Facultad
de
Ciencias

CONTROLADOR DE UN SUBSISTEMA DE OSCILACIÓN DE UNA ANTORCHA DE SOLDADURA CON UN PROCESADOR DE BAJO COSTE

OSCILLATION SUBSYSTEM CONTROLLER OF A WELDING TORCH WITH A LOW-COST PROCESSOR

Trabajo de Fin de Grado
para acceder al

GRADO EN FÍSICA

Autor: Asier Zulueta Barbadillo

Director: Michael González Harbour

Julio 2020

Autor

Asier Zulueta Barbadillo
azb181@alumnos.unican.es
+34 657221900

Resumen

El precio de los microprocesadores está, desde su creación hace casi medio siglo, continuamente reduciéndose. Eso ha ocasionado que, en los últimos años, aparecieran procesadores de bajo coste que pueden ofrecer plataformas de gran capacidad computacional que pueden ser aprovechadas para la implementación de aplicaciones de control industrial. En este trabajo se emplea la plataforma de bajo coste STM32, para la cual, se termina la implementación de un controlador capaz de otorgar capacidad de oscilación a una antorcha de soldadura. El proyecto se ha llevado a cabo en lenguaje Ada sobre máquina desnuda y ha servido para estudiar la viabilidad del uso de la tecnología empleada en controladores industriales.

Palabras Clave

Controlador industrial, STM32, Ada, Tiempo real, Soldadura

Abstract

The price of the microprocessors has been continuously decreasing since their creation half a century ago. This has caused in the last years the appearance of low-cost processors that can offer high-performance platforms which can be used to implement industrial control applications. In this work, the STM32 low-cost platform is used to implement a controller able to provide oscillation ability to a welding torch. This project has been developed in the Ada language on a bare machine and has served to study the viability of the used technology in industrial controllers.

Key Words

Industrial controller, STM32, Ada, Real-time, Welding

Índice

Índice de figuras

Lista de Códigos

Acrónimos

ADC *Analog-to-Digital Converter*, Convertidor analógico-digital. 5

CPU *Central Processing Unit*, Unidad central de procesamiento. 5

HW *Hardware*. 1, 2, 6, 11–13, 20, 21, 23, 26, 27

I/O *Input/Output*, Entrada/Salida. 1, 5, 6

LCD *Liquid-Crystal Display*, Pantalla de cristal líquido. 2, 5, 12, 16, 20, 26, VI

PO *Protected Object*, Objeto protegido. 15, 16, 18

RAM *Random-Access Memory*, Memoria de acceso aleatorio. 2, 5, 27

RMS *Rate-Monotonic Scheduling*, Planificación con prioridad al más frecuente. 20

ROM *Read-Only Memory*, Memoria de solo lectura. 5

SW *Software*. 2, 4, 6, 7, 12, 27

TFT *Thin-Film-Transistor*, Transistor de película delgada. 5

UML *Unified Modeling Language*, Lenguaje unificado de modelado. 12

USB *Universal Serial Bus*, Bus universal en serie. 1, 2

WVGA *Wide Video Graphics Array*, Matriz de gráficos de vídeo en formato ancho. 5

1. Introducción

1.1. Motivación

Desde principios de los años 70, con la creación y comercialización de los primeros microprocesadores promocionados para su uso en calculadoras, cajas registradoras, relojes e instrumentos de medición, la producción y uso de estos se ha expandido hasta tal punto de encontrarse en casi cualquier industria. [2, pp. 1]

La automatización de la industria mediante el empleo de microcontroladores ha supuesto numerosas ventajas, como un aumento de la producción y de la productividad, un uso más eficiente de los recursos y la creación de productos de mayor calidad, así como otras ventajas que benefician a los trabajadores, como la mejora de la seguridad o la disminución de las jornadas laborales. Sin embargo, poder equipar a las empresas con determinado *hardware* (HW) puede llegar a suponer una gran inversión entre el diseño, la fabricación y la instalación. [3]

En vistas a reducir el coste del HW para desarrollos industriales, se propone la implementación de una plataforma de bajo coste para el control de un subsistema de un equipo de soldadura. La efectividad de esta implementación sirve como estudio de viabilidad que podría dar paso al uso de esta plataforma para aplicaciones de control industrial similares a la propuesta.

1.2. Antecedentes

La empresa Equipos Nucleares ha desarrollado numerosos sistemas de soldadura semi-automática de propósito especial, adaptados a la fabricación de componentes de centrales nucleares. En estos sistemas se depositan cordones de material fundido sobre las piezas a soldar. Puesto que suelen ser piezas de grandes espesores, es necesario depositar múltiples de estos cordones, mediante sucesivas pasadas con la máquina de soldar.

Algunos de los sistemas de soldadura desarrollados por Equipos Nucleares disponen de la capacidad de oscilación, que permite mover la antorcha de soldadura en dirección perpendicular al avance del cordón, sucesivamente hacia un lado y otro del centro de la trayectoria de soldadura. Ello permite incrementar el área cubierta por la soldadura, lo que es apropiado en piezas que requieren rellenar con material de soldadura zonas de gran espesor.

Equipos Nucleares ha manifestado su intención de dotar de esta capacidad de soldadura a otras de sus máquinas, creando un subsistema capaz de realizar los movimientos necesarios para la oscilación. Con este motivo, se desarrolló hace tiempo, en colaboración con la Universidad de Cantabria, un proyecto para la implementación de este subsistema utilizando una plataforma HW de bajo coste, basada en el procesador Raspberry PI y el sistema operativo MaRTE OS desarrollado por la Universidad de Cantabria. De ese proyecto se han podido extraer las siguientes conclusiones:

- El HW de *input/output* (I/O) para Raspberry PI de bajo coste utilizado no es lo suficientemente robusto como para soportar un entorno industrial agresivo, con mucho ruido electromagnético y condiciones ambientales difíciles.
- La conexión a red de la tarjeta Raspberry PI se realiza mediante el conocido bus *universal serial bus* (USB). Esto representa un problema para el sistema operativo utilizado, MaRTE OS, que no tiene soporte para este bus.

En línea con estas conclusiones, existen estudios que determinan que aunque en los últimos años la Raspberry PI está dejando de ser simplemente una plataforma de experimentación para adentrarse en el mundo de la industria, no es la mejor opción para aplicaciones críticas. [4]

Figura 1: Pantalla integrada del microcontrolador [1, pp. 21]

Por ello, como sustitución de la Raspberry PI, se utiliza, en el presente trabajo, una plataforma de bajo coste más adaptada al mundo de la industria, como es la plataforma STM32F769I-DISCO de STMicroelectronics. Esta plataforma de carácter industrial ofrece unas mayores prestaciones donde la Raspberry presenta cierta debilidad, como es en las entradas y salidas del microcontrolador. Asimismo, dispone de un dispositivo de red conectado directamente al procesador, no a través de un bus USB. El microcontrolador posee un procesador ARM, una memoria *flash* de 2 megabytes y una *random-access memory* (RAM) de 512+16+4 kilobytes. Entre las características de la plataforma que se emplean en este proyecto destaca su pantalla *liquid-crystal display* (LCD) táctil de 4 pulgadas y su botón de usuario. Se muestra en la Figura (1) la pantalla integrada que posee el microcontrolador.

Aunque en el momento de comenzar este trabajo existía un desarrollo en marcha para portar el sistema operativo de tiempo real MaRTE OS a la plataforma STM32, aún no estaba finalizado. Por ello, se plantea realizar el proyecto con el compilador Ada disponible para STM32 basado en máquina desnuda, por tanto sin ningún sistema operativo. Este cambio de compilador implica a su vez cambios en el *software* (SW) de partida, pues el compilador de Ada disponible para la STM32 está restringido al subconjunto Ada denominado Perfil de Ravenscar.

El Perfil de Ravenscar es un conjunto de restricciones que, manteniendo una filosofía estático-determinista, permite demostrar con más facilidad que el programa está libre de fallos. En la Sección 3 se muestra una descripción más extensa de este perfil con sus principales limitaciones.

La utilización de esta plataforma basada en el Perfil de Ravenscar es otro de los puntos fuertes del estudio de viabilidad desarrollado en el presente trabajo, pues permite estudiar las posibilidades de evitar la dependencia del sistema operativo MaRTE OS.

1.3. Descripción del proyecto

En base a los antecedentes descritos, en este trabajo se desarrolla el controlador de un sistema de soldadura sobre la plataforma STM32, reutilizando el código servible ya desarrollado e implementando nuevo cuando no es posible. Estas partes son, principalmente, las que están ligadas al HW del microcontrolador. Las dos partes principales son las operaciones para manejar la entrada y salida, tanto digital como analógica, y las utilizadas para el uso de la pantalla integrada.

Por otro lado, se tienen que realizar diversos cambios para adaptarse al uso del Perfil de Ravenscar. La implementación existente no seguía las limitaciones debidas a este perfil y es necesaria la realización de cambios más profundos para estar acorde a las restricciones.

Asimismo, se pretende comprobar el correcto funcionamiento del sistema, realizando pruebas tanto de carácter específico como general sobre un entorno simulado, ya que en este momento no se dispone de la máquina de soldadura.

Finalmente se desea extraer conclusiones sobre la viabilidad del uso de la plataforma STM32 y el compilador Ada con la limitación del Perfil de Ravenscar para la implementación de controladores industriales.

1.4. Objetivos

El principal objetivo de este proyecto es completar la implementación de código necesaria para obtener un controlador funcional que permita oscilar la antorcha de un subsistema de soldadura y en

base a los resultados obtenidos estudiar la viabilidad del uso de la plataforma STM32 para implementar controladores industriales en lenguaje Ada. Este objetivo principal, esconde otros objetivos que pretenden ser alcanzados:

- Comprobar el comportamiento de cada funcionalidad del código implementado y del conjunto del controlador. Con esto se pretende obtener un controlador funcional que además asegure el comportamiento que se espera.
- Comprobar la viabilidad del uso del Perfil de Ravenscar para el desarrollo de controladores con requisitos de tiempo real complejos.
- Demostrar la viabilidad del uso de plataformas de bajo coste en ámbitos de desarrollo industriales.

2. Tecnología empleada en el proyecto

En esta sección se explica el entorno empleado en el desarrollo del proyecto. Por un lado se muestra el método optado para la implementación del SW, indicando el lenguaje, entorno de desarrollo, librerías y proceso llevado a cabo para poner a punto el computador y poder cargar el código en el microcontrolador. Por otro lado, se indican los instrumentos de medición y de prueba empleados, (osciloscopio y generador de funciones), y la plataforma que se utilizará para controlar el subsistema de oscilación.

2.1. Sistemas de tiempo real

Los sistemas de tiempo real son aquellos en los que el comportamiento correcto del sistema depende tanto de la lógica de los resultados computacionales como del momento en el que estos resultados han sido producidos. [5, pp. 2]

Es común la distinción de los sistemas de tiempo real en sistemas críticos (*hard*) y acríticos (*soft*). Los sistemas críticos de tiempo real son aquellos en los que es imprescindible que la respuesta se dé dentro del tiempo límite especificado. Los acríticos son los que, aun siendo importante ajustarse los tiempos límite, el sistema puede continuar funcionando si ocasionalmente esto no se cumple. Gran parte de los sistemas están formados tanto por subsistemas críticos como acríticos. Además, muchos de los servicios tienen especificados un tiempo límite acrítico y otro crítico. [6, pp. 2-3]

En las últimas décadas, ha habido un aumento muy importante de la cantidad de sistemas de tiempo real. Hasta el punto en el que, hoy en día, juegan un papel crucial en la sociedad. Su uso está totalmente extendido en los procesos de control, donde un computador de tiempo real se integra en un sistema más complejo en el que se encarga del control de los sensores y actuadores. También en los procesos de manufactura, donde se utilizan para bajar los precios de producción y aumentar la productividad. Sin embargo, también están presentes en casi cualquier otro ámbito, como en el espacial, transporte aéreo y terrestre, médico o militar y en otros más cercanos a la vida cotidiana, como en juguetes inteligentes, comunicaciones, sistemas electrónicos y multimedia. [6, pp. 4-9] [7, pp. 1-2] [8]

2.2. Lenguaje Ada

El lenguaje de programación empleado en este proyecto es Ada. Este lenguaje de alto nivel se caracteriza principalmente por su fiabilidad. [9, pp. 3-4] Su fuerte tipado, con el que detecta los errores más rápido en el proceso de desarrollo, junto con su flexibilidad y portabilidad, hace que sea el lenguaje idóneo para la programación de sistemas embebidos críticos donde se requiere de un control riguroso de los tiempos de respuesta. [6, pp. 20-21]

La estructura de organización de las aplicaciones Ada se basa en los paquetes, mediante los cuales se obtiene una modularización y encapsulación del código. En el proyecto desarrollado la implementación del código se ha llevado a cabo en 12 paquetes distintos, en los que cada uno lleva a cabo una función determinada y específica.

2.3. Instrumentación utilizada

2.3.1. Microcontrolador

Un microcontrolador es un circuito integrado programable que ha sido diseñado para la monitorización o control de determinadas tareas y que contiene todos los componentes para operar de manera independiente. Una de las áreas de mayor uso de los microcontroladores son los sistemas embebidos. En estos sistemas, la unidad de control está integrada en el sistema. [2, pp. 7]

El TMS 1000 introducido en 1974 puede ser considerado como uno de los primeros microcontroladores incluyendo RAM, *read-only memory* (ROM) e I/O. En la actualidad, se crean anualmente miles de millones de microcontroladores para integrarse en multitud de dispositivos. Hoy en día, existen algunos módulos que están presentes en la mayoría de los microcontroladores. Algunos de estos módulos son la *central processing unit* (CPU) del controlador, la memoria, las entradas y salidas digitales y analógicas o los temporizadores/contadores. [2, pp. 1-6]

Para este proyecto se ha utilizado la plataforma STM32F769, que cuenta con un procesador ARM Cortex-M7. En los microcontroladores se encuentran diferentes tipos de memoria. La memoria RAM se utiliza para el almacenamiento temporal de información. La unidad empleada posee una memoria RAM de 512+16+4 kilobytes. Por otro lado, se encuentra la memoria de programa. En este caso, tiene una memoria *flash* de 2 megabytes. [2, pp. 22-27] [10]

Las entradas y salidas son una de las características principales de los microcontroladores ya que se utilizan para monitorizar el sistema a controlar. Por un lado se encuentran las entradas, mediante las que el controlador adquiere las señales eléctricas de los sensores. Por otro lado se encuentran las salidas, mediante las que el controlador define el estado de los actuadores que actúan sobre el proceso o sistema controlado. De las entradas, se distinguen las digitales y las analógicas, según el tipo de señal eléctrica que hay que leer. En el caso analógico es preciso convertir la señal a cifras digitales, que son los únicos datos con los que trabaja el procesador. Similarmente, podemos tener salidas analógicas y digitales. En el caso de las analógicas, es preciso disponer de convertidores capaces de transformar las cifras digitales que maneja el procesador en valores de tensión eléctrica analógica. En este proyecto se utilizan, exclusivamente, entradas digitales y analógicas y salidas digitales. [2, pp. 33-40]

Esta plataforma tiene 6 canales analógicos de entrada y 14 digitales, que pueden emplearse tanto de entrada como de salida. Estos canales I/O pueden tolerar voltajes de 0 a 5V. Sin embargo, para poder trabajar correctamente en *analog-to-digital converter* (ADC), es necesario limitar el rango de voltaje. Por lo tanto, se trabajará teniendo en cuenta los nuevos límites, que serán $V \in [1{,}7, 3{,}3]$. Se han escogido estos límites teniendo en cuenta la ficha técnica del fabricante, donde se indica que se debe emplear un voltaje estándar de operación que se encuentre en el rango $V \in [1{,}7, 3{,}6]$. [11]

Además, la placa empleada presenta algunas incorporaciones menos comunes que se utilizarán en el proyecto, entre las que destacan la pantalla táctil y el botón de usuario que tiene integrado. Esto ahorra la utilización de una pantalla adicional para mostrar la información y de un botón para el control de las alarmas.

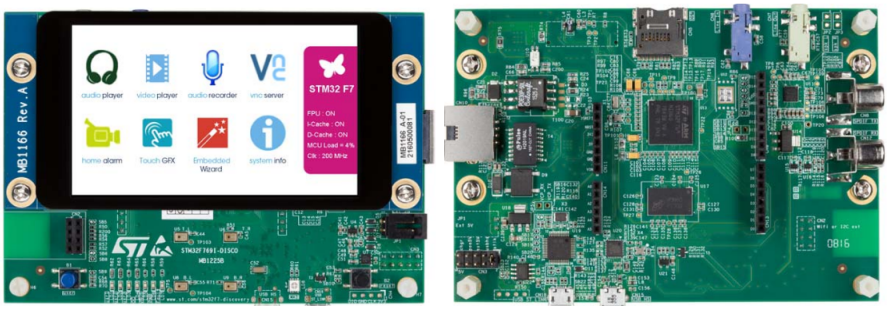

Figura 2: Plataforma STM32F769I [1, pp. 1]

En la Figura (2) se muestra el dispositivo, tanto desde arriba como desde abajo. Desde arriba, se puede apreciar la pantalla *wide video graphics array* (WVGA) *thin-film-transistor* (TFT) LCD de 4 pulgadas, el botón de usuario y el de reinicio de los que dispone. Una vista del microcontrolador desde abajo, muestra, entre otras cosas, la disposición de canales, tanto analógicos como digitales.

La configuración de estos pines es la misma que la de los conectores del Arduino™ UNO V3. [1, pp. 25]

Arduino es una plataforma de código abierto basada en un HW y SW de fácil uso. [12] Aunque en un primer momento este microcontrolador fue diseñado para la enseñanza, su popularización propició la creación de una gran cantidad de HW de I/O orientado a esta plataforma. [13]

2.3.2. Generador de funciones

La función generada por el equipo de soldadura es una función trapezoidal cuya frecuencia y amplitud puede ser fácilmente ajustable al rango de operación del microcontrolador empleado. La posición y el movimiento de la antorcha y del hilo de soldadura irá en concordancia con el momento de la señal trapezoidal. Cuando llega la zona de pendiente nula de la señal, los motores están estáticos, mientras que cuando la señal que llega tiene una pendiente, el motor está en movimiento (para la derecha o para la izquierda en función de si la pendiente es positiva o negativa). Se muestra en la parte inferior de la Figura (3) la posición de la antorcha de soldadura en la placa a soldar de acuerdo a la señal trapezoidal que se emite en ese momento y que se indica en la parte superior de la imagen. La flecha de la imagen de la placa a soldar simboliza el desplazamiento (que no se controla en este proyecto) de la placa a soldar.

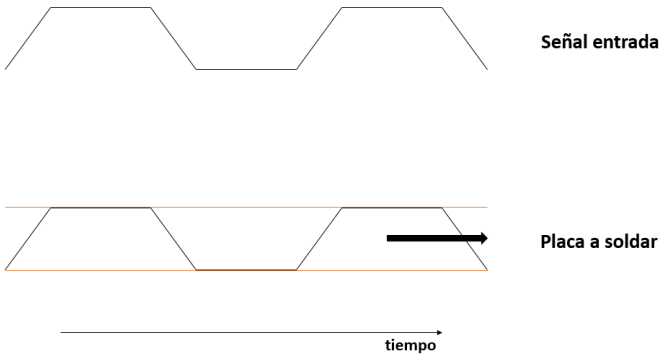

Figura 3: Posición de la antorcha de soldadura en función de la señal de entrada

Se utilizará el generador de funciones HP 33120A para simular las señales que se generarán con el equipo de soldadura y comprobar así, si el programa funciona correctamente. El generador empleado posee unas funciones predefinidas de entre las que no se encuentra una trapezoidal. Sin embargo, se puede modificar las características de la función triangular ya definida para obtener una que tenga una forma similar, como la que se muestra en la Figura (4).

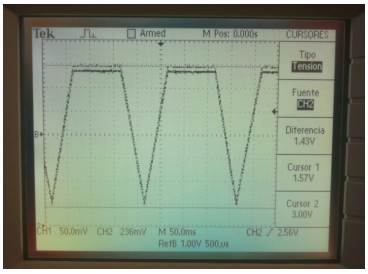

Figura 4: Función triangular deformada

2.3.3. Osciloscopio

Para comprobar las señales se utiliza el osciloscopio de tiempo real Tektronix TDS 210. Este osciloscopio permite tanto la comprobación de las señales generadas, conectando sus sondas al generador de funciones, como analizar las señales digitales de salida, conectando las sondas al microcontrolador.

2.4. Entorno de desarrollo

Para el desarrollo del SW se emplea el entorno de desarrollo integrado GPS (GNAT Programming Studio) en su distribución en Linux. El compilador es respaldado por AdaCore, que se encarga de actualizar y comercializarlo. Los servicios que ofrece AdaCore están orientados a desarrolladores de

aplicaciones embebidas que requieren de cierto nivel de seguridad. [14] Es por ello que GPS está diseñado principalmente para lenguajes como Ada, C o C++, aunque acepta otros lenguajes.

Se emplea un entorno de desarrollo cruzado ya que el SW se compila en un anfitrión de tipo Linux (*host*) y se ejecuta en la STM32 (*target*).

2.5. Proceso de instalación

La instalación se lleva a cabo en Ubuntu y el proceso que se explica a continuación sirve para poder conectarse y cargar programas al microcontrolador, en nuestro caso usando GPS. Las explicaciones parten después de la descarga e instalación de los archivos x86 GNU Linux y ARM ELF de la página https://www.adacore.com/download, y se describen aquí para el lector que desee reproducir el proceso.

En primer lugar se selecciona el servidor principal en la aplicación de actualización de SW y a continuación se realiza un *update* del Ubuntu.

Tras esto, se instala mediante la terminal los siguientes paquetes:

```
$ sudo apt install binutils
$ sudo apt install gcc
$ sudo apt install openocd
$ sudo apt install libusb-1.0-0-dev
$ sudo apt install cmake
```

Para poder conectarse con el microcontrolador se necesita tener STLink[1]. Para ello se va a instalar desde un repositorio en Git-Hub[2] con los siguientes comandos:

```
$ sudo apt install git
$ git clone https://github.com/texane/stlink.git
$ cd stlink.git/
$ make clean
$ make release
$ cd build/Release/
$ sudo make install
$ sudo ldconfig /usr/local/lib/
```

2.6. Uso de librerías

En la creación del código se han empleado dos librerías: Ada_Drivers_Library (ADL) y Drivers_UC. La primera de ellas es una librería escrita en Ada de carácter general que puede encontrarse en Git-Hub[3] y que contiene *drivers* y proyectos de ejemplo para diferentes sensores y microcontroladores de STMicroelectronics. La segunda es una librería específica creada en la UC para la plataforma STM32F769 en lenguaje Ada. Esta librería, que también se encuentra en Git-Hub[4], fue creada para poder interactuar con este microcontrolador de una manera más sencilla que empleando la librería ADL, que resulta muy completa y de bajo nivel, por lo que obliga al usuario a gestionar muchos parámetros, llegando a dificultar su uso a usuarios inexpertos. La librería Drivers_UC ofrece una capa de abstracción que permite utilizar algunos de los procedimientos más comunes de la otra

[1] STLink es un conjunto de herramientas de código abierto utilizadas para programar y depurar plataformas STM32.
[2] URL del repositorio: https://github.com/stlink-org/stlink
[3] URL del repositorio: https://github.com/AdaCore/Ada_Drivers_Library
[4] URL del repositorio: https://github.com/dao703/Robot_Car_UC/tree/master/Drivers_UC

librería. Esta librería se ha utilizado en este proyecto para la configuración y utilización de los canales analógicos y digitales de la plataforma. [15]

3. Perfil de Ravenscar

Como se ha mencionado anteriormente, el compilador de Ada para STM32 sobre máquina desnuda presenta la limitación de obedecer solamente al subconjunto del lenguaje definido en el Perfil de Ravenscar.

En 1997, la *8º International Real-Time Ada Workshop* (IRTAW), tuvo como resolución principal un perfil restrictivo de la parte concurrente del lenguaje Ada que pretendía satisfacer los requisitos de alta fiabilidad de los sistemas de tiempo real. Primeramente, se listaron 18 servicios existentes en Ada que no podrían ser usados por el perfil y se establecieron otras 10 características específicas de tiempo real y concurrentes que sí podrían utilizarse con un comportamiento estático-determinista. [16] En las siguientes reuniones de los sucesivos años (9º y 10º IRTAW) se realizaron algunas alteraciones menores, pero manteniendo la misma filosofía. [17] Siendo la primera de las IRTAW en el pueblo de Ravenscar en Yorkshire, se bautizó a este perfil como el Perfil de Ravenscar.

El Perfil de Ravenscar es un conjunto de restricciones dirigidas a aplicaciones que requieren de implementaciones muy eficientes o que tienen requisitos de alta integridad. [6, pp. 430] Con este modelo más restringido y estático de programación se puede analizar y certificar las aplicaciones con un mayor grado de confianza.

Se puede definir el Perfil de Ravenscar indicando las características prohibidas y compatibles con este perfil. Por un lado, de las prohibidas destacan:

- Tipos protegidos con más de una entrada
- *delays* relativos
- Cualquier tipo de la instrucción `select`
- Entradas con barreras diferentes de una sola variable booleana
- Utilización de referencias al paquete Ada.Calendar

Por otro lado, se permiten:

- Utilización de referencias al paquete Ada.Real_Time;
- Instrucción `delay until`
- Tipos tareas y protegidos y con objetos definidos a nivel de librería
- *Timing events* (eventos temporizados) declarados a nivel de librería

Desde el 2002, el Perfil de Ravenscar es parte formal del lenguaje Ada y a cada actualización del lenguaje, se revisa que el perfil mantiene el conjunto de restricciones adecuadas. [18, pp. 11]

3.1. *Timing Events* (Eventos temporizados)

El paquete `Ada.Real_Time.Timing_Events` permite ejecutar en un cierto instante determinados procedimientos, sin necesidad de usar instrucciones como `delay` o tareas auxiliares. [19]

El paquete incluye los tipos `Timing_Event` y `Timing_Event_Handler` que representan, respectivamente, el evento que ocurrirá en un instante en el futuro e identifican el procedimiento protegido que se va a ejecutar cuando el *timing event* ocurra. Para trabajar con ellos, el paquete incluye procedimientos como `Set_Handler`, que asocia el manejador (*handler*) con el evento y establece el instante en el que se quiere ejecutar el evento o `Cancel_Handler`, que cancela la ejecución futura del evento en caso de que esté establecido por el anterior procedimiento.

El uso de los *timing events* puede resultar de gran utilidad para implementar código con determinadas restricciones (como las del Perfil de Ravenscar). Una de las principales restricciones de este perfil y que tiene implicaciones en este proyecto es la restricción `No_Select_Statements`, que prohíbe la instrucción `select`. Esta instrucción tiene diferentes funcionalidades. Una que afecta directamente a este proyecto es el *timed entry call* (llamada temporizada), donde la llamada a la entrada de un objeto protegido es cancelada después de un período de tiempo si no se ha producido el evento que debería liberarlo. [20] La forma de usar esta funcionalidad en un lenguaje Ada sin restricciones sería:

```
1 select
      PO.Entrada;
3     Timeout := False;
  or
5     delay until Cierto_Tiempo;
      Timeout := True;
7 end select;
```

Código 1: Uso del *timed entry call* mediante la instrucción `select`

Como sustitución de este método se puede crear una tarea adicional que duerme hasta que se le pasa el tiempo de espera como parámetro. [18, pp. 32-33] Otra posibilidad mucho más eficiente, que es por la que finalmente se ha optado en el trabajo, es el uso de los *timing events*. El uso de estos objetos mediante los procedimientos nombrados anteriormente es una alternativa elegante y eficiente al uso del `select`. [21]

4. Implementación

En esta sección se muestra, por un lado, el diseño del proyecto llevado a cabo y su organización como conjunto. Por otro lado, se realiza una explicación más detallada de los paquetes en los que las implementaciones o cambios han sido más significativos.

El diseño del sistema, así como parte de código, estaba efectuado con anterioridad al comienzo de este proyecto. Sin embargo, la decisión de realizar un cambio de plataforma conllevó un nuevo desarrollo de los paquetes relacionados con el HW de microcontrolador (`Entrada_Salida`, `Consola_Avanzada`, `Botones`) así como del paquete `Servos` debido a las limitaciones de Perfil de Ravenscar.

4.1. Descripción del diseño

El equipo de soldadura está formado por uno o dos motores paso a paso controlados por sendos servos digitales. Uno de los servos es el encargado de la oscilación de la antorcha y otro, que se desplaza a la par que el primero, de ir suministrando el hilo necesario para la soldadura. Los servos ordenan el movimiento de la antorcha en función de la señal que se genera en la fuente de soldadura y que se envía al microcontrolador.

En la Figura (5) se muestra el diagrama que resume el comportamiento del sistema.

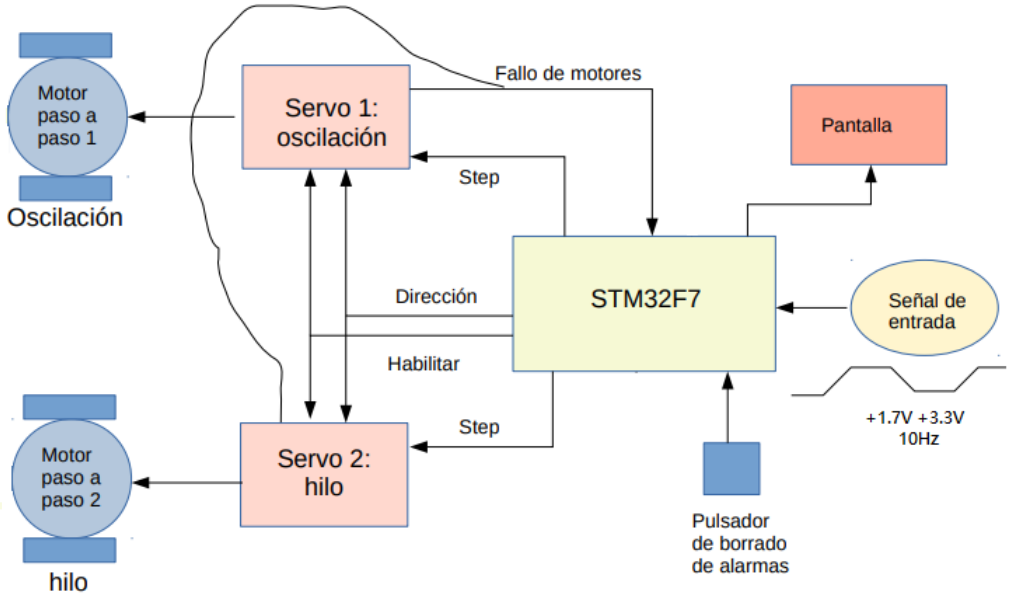

Figura 5: Diagrama de bloques del sistema de control de oscilación

Los motores paso a paso son capaces de avanzar pequeños pasos individuales girando tanto a izquierda como a derecha. Mediante la composición de muchos de estos pasos seguidos es posible establecer movimientos a las posiciones deseadas y con las velocidades deseadas. Cuanta mayor sea la frecuencia de los pasos, mayor velocidad. Los servos generan las señales que necesita el motor para sincronizar estos pasos y con la intensidad requerida. Lo hacen a partir de dos señales digitales.

En primer lugar, una señal cuadrada, llamada *step*, que representa los pasos (se realiza un paso por cada flanco de subida). La otra señal, llamada *dirección*, indica la dirección en que deben darse estos pasos, según su valor digital sea alto o bajo.

Una vez programado el microcontrolador, se conecta a los servos mediante sus pines digitales de salida. Se utilizan 2 pines digitales para controlar los pasos de los servos: uno para el de oscilación y otro para el del hilo. Se emplea otro canal de salida digital para controlar la dirección en la que ambos servos se desplazan. Se utiliza solo una señal de dirección porque ambos motores giran siempre en la misma dirección, o bien hacia la izquierda o a la derecha. Por último se reserva uno de los pines para habilitar los motores o, en caso de enviar una señal con valor digital 0, deshabilitarlos.

También se configura un canal de entrada en la plataforma que se conecta a los servos para que, en caso de que estos detecten un error, lo notifiquen al microcontrolador para deshabilitar los motores.

Además de los pines digitales, se habilita un pin analógico para leer el valor de la señal de entrada trapezoidal de la que dependerá el comportamiento de los servos. En la Figura (3) de la Sección 2.3.2 se muestra el comportamiento del equipo en base a la señal trapezoidal. Se realiza un seguimiento de los valores de la señal analógica para, en función de estos, enviar una consigna de velocidad de actuación a los servos. La rapidez con la que el motor da los pasos depende de la magnitud de la velocidad y la dirección en la que actúan depende de su signo. La sincronización entre esta señal analógica y los pulsos generados para el servo representa una de las mayores dificultades de la aplicación en cuanto a los requisitos de tiempo real, y se describe más adelante en la Sección 4.2.4.

Además de la alerta por fallo de los motores, están implementados otros dos identificadores de alarma diferentes, relacionados con una frecuencia excesiva de la señal de entrada y con una alarma guardián que se activa en caso de no ser reiniciada cada menos de un segundo (este reinicio se realiza cuando llegan nuevas consignas a los servos). Dependiendo de cuál acontezca se tomarán unas medidas u otras. Estas alarmas se mostrarán, junto con otra información relevante (como las consignas a tiempo real, los pasos del motor y los posibles mensajes importantes del funcionamiento del SW) en la pantalla que tiene incorporada el microcontrolador. Para eliminar una vez sean vistas las notificaciones de alarma de la pantalla, se configura el botón de usuario.

4.2. Organización de clases

Se muestra en la Figura (6) el diagrama *unified modeling language* (UML) del paquete controlador encargado del sistema de control de oscilación de la antorcha de soldadura. Los diferentes paquetes que lo conforman son:

- **Alarmas**: alberga las operaciones relacionadas con la notificación de los posibles mensajes y alarmas del HW de los motores o del resto de paquetes.
- **Botones**: realiza la lectura del estado del botón de usuario.
- **Consola_Avanzada**: alberga las operaciones necesarias para poder pintar en la pantalla LCD del microcontrolador.
- **Entrada_Salida**: se encarga de la configuración y manejo de los diferentes canales, tanto digitales como analógicos.
- **Guardian**: se encarga de determinar un posible error causado por la no llegada a tiempo de nuevas consignas.
- **Monitor_Tiempos**: contiene información sobre los tiempos de ejecución de las tareas y operaciones del resto de paquetes.
- **Motores**: controla mediante señales digitales el HW de los motores.

- **Parametros**: incluye los atributos y métodos que pueden variar en un futuro, como por ejemplo los que están ligados al HW del equipo de soldadura.
- **Reportero**: recopila la información necesaria de los otros paquetes para pintarla en pantalla.
- **Seguimiento_Entrada**: envía consignas de velocidad al paquete `Servos` en función del voltaje.
- **Sensor_Entrada**: es el encargado de leer la consigna analógica de entrada y convertirla a voltios.
- **Servos**: en función de las consignas de velocidad que le llegan, ordena realizar a `Motores` una determinada operación.

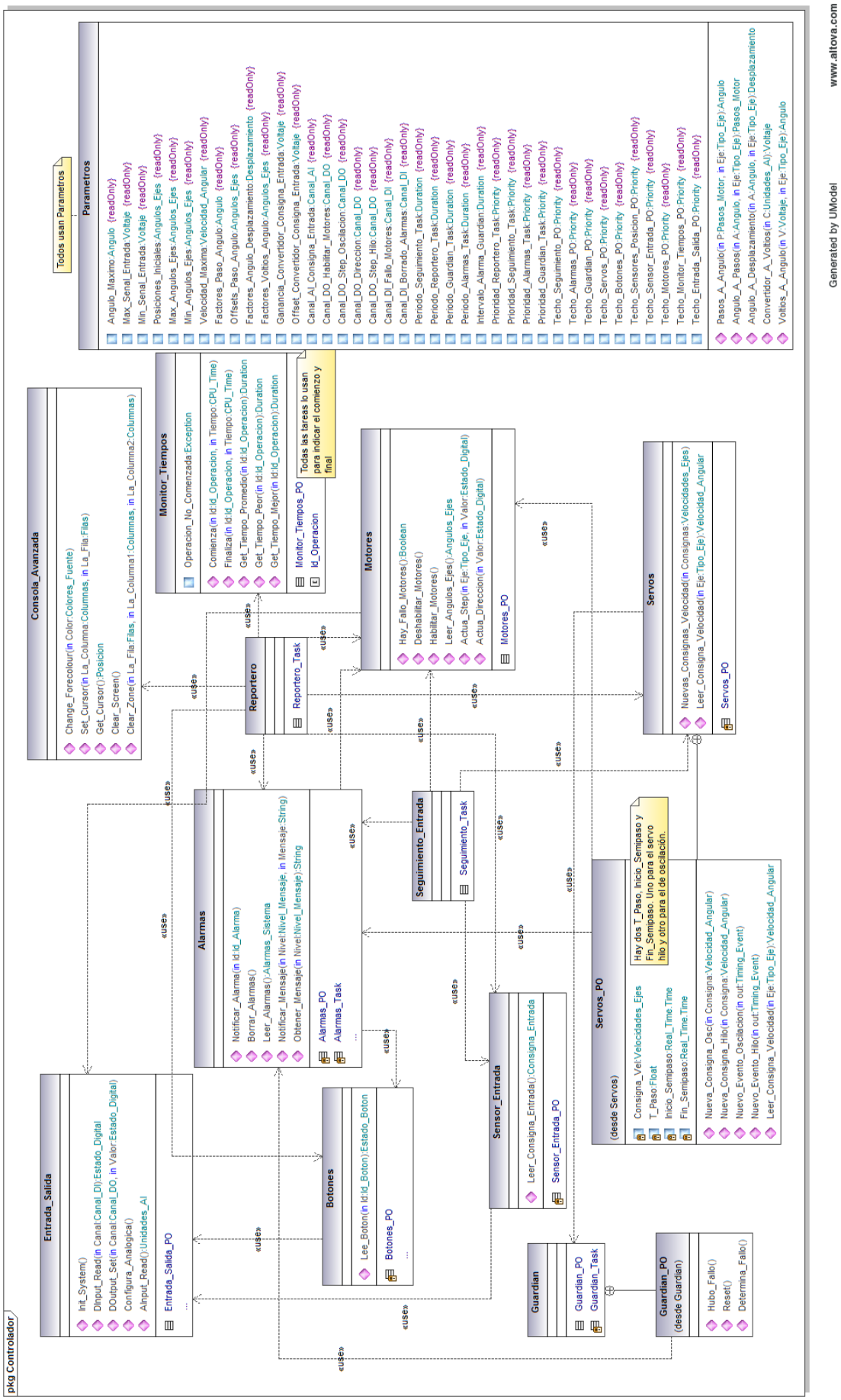

Figura 6: Diagrama de clases del paquete controlador

4.2.1. Archivo de proyecto

Se crea el archivo de proyecto `sistema_soldadura.gpr` para el sistema de compilación cruzada que se necesita. En este archivo se va a establecer el directorio de las librerías y se configuran las propiedades del proyecto, tales como el Perfil de Ravenscar o los *main* ejecutables.

4.2.2. Paquete Entrada_Salida

El paquete `Entrada_Salida` es el encargado de manejar tanto la entrada analógica como la entrada y salida digital del microcontrolador. Se realiza el código adaptándolo a las nuevas librerías de la STM32. Aunque en un principio se crea el objeto protegido `Entrada_Salida_PO` con 4 operaciones, después, para evitar errores en la llamada entre operaciones protegidas, se deja el *protected object* (PO) con solo dos operaciones, sacando fuera las operaciones relacionadas con la salida digital y la lectura analógica, ya que solo son invocadas por el paquete `Motores` y `Sensor_Entrada`, respectivamente. En el objeto protegido se mantiene la operación de inicialización de los pines y de lectura analógica que son llamados por el paquete `Sensor_Entrada` y por el paquete `Reportero` y el *main* `Controlador_Main.adb`, que es el encargado de ejecutar las tareas de los programas (este *main* se explica más detalladamente en la Sección 5.9).

```
1  ------------------------
   -- Entrada_Salida_PO --
3  ------------------------

5  protected Entrada_Salida_PO
     with Priority => Parametros.Techo_Entrada_Salida_PO is
7
      -- Funcion que inicializa los pines de la tarjeta y devuelve un
9     --  Booleano con valor True si todo fue correcto o False en caso de
      --  error.
11    function Init_System return Boolean;

13    -- Funcion que permite leer el estado de una entrada digital para
      --  un canal determinado.
15    function DInput_Read(Canal:Canal_DI) return Estado_Digital;

17 end Entrada_Salida_PO;
```

Código 2: Objeto protegido de `Entrada_Salida.adb`

Las operaciones del objeto protegido tienen como finalidad:

- `Init_System`: se utiliza para configurar y dar un valor lógico a los pines digitales. Se establecen los pines D4, D5, D6 y D7 como pines de entrada y los pines D8, D9, D10 y D11 como de salida. También se habilita el canal analógico PA6 (o A0) para las futuras lecturas (la librería empleada tiene este pin como predeterminado para las lecturas analógicas).
- `DInput_Read`: lee el estado de la entrada digital que se le pasa como parámetro.

Por otro lado se encuentran las operaciones:

- `DOutput_Set`: establece una señal digital de salida en el pin establecido.
- `Configura_Analogica`: habilita, configura y comienza la adquisición del valor de la señal analógica.
- `AInput_Read`: lee la señal de la entrada analógica configurada anteriormente con la operación `Configura_Analogica`. Entre que comienza la adquisición mediante la anterior

operación y termina al leerse el canal mediante este procedimiento, se tiene que dejar un tiempo mínimo de $t = 10\,\mu$s para que el valor obtenido sea un valor correcto. [11, pp. 175]

4.2.3. Paquete Consola_Avanzada

El paquete `Consola_Avanzada` es el encargado de manejar la pantalla LCD que tiene incorporada el microcontrolador. Para el código de este paquete se han utilizado los paquetes `LCD_Std_Out`, `HAL.Bitmap` y `STM32.Board` de la librería Ada_Drivers_Library. Para poder emplear las variables `Char_Count` y `Currrent_Y` (utilizadas para especificar la fila y la columna de la posición actual) que se encuentran en el cuerpo del paquete `LCD_Std_Out`, se han creado dos funciones que devuelven los valores y dos procedimientos para editarlos. También se han creado en este paquete dos nuevas funciones que devuelven las variables `Max_Width` y `Max_Height` que servirán para establecer el límite de columnas y de filas.

Las operaciones de este paquete son:

- `Change_Forecolour`: sirve para modificar el color de la fuente.
- `Set_Cursor` y `Get_Cursor`: se utilizan para establecer una determinada posición (X e Y) en la pantalla y para obtener esa posición.
- `Clear_Screen` y `Clear_Zone`: se emplean para limpiar la pantalla LCD en su totalidad y una parte específica de la misma, respectivamente.

Para la escritura en pantalla se emplea el procedimiento `Put` del paquete `LCD_Std_Out`. Al invocar este procedimiento se le pasa como parámetros el mensaje junto a la posición (fila y columna) donde se desea pintarlo.

4.2.4. Paquete Servos

El paquete `Servos` tiene la función de recibir las nuevas consignas de velocidad y ordenar actuar a los motores.

Para hacer esto se tiene el procedimiento `Nuevas_Consignas_Velocidad`, que recibe periódicamente de la tarea `Seguimiento_Task` (perteneciente al paquete `Seguimiento_Entrada`) unos nuevos valores de velocidad para el movimiento del eje de oscilación y del hilo. Una vez llega la nueva consigna de velocidad, se calculan los nuevos tiempos de semipasos de los servos. El cabezal de soldadura del equipo se desplaza mediante los semipasos. Cuando se modifica de 0 a 1 el estado digital de salida de los canales de paso del microcontrolador, los motores se desplazan un paso, y con ello, un determinado ángulo. Es por eso que desde el paquete `Servos` se van alternando llamadas al procedimiento `Actua_Step` del paquete `Motores` con diferentes estados digitales (0 y 1). Este paquete, se encarga de, si llega una nueva consigna, ordenar realizar un semipaso con la nueva velocidad, o si no ha llegado, de terminar el semipaso. La manera rápida e intuitiva de realizar esto sería con la instrucción `select` (mediante una llamada temporizada). Sin embargo, siguiendo los criterios del Perfil de Ravenscar (Sección 3), no se debe incluir ningún tipo de instrucción del `select`.

En una primera idea para abordar este problema, se diseña un paquete con dos objetos protegidos: `Servos_PO_Oscilacion` y `Servos_PO_Hilo`. Cada uno de estos objetos alberga los datos utilizados en cálculos de temporización (tiempo de paso, tiempo de inicio de semipasos y tiempo final del semipaso) y operaciones para trabajar con ellos y con las consignas de velocidad. Por otro lado, se tiene un tipo tarea que espera a que se abra la barrera `Consigna_Ha_Cambiado` del PO y en cuyo caso, la tarea obtiene el nuevo tiempo de paso del servo. Por otro lado, se utilizan los

timing events para que en caso de que no llegue una nueva consigna, se accione el procedimiento `Nuevo_Evento`, que se encarga de terminar el semipaso.

Pretendiendo simplificar el código e intentando eliminar la tarea, se reduce en una segunda idea (que es la que finalmente se ha implementado) a un solo objeto protegido. Para poder simplificar el código de esta manera también será necesario el uso de los *timing events* (Sección 3.1). El objeto protegido es necesario para evitar problemas de coherencia por accesos concurrentes desde la tarea de seguimiento de la entrada y desde los *timing events*.

En el Código (3) se muestra la especificación del objeto protegido `Servos_PO` del paquete `Servos` y las operaciones y datos que alberga.

```
1
   ------------------------------
3  -- Objeto Protegido Servos_PO --
   ------------------------------
5
   protected Servos_PO with Priority =>  Parametros.Techo_Servos_PO
7  is
      procedure Nueva_Consigna_Osc(Consigna: Velocidad_Angular);
9     procedure Nueva_Consigna_Hilo(Consigna: Velocidad_Angular);
      function Leer_Consigna_Velocidad(Eje: Tipo_Eje) return Velocidad_Angular;
11    procedure Nuevo_Evento_Oscilacion(Event: in out Timing_Event);
      procedure Nuevo_Evento_Hilo(Event: in out Timing_Event);
13 private
      Consigna_Vel: Velocidades_Ejes:=(0.0,0.0);
15
      -- Tiempo de paso actual (segundos por paso)
17    T_Paso_Oscilacion: Float:=Tiempo_Grande;
      T_Paso_Hilo: Float:= Tiempo_Grande;
19
      -- Hora a la que se inicio el actual semipaso
21    Inicio_Semipaso_Oscilacion: Real_Time.Time:=Real_Time.Clock;
      Inicio_Semipaso_Hilo: Real_Time.Time:= Real_Time.Clock;
23
      -- Hora a la que se finalizara el actual semipaso
25    Fin_Semipaso_Oscilacion: Real_Time.Time:=Real_Time.Clock+
        Real_Time.To_Time_Span(Duration(Tiempo_Grande/2.0));
27    Fin_Semipaso_Hilo: Real_Time.Time:=Real_Time.Clock+
        Real_Time.To_Time_Span(Duration(Tiempo_Grande/2.0));
29 end Servos_PO;
```

Código 3: Objeto protegido de `Controlador-Servos.adb`

A nivel de librería se declaran dos procedimientos más: `Nuevas_Consignas_Velocidad` y `Leer_Consigna_Velocidad` que sirven para especificar las nuevas consignas de velocidad que se mantendrán hasta que se llame nuevamente a este método y para leer la velocidad en grados por segundo de la consigna actual, respectivamente.

Cuando las nuevas consignas llegan al paquete por medio del procedimiento `Nuevas_Consignas_Velocidad`, se cancelan los *timing events* de los dos ejes que estaban establecidos para las consignas anteriores. A continuación, siempre que las nuevas consignas de velocidad no sean nulas, se invocan las operaciones protegidas de nuevas consignas tanto de oscilación como de hilo. Además, se hace una llamada al procedimiento `Reset` del paquete `Guardian` con la finalidad de confirmar que el programa se está ejecutando correctamente y está recibiendo nuevas consignas. Esto se hace para evitar que se active la `Alarma_Guardian`.

En el interior del objeto protegido las operaciones que trabajan con el eje de oscilación y del hilo

lo hacen de manera independiente, cada uno con sus procedimientos, pero mantienen un mismo esquema.

Cuando llega una nueva velocidad a una de las operaciones de nueva consigna del interior del PO se asigna la nueva velocidad a la variable protegida `Consigna_Vel` que luego será leída por la función protegida `Leer_Consignas_Velocidad`. A continuación, se actualiza el tiempo de paso de acuerdo a la nueva velocidad. Se comprueba si el nuevo tiempo de paso está dentro del rango posible y se establece un tiempo futuro en el que terminará el semipaso y que empleará el `Timing_Event` para establecer la ejecución futura del procedimiento de nuevo evento. Se establece también la dirección en la que actuarán los motores en base al signo de la consigna de velocidad.

Los procedimientos de nuevo evento (tanto de oscilación como de hilo) tienen como finalidad invocar al procedimiento `Actua_Step` del paquete `Motores` para que estos den un paso en la dirección que anteriormente se ha definido por la consigna de velocidad. Finalmente, realizada la orden de efectuar un paso de motor, se actualiza y establece el tiempo futuro en el que se volverá a ejecutar este procedimiento (en base a las mismas consignas de velocidad y que se modifica al recibir unas nuevas).

Cuanto mayor sea la velocidad que se le pasa al paquete `Servos` más veces se invoca a la operación `Actua_Step` con estados digitales diferentes ya que el tiempo de paso que emplean los *timing events* para ejecutar los procedimientos de nuevos eventos son menores. Por lo tanto, cuanta mayor velocidad, más rápidamente se desplazan los motores. Además, en función del estado digital del canal de dirección (depende del signo de la velocidad), el motor se desplazará en un sentido o en otro. Se muestra en la Figura (7) un resumen del comportamiento de las señales digitales de salida de los canales de pasos del motor (`Canal_Step`) y de dirección (`Canal_Direccion`) y del modo de variación de los pasos del motor.

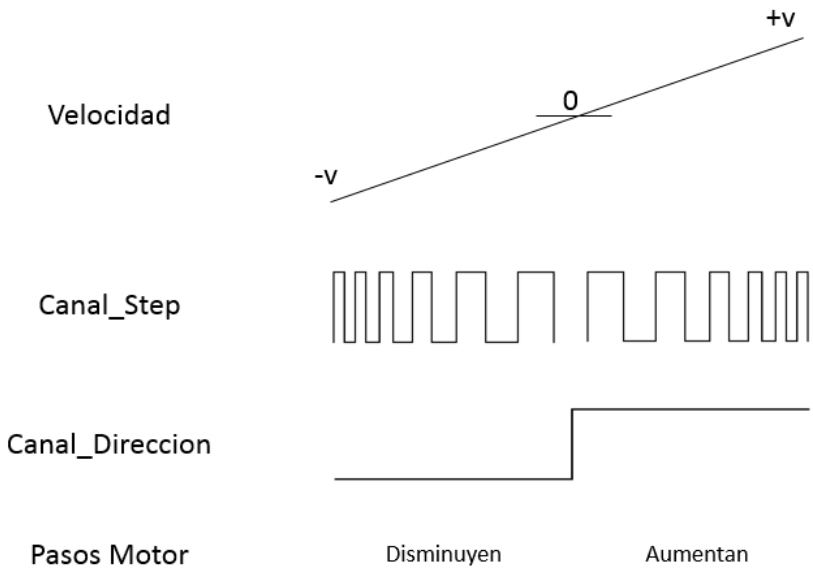

Figura 7: Estado de los canales de salida y de los pasos del motor con la velocidad

Para mostrar el uso de las diferentes operaciones del paquete `Servos` y dar una visión más clara, se modelan las interacciones del paquete mediante la creación de un diagrama de colaboración. Este diagrama se muestra en la Figura (8).

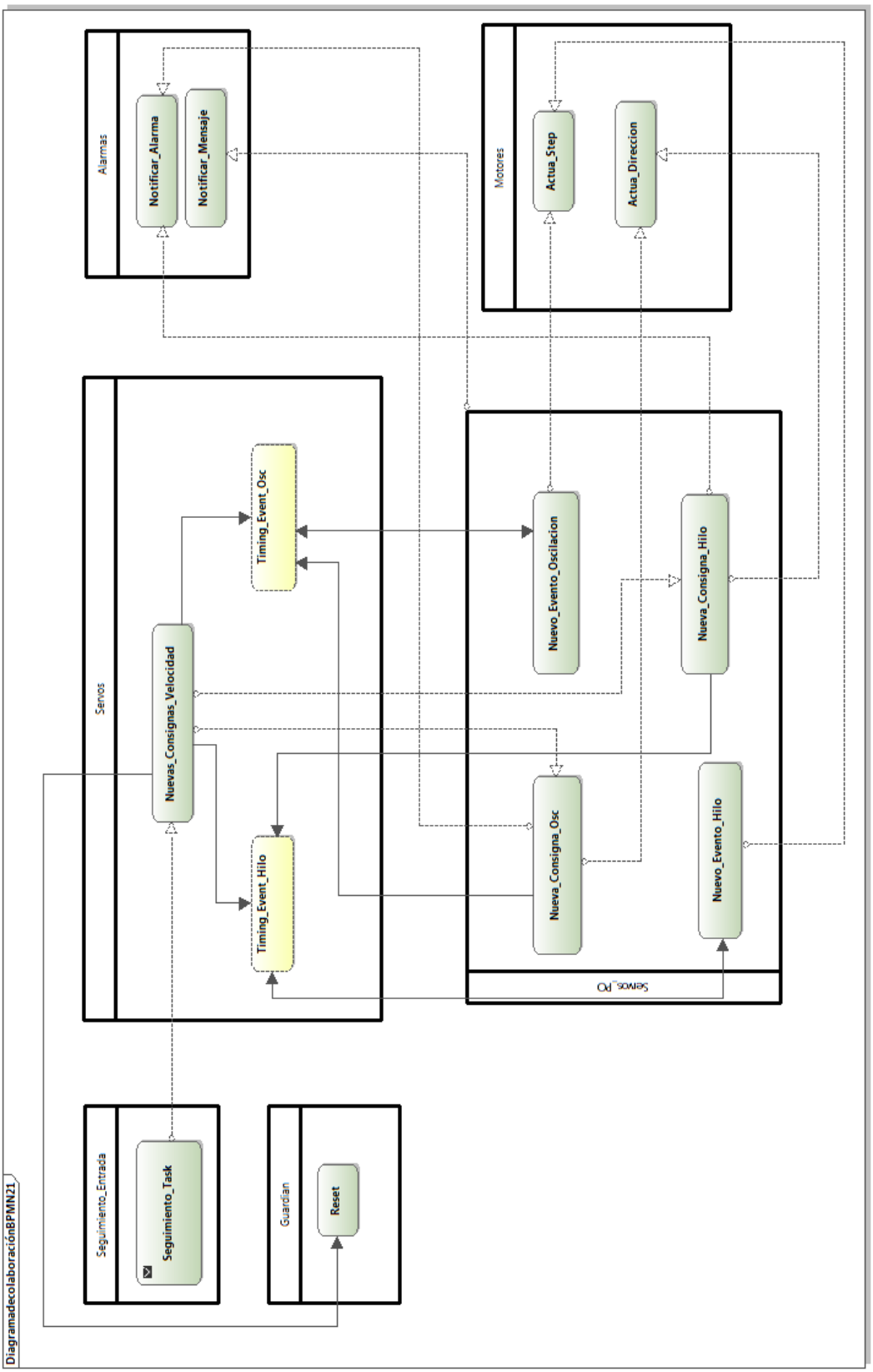

Figura 8: Diagrama de colaboración del paquete `Servos`

4.2.5. Paquete Reportero

El reportero se encarga de obtener la información de los demás paquetes para organizarla y mostrarla en pantalla. El paquete `Reportero` tiene la tarea `Reportero_Task`, que periódicamente actualiza la información de la pantalla.

La información que se muestra en pantalla es, por un lado, la consigna de entrada en grados y los pasos del motor del eje de oscilación de la antorcha de soldadura. Por otro lado, se muestran las alarmas activas que hay (pueden ser `Fallo_Motores`, `Frecuencia_Pulso_Excesiva`, `Alarma_Guardian`), que permanecen en pantalla hasta que se pulsa el botón de usuario y 3 líneas para 3 posibles mensajes relacionados con el funcionamiento del sistema. Las dos primeras se utilizan para notificar un mensaje relacionado con cualquier procedimiento (mensaje de nivel 1) o tarea (mensaje de nivel 2) de otro paquete mientras que la tercera se reserva para notificar un posible fallo de la propia tarea del reportero.

En la Figura (9) se muestra el diseño elaborado para mostrar la información en pantalla. Para visualizar mejor el diseño se activan todas las alarmas y se muestran unos mensajes de prueba y el mensaje de error del reportero.

Figura 9: Pantalla LCD con la información del reportero

4.2.6. Paquete Parametros

En este paquete se albergan los diferentes parámetros de configuración del sistema que pueden ser modificados en un futuro, como pueden ser aquellos que dependen de características específicas del HW o los canales del microcontrolador que se van a emplear. Además, también se encuentran las funciones que se emplean para el cambio de unidades entre las unidades de entrada analógica y el voltaje, entre el voltaje y el ángulo de motor desplazado, entre el ángulo y los pasos de motor efectuados, etcétera.

El paquete aúna también los periodos de las tareas que se encuentran en todo el programa, y se les atribuye una prioridad siguiendo la planificación *rate-monotonic scheduling* (RMS) en la que se fija a cada tarea una prioridad estática que depende de su periodo. A menor periodo, se atribuye una mayor prioridad. Para evitar interacciones excesivas, a cada tarea se le asigna un valor único, que no se utilizará en las prioridades del resto de tareas. Para que se muestre la información correctamente en pantalla se le atribuye a la tarea del reportero la máxima prioridad. Además de las prioridades de las tareas también se encuentran las de los objetos protegidos, que deben asignarse iguales a la máxima

de las prioridades de las tareas y *timing events* que los usan. Todas estas han sido definidas con el prefijo 'Techo' y se les han atribuido la prioridad `System.Priority'Last` a excepción de las definidas en los paquetes `Servos` y `Alarmas` a los que, para una correcta ejecución de los *timing events* se les ha atribuido la prioridad `System.Interrupt_Priority'Last`, que es superior a las del resto y es la prioridad usada por estos *timing events*.

4.2.7. Otros paquetes

Además de la programación de los anteriores paquetes es necesario realizar algunas modificaciones menores en el resto, tanto para adaptarlos a las nuevas librerías empleadas como a las implementaciones de los nuevos paquetes.

En el paquete `Alarmas`, se ha reducido el tamaño de los mensajes de acuerdo con las nuevas dimensiones de la pantalla empleada y debido a que su tarea es la que posee un periodo más corto, se ha elegido para que llame continuamente a la función `Hay_Fallo_Motores` y detectar un posible problema del HW.

En el paquete `Botones` se ha optado por emplear el propio botón de usuario que tiene incorporado el microcontrolador para el borrado de la notificación de alarmas en vez de usar un canal digital.

En la función `Leer_Consigna_Entrada` del paquete `Sensor_Entrada` se configura y se deja un tiempo de $t = 2\,\mathrm{ms}$ antes de que esta llame a la función protegida de su mismo nombre que se encargará de obtener y pasar a voltios el valor de la entrada analógica. El tiempo de espera entre que se invoca el procedimiento de configuración y de adquisición de la entrada analógica es fundamental para poder obtener un valor correcto y real. Este tiempo podría resultar demasiado alto y convendría cambiar la tarjeta de lectura de las entradas analógica para reducirlo.

5. Estrategia de Prueba

Una vez realizadas las implementaciones necesarias, se realiza un proceso de comprobación del código, tanto de los nuevos paquetes implementados, como de los ya existentes (que todavía no habían sido comprobados). Para constatar que el programa se ejecuta satisfactoriamente, se lleva a cabo una estrategia de prueba subdividida por paquetes, para poder localizar de manera más rápida los posibles fallos. Se desarrollan diferentes pruebas para comprobar de manera específica cada paquete. Una vez revisados los paquetes individualmente, se realiza la comprobación general, verificando la ejecución final del programa en su conjunto.

5.1. Paquete Entrada_Salida

Se crea el programa de prueba `Prueba_Entrada_Salida.adb` que comprueba que todos los métodos del paquete `Entrada_Salida` (inicialización de los pines, lectura de los pines analógicos y lectura y salida de los digitales) funcionan correctamente. Para la revisión de este paquete se debe hacer uso del osciloscopio y del generador de funciones. Además, se emplean también varios leds y resistencias de $R = 220\,\Omega$ para evitar sobrepasar la intensidad máxima de salida del canal digital.

El programa de prueba muestra si los pines se han inicializado correctamente e indica si la STM32 recibe señal digital o analógica. Se activa y desactiva una señal digital en un canal de salida del microcontrolador en función de si llega una señal digital de entrada o no. Esto también ocurre con otro canal de salida digital cuando se superan (o no) los $V = 2{,}5\,\mathrm{V}$ de señal analógica de entrada.

Para la comprobación, se han creado las diferentes señales de entrada con un generador de funciones y para la comprobación de las señales digitales de salida se han conectado leds a sus canales. Se muestra en la Figura (10) el montaje utilizado, donde se introduce una señal continua de $V = 2{,}7\,\mathrm{V}$ al canal de entrada analógica y se comprueba que se enciende el led.

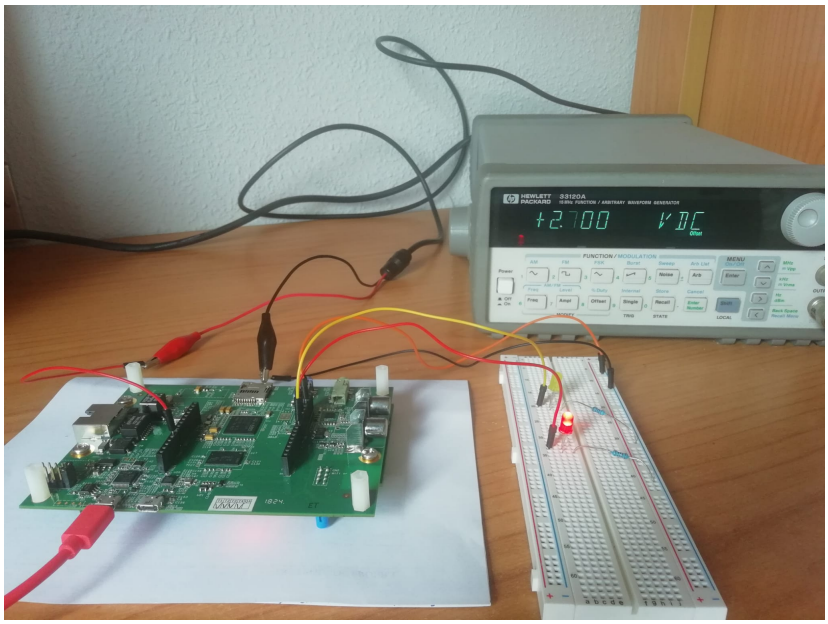

Figura 10: Comprobación del paquete `Entrada_Salida`

Para comprobar que todos los canales que se van a utilizar en el sistema de control funcionan correctamente, se realiza el mismo proceso de comprobación modificando los pines de uso (digitales de salida y de entrada) en la parte declarativa del programa.

Los resultados experimentales han sido satisfactorios: en pantalla se indica que los pines se inicializan correctamente, los valores de las entradas digitales y analógicas corresponden con los generados y las salidas digitales emiten un nivel de tensión alta cuando se les ordena.

5.2. Paquete Consola_Avanzada

El programa `Prueba_Consola_Avanzada.adb` es el encargado de comprobar que las operaciones implementadas en el paquete `Consola_Avanzada` funcionan como se espera.

Para ello, se muestran en pantalla diferentes textos de prueba, en diferentes localizaciones de la misma y en diferentes colores. También, se eliminan y sustituyen textos específicos. Con estas pruebas se verifica que las operaciones para establecer la localización en pantalla, para el cambio de color de fuente y las operaciones para la limpieza total o de una determinada región de la pantalla, funcionan correctamente.

5.3. Paquete Seguimiento_Entrada

El paquete `Seguimiento_Entrada` está formado por la tarea `Seguimiento_Task`. Esta se encarga de pasar a los servos las nuevas consignas de velocidad, que las calcula a partir de obtener la consigna de voltaje del paquete `Sensor_Entrada` y los ángulos de los ejes, del paquete `Motores`.

Para comprobar únicamente si la tarea funciona correctamente, se simulan los valores que devuelven las operaciones de los demás paquetes. Para ello, se crea una copia del paquete `Seguimiento_Entrada` donde se llama a un nuevo paquete llamado `Sim_Operaciones_Seguimiento_Entrada` que simula esos resultados. Este paquete incluye una función que devuelve valores de voltaje comprendidos entre los 2 y los 3 voltios, un procedimiento para actualizar los pasos del motor en función de la velocidad (simula el comportamiento de procedimiento `Actua_Step` del paquete `Motores`) y una función que devuelve la posición del motor en ángulos. Además, este paquete también incluye otras dos operaciones a las que llama la tarea `Seguimiento_Task` (en vez de al paquete `Servos`) para pintar en pantalla la información.

Sabiendo las consignas en voltios que estamos simulando en cada momento y la posición inicial de los motores, se puede calcular para cada nueva consigna, la posición de los motores y la velocidad de movimiento. Se demuestra que la tarea del paquete funciona correctamente comprobando que el cálculo numérico de las velocidades y posiciones de los motores en las diferentes situaciones (diferentes consignas de entrada con diferentes posiciones de los motores) corresponden con las que se muestran en la pantalla.

5.4. Paquete Motores

Se crea el programa `Prueba_Motores.adb` que prueba las diferentes operaciones del objeto protegido `Motores_PO` que se encuentra en el paquete `Motores`.

Esta prueba tiene como finalidad comprobar que las operaciones que controlan el acceso al HW de los motores responden correctamente a las indicaciones. Para examinar esto es necesario el uso del osciloscopio y del generador de funciones.

Por un lado, se ejecutan periódicamente los procedimientos relacionados con la habilitación de los motores (`Habilitar_Motores` y `Deshabilitar_Motores`) y con su actuación (`Actua_Direccion y Actua_Step`), pasándoles como parámetros diferentes valores. La habilitación de los motores, la actuación de pasos y la dirección en la que los tienen que realizar tienen dominio sobre 4 canales digitales de salida diferentes. Se utiliza la sonda del osciloscopio para comprobar si

cuando se envía una determinada orden a uno de los procedimientos, se envía una señal digital al canal de salida correspondiente.

El paquete `Motores` posee un objeto protegido con dos funciones: `Hay_Fallo_Motores` y `Leer_Angulos_Ejes`. Se utiliza la pantalla integrada del microcontrolador para mostrar periódicamente los ángulos, tanto del eje de oscilación como del hilo y si hay un fallo de los motores. También se muestran los pasos del motor para los dos casos. El ángulo depende de los pasos del motor que variarán en función de las llamadas al procedimiento `Actua_Step`. El fallo de los motores viene determinado por la llegada de una señal digital al canal `Canal_DI_Fallo_Motores`. Esta función se puede testar generando una señal de entrada en ese canal.

En la Figura (11) se muestra tanto el montaje llevado a cabo para comprobar la variación de las señales digitales en los canales de salida, como la información que se muestra en pantalla. Al no estar usando el generador de funciones para introducir una señal en el canal de fallo de motores, se muestra que no hay fallo en los mismos.

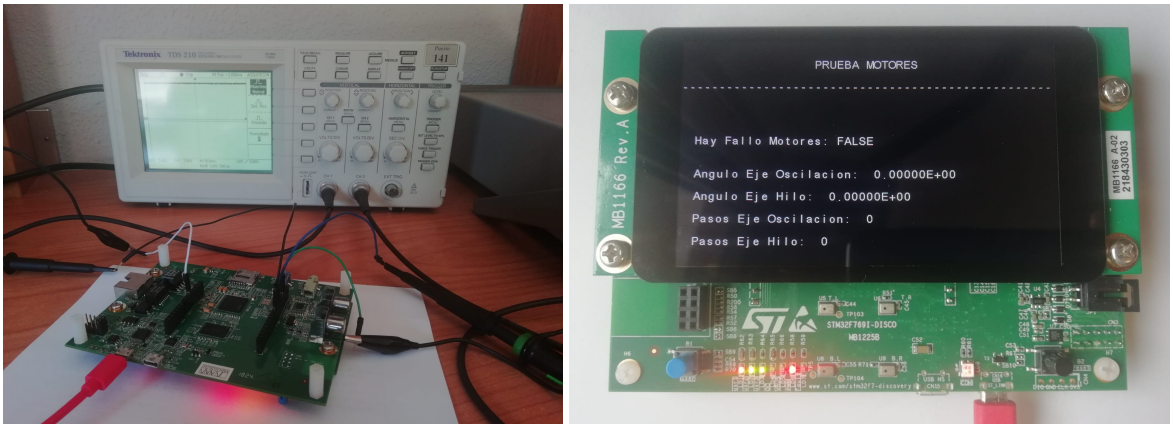

Figura 11: Comprobación del paquete `Motores`

5.5. Paquete Servos

Para comprobar el paquete `Servos` se realiza un *main* en el que se simula el comportamiento de la tarea `Seguimiento_Task` que, periódicamente, envía nuevas consignas de velocidad. Para comprobar este paquete es necesario el uso del osciloscopio.

Se han implementado tres procedimientos diferentes para revisar la respuesta en los 3 casos diferentes que pueden darse. Uno de ellos envía repetidamente consignas nulas de velocidad. Otro envía consignas constantes. El último, envía consignas que van variando.

Como se conoce el valor de la consigna que se envía, se puede prever la respuesta del microcontrolador y ver si corresponde con lo que acontece. Para ello, se conecta la sonda a las salidas digitales del microcontrolador. Se comprueba que cuando la consigna de velocidad es positiva, se detecta una señal en el `Canal_DO_Direccion` mientras que cuando es negativa, no se detecta. También se comprueba que en función de la magnitud de la consigna, en los canales de pasos de motor:

- No se detecta ninguna señal cuando las consignas que se envían son nulas.
- Se detecta una señal en cada canal cuyo periodo depende de la consigna de velocidad que le hayamos enviado. A más velocidad, más rápidamente varía la señal de salida entre 0 y 1.
- Se detecta una señal en cada canal cuyo periodo no es constante ya que las consignas de

velocidad que recibe también varían con el tiempo. Este caso es el que más se asemeja a la realidad.

5.6. Paquete Alarmas

El paquete `Alarmas` es el encargado de notificar y eliminar las tres posibles alarmas que se pueden dar y de manejar los posibles mensajes que se tienen que mostrar.

Para ver si el manejo de las alarmas y de los mensajes funciona correctamente, se ha realizado un programa que pinta en pantalla el estado de las alarmas y los mensajes.

Para simular una situación real, se le va a notificar desde el propio *main*, usando los procedimientos `Notificar_Alarma` y `Notificar_Mensaje` del paquete `Alarmas`, unas determinadas alarmas y unos mensajes. Después con los procedimientos `Leer_Alarmas` y `Obtener_Mensaje`, del mismo paquete, se obtienen los estados actuales de las alarmas y los mensajes que se pintan en pantalla. Se crean varias posibilidades, cambiando el estado de las alarmas y de los mensajes, y se comprueba que la información mostrada en pantalla corresponde con la enviada.

Para comprobar que el procedimiento `Borrar_Alarmas` también funciona, se notifica en pantalla cuando se puede pulsar (momento en el que no se notifican nuevas alarmas) para que, tras unos segundos, aparezcan como 'ACTIVA' o 'INACTIVA' el estado de cada alarma, en función de si se ha pulsado o no el botón de usuario.

5.7. Paquete Botones

Con la sustitución de la Raspberry PI por la STM32F769, se puede simplificar el controlador en lo referente al botón que elimina las alarmas, ya que el nuevo microcontrolador tiene incorporado un botón de usuario. Gracias a este botón, se puede omitir el canal digital de entrada `Canal_DI_Borrado_Alarmas`, que es al que estaría conectado el botón en la Raspberry. Se crea un programa `Prueba_Botones.adb` con el fin de comprobar que la función `Lee_Boton` devuelve en cada momento el correcto estado (pulsado o no pulsado) del botón de usuario.

5.8. Paquete Guardian

El paquete `Guardian` alberga una tarea concurrente que determina si se ha producido un fallo y se ha de notificar (`Alarma_Guardian`) y deshabilitar los motores o, por el contrario, todo funciona correctamente. Esta decisión es tomada en función de si se ha llamado al procedimiento `Reset` del paquete `Guardian` cada menos de un tiempo establecido de $t = 1\,\mathrm{s}$. En caso de que entre una llamada y la siguiente a este procedimiento haya pasado más de un segundo, se produce el fallo. Este procedimiento `Reset` es invocado desde `Nuevas_Consignas_Velocidad` que se encuentra en el paquete `Servos`. Sin embargo, a efectos de prueba se va a llamar a la operación desde el propio *main* `Prueba_Guardian.adb`. Haciendo esto, se controla la periodicidad con la que se invoca al procedimiento y se comprueba si, cuando el tiempo entre dos llamamientos es superior a un segundo, los motores se deshabilitan (se emite una señal digital de valor 0 por el pin `Canal_DO_Habilitar_Motores`). Además, se pinta en pantalla el estado de alarma que, llegados a ese punto, se mantiene como 'ACTIVA' hasta que se pulsa el botón de usuario.

5.9. Comprobación General

Una vez comprobado el correcto funcionamiento de cada paquete, se tiene que revisar que el programa en su conjunto, también responde de manera adecuada.

Para ello, se crea el archivo `Controlador_Main.adb`. En este *main*, se añaden, antes del procedimiento principal, el resto de paquetes mediante la instrucción `with`. Haciendo esto, empiezan a elaborarse las tareas albergadas en esos paquetes, comenzándose a ejecutar el programa. En el procedimiento principal de este *main* se inicializa la pantalla LCD del microcontrolador, se establecen los canales digitales a 0 y se invoca una vez a los procedimientos de habilitación de los motores, de actuación de pasos (del eje de oscilación y de hilo con valor 0) y de dirección (también con valor 0).

Con este programa cargado en el microcontrolador se comprueba que las diferentes funcionalidades implementadas se están ejecutando. La pantalla del reportero se muestra correctamente. El canal de habilitación de los motores se enciende. Se revisa (haciendo la conversión de grados a voltaje) que las consignas en grados que se muestran en pantalla a tiempo real corresponden con el voltaje suministrado a la plataforma. Se comprueba también que, en función de ese voltaje, se emite una señal digital en el canal de dirección de motores y que en los canales de pasos de los motores, se mantienen, tal y como se espera, constantemente variando. Dejando el programa ejecutándose durante un tiempo se comprueba que los pasos van variando y que las alarmas `Alarma_Guardian` y `Fallo_Motores` no se activan. Además, tampoco se muestra en pantalla ningún mensaje de error fruto de alguna excepción en las tareas o procedimientos de los paquetes.

Sin embargo, sí aparece la alarma de `Frecuencia_Pulso_Excesiva`. Esta alarma acontece cuando el tiempo de paso que se le exige al motor en base a las consignas de entrada es demasiado pequeño. En estos casos salta la alarma `Frecuencia_Pulso_Excesiva` y se reduce el tiempo de paso al mínimo. Al ocurrir esto, los pasos de los motores aumentan rápidamente. Esta alarma se da al no tener los valores exactos para determinados parámetros necesarios para las conversiones entre el voltaje de entrada y acciones que implican el HW del equipo de soldadura. Disponiendo del equipo de soldadura, se podrán atribuir los valores y especificaciones reales a las constantes del paquete `Parametros` relacionadas el HW del equipo.

6. Conclusiones

El procesador de 32 bits ARM Cortex-M7 de alto rendimiento, la memoria *flash* (de 2 megabytes) y la RAM (de 512+16+4 kilobytes), cubren holgadamente las necesidades de proyectos de magnitud similar al realizado. Esto, junto a las diferentes funciones de carácter general que posee la STM32 y su robustez al poder operar en un rango amplio de temperaturas ($t \in [-40, 105]\,^{\circ}\mathrm{C}$), nos lleva a concluir que esta tarjeta constituye una plataforma adecuada para el uso de aplicaciones industriales con necesidades de accionamiento de motores y lectura de sensores. [11, pp. 14-15]

También podemos extraer como conclusión que las limitaciones del Perfil de Ravenscar no han supuesto mayores dificultades en la implementación de la aplicación de control de seguimiento de oscilación. Ha sido posible reemplazar la instrucción `select` inicialmente prevista para hacer llamadas temporizadas por un mecanismo basado en un objeto protegido y un par de *timing events*.

En este trabajo, se ha completado la implementación de código para otorgar capacidad de oscilación a la antorcha de un equipo de soldadura. Aunque los principales paquetes de código para que el sistema funcione correctamente han sido implementados, se puede realizar en un trabajo futuro las siguientes continuaciones:

- A la hora de comenzar a ejecutarse el programa, se ha seleccionado la posición inicial de los motores como 0. Se puede implementar un procedimiento de inicialización para que los motores comiencen en una determinada posición.
- En el paquete `Parametros` se encuentran factores y constantes que se utilizan para la conversión y cambio de unidades entre el voltaje que llega al microcontrolador y el ángulo, desplazamiento, etcétera, que implica al HW del equipo de soldadura. Estos parámetros han de ser ajustados para que el SW funcione correctamente.
- Para que se ejecuten las tareas de los paquetes de una manera más eficiente se pueden ajustar las prioridades de las mismas en el paquete `Parametros`. Para modificar y atribuir la prioridad una prioridad más exacta, se realiza un análisis, con ayuda del paquete `Monitor_Tiempos` de los tiempos de ejecución de cada una. Con un estudio temporal también se pretende verificar matemáticamente el cumplimiento de los requisitos temporales que hasta ahora han sido verificados exclusivamente mediante tests.
- Realización de pruebas en el sistema de soldadura real.
- Incorporación de sensores de posición a los ejes de los motores a fin de conocer sus posiciones reales. Estos sensores servirían para detectar un posible error en el desplazamiento del motor (que actualmente es calculado mediante un contador de los pasos del motor).
- Estudio de la posibilidad de usar convertidores analógico-digital más rápidos.

Agradecimientos

En primer lugar quiero dar las gracias a Equipos Nucleares que se han encargado de proporcionar las especificaciones de la aplicación. El desarrollo de este proyecto ha sido posible gracias a ellos.

Quiero mostrar también mi agradecimiento a mi director Michael, quien me ha guiado a lo largo de todo el proyecto y ha estado siempre disponible para ayudarme. Haber podido trabajar con él ha sido una experiencia muy enriquecedora que me ha dotado de una perspectiva real del trabajo que se desempeña en la programación de microcontroladores.

También quiero agradecer a mis compañeros de carrera, Aritz, Ixaka, Santi y Toño, con los que he recorrido este camino.

Por último, quiero agradecer a mi familia: mis padres, Juan Carlos y Yolanda; mi hermana, Maider; mis abuelos, Beni y Julio; mis tíos, Nerea y Ramón. Ellos me han ayudado y apoyado cuando lo he necesitado.

Referencias

[1] STMicroelectronics, *Discovery kit with STM32F769NI MCU - User Manual*, 2018. Rev. 3.

[2] G. Gridling and B. Weiss, *Introduction to Microcontrollers*. 2007.

[3] M. P. Groover, "Automation." https://www.britannica.com/technology/automation, 2019.

[4] D. Greenfield, "Is raspberry pi ready for industry?," *Automation World*, 2019. Rescatado el 20/04/2020.

[5] H. Kopetz, *Real-time systems: design principles for distributed embedded applications*. New York: Springer, 2nd ed., 2011.

[6] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*. Addison Wesley, 4nd ed., 2009.

[7] G. C. Buttazzo, *Hard Real-Time Computing Systems*. Springer, Boston, MA, 3rd ed., 2011.

[8] F. Zhang and A. Burns, "Schedulability analysis for real-time systems with edf scheduling," *IEEE Transactions on Computers*, vol. 58, no. 9, 2009.

[9] J. Barnes, *Programming in Ada 2012*. Cambridge University Press, 2014.

[10] STMicroelectronics, *32F769IDISCOVERY Data brief*, 2016. Rev. 2.

[11] STMicroelectronics, *STM32F765xx STM32F767xx STM32F768Ax STM32F769xx Datasheet - production data*, 2016. Rev. 4.

[12] Arduino, "What is arduino?." https://www.arduino.cc/en/Guide/Introduction. Rescatado el 03/07/2020.

[13] D. Kushner, "The making of arduino," *IEEE Spectrum*, 2011. Rescatado el 03/07/2020.

[14] AdaCore, "Languages support." https://www.adacore.com/products/languages. Rescatado el 20/04/2020.

[15] D. Arranz, "Marco para el desarrollo de aplicaciones ada sobre microcontroladores stm32," *Universidad de Cantabria*, 2019.

[16] L. Asplund, B. Johnson, and K. Lundqvist, "Session summary: The ravenscar profile and implementation issues," in *Proceedings of the 9th International Real-Time Ada Workshop* (A. Burns, ed.), pp. 12–14, Ada Letters, 1999.

[17] A. Burns, "The ravenscar profile," *ACM Ada Letters*, 1999.

[18] A. Burns, B. Dobbing, and T. Vardanega, "Guide for the use of the ada ravenscar profile in high integrity systems," tech. rep., University of York, 2017.

[19] A. R. Manual, *D.15 Timing Events*, 3rd ed.

[20] B. S. Crawford, *Ada Essentials:Overview, Examples and Glossary*, ch. The select Statement.

[21] M. G. Harbour, "Gem 4: Pwm position control for radiocontrol servos." https://www.adacore.com/gems/gem-4, 2017. Rescatado el 27/04/2020.