CEA Comité Español de Automática

XLVI JORNADAS AUTOMÁTICA CARTAGENA 2025

UNIVERSIDADE DA CORUÑA

Jornadas de Automática

Sistema de control de un robot con Ada y Xtratum

Guasque, Ana[a,*], Fontalba, Marc[a], Ortiz, Luis[a], Simó, José[a], Balbastre, Patricia[a], Crespo, Alfons[a]

[a]*Instituto de Automática e Informática Industrial (ai2), Universitat Politècnica de València, Camino de Vera, s/n, 46022, Valencia, España*

To cite this article: Guasque, Ana, Fontalba, Marc, Ortiz, Luis, Simó, José, Balbastre, Patricia, Crespo, Alfons. 2025. Robot control system using Ada and Xtratum. Jornadas de Automática, 46. https://doi.org/10.17979/ja-cea.2025.46.12196

Resumen

Este artículo describe el diseño y programación de un robot móvil mediante dos configuraciones software distintas: una utilizando el lenguaje Ada con soporte de ejecución (Run-Time Support) y otra empleando un sistema particionado basado en el hipervisor XtratuM y el sistema operativo LithOS. Se exploran distintas funcionalidades de control del robot, integración con sensores y planificación en tiempo real, demostrando la aplicabilidad de tecnologías críticas en entornos embebidos. Además, se plantean las ventajas e inconvenientes de cada una de las configuraciones.

Palabras clave: Control en tiempo real, Algoritmos de tiempo real, planificación y programación, Guiado, navegación y control de vehículos, Sistemas informáticos de control integrados y aplicaciones, Diseño lógico, diseño físico e implementación de sistemas informáticos empotrados, Arquitecturas informáticas empotradas.

Robot control system using Ada and Xtratum

Abstract

This paper describes the design and programming of a mobile robot using two different software configurations: one using the Ada language with Run-Time Support and the other using a partitioned system based on the XtratuM hypervisor and the LithOS operating system. Different robot control, sensor integration and real-time scheduling functionalities are explored, demonstrating the applicability of critical technologies in embedded environments. In addition, the advantages and disadvantages of each of the configurations are discussed.

Keywords: Real-time control, Real-time algorithms, scheduling, and programming, Guidance, navigation and control of vehicles, Embedded computer control systems and applications, Logical design, physical design, and implementation of embedded computer systems, Embedded computer architectures.

1. Introducción

Los sistemas empotrados de tiempo real desempeñan un papel fundamental en aplicaciones críticas como robótica, aviónica, satélites, etc., donde la fiabilidad y la predictibilidad del comportamiento son requisitos esenciales. En estos entornos, los sistemas deben ser capaces de procesar señales de sensores y ejecutar algoritmos de control bajo estrictas restricciones temporales. El incumplimiento de los plazos puede dar lugar a comportamientos no deterministas o inseguros, especialmente cuando los robots interactúan con humanos en entornos colaborativos o se emplean en aplicaciones de alto riesgo, como vehículos autónomos o el sector aeroespacial (Burns and Wellings, 2009).

El desarrollo de sistemas embebidos en el ámbito de la robótica requiere gestionar múltiples tareas concurrentes con diferentes prioridades. Por ejemplo, un robot móvil puede necesitar adquirir y procesar datos de sensores, planificar trayectorias, controlar actuadores y comunicarse con otros sistemas, todo ello de forma paralela. Esta complejidad exige herramientas y lenguajes de programación que faciliten una

*Autor para correspondencia: anguaor@upv.edu.es

implementación concurrente segura y eficiente.

En el ámbito aeroespacial, el lenguaje Ada ha tenido históricamente un papel destacado, siendo en muchos casos la opción preferente en los desarrollos realizados por agencias espaciales. En las últimas dos décadas, la adopción de sistemas particionados basados en hipervisores ha permitido diseñar sistemas aviónicos y espaciales más robustos y confiables, con ventajas adicionales en términos de reducción de hardware, costes e incremento de la certificabilidad.

El desarrollo del sistema robótico presentado en este trabajo se enmarca en el contexto aeroespacial y aborda aspectos relacionados con el diseño, validación y certificación de software para sistemas embarcados. Por ello, se propone un enfoque dual: por un lado, el diseño tradicional basado en Ada, y por otro, el diseño de un sistema particionado.

En este contexto, Ada se posiciona como una solución idónea para el desarrollo de sistemas críticos de tiempo real, gracias a su fuerte tipado, semántica determinista y soporte nativo para la concurrencia. Ada ofrece mecanismos estructurados para la sincronización y el control de tareas, permitiendo la implementación predecible de sistemas paralelos. Además, su compatibilidad con técnicas de análisis estático facilita la verificación de propiedades temporales, aspecto esencial en sistemas críticos (Taft et al., 2022). Estudios comparativos han demostrado que los desarrollos en Ada presentan mayor mantenibilidad y menor propensión a errores que aquellos realizados en C/C++ (Kornecki and Zalewski, 2009).

Una de las principales ventajas de Ada es su capacidad para ejecutar aplicaciones sin requerir un sistema operativo de tiempo real (RTOS), mediante su entorno de ejecución (*Run-Time Support*, RTS). Este proporciona servicios esenciales como gestión de tareas, temporizadores y sincronización, permitiendo el desarrollo de aplicaciones autónomas sobre hardware específico. Cuando se requieren mayores garantías de certificabilidad, puede emplearse el perfil Ravenscar, una configuración restringida del lenguaje que asegura el análisis temporal y elimina construcciones complejas (Burns et al., 2004). Este perfil ha sido ampliamente adoptado en sectores como el aeroespacial y el automotriz, demostrando su eficacia en misiones críticas como *ExoMars* de la ESA.

No obstante, cuando se utilizan lenguajes como C para implementar la lógica del sistema, o cuando las necesidades de concurrencia y comunicación superan lo que puede ofrecer un RTS, resulta necesario emplear un RTOS. En estos casos, la interfaz de programación del sistema operativo (API) adquiere un rol fundamental, al definir la interacción entre la aplicación y el sistema operativo. Para garantizar portabilidad, interoperabilidad y certificación, es crucial que dicha API cumpla con estándares reconocidos como ARINC-653 (en entornos aeronáuticos) o POSIX (en sistemas embebidos en general) (Aeronautical Radio, 2010; Society, 2008).

A medida que los sistemas embebidos integran funcionalidades con diferentes niveles de criticidad, garantizar su aislamiento y certificabilidad se vuelve más complejo. Esto ha impulsado la adopción de arquitecturas particionadas, en las que el sistema se divide en particiones independientes, con aislamiento espacial y temporal. Estas arquitecturas, inspiradas en el concepto MILS (*Multiple Independent Levels of Security/-Safety*) (Rushby, 1999), permiten contener fallos y asegurar que errores en una partición no afecten al resto del sistema.

El componente clave de estas arquitecturas es el hipervisor, que actúa como una capa intermedia entre el hardware y las particiones, gestionando su planificación, comunicación y control. XtratuM (Masmano et al., 2009) es un hipervisor *bare-metal* diseñado específicamente para sistemas de tiempo real críticos, que proporciona un aislamiento estricto y soporta configuraciones mixtas. Combinado con un RTOS ligero como LithOS (Masmano et al., 2010), permite implementar sistemas embebidos robustos que integran funcionalidades con distintos niveles de criticidad.

En este trabajo se propone el diseño y desarrollo de una aplicación orientada al control de un sistema de tiempo real crítico, representado por un robot móvil con múltiples funcionalidades: detección de obstáculos, seguimiento de trayectorias, monitorización de eventos, grabación de datos, entre otras. Para ello, se contemplan dos configuraciones. En la primera, se emplea Ada junto con su RTS y el perfil Ravenscar para implementar tareas concurrentes de control, navegación y supervisión. En la segunda, se adopta una arquitectura particionada basada en el hipervisor XtratuM y el sistema operativo LithOS, utilizando el lenguaje C para desarrollar las funcionalidades dentro de cada partición. Finalmente, se lleva a cabo un análisis comparativo entre ambas configuraciones, evaluando aspectos como la facilidad de implementación, la mantenibilidad del código y el cumplimiento de los requisitos temporales del sistema.

2. Descripción del sistema a controlar

En este trabajo se desea diseñar e implementar un sistema empotrado en un robot móvil cuya imagen se muestra en la Figura 1. El robot Alphabot está formado por un chasis y un adaptador para la placa de desarrollo. La placa con la que se va a trabajar es la Cora Z7, que es una plataforma de desarrollo que combina hardware y software, diseñada por Digilent Inc. La arquitectura Zynq-7000 integra un procesador ARM Cortex-A9 de núcleo simple o doble de 667 MHz con lógica programable FPGA (Field-Programmable Gate Array) en un solo chip. Esta combinación permite aprovechar las ventajas de un procesador de propósito general junto con la flexibilidad y capacidad de procesamiento paralelo proporcionadas por una FPGA.

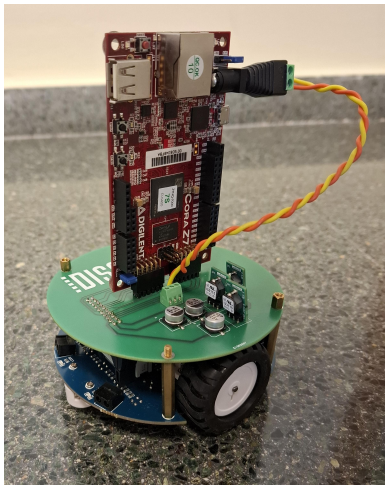


Figura 1: Alphabot controlado a través de una Cora z7 de Xilinx.

Para el diseño de la FPGA se utiliza el software Vivado, que permite crear y adaptar diseños de FPGA y SoC de AMD. Para este proyecto, se van a incluir en el diseño de la FPGA los LEDs RGB, los pulsadores y los dos conectores Pmod, que se describirán más adelante. Una vez realizado el diseño, se genera el Bitstream, que es el fichero final requerido para programar la FPGA. Este fichero es común a las dos configuraciones.

La plataforma móvil a la que se conecta la placa Cora cuenta con sensores fotoeléctricos infrarrojos para la detección de obstáculos y sensores para el seguimiento de líneas. Además, posee motores controlados por modulación de ancho de pulso (PWM), donde se actúa sobre la duración de los impulsos de tensión que se le aplican. De este modo, se controla tanto la velocidad como la dirección de los motores. Las funciones a implementar en el sistema empotrado son: seguimiento de un camino (navegación controlada), navegación libre con detección de obstáculos, navegación emulada, monitorización de eventos y generación de alarmas.

Para la implementación de estas funcionalidades se van a seguir las dos configuraciones anteriormente comentadas (Figura 2) y que se detallan a continuación.

Los distintos sensores y actuadores de la plataforma móvil se integran en la placa a través de conectores PMOD (Peripheral Module) que para su acceso desde el procesador requieren la programación de la FPGA de la placa. El diseño de la FPGA para esta integración se realiza a través de la herramienta de Vivado que proporciona Xilinx. Además, para el proceso de desplegado del software en la placa y la comunicación entre el sistema de desarrollo y el de ejecución se utilizan las herramientas que el fabricante del hardware proporciona. (Xilinx Inc., 2021).

Para el desarrollo de la navegación emulada, se va a hacer uso de los dos módulos periféricos PMODs de la placa Cora. Digilent dispone de un gran número de dispositivos auxiliares Pmods que pueden conectarse a la placa para ofrecer una mayor funcionalidad a la misma. En este proyecto, para emular los sensores fotoeléctricos y los motores, se van a utilizar el Pmod SWT y Pmod 8LD, respectivamente. Ambos dispositivos utilizan un protocolo de comunicación GPIO. El Pmod SWT ofrece 4 switches (ON/OFF) que funcionan como un conjunto de entradas binarias, emulando la presencia detectada en un sensor activando un switch. El Pmod 8LD tiene 8 LEDs de alto brillo, de forma que cada LED se puede iluminar de manera individual mediante una señal lógica en alto. El encendido y apagado de LEDs emulará la puesta en marcha y paro de los motores, así como el sentido de giro de los mismos. Ambos Pmods tienen interfaz GPIO (General Purpose Input/Output). En la Figura 3 se muestra la conexión de los Pmods para la navegación emulada.

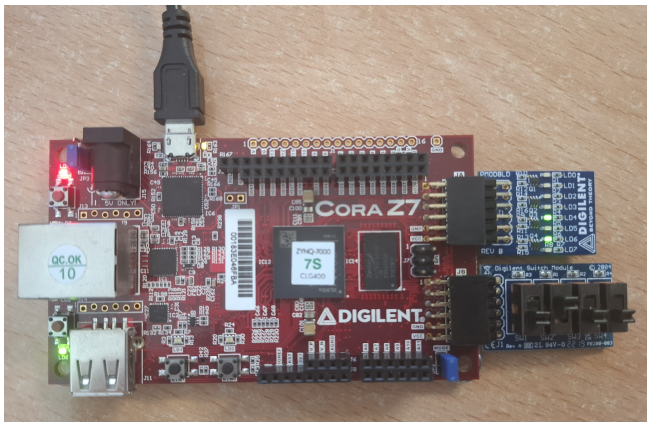


Figura 3: Conexiones de la placa Cora para la navegación emulada.

A continuación se describen las dos configuraciones seguidas para implementar las diferentes funcionalidades del sistema.

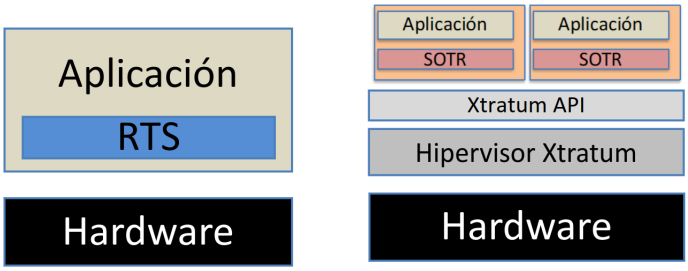
3. Configuración 1: Aplicación en Ada sin RTOS

En esta configuración, se va a implementar el control del robot móvil en Ada con el Run-Time support. Como se ha comentado anteriormente, el RTS ofrece mecanismos que proporcionan las funcionalidades necesarias para ejecutar programas en Ada, sin necesidad de contar con un RTOS. A diferencia de otros lenguajes que dependen del sistema operativo, Ada es ampliamente utilizado en entornos donde el sistema operativo puede no existir o tener algunas funcionalidades mínimas. El RTS de Ada se encarga de la gestión de la concurrencia (con tareas), gestión de memoria, manejo de excepciones y mecanismos de entrada y salida básica. Además, cuando el ámbito de aplicación es crítico, Ada permite seleccionar el perfil Ravenscar, que tiene recursos limitados y evita diferentes fuentes de indeterminismo, lo que asegura un comportamiento predecible.

Las herramientas de software que se utilizan, además de las imprescindibles de Xilinx para la comunicación con la placa Cora, son el compilador cruzado del lenguaje Ada para la arquitectura ARM y software para la comunicación serie entre la placa y la estación de desarrollo. El procedimiento de trabajo es el siguiente: en la estación de desarrollo se implementan los programas en Ada, se compilan para la arquitectura ARM y el perfil Ravenscar generando un fichero ejecutable que se despliega en la placa.

Con esta configuración, se implementa el control de movimientos básicos del robot (avance, retroceso, giro) empleando procedimientos. Para la implementación del PWM para el control de la velocidad y dirección de los motores, se utilizan tareas periódicas en Ada. Las tareas se ejecutan periódicamente siguiendo un algoritmo de planificación basado en prioridades fijas. Se utiliza un temporizador y se asegura la temporización mediante el uso del paquete Ada.Real-Time.

A modo de ejemplo, en la Figura 4 se muestra el código en Ada para la definición de la variable RGB, que hace referencia a los LEDs RGB integrados en la cora. En la izquierda se muestra el fichero de especificación (.ads), que muestra la definición de la variable RGB de tipo GPIO, con el campo de datos que define los bits que corresponden a cada uno de los



(a) Configuración 1: Aplicación en Ada sin RTOS.

(b) Configuración 2: Sistema particionado con Xtratum y Lithos en lenguaje C.

Figura 2: Configuraciones aplicadas para el control del robot.

```
with System.storage_elements;

package GPIO is

        --leds GRB
        type RGBtype is (red, green, blue);

        for RGBtype use (
                red=>2#001#,
                green=>2#010#,
                blue=>2#100#);

        for RGBtype'Size use 3;

        type Ctrl_RGB is record
                rgbColor0:RGBtype;
                rgbColor1:RGBtype;
        end record;

        for Ctrl_RGB use record
                rgbColor0 at 0 range 0..2;
                rgbColor1 at 0 range 3..5;
        end record;

        type GPIO_type is record
                datos:Ctrl_RGB;
                control: integer;
        end record;

        RGB: GPIO_type2;
        for RGB'address use system.storage_elements.To_address(16#41200000#);

        procedure InitRGB;
        procedure writeRGB0 (color:RGBtype);
        procedure writeRGB1 (color:RGBtype);


end GPIO;
```

(a) Fichero de especificación.

```
package body GPIO is

   procedure InitRGB is
   begin
      RGB.control:=16#00#;
   end InitRGB;

   procedure writeRGB0 (color:RGBtype) is
   begin
      RGB.datos.rgbColor0:=color;
   end writeRGB0;

   procedure writeRGB1 (color:RGBtype) is
   begin
      RGB.datos.rgbColor1:=color;
   end writeRGB1;


end GPIO;
```

(b) Fichero de implementación.

Figura 4: Código en Ada para el acceso a los LEDs RGB mediante GPIO.

LEDs, y el campo de control, que se inicializa a modo salida. También se especifica la dirección de memoria a la que se asigna esta variable, que tiene que ser coherente con el diseño previo de la FPGA. En la derecha se muestra el fichero de implementación (.adb) de los procedimientos de inicialización y escritura en cada uno de los LEDs. Basta con hacer uso de este paquete y de los procedimientos que incluye para inicializar los LEDs y, posteriormente, asignar un color.

Además, el robot es capaz de seguir una línea negra en el suelo utilizando sensores reflectivos. Se introduce el uso de tareas esporádicas para el manejo de interrupciones generadas por cambios en los sensores.

4. Configuración 2: Sistema particionado con Xtratum y LithOS.

En esta configuración se pretende implementar el mismo sistema pero mediante el uso de un sistema particionado basado en el hipervisor Xtratum. XtratuM es un hipervisor de tipo 1 (bare-metal) diseñado específicamente para sistemas embebidos de tiempo real críticos. Entre sus características principales se destaca la virtualización para tiempo real, proporcionando aislamiento espacio-temporal estricto entre particiones y planificación cíclica, canales de comunicación entre particiones, soporte para multi-arquitectura, gestión de interrupciones, temporizadores y servicios de tiempo así como manejo de fallos (health monitoring). XtratuM garantiza el aislamiento entre particiones y LithOS facilita la configuración del sistema.

Se divide el sistema en particiones independientes: se define una partición que se encargará de la sensorización y otra que se encargará del control de motores. Los mecanismos de comunicación entre particiones (puertos y canales) que define XtratuM se basan en la especificación de ARINC-653 y se realiza a través de mensajes. Un puerto permite que una partición específica escriba y lea mensajes a través de un canal, entre un puerto fuente y un puerto destino, especificado en el fichero de configuración de XtratuM. Tanto los canales, puertos, máximo tamaño de los mensajes y máximo número de mensajes se definen completamente durante la fase de configuración del sistema. Estos servicios incluyen puertos *sampling* y puertos *queuing*, escogiendo los *sampling* para el desarrollo de este trabajo, ya que únicamente se va a enviar un dato de manera periódica. En la Figura 5 se representa la configuración del sistema particionado y el canal de comunicación entre particiones.

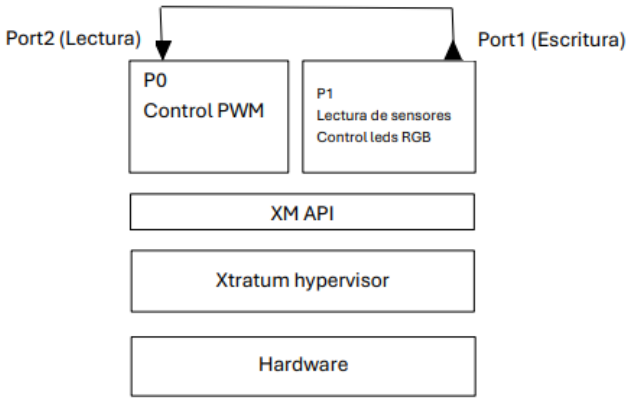


Figura 5: Estructura de las particiones para esta configuración

El dato que se va a enviar a través del canal se ha definido como una estructura de datos con dos campos, que tienen que ver con la caracterización del PWM: por un lado, el valor cuando el pulso está en alto y, por otro lado, durante cuánto tiempo tiene que estar en alto. Por ejemplo, cuando se detecte un obstáculo al frente, se establece el valor del pulso a cero y, cuando no se detecte, se establecen el valor y tiempo del pulso a la velocidad nominal deseada.

Cada partición tiene asignado un slot temporal en un plan mayor de hiperperiodo. Se depura el comportamiento mediante la monitorización de mensajes entre particiones.

A continuación, se van a describir en detalle las fases de diseño e implementación de cada una de las particiones.

Partición para el control de los motores.

En esta partición se ha implementado un proceso periódico, cuyo período viene definido en el fichero de configuración de XtratuM. El pseudocódigo de este proceso se indica a continuación:

```
#include <xm.h>
#include <gpio.h>
#other includes
void main()
{
    /* Creacion de puerto sampling
    como destino */
    CREATE_SAMPLING_PORT(params);

    while (1)
    {
        /*Leer dato recibido en el
        puerto sampling*/
        READ_SAMPLING_MESSAGE(params);
        //Tomar el tiempo actual
        XM_get_time(params);
        /*Escribir el dato recibido en el
        PWM (valor y duracion)*/
        gpio_write(dato);
        /*Cuando finalice el tiempo en alto,
        escribir un 0 en el PWM*/
        gpio_write(0);
        /*Esperar al siguiente
        periodo de particion*/
        XM_idle_self;
    }
}
```

Partición para la sensorización

En esta partición se han implementado dos procesos: por un lado, el proceso de lectura de los sensores fotoeléctricos y por otro, el control de los LEDs RGB de la cora para la generación de alarmas. Se debe tener en cuenta que los sensores fotoeléctricos funcionan con lógica negada. El proceso de lectura de sensores determina el valor del dato a enviar por el puerto sampling, en función del valor leído en el sensor, tal y como se indica a continuación:

```
#include <xm.h>
#include <gpio.h>
#other includes
void LeerSensores()
{
    while(1) {
        /*Leer dato de sensores*/
        GPIO_Read(sensores);
        switch (sensores) {
            case 0:
                // Obstaculo en 2 sensores
                valorAlto = 0;
                tAlto = 0;
                break;
            case 1:
                // Obstaculo en derecha
                valorAlto = Avance_Drecha;
                tAlto = Velocidad_Giro;
                break;
            case 2:
                // Obstaculo en izquierda
                valorAlto = Avance_Izquierda;
                tAlto = Velocidad_Giro;
                break;
            case 3:
                // No hay obstaculo
                valorAlto = Avance_Ambos;
                tAlto = Velocidad_Nominal;
                break;
        }
        /*Escribir dato calculado en el
            puerto sampling*/
        WRITE_SAMPLING_MESSAGE(params);
    }
}
```

Al igual que en la configuración 1, en esta configuración también se implementa el control de los LEDs RGB mediante la interfaz GPIO. Para ello, se ha implementado el código fuente en C, cuyo fichero de cabeceras se muestra en la Figura 6.

```
#include <xm.h>

// Function to initialize the GPIO device
int gpio_init(int base_addr);

// Function to set the direction of a GPIO pin (0 for output, 1 for input)
int gpio_set_direction(int did, int direction);

// Function to write a value to a GPIO pin (1 for high, 0 for low)
int gpio_write(int did, int value);

// Function to read the value of a GPIO pin (returns 1 for high, 0 for low)
int gpio_read(int did, int *value);
```

Figura 6: Cabecera del paquete GPIO.h

Para manejar los LEDs, se debe asignar la dirección de memoria a la variable y, a continuación, establecer la dirección del pin a modo salida. Entonces ya se puede usar la función de escritura, tal y como se indica en la Figura 7.

```
#define GPIO_ADDRESS_RGB 0x41200000

//Port initialization
rgbs = gpio_init(GPIO_ADDRESS_RGB);
gpio_set_direction(rgbs, 0x00);

retCode = gpio_write(rgbs, val);
```

Figura 7: Uso del código fuente del GPIO.

5. Comparativa entre enfoques

En esta sección se comparan los dos enfoques: por un lado, el uso del lenguaje Ada junto con su Run-Time Support (RTS), y por otro, una arquitectura particionada basada en el hipervisor XtratuM, con el sistema operativo LithOS y desarrollo en C.

La primera opción, utilizar Ada con RTS, se apoya en las capacidades del propio lenguaje para ofrecer concurrencia, sincronización y control temporal sin necesidad de un sistema operativo. Ada incluye construcciones del lenguaje para definir tareas concurrentes, manejar su sincronización mediante objetos protegidos, y controlar el tiempo de ejecución con instrucciones como delay. Esto permite desarrollar sistemas de control reactivos, como los necesarios para un robot móvil, de manera sencilla y verificable. Además, cuando se utiliza el perfil Ravenscar, se garantiza la capacidad de análisis temporal y se facilita la certificación. Al ejecutarse sin un RTOS, gracias al RTS, esta configuración implica una menor sobrecarga de sistema, mayor control sobre el hardware y una arquitectura más simple, ideal para sistemas con un número reducido de tareas de criticidad bien delimitada.

Sin embargo, cuando el sistema requiere mayor robustez frente a fallos, escalabilidad o la ejecución conjunta de componentes con distintos niveles de seguridad o fiabilidad, resulta más adecuado optar por una arquitectura particionada. En este segundo enfoque, el hipervisor XtratuM permite dividir el sistema en particiones independientes, cada una con su propio entorno de ejecución. Una de las ventajas fundamentales de este diseño es el aislamiento espacio-temporal entre particiones, lo que impide que un fallo en una afecte al resto. En este caso, dentro de cada partición puede ejecutarse un RTOS como LithOS, que proporciona los servicios básicos de planificación, sincronización, comunicación y temporización. El desarrollo de la lógica de control se realiza en lenguaje C y la interacción con el sistema operativo se produce a través de una API que, para garantizar la portabilidad y certificabilidad del sistema, debe seguir un estándar como ARINC-653 o POSIX.

La principal diferencia entre ambos enfoques radica en su complejidad y grado de aislamiento. Mientras que la configuración con Ada y RTS es más ligera y directa, adecuada para sistemas sencillos y con requisitos temporales exigentes, la arquitectura con XtratuM y LithOS ofrece una base más robusta para sistemas heterogéneos o críticos, a costa de una mayor dificultad en la configuración e implementación. Este tipo de soluciones es habitual en sectores donde la seguridad es prioritaria, como la aeronáutica, el ferroviario o el espacio, y empieza a extenderse también a entornos como la robótica avanzada.

6. Conclusiones

Este trabajo demuestra cómo es posible programar un robot móvil utilizando diferentes tecnologías de software embebido. La experiencia práctica con Ada y con XtratuM-LithOS permite entender las implicaciones del diseño en tiempo real, el uso de perfiles restringidos como Ravenscar y la planificación particionada. La elección entre las dos configuraciones presentadas en este trabajo dependerá del grado de criticidad del sistema, el número de funcionalidades a integrar y la necesidad o no de aislamiento entre componentes. Para un robot móvil con funcionalidades bien delimitadas y de criticidad homogénea, Ada con RTS es una solución eficaz y eficiente. En cambio, para robots complejos que integren funciones críticas junto a otras no críticas, o que requieran una arquitectura escalable y segura, la opción basada en XtratuM, LithOS y C será la más adecuada.

Referencias

Aeronautical Radio, I. A., 2010. ARINC Specification 653: Avionics Application Software Standard Interface. https://www.aviation-ia.com/products/arinc-653, part 1 - Required Services.

Burns, A., Dobbing, B., Vardanega, T., Jun. 2004. Guide for the use of the Ada Ravenscar profile in high integrity systems. Ada Lett. XXIV (2), 1–74.
URL: https://doi.org/10.1145/997119.997120
DOI: 10.1145/997119.997120

Burns, A., Wellings, A. J., 2009. Real-time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX. Addison-Wesley.

Kornecki, A. J., Zalewski, J., 2009. Certification of software for real-time safety-critical systems: state of the art. Innov. Syst. Softw. Eng. 5 (2), 149–161.
URL: https://doi.org/10.1007/s11334-009-0088-1
DOI: 10.1007/S11334-009-0088-1

Masmano, M., Ripoll, I., Crespo, A., Metge, J., 2009. XtratuM: a hypervisor for safety critical embedded systems. In: 11th Real-Time Linux Workshop. Vol. 9. Citeseer.

Masmano, M., Valiente, Y., Balbastre, P., Ripoll, I., Crespo, A., Metge, J., 2010. LithOS: a ARINC-653 guest operating for XtratuM. In: Proc. of the 12th Real-Time Linux Workshop.

Rushby, J., 1999. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Tech. rep., NASA Langley Technical Report.

Society, I. C., 2008. IEEE Std 1003.1-2008 (POSIX): Standard for Information Technology – Portable Operating System Interface. https://standards.ieee.org/standard/1003_1-2008.html.

Taft, S. T., Duff, R., Brukardt, R., Ploedereder, E., 2022. Ada reference manual.

Xilinx Inc., 2021. Zynq-7000 SoC Technical Reference Manual. San Jose, CA, USA.