finally returned.
858     """
859     res = ft.result()
860     self.progress_bar.hide()
861     if isinstance(res, ErrorData):
862         self.show_popup(res.err_msg)
863         self.execute_button.State = 0
864         self.execute_button.setText("Execute Movement")
865         self.log.error(f'Error happened during movement: {res.err_msg}')
866     elif isinstance(res, ControlInterface):
867         self.log.info('Movement was completed successfully')
868         if self.combo_box_coordinates.currentIndex() == 0:
869             self.spin_box_1.setValue(res.theta1)
870             self.spin_box_2.setValue(res.theta2)
871             self.spin_box_3.setValue(res.theta3)
872         elif self.combo_box_coordinates.currentIndex() == 1:
873             self.spin_box_1.setValue(res.x)
874             self.spin_box_2.setValue(res.y)
875             self.spin_box_3.setValue(res.z)
876
877     def move_to_origin(self,
878                         button: QtWidgets.QPushButton,
879                         sliders: QtWidgets.QSlider,
880                         spin_boxes: QtWidgets.QDoubleSpinBox,
881                         index):
882     """
883     This method is used to return the sliders, spinboxes and graphic
884     views to its origin position, taking into account if the coordinates
885     are angular or cartesian
886
887     :param button: Button widget assigned to origin event.
888     :param sliders: List of all sliders.
889     :param spin_boxes: List of all spinboxes
890     :param index: indicates if the coordinates are angular or cartesian
891     """
892     if index == 0:
893         spin_boxes[0].setValue(90)
894         spin_boxes[1].setValue(0)
895         spin_boxes[2].setValue(0)
896     elif index == 1:
897         spin_boxes[0].setValue(0)
898         spin_boxes[1].setValue(0)
899         spin_boxes[2].setValue(0)
900     self.log.info('The arm was sent to its origin position')
901
902     def disable_execute_button(self, enabler):
903         """
```

```
904     This method is used to disable/enable the execute movement button and to
905     notify the user that the current selected position is unreachable.
906
907     :param enabler: flag that indicates if the button is enabled or not.
908     """
909     if not enabler:
910         self.execute_button.setText("Unreachable position")
911         self.execute_button.setEnabled(False)
912     else:
913         self.execute_button.setText("Execute Movement")
914         self.execute_button.setEnabled(True)
915
916 def open_browser_info(self, action: QAction):
917     """
918     This method is used open some links using the browser when the users click
919     the 'extra info' menu.
920
921     :param action: flag that indicates if the button is enabled or not.
922     """
923     if action.iconText() == 'GitHub project':
924         webbrowser.open("https://github.com/pArm-TFG")
925     elif action.iconText() == 'Documentation':
926         webbrowser.open('https://github.com/pArm-TFG/Memoria')
927     elif action.iconText() == 'About us':
928         webbrowser.open('https://www.linkedin.com/in/jose-alejandro-moya-blanco-78952a126/')
929     )
930         webbrowser.open('https://www.linkedin.com/in/javinator9889/')
931         webbrowser.open('https://www.linkedin.com/in/mihai-octavian-34865419b/')
932
933 def check_list(self, theta_0, theta_1, theta_2, x_coord, y_coord, z_coord):
934     """
935     This method is used to verify whether a given arm's position is
936     reachable
937     or not.
938     :param theta_i coords: angular coordinates of the arm.
939     :param x, y, z coords: cartesian coordinate of the arm.
940     """
941     result = True
942     if theta_0 > 151:
943         result = False
944         if self.counter == 200:
945             self.log.warning('Base joint angle (t0) is over 151°')
946             self.counter = 0
947             self.counter += 1
948
949     if theta_1 > 135:
950         result = False
951         if self.counter == 200:
952             self.log.warning('Shoulder joint angle (t1) is over 135°')
953             self.counter = 0
954             self.counter += 1
955
956     if theta_2 > 120:
957         result = False
958         if self.counter == 200:
959             self.log.warning('Elbow joint angle (t2) is over 120°')
```

```

959         self.counter = 0
960         self.counter += 1
961
962     if math.sqrt(x_coord**2 + y_coord**2 + z_coord**2) > 261:
963         result = False
964
965     if theta_2 > (theta_1 + 55):
966         result = False
967     if self.counter == 200:
968         self.log.critical('Physical structure limit!')
969         self.counter = 0
970     self.counter += 1
971
972     if 60 > x_coord > -60 and z_coord < 0 and 60 > y_coord > -60:
973         result = False
974     if self.counter == 200:
975         self.log.critical('End-effector colliding with pArm base')
976         self.counter = 0
977     self.counter += 1
978
979 return result

```

Listing E.8: pArm-S1/pArm/GUI/GUI.py

```

1 #                                     pArm-S1
2 #
3 # This program is free software: you can redistribute it and/or modify
4 # it under the terms of the GNU General Public License as published by
5 # the Free Software Foundation, either version 3 of the License, or
6 # (at your option) any later version.
7 #
8 # This program is distributed in the hope that it will be useful,
9 # but WITHOUT ANY WARRANTY; without even the implied warranty of
10 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
11 # GNU General Public License for more details.
12 #
13 # You should have received a copy of the GNU General Public License
14 # along with this program. If not, see <http://www.gnu.org/licenses/>.
15 from __future__ import annotations
16 from PyQt5 import QtCore
17 from PyQt5.QtCore import QObject
18 from PyQt5.QtWidgets import QProgressBar
19 from time import time, sleep
20 from typing import Tuple, Optional, Union
21 from ..utils import AtomicFloat
22
23
24 class Worker(QObject):
25     """
26     Worker class for counting the remaining time until a given value. Provides
27     three signals for handling when a new value is set, the lower and upper
28     limits and when it has finished.
29
30     Those signals are:
31     - ``update_progress [float]``
32     - ``limit_values [tuple of floats]``

```

```
33     - “finished [bool]“
34 """
35 update_progress = QtCore.pyqtSignal(float)
36 """ Signal called when a new value is available """
37
38 limit_values = QtCore.pyqtSignal(tuple)
39 """ Signal called when the limits are known """
40
41 finished = QtCore.pyqtSignal(bool)
42 """ Signal called when the processing has finished """
43
44 def __init__(self, time_object: AtomicFloat):
45     super().__init__()
46     self.time = time_object
47
48 def work(self):
49 """
50     Notifies the worker to start. Waits until a value is available
51     in the ‘‘AtomicFloat’‘ object (defaults to -1.0) and then emits the
52     lower and upper limit. Finally, starts counting the elapsed time and
53     notifies through ‘‘update_progress’‘ the new value that must be
54     displayed. At the end, notifies that it has finished processing
55     and quits.
56 """
57     while not self.time.value != -1.0:
58         sleep(.01)
59
60     self.limit_values.emit((0, self.time.value))
61     start_time = time()
62     finish_time = start_time + self.time.value
63     while time() < finish_time:
64         self.update_progress.emit(time() - start_time)
65
66     self.finished.emit(True)
67
68
69 class ProgressWidget(QProgressBar):
70 """
71     ProgressBar class that inherits from QProgressBar implementing an
72     anonymous worker for updating the UI with a given value.
73
74     This class basically works the same as QProgressBar but with the addition
75     of the two following methods:
76     - ‘‘create_worker(time_object: AtomicFloat)“
77     - ‘‘run_worker(time_object: Optional[AtomicFloat])“
78
79     Those methods must be called in order to update the progress bar by the
80     value stored in ‘‘time_object’‘.
81 """
82
83 @classmethod
84 def from_bar(cls,
85             progress_bar: Union[QObject, QProgressBar]) -> ProgressWidget:
86     return cls(base=progress_bar)
87
88 def __init__(self, base: Optional[QProgressBar] = None):
```

```
89     super().__init__()
90     self.__base = base
91     self._worker: Optional[Worker] = None
92     self._thread: Optional[QtCore.QThread] = None
93
94     def __getattr__(self, attr):
95         if self.__base:
96             return getattr(self.__base, attr)
97         return getattr(self, attr)
98
99     def __setattr__(self, attr, value):
100        if attr == '_ProgressWidget__base':
101            return object.__setattr__(self, attr, value)
102
103        if self.__base:
104            return setattr(self.__base, attr, value)
105        return setattr(self, attr, value)
106
107    def __getattribute__(self, item):
108        if item == '_ProgressWidget__base':
109            return super(ProgressWidget, self).__getattribute__(item)
110        if hasattr(self.__base, item):
111            return object.__getattribute__(self.__base, item)
112        return super(ProgressWidget, self).__getattribute__(item)
113
114    def create_worker(self, time_object: AtomicFloat):
115        """
116            Creates a new worker overwriting any existing one. A reference to an
117            existing “time_object” must be given every time the worker is created.
118
119            This function attaches signals for knowing which are the limits for the
120            progress bar, the new value that must be shown and whether it reaches
121            100%.
122
123            :param time_object: the atomic float holder value.
124            :type time_object: AtomicFloat
125            """
126
127        thread = QtCore.QThread()
128        worker = Worker(time_object)
129        worker.update_progress.connect(
130            lambda progress: self.handle_progress(progress)
131        )
132        worker.moveToThread(thread)
133        thread.started.connect(worker.work)
134        worker.limit_values.connect(
135            lambda limits: self.handle_limits(limits)
136        )
137        worker.finished.connect(thread.quit)
138        QtCore.QMetaObject.connectSlotsByName(self)
139
140        self._worker = worker
141        self._thread = thread
142
143    def run_worker(self, time_object: Optional[AtomicFloat] = None) -> Worker:
144        """
145            Starts an existing worker (if any) or creates a new one. In that case,
```

```

145     this function checks if "time_object" is not "None" and then calls
146     "create_worker" function.
147
148     :param time_object: the atomic float holder value. Can be "None" if
149     "create_worker" was called before.
150     :return: the created Worker.
151     :raises ValueError: if "time_object" is None and no worker is
152     available.
153     """
154     if not self._worker:
155         if not time_object:
156             raise ValueError('When creating a worker, time_object cannot '
157                             'be None')
158             self.create_worker(time_object)
159             self._thread.start()
160             return self._worker
161
162     def handle_limits(self, limits: Tuple[float, float]):
163         """
164             Function that is called whether the running worker has received the
165             value from the "time_object".
166
167             As the worker works with "float", a conversion to "int" must be
168             done. In this case, the progress bar works with 4 decimals.
169
170             :param limits: a tuple containing both the lower limit and upper limit.
171             """
172             self.setMinimum(int(round(limits[0], 4) * 10000))
173             self.setMaximum(int(round(limits[1], 4) * 10000))
174             self.show()
175
176     def handle_progress(self, value: float):
177         """
178             Function that is called when a new value must be displayed in the
179             progress bar. When the progress bar reaches the top, it is hidden.
180
181             As the worker works with "float", a conversion to "int" must be
182             done. In this case, the progress bar works with 4 decimals.
183
184             :param value: the new value that must be displayed.
185             """
186             if value >= self.maximum():
187                 self.hide()
188             else:
189                 self.setValue(int(round(value, 4) * 10000))

```

Listing E.9: pArm-S1/pArm/GUI/progress_widget.py

```

1 #                                     pArm-S1
2 #
3 # This program is free software: you can redistribute it and/or modify
4 # it under the terms of the GNU General Public License as published by
5 # the Free Software Foundation, either version 3 of the License, or
6 # (at your option) any later version.
7 #
8 # This program is distributed in the hope that it will be useful,

```

```
9 #      but WITHOUT ANY WARRANTY; without even the implied warranty of
10 #      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
11 #      GNU General Public License for more details.
12 #
13 #      You should have received a copy of the GNU General Public License
14 #      along with this program. If not, see <http://www.gnu.org/licenses/>.
15 import pyqtgraph as qtg
16 from PyQt5 import QtCore, QtGui
17
18
19 class RectItem(qtg.GraphicsObject):
20     """
21     Custom graphics object for drawing a rectangle in Qt.
22     Receives a QRectF containing the points in which the rectangle
23     must start and must end.
24
25     :param rect: the rectangle vertices.
26     :param parent: optional parent at which is attached.
27     """
28     def __init__(self, rect: QtCore.QRectF, parent=None):
29         super(RectItem, self).__init__(parent)
30         self.__rect = rect
31         self.picture = QtGui.QPicture()
32         self._gen_picture()
33
34     @property
35     def rect(self):
36         """
37             Access the rectangular vertices item.
38             :return: QtCore.QRectF object.
39         """
40         # Read-only property
41         return self.__rect
42
43     def _gen_picture(self):
44         """
45             Generates the rectangle and paints it in pyqtgraph.
46         """
47         painter = QtGui.QPainter(self.picture)
48         painter.setPen(qtg.mkPen('w'))
49         painter.setBrush(qtg.mkBrush((130,130,130)))
50         painter.drawRect(self.rect)
51         painter.end()
52
53     def paint(self, painter: QtGui.QPainter, options, widget=None):
54         """
55             Starts painting the rectangle.
56             :param painter: the QPainter object.
57             :param options: options to pass to the painter (actually ignored).
58             :param widget: widget to use with the painter (actually ignored)
59         """
60         painter.drawPicture(0, 0, self.picture)
61
62     def boundingRect(self):
63         """
64             Generates the bounding rectangle by using the generated picture.
```

```
65
66     :return: QtCore.QRectF
67     """
68
69     return QtCore.QRectF(self.picture.boundingRect())
```

Listing E.10: pArm-S1/pArm/GUI/rect_item.py

```
1 #                                     pArm-S1
2 #
3 # This program is free software: you can redistribute it and/or modify
4 # it under the terms of the GNU General Public License as published by
5 # the Free Software Foundation, either version 3 of the License, or
6 # (at your option) any later version.
7 #
8 # This program is distributed in the hope that it will be useful,
9 # but WITHOUT ANY WARRANTY; without even the implied warranty of
10 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
11 # GNU General Public License for more details.
12 #
13 # You should have received a copy of the GNU General Public License
14 # along with this program. If not, see <http://www.gnu.org/licenses/>.
15 import logging
16 import os
17 from logging.handlers import RotatingFileHandler
18 from typing import Optional
19
20
21 __formatter = None
22
23
24 def init_logging(logger_name: Optional[str] = None,
25                  log_file: Optional[str] = None,
26                  console_level: int = logging.DEBUG,
27                  file_level: int = logging.WARNING,
28                  log_format: str = "%(process)d - %(asctime)s | [%(" +
29                               "levelname)s]: %(message)s") -> logging:
29
30     """
31     Creates a custom logging that outputs to both console and file, if
32     filename provided. Automatically cleans-up old logs during runtime and
33     allows customization of both console and file levels in addition to the
34     formatter.
35
36     :param logger_name: the logger name for later obtaining it.
37     :param log_file: a filename for saving the logs during execution - can be
38                      'None'
39     :param console_level: the logging level for console.
40     :param file_level: the logging level for the file.
41     :param log_format: the logging format.
42
43     :return: the created logging instance
44     """
45
46     global __formatter
47     __formatter = logging.Formatter(log_format)
48     logger = logging.getLogger(logger_name)
49     if getattr(logger, 'created', False):
50         return logger
```

```
50     setattr(logger, 'created', True)
51     for handler in logger.handlers:
52         if type(handler) is logging.StreamHandler:
53             handler.setLevel(console_level)
54             handler.setFormatter(__formatter)
55
56     def file_rotator(source: str, dest: str):
57         """
58             Custom file rotator for creating compressed logging files.
59
60             :param source: source filename.
61             :param dest: destination filename.
62             """
63             import gzip
64             import shutil
65
66             with open(source, "rb") as in_file:
67                 with gzip.open(dest, "wb") as out_file:
68                     shutil.copyfileobj(in_file, out_file)
69
70     def namer(name: str) -> str:
71         """
72             Custom namer implementation as we are gzipping files.
73
74             :param name: the name to append .gz
75             :return: the name with .gz extension
76             """
77             return f"{name}.gz"
78
79     if log_file:
80         old_log = os.path.exists(log_file)
81         file_handler = RotatingFileHandler(log_file,
82                                             mode='a',
83                                             maxBytes=2 << 20,
84                                             backupCount=5)
85         file_handler.rotator = file_rotator
86         file_handler.namer = namer
87         file_handler.setLevel(file_level)
88         file_handler.formatter = __formatter
89         if old_log:
90             file_handler.doRollover()
91         logger.addHandler(file_handler)
92
93     return logger
94
95
96     def add_handler(handler: logging.Handler,
97                     logger_name: Optional[str] = None,
98                     level: int = logging.DEBUG,
99                     log_format: Optional[str] = None):
100         """
101             Adds a new handler to an existing logger, with the specified formatter
102             in ``init_logging``. If a new format is specified (is not None) then
103             it will be used for this handler.
104
105             :param handler: the new handler to be added.
```

```

106     :param logger_name: the logger name to which add the handler.
107     :param level: the logging level for that formatter.
108     :param log_format: the log format used if not formatter was created.
109     """
110     logger = logging.getLogger(logger_name)
111     global __formatter
112     fmt = logging.Formatter(log_format) if log_format else __formatter
113     handler.setFormatter(fmt)
114     handler.setLevel(level)
115
116     logger.addHandler(handler)

```

Listing E.11: pArm-S1/pArm/logger/logger.py

```

1 #                                     pArm-S1
2 #
3 # This program is free software: you can redistribute it and/or modify
4 # it under the terms of the GNU General Public License as published by
5 # the Free Software Foundation, either version 3 of the License, or
6 # (at your option) any later version.
7 #
8 # This program is distributed in the hope that it will be useful,
9 # but WITHOUT ANY WARRANTY; without even the implied warranty of
10 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
11 # GNU General Public License for more details.
12 #
13 # You should have received a copy of the GNU General Public License
14 # along with this program. If not, see <http://www.gnu.org/licenses/>.
15 from logging import Handler, LogRecord
16 from PyQt5 import QtWidgets
17 from typing import Optional
18
19
20 class QTextEditLogger(Handler):
21     """
22     Custom "logging.Handler" class which outputs the log
23     to the in UI console. Receives the parent to which it is attached
24     and defaults to read-only (can be changed accessing "widget" attr).
25
26     :param edit_text: the source edit text to use if no parent specified.
27     :param parent: the parent in which the new widget will be hosted.
28     :raises AttributeError if both "edit_text" and "parent" are None.
29     """
30     def __init__(self,
31                  edit_text: Optional[QtWidgets.QPlainTextEdit] = None,
32                  parent: Optional[QtWidgets.QWidget] = None):
33         super(QTextEditLogger, self).__init__()
34         if edit_text:
35             self.widget = edit_text
36         elif parent:
37             self.widget = QtWidgets.QPlainTextEdit(parent)
38         else:
39             raise AttributeError("Either edit_text or parent must not be None")
40         self.widget.setReadOnly(True)
41
42     def emit(self, record: LogRecord):

```

```

43     """
44     Formats the record and outputs it to the plain text edit widget.
45     :param record: the record to be formatted.
46     """
47     msg = self.format(record)
48     self.widget.appendPlainText(msg)
49     self.widget.ensureCursorVisible()

```

Listing E.12: pArm-S1/pArm/logger/PyQtHandler.py

```

1 #                                     pArm-S1
2 #
3 # This program is free software: you can redistribute it and/or modify
4 # it under the terms of the GNU General Public License as published by
5 # the Free Software Foundation, either version 3 of the License, or
6 # (at your option) any later version.
7 #
8 # This program is distributed in the hope that it will be useful,
9 # but WITHOUT ANY WARRANTY; without even the implied warranty of
10 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
11 # GNU General Public License for more details.
12 #
13 # You should have received a copy of the GNU General Public License
14 # along with this program. If not, see <http://www.gnu.org/licenses/>.
15 from typing import Union
16
17
18 class RSA:
19     def __init__(self, e: int, n: int):
20         self.e = e
21         self.n = n
22
23     def verify(self, message: Union[int, str]) -> Union[int, str]:
24         """
25             This function is used to "un-sign" the control string received from the
26             arm controller.
27             :param message: the string to be "un-signed"
28             :return: the unsigned string
29         """
30         if self.n == 1:
31             return ''
32
33         if type(message) == int:
34             return right_to_left(message, self.e, self.n)
35         elif type(message) == str:
36             verified_message = list()
37             for msg_val in message.split():
38                 try:
39                     val = int(msg_val)
40                     verified_message.append(
41                         chr(right_to_left(val, self.e, self.n)))
42                 )
43             except ValueError:
44                 del verified_message
45                 raise AttributeError(
46                     f'message contents must be an integer - {msg_val} is '

```

```

47             f'invalid')
48     return ''.join(verified_message)
49 else:
50     raise AttributeError(
51         f'message must be int or str, not {type(message)}')
52
53 def encrypt(self, message: Union[int, str]) -> Union[int, str]:
54     """
55     This function is used to encrypt either an integer or str. This is
56     used mainly for encrypting the signed message sent by S2.
57
58     :param message: a string or int to be encrypted.
59     :return: the encrypted message.
60     """
61     if self.n == 1:
62         return ''
63
64     if type(message) == int:
65         return right_to_left(message, self.e, self.n)
66     elif type(message) == str:
67         encrypted_message = list()
68         for char in message:
69             encrypted_message.append(
70                 str(right_to_left(ord(char), self.e, self.n)))
71         )
72     return ' '.join(encrypted_message)
73 else:
74     raise AttributeError(
75         f'message must be int or str, not {type(message)}')
76
77
78 def right_to_left(number: int, exp: int, mod: int) -> int:
79     """
80     This function does the math of the encryption process.
81     :param number: A number to be encrypted
82     :param exp: The exponent of the mathematical operation used to do the
83     actual encryption
84     :param mod: The module of the mathematical operation used to do the
85     actual encryption
86     :return: the encrypted number.
87     """
88     ret: int = 1
89     while exp > 0:
90         if exp % 2 == 1:
91             ret = (ret * number) % mod
92         exp >>= 1
93         number = (number * number) % mod
94
95     return ret

```

Listing E.13: pArm-S1/pArm/security/rsa.py

```

1 from collections import namedtuple
2
3 ErrorData = namedtuple('ErrorData', 'err_level err_msg')

```

Listing E.14: pArm-S1/pArm/utils/error_data.py

```
1 #                                     pArm-S1
2 #
3 # This program is free software: you can redistribute it and/or modify
4 # it under the terms of the GNU General Public License as published by
5 # the Free Software Foundation, either version 3 of the License, or
6 # (at your option) any later version.
7 #
8 # This program is distributed in the hope that it will be useful,
9 # but WITHOUT ANY WARRANTY; without even the implied warranty of
10 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
11 # GNU General Public License for more details.
12 #
13 # You should have received a copy of the GNU General Public License
14 # along with this program. If not, see <http://www.gnu.org/licenses/>.
15 import itertools
16 from abc import ABC, abstractmethod
17 from threading import Lock
18 from typing import Optional, Generic, TypeVar
19
20 T = TypeVar('T')
21
22
23 class Atomic(ABC, Generic[T]):
24     """
25     Base class for an Atomic container. It can be initialized with any type
26     'T', but then that type must be respected and cannot be changed.
27     """
28     def __init__(self,
29                  initial_value: Optional[T] = None,
30                  step: Optional[T] = None):
31         self.rlock = Lock()
32
33     @property
34     @abstractmethod
35     def value(self) -> T:
36         """
37             Atomically obtains the stored value.
38             :return: the stored value.
39         """
40         pass
41
42     @value.setter
43     @abstractmethod
44     def value(self, val: T):
45         """
46             Atomically sets the stored value.
47             :param val: the new value to store.
48             :type val: T
49         """
50         pass
51
52
53 class AtomicFloat(Atomic[float]):
54     """
55     Atomically stores a floating number.
```

```
56 """
57 def __init__(self, initial_value=.0):
58     super().__init__(initial_value)
59     self._val = initial_value
60
61 @property
62 def value(self) -> float:
63     """
64     Atomically obtains the stored value.
65     :return: the stored value as float.
66     """
67     with self.rlock:
68         return self._val
69
70 @value.setter
71 def value(self, val: float):
72     """
73     Atomically sets the stored value.
74     :param val: the new value to store.
75     :type val: float
76     """
77     with self.rlock:
78         self._val = val
79
80
81 class AtomicInteger(Atomic[int]):
82     """
83     Atomically stores an integer.
84     """
85     def __init__(self, initial_value=0, step=1):
86         super().__init__(initial_value, step)
87         self._step = step
88         self._number_of_read = 0
89         self._counter = itertools.count(initial_value, step)
90         self._read_lock = Lock()
91
92     @property
93     def value(self) -> int:
94         """
95         Atomically obtains the stored value.
96         :return: the stored value as int.
97         """
98         with self._read_lock:
99             value = next(self._counter) - self._number_of_read
100            self._number_of_read += 1
101        return value
102
103 @value.setter
104 def value(self, val: int):
105     """
106     Atomically sets the stored value.
107     :param val: the new value to store.
108     :type val: int
109     """
110     with self.rlock:
111         self._counter = itertools.count(val, self._step)
```

```

112         self._number_of_read = 0
113
114     def increment(self):
115         """
116             Increments the stored value with the given step.
117         """
118         next(self._counter)

```

Listing E.15: pArm-S1/pArm/utils/atomics.py

```

1 #                                     pArm-S1
2 #
3 # This program is free software: you can redistribute it and/or modify
4 # it under the terms of the GNU General Public License as published by
5 # the Free Software Foundation, either version 3 of the License, or
6 # (at your option) any later version.
7 #
8 # This program is distributed in the hope that it will be useful,
9 # but WITHOUT ANY WARRANTY; without even the implied warranty of
10 # MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
11 # GNU General Public License for more details.
12 #
13 # You should have received a copy of the GNU General Public License
14 # along with this program. If not, see <http://www.gnu.org/licenses/>.
15 import logging
16 from . import init_logging
17 from PyQt5 import QtWidgets, QtGui
18 from .GUI import GUI
19 from .control.control import Control
20 import sys
21 from concurrent.futures import ThreadPoolExecutor
22
23 logging.basicConfig(level=logging.NOTSET)
24
25
26 def main():
27     try:
28         init_logging("Roger", log_file="p-Arm.log")
29         app = QtWidgets.QApplication(sys.argv)
30         app.setWindowIcon(QtGui.QIcon("yo.jpg"))
31
32         executor = ThreadPoolExecutor()
33
34         sys_control = Control(executor)
35
36         ui = GUI.Ui(sys_control)
37         ui.setupGUI()
38         ui.show()
39
40         sys.exit(app.exec_())
41     except Exception as e:
42         log = logging.getLogger("Roger")
43         log.critical(f"Unexpected error '{e}' while executing application!",
44                         exc_info=True)
45
46

```

```
47 if __name__ == '__main__':
48     main()
```

Listing E.16: pArm-S1/pArm/_main__.py

```
1 from pArm.__main__ import main
2
3 main()
```

Listing E.17: pArm-S1/app.py

Anexo F

Diagrama de Gantt al completo

Paquetes de trabajo

