Universidad de Valladolid

UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería Electrónica Industrial y Automática

Control por computador basado en C++

Autor:

Rico Ramón, Jorge

Tutor:

Mazaeda Echevarria, Rogelio
Departamento de Ingeniería de Sistemas y Automática

Valladolid, febrero 2019

Resumen

Es frecuente utilizar programación estructurada cuando queremos acometer tareas de control y simulación digital en la ingeniería.

Sin embargo, el lenguaje C++ ha adquirido un importante papel en el ámbito de la programación de propósito general, gracias a sus potentes características como herencia, polimorfismo, clases, objetos y estructuras dinámicas con los que se crean aplicaciones de gran interés.

Si a esto le añadimos ejecutar múltiples hilos de ejecución, donde cada entidad es autónoma y se relaciona con los demás mediante reglas, entonces C++ es también idóneo para conformar sistemas de control y simulación, en muchos campos de la ingeniería.

En el presente trabajo demostraremos las capacidades de este lenguaje de programación en el ámbito del control de sistemas.

Palabras clave: C++, multitarea, control, simulación, sistemas

Abstract

Is common practice to use structured programming in Control Engineering and simulation tasks.

Therefore, the C++ language has acquired an important role in general-purpose programming, due its powerful features like inheritance, polimorfism, objects and dynamic data structures, so we can make interesting applications.

In addition, we can run several threads. Entities will cooperate with each other using simple rules.

The present document will show C++ capabilities in a control system enviroment.

Keywords: C++, multitasking, control, simulation, systems

Índice

1. Introducción

Los controladores digitales en la industria han experimentado un rápido desarrollo que han permitido pasar de los relés y los cableados a los microprocesadores.

Los microprocesadores hacen que los controladores sean versátiles, fácilmente configurables, sin embargo, los procesos a controlar a su vez se han hecho cada vez más complejos, necesitando procesar crecientes cantidades de datos, con rapidez y fiabilidad.

Así mismo los microprocesadores han experimentado grandes mejoras y en una pequeña placa de controlador podemos disponer de chips potentes con características interesantes a un precio asequible. Desde hace décadas existen los microprocesadores que son capaces de ejecutar varios hilos de forma simultánea y eso puede suponer un mayor rendimiento al que hemos de dar provecho.

C++ que se ha hecho célebre como lenguaje de programación de propósito general, consolidado y en continua evolución. Nos centraremos en el estándar C++11 y su librería STL, donde reside sus características más notables y que nos proporcionará funcionalidades en la gestión de hilos, memoria y comunicación. En las páginas de este trabajo, se mostrará como C++ permite diseñar toda clase de sistemas y controladores fiables, en un entorno multitarea, de una forma eficaz.

Es habitual que C++ suponga al principio una cierta dificultad para el ingeniero de sistemas que quiera sacar provecho de todas las características útiles para su campo de actuación. Solventar esta dificultad y conseguir que el lector se familiarice con el lenguaje para que pueda programar sus propias aplicaciones en el campo de los sistemas y la automática se convierte en el principal objetivo de este trabajo.

Iniciamos con una breve introducción en C y C++ en el ámbito de los sistemas, para sumergirnos de lleno en el diseño de una aplicación de simulación y control mediante programación orientada a objetos. Pretendemos satisfacer todas las necesidades; desde la simulación multitarea de sistemas físicos, pasando por la interconexión de sus elementos, buses, controladores, recopilación de datos, e interactividad con el usuario. Con esto, pretendemos completar un recorrido relevante para quien quiera profundizar en el uso de este potente lenguaje, en lo académico y profesional.

El entorno de desarrollo de los archivos fuente que acompañan el trabajo es el sistema operativo GNU/Linux Debian. Nos hemos valido de componentes estándar como el célebre compilador gcc y g++ del proyecto GNU. No hay mayores exigencias que la utilización de la consola de comandos, ya sea para

compilar o para interaccionar con la aplicación. En el anexo final hay una pequeña guía sobre cómo utilizar el software.

1.1. C++ en la ingeniería de sistemas

Aun existiendo herramientas bastante potentes como Matlab o Simulink, son muchos los usuarios que implementan y analizan cualquier sistema utilizando un lenguaje de propósito general. Existen condiciones que justifican el uso de lenguajes de este tipo, ya que son altamente flexibles y pueden adaptarse a tareas extremadamente específicas que no pueden cubrir los programas profesionales.

El enfoque tradicional son los lenguajes estructurados, procedimentales, en los cuales se emplean estructuras y funciones, susceptibles de ser reutilizadas, mejoradas y ampliadas. En este contexto, es inevitable recordar las posibilidades que ofrece el lenguaje ANSI C (C estándar), cuando queremos implementar la simulación y el control de un sistema. Hagamos un repaso:

Requerimientos clásicos en la implementación de sistemas	Herramientas de C disponibles
Almacenado y lectura de datos en memoria	• Variables en array. • Punteros. • Asignación dinámica de memoria (**`malloc, realloc, free`**). • Acceso a cada elemento mediante índice (por ejemplo, en un bucle **`for`**). • Listas encadenadas.
Entrada/Salida	• Introducción por teclado, función **`scan().`** • Archivos de texto y binarios. Acceso secuencial y aleatorio. • Flujos de datos de entrada/salida mediante descriptores. • Presentación tabulada o gráfica de datos. • Cola de mensajes y tuberías en las comunicaciones entre procesos.
Cálculo, análisis y métodos matemáticos	• Toda clase de funciones matemáticas en **`math.h`**. • Métodos iterativos, álgebra matricial y vectorial para conformar sistemas continuos y discretos. Ecuaciones diferenciales y en diferencias. • Control del flujo de ejecución para satisfacer los algoritmos de cálculo.

Al ser C un lenguaje maduro, consolidado en décadas, ha tenido oportunidad de ampliarse y, por ejemplo, se han desarrollado herramientas y se han creado librerías que evitan mucho trabajo en la programación.

Una librería destacable es GLib, donde se pueden gestionar hilos de ejecución, perfilar definiciones numéricas, programar temporizadores, gestionar árboles, listas encadenadas, ordenar datos, y un largo etcétera.

Sin embargo, todas estas mejoras no bastan cuando aumenta la complejidad de los sistemas.

- Al ampliar funcionalidades o depurar, la gestión de las variables y procedimientos puede terminar siendo inabarcable. Se necesita otra forma de relacionar variables y procedimientos, para facilitar la tarea del programador. La programación orientada a objetos da una solución más intuitiva, más cercana a la forma habitual (cotidiana) de estructurar conjuntos, de encapsular variables y métodos, de tal forma que no interfieran unos con otros.
- En el mundo físico, las pequeñas piezas de un sistema, aunque sujetas a reglas, son autónomas, y esto para nada concuerda con la ejecución tradicional en computadores de un solo hilo (monotarea). Existirían serias complicaciones a la hora de depurar o ampliar las funcionalidades del software. Aún peor, desaprovecharíamos las características más notables de los microprocesadores modernos, que admiten el multiproceso y los múltiples hilos.

Algo que nos va a facilitar este primer contacto con C++ es considerarlo como un superconjunto de C, y no como un sustituto. Aunque se han añadido muchas nuevas características, éstas para nada pretenden forzar cambios drásticos ni romper con lo que ya sabíamos. Hay nuevos conceptos como las clases y los objetos; nuevas facilidades como las referencias y las plantillas que, naturalmente, incorporaremos. Tenemos tres conjuntos anidados:

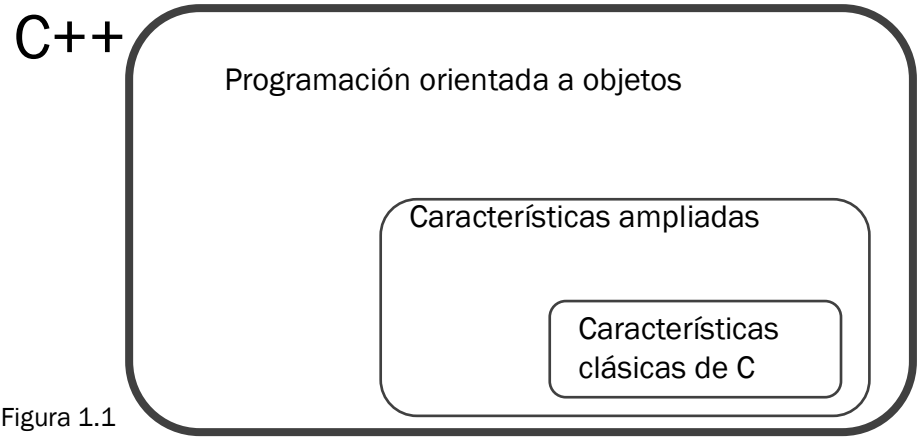


Figura 1.1

Veamos con profundidad en qué consisten estos tres conjuntos y así tendremos el esquema general del lenguaje. Se recomienda echar un primer vistazo a las cabeceras del código fuente como base.h o el programa bus.cpp, que nos muestra características clásicas del lenguaje en la cola de mensajería.

- **Características clásicas de C**. Las que se pueden usarse sin necesidad de utilizar un compilador C++, pero que se admiten en C++. Es todo lo que hemos aprendido antes de iniciarnos en este entorno. Las utilizaremos en este trabajo, por ejemplo, el uso de archivos con **`fopen(), fprintf()`**, etc. y las colas de mensajes System V que nos permitirán realizar la comunicación entre procesos. También se usa la práctica totalidad de operadores, símbolos de comparación, bucles **`for, while`** y tipos de datos.
- **Características ampliadas**. Muchos usuarios creen que C++ ha aportado, como novedad única, la programación orientada a objetos, pero no es exactamente así. Hay muchas nuevas características del lenguaje que se pueden utilizar sin necesidad de que pensemos en organizar nuestro software en clases y objetos.
  - **Plantillas**, que permiten una programación más flexible. Por ejemplo, se puede hacer una función que opere números enteros o de coma flotante, o con tipos de datos que definamos, sin necesidad de reprogramarla, gracias al uso de plantillas. Es una forma "estática" de polimorfismo que se realiza en tiempo de compilación. Cuando operemos con clases dinámicas, podemos escoger que éstas operen con números en coma flotante de precisión simple **`Dinamico<float>`** o con números, también en coma flotante, de precisión doble **`Dinamico<double>`**. Es más, si programamos nuestros propios tipos o recurrimos a álgebras más específicas como la del cuerpo de números complejos, podemos utilizar estas clases sin ninguna modificación. Tan solo hay que redefinir las operaciones de suma, resta, multiplicación y división al crear esos tipos de números.
  - **Standard Template Library (STL)** es una biblioteca en la que podemos gestionar todo tipo de datos como vectores **`std::vector<>`** que son de alojamiento dinámico, colas y listas encadenadas con facilidad. También usaremos la multitarea y el arbitraje de hilos concurrentes, como nuevo requerimiento en este entorno de programación. Los elementos que usan son objetos, sin embargo, los podemos utilizar en lenguaje procedimental puro.
  - **Sobrecarga de funciones**. La misma función puede servir para propósitos diferentes según los parámetros con los que se llama. Es otra manera básica de hacer polimorfismo. Se empleará, por ejemplo, al definir constructores de clase. Un ejemplo lo tenemos en la clase **`Dinamico`**. Cada método, aunque tenga el mismo nombre, es único para el compilador porque tiene otros argumentos.
  - **Referencias.** Si conocemos el uso de punteros, las referencias tienen un comportamiento similar y nos permite tener un código mucho más claro. Tan solo hemos de definirlas con el símbolo **`&`** a la derecha del tipo de datos, como por ejemplo **`int& ref;`** cuando definimos; además, pueden ser de solo lectura, y pasarse por argumentos como hemos hecho con los punteros.
  - **Nuevos tipos de datos y constantes.** Como, por ejemplo, los tipos de datos como **`byte`** y **`boole`** (número booleano), así como las nuevas constantes como los punteros nulos **`nullptr`** que sustituye a la antigua

macro **NULL**; y las constantes booleanas **true** y **false**, ampliamente utilizadas en este trabajo.

- **Reserva dinámica de memoria.** Gracias a los operadores **new** y **delete**, aunque también se dispone de las conocidas **malloc** y **free** con **std::malloc** y **std::free**.

La tercera característica merece una mención aparte, en el siguiente apartado, por ser la más interesante y en la que más profundizaremos. Nos ofrece toda la potencia del lenguaje C++.

1.1.1. Programación orientada a objetos.

Es la característica notable y la base con la que estructuraremos la aplicación. Esta forma de programar es intuitiva, en el sentido que nos permite hacer una estructuración cercana al mundo real que conocemos. Son tres las propiedades de este estilo de programación:

- **Encapsulamiento**. Limita la accesibilidad de datos y métodos en las clases, permitiendo un código más limpio y menos proclive a fallos. Si, por, ejemplo, tenemos una variable importante o método que sólo puede ser accedido o cambiada internamente por la clase, debería estar en la zona de la declaración conocida como **private** y si queremos que sea accesible sólo por las clases derivadas, la marcamos como **protected.** Las que son accesibles desde fuera se marcarán como **public**. Es muy fácil de entender si lo empleamos en el campo de los sistemas. Imaginemos que tenemos una clase base llamada Dispositivo con variables muy sensibles como la que permite la detención del hilo.

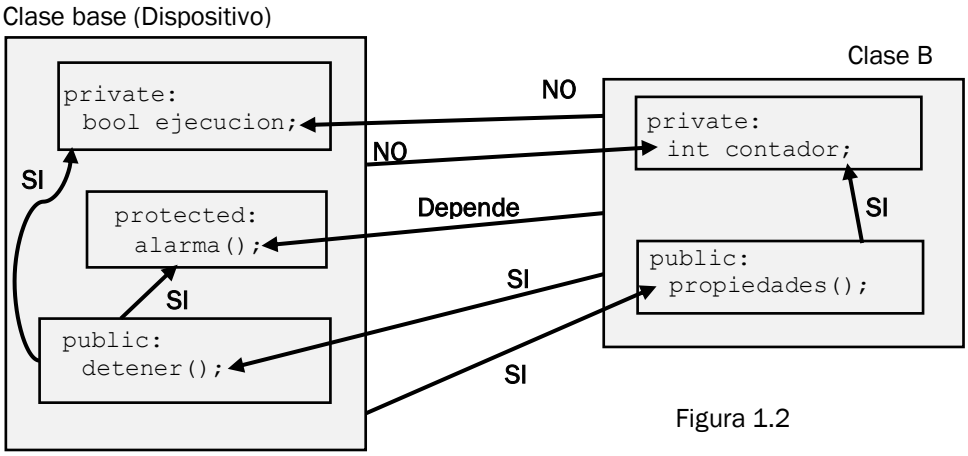


Figura 1.2

Si actuamos directamente sobre su variable, corremos el riesgo de equivocarnos (el hilo no era ejecutable), hacerlo en el momento menos apropiado (por estar desarrollando una tarea crítica), o por no modificar otras variables necesarias para realizar la ejecución con éxito. Es decir, el programa puede terminar fallando. La clase **Dispositivo** sabe cómo proceder, indicando a quien quiera llamar por qué no se puede modificar (por ejemplo, devolviendo

un valor de error). Por tanto, en vez de permitir modificar esa variable directamente, la etiquetamos en la zona privada y, si de alguna forma queremos detener el hilo desde el exterior, llamamos al método correspondiente que es **`acaba()`**. Así se evitan fallos, descuidos y problemas de seguridad.

El encapsulamiento es un mecanismo muy sencillo y potente de seguridad. Otra situación que se puede dar es que los hilos derivados (que heredan características, como veremos en el siguiente punto) puedan hacer uso de variables y métodos de las clases de los que deriva, pero que sigan siendo privados para clases externas. Entonces utilizaremos **`protected`**.

- Herencia. La organización de los datos es fundamental en el mundo informático. Al realizar programas de cierta envergadura reaprovechamos características comunes mediante funciones y subrutinas. Son los cimientos de la programación procedimental. Con la herencia damos un paso más allá y englobamos no solo métodos en las clases, sino también las variables asociadas a esos métodos. Se ha habilitado un mecanismo en el cual creamos nuevas clases a partir de las anteriores, de una forma sencilla.

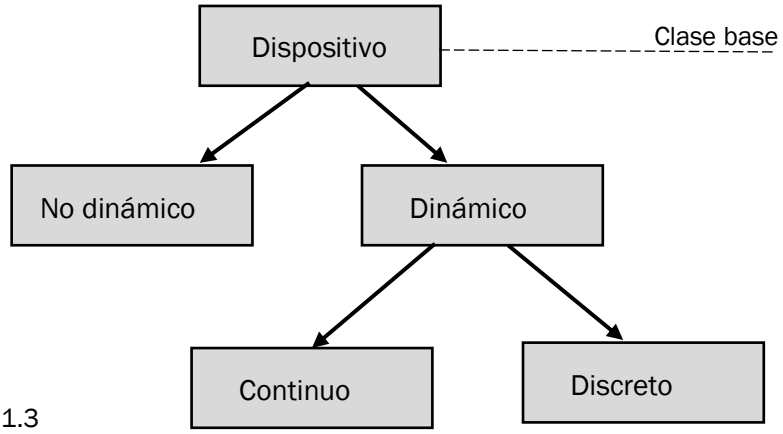


Figura 1.3

Cuando se ha diseñado el programa, se ha tenido en cuenta qué características comunes tienen todos los objetos; ¿qué es lo que queremos conseguir? Pues objetos que puedan ejecutar sus propios hilos, ser iniciados y detenidos, que muestren el tiempo que llevan funcionando, si tienen algún problema interno (alarma). Con ello se crea la clase base llamada **`Dispositivo`**.

A partir de ahí, reproducimos las características comunes de la clase base para aplicarlas a clases específicas que queramos construir. Todos ellos han heredado propiedades fundamentales como poder ejecutar hilos (tanto en métodos como en variables); y se pueden modificar otras propiedades, a conveniencia. Sabemos que puede haber dispositivos dinámicos o no. Dentro de los dispositivos dinámicos tendremos los continuos (para la planta) y los

discretos que además de métodos y variables de la clase base **Dispositivo** también heredan métodos y variables de la clase **Dinamico**.

La herencia, sin duda tiene un impacto enorme en un programa escrito en C++. Si un programador está acostumbrado a un entorno procedimental ahora tiene que pensar que la estructura jerárquica no sólo afecta a procedimientos sino a variables. Es decir, afecta a cada uno de los objetos que se puedan crear, tanto en código como en datos.

Naturalmente, existen en C++ muchos mecanismos para heredar, e incluso la existencia de clases amigas, y herencia múltiple desde varias clases base.

- **Polimorfismo.** Hay varias formas de polimorfismo. Según hemos adelantado, las plantillas se pueden considerar polimorfismo, pues programamos clases que son versátiles, se pueden usar con tipos de datos diferentes. También lo es la sobrecarga de funciones, admitida en C++, en la que se puede crear funciones con los mismos nombres con la condición de que tengan argumentos o variables de retorno diferentes. Lo veremos en algunos constructores. Sin embargo, la característica más potente de polimorfismo queda reflejado de forma dinámica gracias al modificador **virtual**.

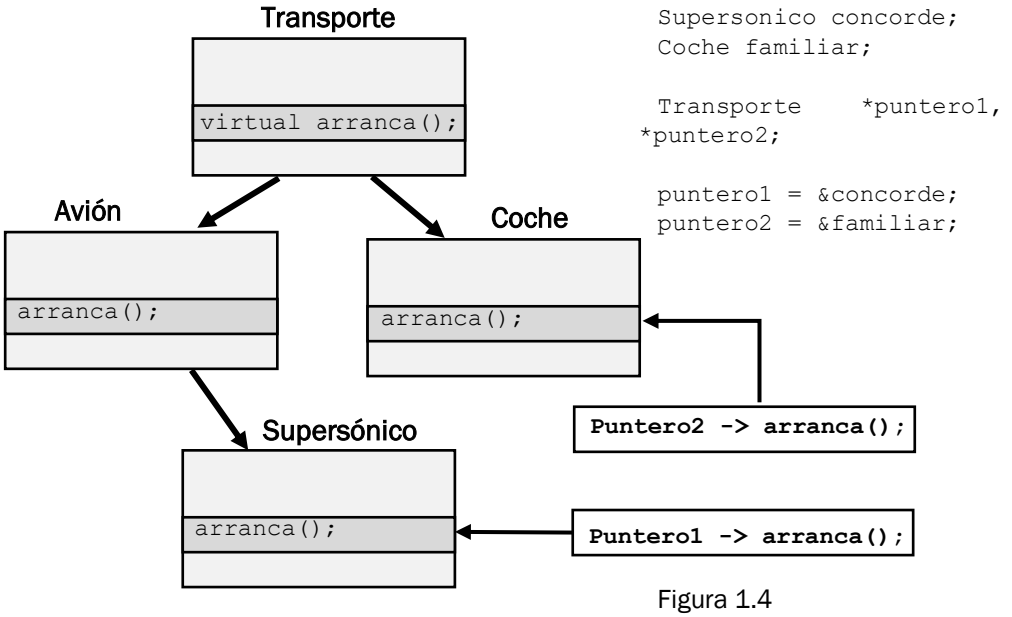


Figura 1.4

Básicamente, si disponemos de un puntero a la clase base y un conjunto de objetos derivados con características diferentes, podemos referirnos a una misma operación, con la garantía de que utilizaremos el método adecuado.

No es lo mismo arrancar un coche familiar que un avión supersónico. Necesitan métodos muy diferentes, aunque ambos sean transportes. En nuestro trabajo, utilizaremos una forma similar de enfocar el polimorfismo. Hay dispositivos con rutinas de ejecución y propiedades diferentes.

1.2. Los contenedores STL. Vectores y colas.

En nuestro primer contacto con C++ es imposible prescindir de la Standard Template Library (STL). Nos ofrece una excelente biblioteca de clases para manipular toda clase de estructuras habituales en informática. No se pretende profundizar en exceso, pero sí debemos conocer el uso de los contenedores que nos permitirá implementar el sistema.

Son estructuras muy frecuentes en programación. Se definen como una colección de elementos que se pueden acceder mediante indexación. En C tenemos los vectores estáticos, cuyo espacio de memoria se reserva de antemano, y también los vectores dinámicos que pueden hacer uso de espacios de memoria asignados mediante **`std::malloc`**. Sin embargo, se hace desaconsejable utilizar memoria dinámica "a la vieja usanza".

- Donde hay un **`malloc`** tiene que haber un **`free`**. Si no liberamos memoria, ésta queda retenida y sin uso. Tenemos una fuga de memoria.
- En ocasiones, por accidente, se escribe y se lee en zonas que no fueron reservadas provocando excepciones como la célebre *segmentation fault*.
- El manejo de punteros se hace engorroso, principalmente si utilizamos estructuras de datos de cierta complejidad (matrices, árboles, etc.).
- Es probable que tengamos que depurar, con el consiguiente gasto de tiempo.

Todos estos problemas son resueltos con unos objetos llamados contenedores, que se adaptan a las necesidades de uso. Reservan la memoria si son ampliados, y la dejan de nuevo disponible si son reducidos. Cuando terminan fuera de ámbito o el objeto que los alberga es destruido, se liberan automáticamente de la memoria.

Los vectores o arrays son los elementos más importantes. Este trabajo los utiliza constantemente, por ejemplo, en la elaboración de una lista de dispositivos para el menú, o de estructuras de parámetros para el bus. El vector más básico es el declarado estáticamente.

```
int lista[4] = {18,16,7,-2};
```

Sin embargo, no podemos ampliar ese vector fácilmente. Ahora pensemos en la posibilidad de que tengamos que aumentar una lista de puertos para conectar en un dispositivo, como haremos más adelante. Tendríamos que recurrir a una combinación de **`malloc`** y **`realloc`**. Y si queremos insertar un número en una posición intermedia, tendríamos que desplazar el resto de elementos, utilizando bucles. Se hace muy engorroso.

La solución que nos propone la librería STL son los contenedores. Objetos que albergan datos y métodos para manipular estos datos. Si queremos declarar el vector de antes al estilo STL, y para la versión que usamos (C++11) escribiríamos:

```
std::vector<int> lista = {18,16,7,-2};
```

Por supuesto, los contenedores son plantillas y podemos definir cualquier tipo numérico, objeto o estructura que queramos.

Las colas son entidades en las que se van encadenando elementos y donde es habitual introducir un nuevo elemento por un extremo y extraerlo por el otro

En STL hay dos tipos de cola: la cola **`std::queue`** simple, que no permite acceso individual e indexado de sus elementos, y la cola **`std::deque`**, que, a semejanza de una lista doblemente encadenada permite iteradores y por tanto un recorrido mediante índice, como se hace en los vectores o arrays.

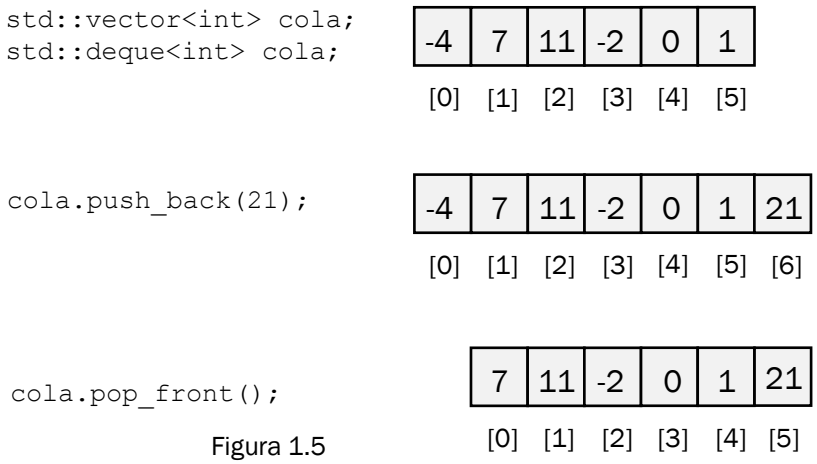


Figura 1.5

Se hace conveniente en este trabajo que utilicemos una cola **`std::deque`**, puesto que, como iremos viendo, hemos de acceder a los diferentes estados de un dispositivo dinámico, y estos están implementados en colas.

También es interesante darse cuenta que aparentemente podríamos emplear colas doblemente enlazadas **`std::deque`** y vectores **`std::vector`** como si fuesen lo mismo, sin embargo conviene recordar que estos contenedores están optimizados para su uso propio. Las colas reservan memoria por bloques, de varios elementos, mientras que los vectores lo hacen por elemento. Si estamos frecuentemente introduciendo y sacando elementos, una cola **`std::deque`** es más rápida que un vector **`std::vector`**.

La nomenclatura de indexación que utilizaremos para vectores, colas dobles, etc. es la tradicional en C, aunque se sabe que es posible que salte una excepción si el índice está fuera de rango y accedemos a memoria no permitida. Otra forma es utilizar el método del contenedor **`at()`**. Ambas formas son igual de válidas, aunque lanzan diferentes excepciones cuando se lee/escribe fuera de rango.

```
cola[1]  ≈  cola.at(1)
```

Es interesante inspeccionar todos los tipos de contenedores existentes, y que permiten diseñar listas **`std::list`**, conjuntos **`std::set`**, o pilas (LIFO) **`std::stack`**.

1.3. Multitarea. Hilos de ejecución.

La programación basada en hilos puede llegar a ser muy potente, porque estamos creando pequeñas unidades autónomas, con su propio flujo de programa, que en su conjunto resuelven un problema.

Sin embargo, esta forma de programar necesita precauciones especiales, puesto que estamos accediendo espacios de memoria y recursos compartidos.

Al crear unidades autónomas de ejecución, las ponemos a disposición para

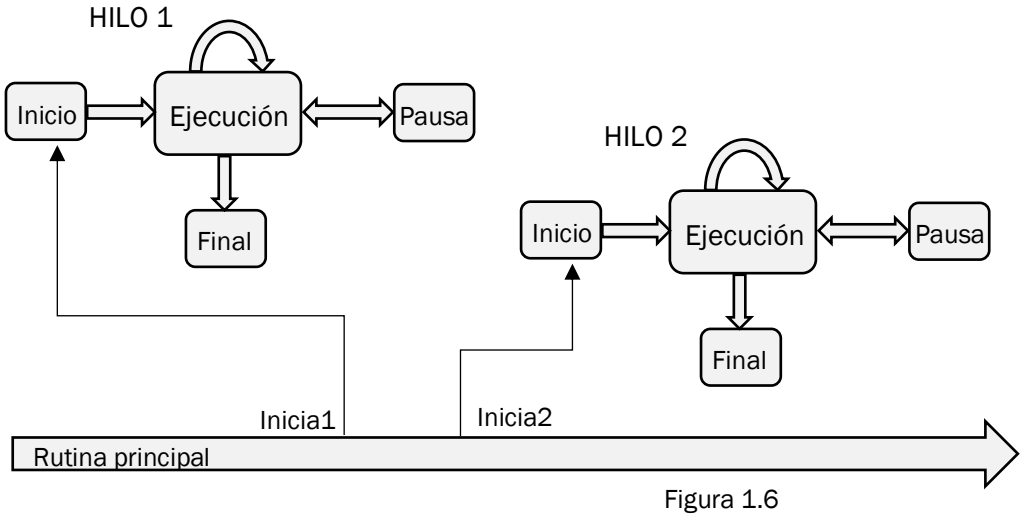


Figura 1.6

leer o escribir cuando éstas están disponibles y van cumpliendo su rutina, pero, mientras con un hilo de ejecución sabemos que no se puede leer y escribir a la vez una zona de memoria, con hilos múltiples. Se producen situaciones indeseables. Los hilos tienen varios estados:

- Estado de inicio. Se prepara el hilo, se carga su rutina en memoria y se espera a una orden para que empiece la ejecución. Esta ejecución empieza en la rutina principal (o en otro hilo) y se ha representado en la figura como init1 e init2.
- Estado de ejecución. Es el habitual y donde se ejecutan las instrucciones de la rutina asociada al hilo.
- Estado de pausa. Puede ser por dos motivos: pausa planificada, como por ejemplo un método de la clase **`std::thread`** llamado **`sleep()`** que hace "dormir el hilo un tiempo determinado; o una pausa forzada al encontrarse con un método de **`std::mutex`** (exclusión mutua) llamado **`lock()`**, el cual indica que otro hilo está ocupando ese recurso, o porción del programa, y ha de esperar. En ambos casos, se retoma la ejecución, cuando el otro hilo finaliza y llama al método **`unlock()`**.

- **Estado final**. Se finaliza el hilo y, en última instancia, se liberan sus recursos. La rutina ya ha cumplido su cometido, o el programa principal ha decidido que acabe.

Se pueden crear infinidad de hilos que realicen sus rutinas mientras el hilo principal continua con su ejecución, creando o cerrando hilos. A diferencia de la comunicación entre procesos IPC que se realiza con la instrucción **`fork()`**, los hilos son unidades mucho más integradas y pueden convivir en el mismo programa, acceder a espacios de memoria comunes, métodos, subrutinas, etc.

Sin embargo, como se ha apuntado, hay que conocer bien esas zonas comunes de memoria y de acceso a dispositivos críticos, pues de lo contrario, ocurren problemas.

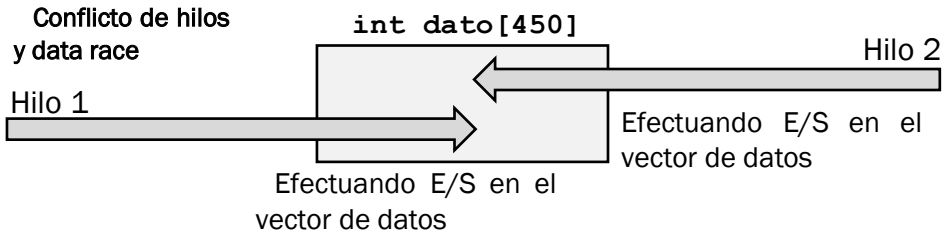


Figura 1.7

Otras situaciones para evitar son los interbloqueos o *deadlocks*, que ocurren cuando un hilo acapara un recurso que otro hilo necesita, dejándole bloqueado en espera; y a la vez, ese hilo que acapara el recurso necesita que el otro hilo finalice una tarea antes de liberarlo.

Se forma un lazo que finalmente bloquea todo el programa. En la práctica intentaremos que nuestros dispositivos sean poco interdependientes. Aun así, conviene tener presente estas posibles complicaciones.

1.4. Antes de empezar

El lenguaje C++ tiene innumerables características atractivas para los sistemas, el control y la automática. Aunque, a partir de C++11 se pueden definir bucles **`for`** diferentes, determinados por rango, por ejemplo, en la línea **`for (int i : v)`**, intentaremos que el código sea legible para quien está aprendiendo C++ desde C; por tanto, en bucles y control, utilizaremos las instrucciones habituales como **`while`**, **`do while`**, **`for`**, **`if else`**, **`switch`**, etc.

El uso de referencias tiende a sustituir el uso de punteros ya que éstos, al acceder directamente a memoria son proclives a errores que pueden desencadenar comportamientos imprevisibles del programa, excepciones y fallos de seguridad. Aunque se siguen utilizando punteros a la hora de confeccionar listas (ya que los punteros han de estar previamente declarados), en ocasiones es ventajoso utilizar referencias y su notación es muy transparente.

Se recomienda adquirir unos mínimos conocimientos sobre programación orientada a objetos y una base en C++. En la Bibliografía hay referencias interesantes para comprender el lenguaje C++, en especial el libro de Anthony Williams: *C++, Concurrency In action*, que detalla el funcionamiento de la multitarea. Muchos conceptos y clases de la librería estándar irán surgiendo, a medida que añadamos características y tengamos que resolver los problemas que se nos vayan planteando. El objetivo, indudablemente, es aplicar C++ en el campo de la Ingeniería de Sistemas.

En la lista se enumeran algunos fundamentos de programación, para los capítulos siguientes:

- Declaración de una clase.
- Encapsulamiento y herencia.
- Sobrecarga de funciones, plantillas y polimorfismo.
- Código fuente en varios archivos.
- Instancia de una clase. Objeto.
- Constructores y destructores. Constructor copia
- Procesos e hilos. Formas de comunicación entre hilos y procesos.
- Salida por pantalla y por archivo
- Contenedores de datos: vectores, colas y cadenas de caracteres.
- Cola de mensajes estándar System V.
- Entrada/Salida en Raspberry.
- Programación en Arduino.

Se adjuntan tres anexos:

- Anexo I. Ayudará a comprender la jerarquía de clases y la relación entre éstas.
- Anexo II. Enumeración de archivos y directorios. Ayudas para la compilación y la depuración.
- Anexo III. Funcionamiento de los procesos de planta y control programados, mediante ensayos y simulaciones, con modelos habituales en la ingeniería, que ayudarán a consolidar el objetivo de este trabajo.

También es aconsejable tener disponible el código que acompaña al trabajo, para realizar consultas, revisar la implementación o comprobar su funcionamiento.

2. Implementación de los modelos físicos de planta

Una vez que se han descrito todas las características que ha de cumplir nuestro sistema de control, y tras hacer un recorrido por todas las herramientas de programación en C++ que emplearemos, nos planteamos la duda de cómo empezar la implementación.

No es una tarea fácil, puesto que hemos de combinar el uso de hilos concurrentes con una adecuada abstracción (o generalización de conceptos) que permita hacer reutilizable el código. El lenguaje nos brinda muchas facilidades que podemos compatibilizar con nuestra visión tradicional de los lenguajes de programación.

Por el momento, no veremos ningún aspecto de la programación concurrente, con varios hilos ejecutándose en paralelo, sino que intentaremos satisfacer las características estructurales, en la implementación de sistemas, creando las primeras entidades y sistemas dinámicos, que luego podremos interconectar y finalmente sincronizar. Para ello haremos uso de tres clases consideradas esenciales: **`Dispositivo`**, **`Dinamico`** y **`Continuo`**. Si nos enfocamos en la programación orientada a objetos, la jerarquía es sencilla:

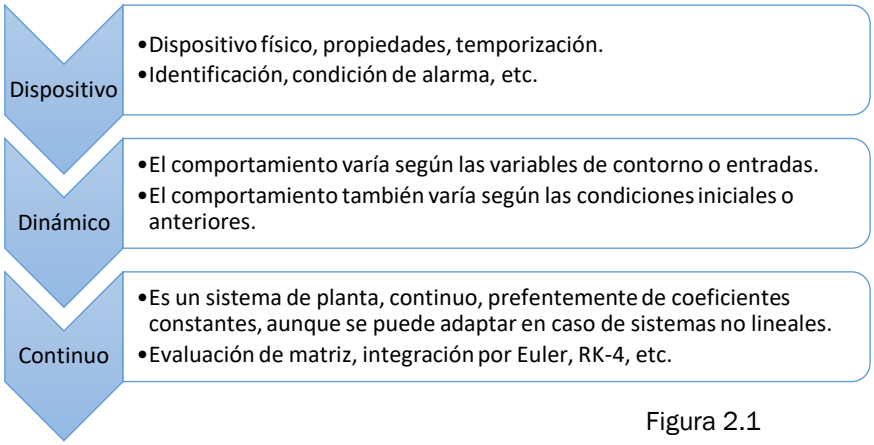


Figura 2.1

¿Por qué elegimos este orden? El motivo es muy intuitivo. Aquí es donde empezamos a usar la programación orientada a objetos.

2.1. De lo general a lo particular

Al diseñar algoritmos, intentamos unir piezas y no desaprovechar trabajo. El surgimiento de las bibliotecas de funciones, cuando empleamos programación procedimental, no es casual y responde a la necesidad de agilizar la escritura de programas. Un ejemplo sencillo es la construcción de las operaciones aritméticas y funciones matemáticas básicas:

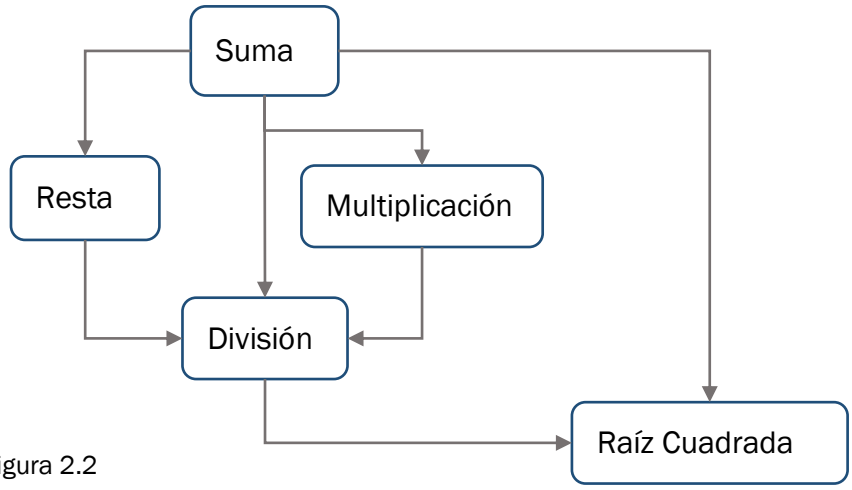


Figura 2.2

- La suma es la operación aritmética por excelencia y se utiliza en todas las demás.
- La resta es la suma de un número y su negativo (o complementario a dos, si operamos con números binarios).
- La multiplicación se puede definir como una secuencia de sumas.
- La división, tal como se hace tradicionalmente, es una combinación de sumas, restas y multiplicaciones.
- La raíz cuadrada se puede calcular de varias formas. Un método bastante poderoso es el de Newton-Raphson, que a partir de N genera una sucesión numérica recurrente $\{n_i\}_{i\in\mathbb{N}}$ que converge con rapidez hacia $\sqrt{N}$:

$$n_{i+1} = \frac{1}{2}\cdot\left(\frac{N}{n_i} + n_i\right) \tag{2.1}$$

Es decir, para calcular la raíz cuadrada tan solo necesitamos saber sumar y dividir, hasta alcanzar la convergencia del resultado. Como primer valor tenemos que tomar un número lo más aproximado a la raíz. Por ejemplo, si N no es muy grande, una buena aproximación es $n_1 = {}^{N}/_{2}$

Haciendo uso de las series de Taylor, podemos implementar toda clase de funciones matemáticas, utilizando sumas, restas, multiplicaciones y divisiones.

Un camino similar vamos a tomar a la hora de escribir las clases en C++. Cuando queremos programar basándonos en objetos, es aconsejable empezar por la generalización (abstracción) y terminar por la concreción, especializando las clases; o también se puede seguir la corriente contraria, e ir de lo particular a lo general. Tomemos el primer camino y empecemos concibiendo las variables y los métodos que compartirán **todos** los dispositivos que conformarán nuestro sistema de control y simulación.

2.2. El nivel más alto de generalización. La clase base "Dispositivo"

¿Qué es lo que necesitamos de un dispositivo en nuestro sistema? En primer lugar, sabemos que pueden tener rutinas de ejecución, y que se irán actualizando en cada iteración, pues al fin y al cabo estamos usando un computador, aplicando métodos numéricos para hacer los cálculos. Esos métodos numéricos serán de varios tipos, según el contexto. No es lo mismo simular un dispositivo de planta continuo que un controlador discreto.

Los dispositivos además tienen unas características muy diferentes que conviene reflejar; el tipo de dispositivo (si es planta, controlador, si es de discretización o bus; o si es un conector…). También es interesante saber cuándo el dispositivo requiere especial atención, como puede ser una condición de alarma. Por tanto, iniciamos la tarea de escritura del código, con este esquema, donde se reflejan las características más importantes:

Clase Dispositivo	

Variables	• Número de identificación de dispositivo.
	`int idslot;`
	• Frecuencia de muestreo (cero si es un dispositivo pasivo).
	`unsigned int muestreo;`
	• Bit de alarma que indica que existe un problema en el dispositivo.
	`bool alarma = false;`
	• Instancia de la clase thread (hilo)
	`std::thread hilo;`
	• Instancia de la clase mutex (exclusión)
	`std::mutex exclusion;`

Métodos	• Constructor. Se crea el dispositivo con una identificación numérica id y con un tiempo de t que marca la frecuencia de actualización en Hz o cero si es pasivo.
	`Dispositivo(unsigned int t, int id);`
	• Actualiza. Para dispositivos iterados (procesos en planta o controladores, por ejemplo). Cada subclase se actualiza en cada ciclo, como por ejemplo variables de estado, alarmas, etc.
	`virtual int actualiza();`
	• Rutina. Método virtual que alberga la rutina de ejecución del dispositivo.
	`virtual int rutina();`
	• Inicia. Iniciador de la rutina del dispositivo.
	`int inicia(int tiempo);`
	• Propiedades. Método virtual (polimórfico) que nos muestra en pantalla las propiedades del objeto instanciado por la clase.
	`virtual int propiedades();`

La clase base **Dispositivo** proporciona toda clase de servicios a las clases derivadas o externas como información sobre su estado (si es ejecutable, si está pausado, si está en alarma, sus parámetros de funcionamiento, su tiempo de ejecución, período, etc.) que se usan de forma frecuente.

Al tratarse de una clase muy extensa, en la tabla se han omitido características. Para facilitar la comprensión, se irán introduciendo más, a medida que se desarrolle la aplicación y definamos nuevos conceptos.

2.2.1. Hilo de ejecución. Temporización. La función lambda.

En la tabla anterior hay variables relacionadas con la ejecución y la exclusión mutua. La exclusión mutua es un mecanismo básico de sincronización de hilos que concurren en rutinas en las que puede haber conflicto entre ellos. Tendrá un tratamiento extenso en el capítulo 3.

Para que nuestros sistemas funcionen, nos centramos en la ejecución y la temporización de la clase base **Dispositivo**. Ya habíamos definido lo que es un hilo. Los hilos son las unidades de ejecución de nuestros dispositivos, y recurren a la clase **std::thread**. Esta clase dos métodos a los que les daremos uso: su constructor y el método **join()**. Pensemos en el arranque, la pausa y la finalización del hilo para hacernos una idea.

El arranque se usa en el mismo constructor y hemos de proporcionar la rutina en la que arrancará el hilo. Cada rutina es diferente, dependiendo de la clase, de ahí su declaración **virtual**. Para llevar a cabo este inicio por constructor, sin importar qué clase hace uso del constructor, hacemos uso de una característica interesante en C++11 como son las funciones lambda.

```
std::thread RutinaThread() {
            return std::thread([=] { rutina(); });}
```

De forma resumida, las funciones lambda son una forma *inline* (código insertado directamente en la declaración) de apuntar a una función para que, de esta forma, podamos utilizarla en llamadas a otras funciones o constructores.

Para iniciar el hilo tan solo hemos de utilizar nuestra variable declarada como **std::thread hilo,** con esta llamada:

```
hilo = this->RutinaThread();
```

Recordemos que **this** es el puntero al objeto. Desde ese mismo momento el hilo empieza a ejecutarse. Hemos “encendido” el dispositivo.

Queda claro que también necesitamos una temporización adecuada de las ejecuciones. Entre cada iteración hay un tiempo de espera que permite que transcurran los hilos en tiempo real, dependiendo de su frecuencia de muestreo. Si no lo hiciéramos así, no habría ninguna sincronización entre dispositivos, y los datos transmitidos/recibidos por cada uno serían caóticos.

Entre cada iteración hacemos uso de:

```
std::this_thread::sleep_for(std::chrono::milliseconds((int)(Dispos
itivo::period*1000)));
```

Que a su vez realiza una llamada a **`std::chrono`**. El hilo se queda detenido ("durmiendo") un tiempo prefijado en milisegundos. Después despierta y ejecuta otro ciclo del dispositivo; se vuelve a dormir, y así sucesivamente.

Hay que tener en cuenta que ésta es una pausa planificada. Si queremos pausar el hilo externamente un tiempo indefinido, tendremos que utilizar nuestros propios medios, mediante bloqueo por **`std::mutex`** o mediante semáforo, pues no existe un método específico que lo haga.

Tras cumplir su función, el hilo vuelve de nuevo a unirse al hilo principal del programa, de lo contrario quedaría indefinidamente en espera hasta ser cerrado. Esto hace con el método **`join()`** en la rutina principal. Así finaliza el ciclo del hilo que conforma nuestro dispositivo.

2.3. Dispositivos no dinámicos. Generadores de funciones

Los dispositivos más básicos se pueden representar con funciones analíticas, algebraicas (continuas o definidas a trozos) que pueden servir de variables de contorno para los sistemas dinámicos que veremos posteriormente. En este esquema tenemos los tres dispositivos: dos generadores (uno cíclico, otro constante); y un dispositivo de cálculo, conectados de tal forma que se hace una operación de suma.

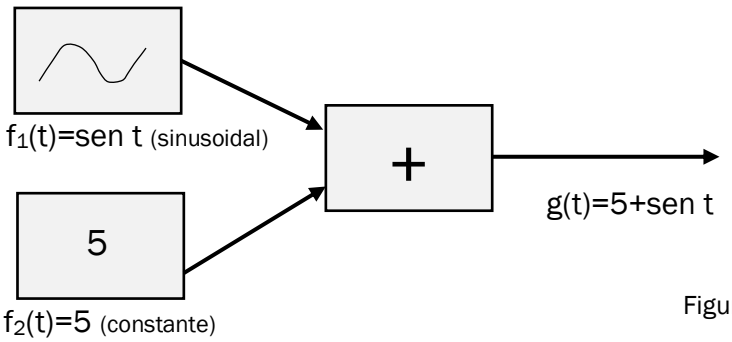


Figura 2.3

Por tanto, podemos disponer de una combinación de fuentes generadoras.

- El dispositivo de señal cíclica implementado en funciones_periodicas.cpp que nos proporciona señales triangulares y cuadradas, de amplitud, frecuencia y asimetría deseadas.
- El dispositivo de número constante, al que le podemos hacer un uso como consigna en controladores o como una función escalón, modificando su valor durante la ejecución del sistema en su menú de propiedades. Lo tenemos implementado en constante.cpp.

- El dispositivo que realiza la suma hace operaciones más generales y está implementado en combinacion_lineal.cpp. Suministramos dos entradas a y b y como parámetros internos del dispositivo tenemos λ y μ entonces la salida c será una combinación lineal de las entradas tal que:

$$c = \lambda \cdot a + \mu \cdot b \tag{2.2}$$

Esto es suficiente para hacer sumas, restas, realizar conmutación ($\lambda = 0$ ó $\mu = 0$), y para multiplicar valores por un número.

Clase Periodica	
Métodos	• Constructor (Onda triangular, cuadrada y cuadrada inversa) `Periodica(unsigned int t, int id, int modo, unsigned int cuenta_asc, unsigned int cuenta_desc, float superior, float inferior);`
Clase Constante	
Métodos	• Constructor `Constante(unsigned int t, int id, float valor, float superior, float inferior);`
Clase Combinacion	
Métodos	• Constructor `Combinacion(unsigned int t, int id, float a, float b);`

*La clase Periodica realiza conteos cada intervalo de muestreo así que la frecuencia de la señal depende de la frecuencia de muestreo y de las cuentas ascendentes y descendentes. La clase Constante, además de su valor, también tiene dos límites superior e inferior que no se pueden sobrepasar.

2.4. Dispositivos dinámicos. Primer uso de los contenedores STL.

Ya hemos definido lo que es un dispositivo, así que empezamos a concretar. Hay muchos tipos de dispositivos, pero nos vamos a centrar en dos categorías importantes: los dispositivos estáticos y los dispositivos dinámicos.

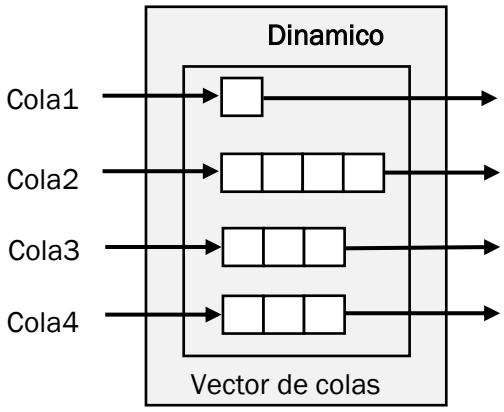


Figura 2.4

Por lo general, vamos a utilizar sistemas dinámicos cuya respuesta depende, entre otras cosas, del estado previo del dispositivo o de sus condiciones iniciales. Son los modelos que mejor corresponden con los fenómenos naturales y con el comportamiento físico de los dispositivos industriales.

Matemáticamente se representan por ecuaciones diferenciales (variable continua) o ecuaciones en diferencias (variable discreta).

La figura representa un dispositivo dinámico que hemos definido arbitrariamente. Observamos que hay cuatro colas y cada cola tiene un número de elementos. Si queremos construir un objeto de esta clase lo podemos hacer pasando como argumento un vector de n elementos en el que cada elemento representa la longitud de la cola. En este caso el vector se definiría así:

```
std::vector<int> v = {1,4,3,3};
```

Y ya podríamos pasar este vector como argumento para el constructor del objeto de la clase **`Dinamico`**.

En cada ciclo, esta clase tiene como función escribir al final de las colas los nuevos valores que han calculado las clases derivadas y que se incorporarán en su estado. Entonces, una a una, cada cola elimina su primer valor, haciendo que, si un valor estaba en la posición n de la cola, pase a estar en la posición $n-1$. Es decir, estamos realizando constantes desplazamientos de adelante hacia atrás.

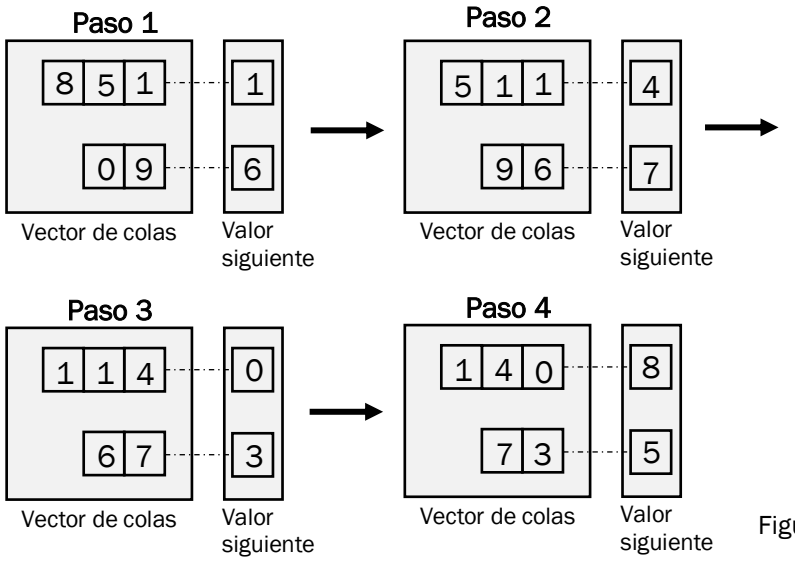


Figura 2.5

Estas sencillas sustituciones y desplazamientos son suficientes para satisfacer las necesidades de los dos principales sistemas dinámicos que implantaremos:

- **En los sistemas continuos**, guardando el estado actual y anteriores, según los pasos de integración que estemos efectuando.
- **En los sistemas discretos**, las colas y desplazamientos se asemejan a retardadores que en el dominio Z se representan por z^{-1} y que son los elementos con los que construir toda clase de controladores, filtros, etc.

Como se puede observar en el código fuente, en C++ hemos de utilizar un vector de colas de estado; es decir, un anidamiento de dos contenedores STL, con la siguiente declaración:

```
std::vector< std::deque<T> > estado;
```

Cada elemento **estado[i][j]** puede ser accedido mediante dos índices, como ocurriría con cualquier matriz, con precaución de no leer fuera de límites.

Cuando queramos sustituir y realizar el desplazamiento, hacemos uso de los métodos implementados en el objeto **std::deque** (cola) como es **push_back()** (añadir último elemento) y **pop_front()** (quitar primer elemento), de esta forma conformamos el sistema dinámico.

Ésta es la primera clase que hace uso de las plantillas y podremos utilizarla sin modificaciones con cualquier tipo de número. En el presente trabajo se usa el tipo **float**, pero podemos implementar sistemas dinámicos con números **double**, complejos, vectores, etc.

Las ideas principales que hemos ido recogiendo sirven para implementar la clase, que hereda las características de los dispositivos.

Clase Dinamico	

Variables	• Estado del sistema (vector de colas)
	`std::vector< std::deque<T> > estado`
	• Siguiente estado (vector)
	`std::vector<T> siguiente;`

Métodos	• Constructor (principal)
	Se pasa como argumento, además de los parámetros de dispositivo, un vector que determinará las longitudes de cola.
	`Dinamico(unsigned int t, int id, const std::vector<unsigned int>& v);`
	• Actualiza. Actualización de las colas de estado con los valores del vector siguiente.
	`int actualiza();`

Observación: La clase puede manejar cualquier tipo de datos por usar plantilla.

Como vemos, se trata de una clase muy sencilla, pero con características fundamentales.

2.4.1. Generador de números pseudoaleatorios

En esencia, este generador es un sistema dinámico discreto; sin embargo, podemos utilizarlo para generar perturbaciones de índole aleatoria en circuitos eléctricos, mecánicos, etc. Aunque la librería STL nos proporciona una colección de funciones para la generación de números pseudoaleatorios y herramientas estadísticas, se ha preferido, por motivos didácticos, crear una rutina propia en *aleatorio.cpp.*

Utilizamos el método congruencial aditivo de cuatro pasos. Por tanto haremos uso de los métodos implementados en la clase **`Dinamico`**.

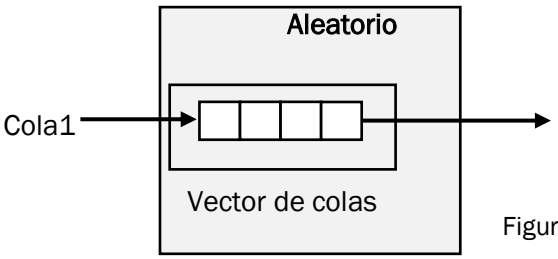


Figura 2.6

Los requerimientos de memoria para este método son básicos: una cola de cuatro elementos, por lo que, al llamar al constructor (o configurador) de la clase Dinamico, le pasamos este vector:

```
std::vector<unsigned int> inicial(1,4);
```

Se parte de cuatro semillas enteras y positivas x_1, x_2, x_3, x_4 que escogeremos libremente, al llamar al constructor de la clase, y un módulo $m > 0$ también entero. La iteración i con $i \geq 4$ se calcula (aritmética de números naturales):

$$x_{i+1} = x_i + x_{i-3}\ mod\ m \qquad (2.3)$$

y el resultado de cada iteración, que sigue una distribución pseudoaleatoria uniforme $U(0,1)$ de números racionales es:

$$r_i = \frac{x_i}{m-1} \qquad (2.4)$$

El método de clase permite que, en cada actualización por temporización, se generen números en un intervalo $[sup, inf]$ prefijado, con lo cual, una vez que tenemos un número $r_i \approx U(0,1)$ multiplicamos y sumamos para que se ajuste a ese intervalo.

Clase Aleatorio	
Método	• Constructor **`Aleatorio(unsigned int t, int id, unsigned int k1, unsigned int k2, unsigned int k3, unsigned int k4, unsigned int modulo, float sup, float inf)`**

La experiencia indica que hay combinaciones de semillas que producen secuencias periódicas, por lo que es conveniente ensayar el sistema, e incluso hacer un análisis estadístico de la secuencia que vamos obteniendo para comprobar que estamos generando una serie pseudoaleatoria que se aproxima de forma satisfactoria a una distribución uniforme.

2.5. Sistemas continuos de planta

En la clase Continuo englobamos los métodos más empleados para simular dispositivos físicos de planta. Procuraremos, siempre que nos sea posible,

trabajar con modelos lineales de coeficientes constantes, por lo que se han escrito los métodos para estos modelos, como es la evaluación de un sistema matricial, método clásico de integración de Euler, y el método de Runge-Kutta de orden cuatro.

2.5.1. Métodos matemáticos e implementación

Tengamos el paso h, la matriz del sistema dinámico A, el vector de variables de términos independientes $\vec{b}(t)$ que puede incluir funciones de contorno; el vector de variables de estado $\vec{y}(t)$; e $\overrightarrow{y'}(t)$ su vector derivada. La ecuación diferencial del modelo físico cumple que:

$$\vec{y}'(t) = A \cdot \vec{y}(t) + \vec{b}(t) \tag{2.5}$$

Si establecemos un paso h para hacer resolución numérica, la integración iterada más sencilla de esta ecuación diferencial es: $\vec{y}_{n+1} = h\left[A \cdot \vec{y}_n + \vec{b}_n\right]$ (2.6)

Este es el método de Euler, implementado en la función **`euler()`**. Por cada iteración realiza una sola evaluación de la matriz del sistema y una sola suma de los términos independientes.

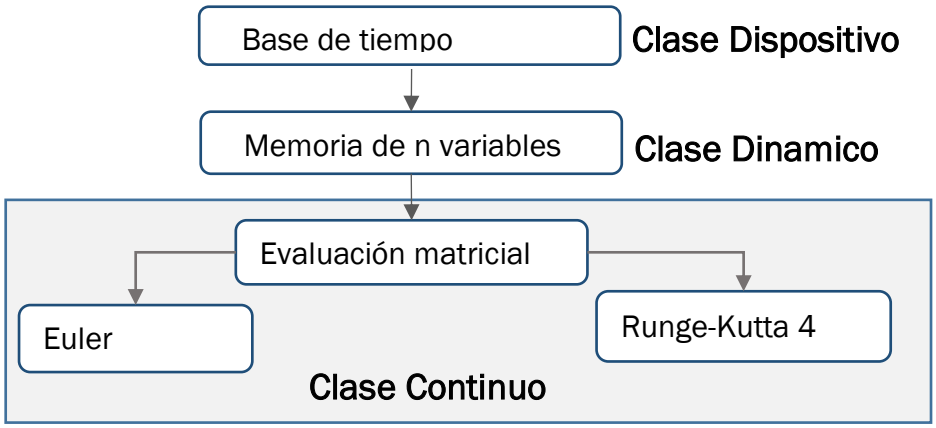


Figura 2.7

El método de Runge Kutta es más elaborado, y su propósito es conseguir una mayor precisión que el método de Euler con el mismo paso h, puesto que se realizan cuatro evaluaciones de la ecuación matricial por cada iteración. Además de precisión, el sistema gana en estabilidad. Utilizaremos el algoritmo estándar RK-4 (Runge-Kutta orden cuatro), cuyo uso es muy frecuente en el software de simulación. Su cálculo con una ecuación general $y' = f(y,t)$ es:

$$\begin{aligned} k_1 &= f(y_n, t_n) \\ k_2 &= f\left(y_n + \frac{h}{2} \cdot k_1, t_n + \frac{h}{2}\right) \\ k_3 &= f\left(y_n + \frac{h}{2} \cdot k_2, t_n + \frac{h}{2}\right) \end{aligned} \tag{2.7}$$

$$k_4 = f(y_n + h \cdot k_3, t_n + h)$$

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Generalmente, la ecuación matricial depende del tiempo (variables de contorno en el término independiente). A nivel de paso consideraremos una retención de orden cero (para simplificar, sin interpolación con los valores anteriores), con lo que $t_n = t_n + \lambda h$ con $\lambda \in [0,1]$. Por tanto, en la implementación del programa emplearemos:

$$\begin{aligned}
\overrightarrow{k_1} &= A \cdot \overrightarrow{y_n} + \vec{b}(t_n) \\
\overrightarrow{k_2} &= A \cdot \left(\overrightarrow{y_n} + \frac{h}{2} \cdot \overrightarrow{k_1}\right) + \vec{b}(t_n) \\
\overrightarrow{k_3} &= A \cdot \left(\overrightarrow{y_n} + \frac{h}{2} \cdot \overrightarrow{k_2}\right) + \vec{b}(t_n) \\
\overrightarrow{k_4} &= A \cdot \left(\overrightarrow{y_n} + h \cdot \overrightarrow{k_3}\right) + \vec{b}(t_n) \\
\overrightarrow{y_{n+1}} &= \overrightarrow{y_n} + \frac{h}{6}\left(\overrightarrow{k_1} + 2\overrightarrow{k_2} + 2\overrightarrow{k_3} + \overrightarrow{k_4}\right)
\end{aligned} \tag{2.8}$$

Comparando con el método de Euler, la evaluación de las matrices se complica un poco, pero la aproximación es muy buena. El algoritmo está en el método **`rk4()`**.

Clase Continuo	

Variables	• Variables auxiliares para los métodos de integración
	`std::vector<T> eul, k1, k2, k3, k4;`
	• Matriz asociada y vector de términos independientes
	`std::vector< std::vector<T> > asoc;`
	`std::vector<T> indep;`

Métodos	• Constructor
	Además de los parámetros de dispositivo, se proporciona referencia de la matriz asociada y el vector de términos independientes
	`Continuo(unsigned int t, int id, const std::vector< std::vector<T> >& a, const std::vector<T>& b);`
	• Métodos de integración numérica (Euler y RK4)
	`void euler();`
	`void rk4();`
	• Funciones de evaluación (sobrecarga)
	`void eval(std::vector<T>& salida);`
	`void eval(std::vector<T>& entrada, std::vector<T>& salida, T propor);`
	• Actualiza
	`int actualiza();`

Observación: Como ocurre en Dinamico, la clase puede manejar cualquier tipo de datos por usar plantilla.

Estos métodos también han de hacer uso de la memoria en **`Dinamico`**, que será un vector de n elementos, de colas unitarias.

Es interesante ver cómo se declara mediante **`std::vector`** una matriz. Al igual que en C, una matriz es considerada un vector de vectores, con **`std::vector<std::vector<T>>`** y el manejo de cada término se hace con la clásica indexación **`asoc[i][j]`**.

Los métodos de integración no necesitan argumentos porque actualizan el vector **`Dinamico::siguiente`** a partir del estado actual, la matriz asociada, y los términos independientes (actualizados por el dispositivo derivado que use esta clase).

Se proporcionan dos formas de evaluar la ecuación matricial, mediante el método **`eval()`**. La evaluación sencilla es la usada en Euler y consiste en calcular el vector $\vec{r} = A \cdot \overrightarrow{y_n} + \vec{b}(t_n)$ con $\overrightarrow{y_n}$ como el estado presente del dispositivo.

Sin embargo, hemos visto que para hallar $\overrightarrow{k_2}, \overrightarrow{k_3}, \overrightarrow{k_4}$ en Runge-Kutta hemos de hacer otras operaciones complementarias, por eso **`eval()`**, para RK4, admite una constante λ y un vector $\vec{w}$ de tal forma que se realiza la operación:

$$\vec{r} = A \cdot (\overrightarrow{y_n} + \lambda \cdot \vec{w}) + \vec{b}(t_n) \tag{2.9}$$

Con estas evaluaciones auxiliares los métodos de integración calculan $\overrightarrow{y_{n+1}}$

También apreciamos que con esta clase hemos hecho uso de las referencias, concretamente con **`eval(std::vector<T>& entrada, std::vector<T>& salida, T propor);`**

El uso de las referencias es muy similar a los punteros. Cuando pasamos un valor por referencia en el argumento la función puede cambiar el valor de la variable, en la rutina que llamó a la función, siempre y cuando este no sea **`constant`** (declarado constante; de solo lectura).

La ventaja de las referencias es que nos ahorran tener que anteponer el asterisco **`*`** en la variable cuando se cambia su valor. Se simplifica mucho la notación. En algunas funciones, en vez de pasar punteros (puesto que son zonas de memoria) incorporaremos referencias.

2.5.2. Motor eléctrico de corriente continua acoplado a una carga mecánica

Una vez que se han confeccionado las clases que ofrecen un soporte de cálculo adecuado, es el momento de hacer uso de ellas. Lo apropiado es crear subclases derivadas, ya que éstas tendrán un acceso inmediato a los métodos **`public`** y **`protected`** de las implementaciones realizadas. Tan solo han de cumplir algunos requisitos que mostraremos en el capítulo 3.

El caso habitual de modelado de un sistema de ecuaciones diferenciales es mediante una ecuación matricial, con matriz del sistema de coeficientes constantes.

Un motor eléctrico es un dispositivo que transforma la energía eléctrica en movimiento, generalmente rotatorio. Esta implementación es sencilla y servirá para luego simular un carro unido al motor con un tornillo sinfín. El motor eléctrico, equivale, desde el punto de vista de la fuente de voltaje, a una bobina y una resistencia. El sistema mecánico con respecto al rotor tiene un momento de inercia I_L y la carga acoplada al motor se caracteriza por un par variable $T_{SF}(t)$ que tiende a frenar el motor. La fuerza de rozamiento es proporcional a $b \cdot \omega(t)$.

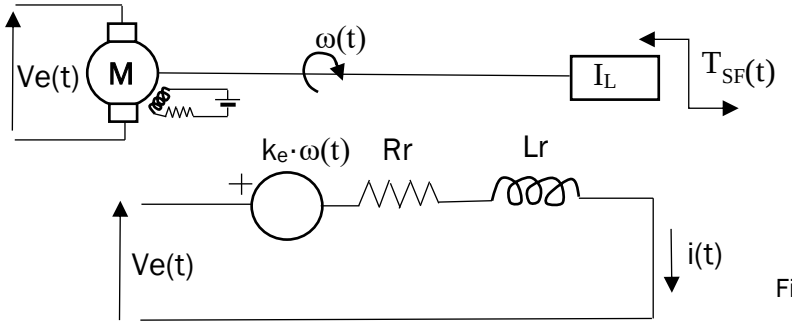


Figura 2.8

Las ecuaciones de cada variable de estado del modelo físico son:

$$V_e(t) = L_r \cdot \frac{di(t)}{dt} + R_r \cdot i(t) + k_e \cdot \omega(t)$$

$$I_L \cdot \frac{d\omega(t)}{dt} = k_s \cdot i(t) - b \cdot \omega(t) - T_{sf}(t) \qquad (2.10)$$

$$\frac{d\theta(t)}{dt} = \omega(t)$$

Hay que tener en cuenta que el movimiento del motor induce una tensión en el circuito de valor $k_e \cdot \omega(t)$ y que el par que produce el motor es proporcional a la corriente del circuito eléctrico y tiene un valor $k_s \cdot i(t)$.

Las dos variables de contorno son el voltaje que proporcionamos al motor $v_e(t)$ y el par $T_{SF}(t)$. Ambos son actualizados externamente.

Nos interesa la forma matricial, y tenemos tres variables de estado que son la corriente, la velocidad angular y la posición angular del eje del motor. Despejando las tres derivadas de las fórmulas anteriores obtenemos la ecuación diferencial en su forma vectorial $\vec{y}'(t) = A \cdot \vec{y}(t) + \vec{b}(t)$

$$\begin{pmatrix} i'(t) \\ \omega'(t) \\ \theta'(t) \end{pmatrix} = \begin{pmatrix} -R_r/L_r & -k_e/L_r & 0 \\ k_s/I_L & -b/I_L & 0 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} i(t) \\ \omega(t) \\ \theta(t) \end{pmatrix} + \begin{pmatrix} V_e(t)/L_r \\ -T_{sf}(t)/I_L \\ 0 \end{pmatrix} \qquad (2.11)$$

La matriz característica A solo depende de los parámetros. Es decir, con un diseño determinado el sistema tiene una estabilidad constante, sin importar el

estado; y también una respuesta en frecuencia constante. La variable de contorno (tensión de entrada) se halla en el vector de términos independientes $\vec{b}(t)$. Este tipo de sistemas son los más sencillos de analizar y de simular, haciendo uso de los métodos de la clase Continuo. El archivo motorcc.cpp contiene la implementación del modelo.

2.5.3. Suspensión mecánica con elemento actuador

Este modelo, de carácter mecánico, servirá de ejemplo en el siguiente capítulo para describir los conectores y la concurrencia. Al igual que el motor eléctrico anterior tenemos dos variables de contorno: un actuador hidráulico de fuerza f(t) y un perfil de calzada $x_3(t)$.

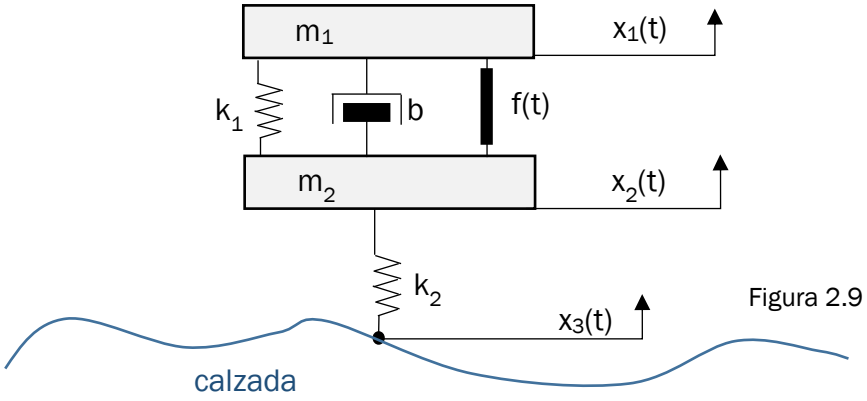


Figura 2.9

Tan solo tenemos que formalizar el equilibrio de fuerzas en las dos masas (m_1: vehículo, m_2: rueda); k_1 y k_2 son las constantes elásticas del resorte y de la cámara de aire de la rueda; b es la constante del amortiguador (disipador de energía). Obtenemos:

$$m_1 \frac{dv_1}{dt} = -k_1(x_1 - x_2) - b(v_1 - v_2) - m_1 g + f(t)$$

$$m_2 \frac{dv_2}{dt} = k_1(x_1 - x_2) - k_2[x_2 - x_3(t)] + b(v_1 - v_2) - m_2 g - f(t) \qquad (2.12)$$

$$v_1 = \frac{dx_1}{dt} \qquad v_2 = \frac{dx_2}{dt}$$

Despejando las derivadas, obtenemos la ecuación diferencial matricial, que como en el caso anterior, tiene una matriz característica con coeficientes constantes, pero con dos funciones de contorno en el vector de términos independientes:

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{v}_1 \\ \dot{v}_2 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -k_1/m_1 & k_1/m_2 & -b/m_1 & b/m_1 \\ k_1/m_2 & (-k_1 - k_2)/m_2 & b/m_2 & -b/m_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ v_1 \\ v_2 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \frac{f(t)}{m_1} - g \\ \frac{k_2 \cdot x_3(t) - f(t)}{m_2} - g \end{pmatrix}$$

(2.13)

De nuevo, como en el apartado anterior, se puede analizar la estabilidad, frecuencias propias, etc. con las constantes de la matriz asociada.

$\mathrm{f(t)}$ es la fuerza ejercida por el actuador hidráulico. Se ha considerado que esta fuerza tiene una componente estocástica y se puede descomponer en la fuerza efectiva $g(t)$, y en una componente de ruido $h(t)$, es decir $\mathrm{f(t)} = \mathrm{g(t)} + \mathrm{h(t)}$.

Al haber tres entradas, describiremos en el capítulo siguiente como podemos hacer funcionar este dispositivo junto con los dispositivos generadores de funciones y números aleatorios, gracias a los conectores.

2.5.4. Dispositivos discontinuos. Carro motorizado.

El motor de corriente continua del apartado 2.5.2 puede ser empleado para mover un pequeño tren eléctrico, para mover un péndulo inverso, un brazo robótico y un largo etcétera. Una vez que hemos programado la clase en C++ motorcc.cpp. con el modelo matemático correspondiente, se pueden derivar clases más especializadas, que hacen un uso del motor.

Es habitual encontrarse con discontinuidades en el mundo físico. Por ejemplo, las paredes de un recinto o los topes de un carril que impiden a un objeto seguir avanzando. Implementaremos un carro motorizado que transcurre por un carril y se mueve gracias a un tornillo sinfín.

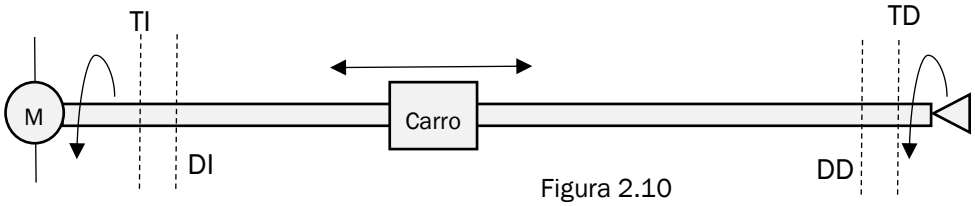


Figura 2.10

El carro se mueve de derecha a izquierda accionado por el giro del motor. Existen dos detectores conectados con el controlador (derecho e izquierdo) que hemos etiquetado como DD y DI. Y hay dos topes TD y TI a partir de los cuales físicamente el carro no puede avanzar.

Esto es un ejemplo de especialización del motor del apartado anterior. Las variables físicas pasan de ser velocidad y posición de giro a ser velocidad lineal y posición lineal, con tan solo multiplicar por un coeficiente c que depende del tornillo sinfín elegido (paso, ángulo de hélice, diámetro, etc.).

$$\begin{aligned} v(t) &= c \cdot \omega(t) \\ x(t) &= c \cdot \theta(t) \end{aligned} \tag{2.14}$$

Aumenta el rozamiento b (por el tornillo sinfín) y no hay par externo $T_{sf}(t)$. Sin embargo, la peculiaridad está en hacer real el modelo cuando alcanza uno de los topes. Físicamente, el motor deja de girar. Suponiendo que el choque es perfectamente inelástico, la velocidad angular se anula inmediatamente:

$$\omega(t) = 0$$

Con esto, varían notablemente las ecuaciones del modelo físico del motor:

$$V_e(t) = L_r \cdot \frac{di(t)}{dt} + R_r \cdot i(t) \tag{2.15}$$

Al no haber velocidad angular, desaparece la fuerza contraelectromotriz que era $k_e \cdot \omega(t)$ y, por tanto, la corriente se mantiene en valores máximos. Tener la corriente así por mucho tiempo es peligroso. Se puede quemar la bobina.

En esencia, el escenario cambia notablemente al anular una de las variables de estado. Ahora tenemos sólo una variable de estado (intensidad) y una ecuación algebraica que relaciona el par del motor estático con la intensidad.

$$M(t) = k_s \cdot i(t) \tag{2.16}$$

Esta condición especial sólo se abandonará cuando la tensión de entrada cambie de polaridad y, por tanto, el motor se ponga a girar en sentido contrario. Entonces la ligadura desaparece y el motor vuelve a su modo de operación normal.

El par de torsión que resiste el movimiento cambia aquí de naturaleza. Además del rozamiento implícito b que tiene el eje del motor, se ha definido un puerto de entrada de datos $r(t)$ que representa coeficiente de rozamiento del carro por el rail. Se relaciona con el par de resistencia $T_{sf}(t)$ de tal forma que:

$$T_{sf}(t) = \mathrm{m} \cdot \mathrm{g} \cdot r(t) \tag{2.17}$$

Con m como la masa del carro y g la aceleración gravitatoria. Este par de resistencia por rozamiento nunca puede ser, en valor absoluto, mayor que el par motor. Para determinar $r(t)$ crearemos una señal aleatoria que representa la rugosidad del raíl por donde trascurre el carro.

El uso de restricciones es muy habitual y fácil de efectuar utilizando un lenguaje de programación de propósito general con tal solo poner condicionales. Este ejemplo propuesto está en carro_motorizado.cpp, que es una clase derivada de motorcc.cpp.

2.5.5. Dispositivos no lineales. Depósito de agua.

Hemos visto en el apartado anterior que existen modelos que tienen discontinuidades, ligaduras, y falta de libertad de movimientos. Ello hace que cambien sus ecuaciones y sus variables, en según qué condiciones.

Aquí nos centraremos en otra particularidad bastante común en sistemas físicos que es la falta de linealidad, y que nos obligará a operar de otra forma sobre la matriz característica.

Ideamos un depósito de perfil curvo y de sección variable, en función de la altura h, de altura máxima de desbordamiento H que se llena con un caudal de agua y que se vacía por un sumidero cuyo coeficiente de descarga es k_s.

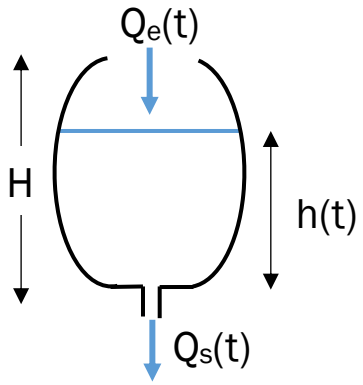


Figura 2.11

Por Ley de conservación de la masa de un fluido incompresible (densidad constante) en un volumen de control, de altura h y de sección $A(h)$, tenemos:

$$A(h) \cdot \frac{dh}{dt} = Q_e(t) - Q_s(t) \tag{2.18}$$

Por otra parte, se ha de cumplir el principio de conservación de la energía, cinética y potencial, a presión atmosférica. Consideraremos un comportamiento cuasiestático del depósito para satisfacer la ecuación estacionaria de Bernoulli, entre un punto en la superficie del agua y la salida del sumidero:

$$g \cdot h(t) - \frac{1}{2}\left(\frac{Q_s(t)}{k_s A_s}\right)^2 = 0 \tag{2.19}$$

g es la aceleración gravitatoria y A_s es la sección de salida. Despejando Q_s en la primera ecuación y sustituyendo en la segunda, obtenemos la ecuación diferencial que describe el sistema. Tras operar y simplificar:

$$k_s \cdot A_s \cdot \sqrt{2g \cdot h(t)} = Q_e(t) - A(h) \cdot \frac{dh}{dt} \tag{2.20}$$

Para completar el modelo, definimos A(h) que es la sección del depósito en función de la altura, según el contorno. Elegiremos, por ejemplo, uno de perfil parabólico, de sección de fondo de 1 m²: $A(h) = \left(1 + \sqrt{h}\right)^2$ (2.21)

Sustituimos en la ecuación diferencial, y para simplificar definimos la constante de diseño $c = k_s \cdot A_s \cdot \sqrt{2g}$ que depende del coeficiente de descarga y la sección del sumidero:

$$\left[1 + \sqrt{h(t)}\right]^2 \frac{dh}{dt} + c\sqrt{h(t)} = Q_e(t) \tag{2.22}$$

Aunque estamos ante una ecuación diferencial de primer orden, es evidente que ésta no es de coeficientes constantes, y ni siquiera es lineal.

Sin embargo, esto no nos va a evitar utilizar los métodos que hemos usado en las anteriores situaciones. Disponemos de dos herramientas:

1) Hacer lineal la ecuación diferencial, a partir de un punto de operación nominal como es el nivel habitual del depósito, o el nivel estacionario h_0, aplicando la serie de Taylor, tras lo cual, podemos usar los métodos habituales implementados en la clase Continuo. Al ser una aproximación local, no puede diferir excesivamente el punto de operación $h(t)$ de h_0 lo cual puede suponer una desventaja si tenemos grandes variaciones de nivel del depósito.

$$\left[1+\sqrt{h(t)}\right]^2\frac{dh}{dt}+c\sqrt{h(t)}=Q_e(t) \tag{2.23}$$

El sistema es función de la altura del líquido, de su derivada y de la función de contorno del caudal de entrada; es decir:

$$f(h',h,Q_e)=0 \tag{2.24}$$

Y linealizamos en torno a los puntos h_0 y Q_{e0} utilizando la fórmula de Taylor para múltiples variables, de tal forma que:

$$\left[\frac{\partial f}{\partial h'}\right]_{h'=h_0'}\cdot(h'-h_0')+\left[\frac{\partial f}{\partial h}\right]_{h=h_0}\cdot(h-h_0)+\left[\frac{\partial f}{\partial Q_e}\right]_{Q_e=Q_{e0}}\cdot(Q_e-Q_{e0})=0 \tag{2.25}$$

Es frecuente hacer estas aproximaciones, eligiendo convenientemente los puntos de linealización h_0', h_0 y Q_{e0}. Como es muy fácil conocer la estabilidad del sistema en el rango de altura H, nos guiaremos por el método que describimos a continuación.

2) Variar los coeficientes de la matriz A del sistema, según el nivel del depósito, y, por tanto, crear una clase específica que varíe el coeficiente del sistema en cada iteración que está en función de $h(t)$. Es sencillo, pues hay una variable de estado y una matriz unitaria que definir al utilizar la clase **`Continuo`**. Es decir, en vez de una matriz, consideramos un escalar. Despejamos la derivada:

$$\frac{dh}{dt}=\left[-\frac{c\sqrt{h(t)}}{\left(1+\sqrt{h(t)}\right)^2}\right]+\frac{Q_e(t)}{\left(1+\sqrt{h(t)}\right)^2} \tag{2.26}$$

La función entre corchetes del miembro de la derecha es continua en el ámbito de operación del depósito, con $h(t)>0$. Es también una función negativa, mientras la altura $h(t)$ sea finita, lo que implica que el sistema es estable. De hecho, si no hubiera caudal entrante, el depósito tendería a vaciarse y a alcanzar la altura cero.

Finalmente, el método de integración, de paso fijo, ya sea Euler o Runge-Kutta orden cuatro, ha de ser también estable. La precisión de integración varía al ser variable el coeficiente. Para integración por Euler se necesita que:

$$|1 + \lambda p| < 1 \tag{2.27}$$

Al ser $\lambda < 0$ y $p > 0$ el criterio de estabilidad se transforma en:

$$|\lambda p| < 2 \tag{2.28}$$

Siendo p el paso y λ el autovalor de la matriz del sistema. El autovalor es fácil de hallar, con tan solo anular la variable de contorno $Q_e(t)$:

$$h'(t) = \lambda \cdot h(t) \tag{2.29}$$

Y teniendo en cuenta la expresión del modelo del depósito, el autovalor es:

$$\lambda = -\frac{c}{\sqrt{h(t)} \cdot \left(1 + \sqrt{h(t)}\right)^2} \tag{2.30}$$

Es decir, es el coeficiente del sistema a actualizar en cada iteración. Por tanto, para garantizar la estabilidad del método numérico se ha de cumplir que:

$$p < \frac{2 \cdot \sqrt{h(t)} \cdot \left(1 + \sqrt{h(t)}\right)^2}{c} \tag{2.31}$$

Con $c = k_s \cdot A_s \cdot \sqrt{2g}$

Si, por ejemplo, tenemos un depósito con un coeficiente de descarga a la salida de $k_s = 0.8,$ y de área en el conducto de salida $A_s = 70\ cm^2$, para poder integrar con garantía la ecuación diferencial, con un paso $p = 0.01$ (frecuencia de muestreo de 100 Hz) entonces ha de ser $h(t) > 10^{-7}\ m$. En Runge-Kutta orden cuatro, la cota mínima de altura para garantizar la estabilidad es aún menor, pero nunca llega a ser cero.

Por tanto, se puede utilizar el modelo mediante integración de matiz de coeficientes variables, en un amplio margen. Evitaremos integrar en valores muy próximos a la altura del agua cero. La implementación está en deposito.cpp.

Se pueden integrar sistemas no lineales variando los coeficientes de la matriz, según el estado, pero hay que hacer un análisis previo que garantice la convergencia del método numérico aplicado, en unos márgenes establecidos.

Cuando se implemente el depósito y se ensaye, se puede poner k_s como señal de entrada, al igual que el caudal. Esa señal puede representar una válvula que se abre o cierra. Con $k_s = 1$ es como tener una válvula abierta sin rozamiento, y con $k_s = 0$ es como tenerla cerrada. También se puede asignar una distribución aleatoria en torno a un valor central, a semejanza de un flujo turbulento.

2.6. Tolerancias. Alarma del dispositivo.

El concepto de alarma es general y, por lo general, consideraremos una condición de alarma el estado de dispositivo que requiere atención del usuario. Iremos viendo cómo la alarma de dispositivo varía según el dispositivo y el contexto. En los dispositivos de planta es evidente que un depósito que se llena no debe rebosarse, o que por una bobina no puede pasar un exceso de corriente, puesto que ésta se fundiría, por efecto Joule.

Las alarmas son irreversibles durante la ejecución del proceso. Existe un solo método ubicado en **`Dispositivo`** que habilita la alarma. Su variable, encapsulada y de carácter privado sólo puede ser modificada para ser activada, con el método correspondiente, al que solo pueden acceder las clases derivada:

```
void activa_alarma(); // Activar alarma
```

Este estatus, junto con el de ejecución, pausa, duración y detención de hilos es considerado relevante y así se mostrará en el menú de dispositivos, cuando abordemos la interacción con el usuario.

En los dispositivos de planta podemos variar las tolerancias (máxima fuerza en el cilindro hidráulico, máxima corriente en la bobina del motor y máxima altura del depósito hasta el desbordamiento) para evitar que salten las alarmas. Aquí nos indican simplemente que se ha rebasado una tolerancia, para así hacer ajustes sobre el modelo.

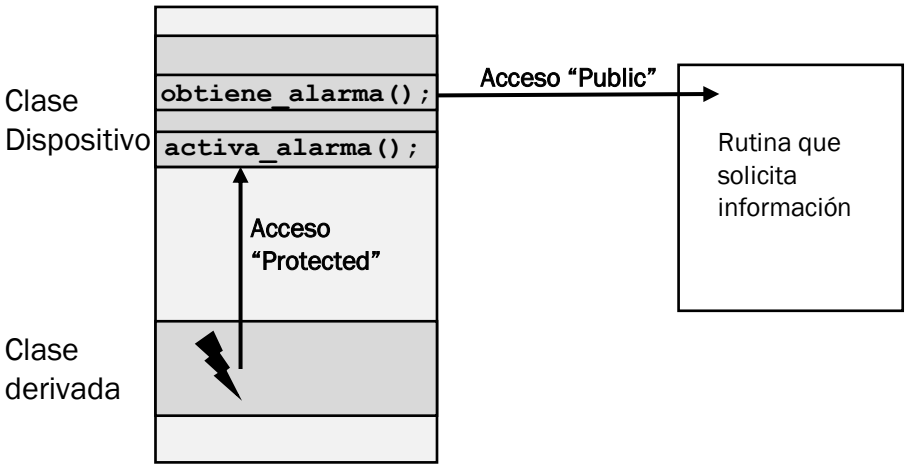


Figura 2.12

3. Conexión y multitarea

Hemos definido una colección de clases con las que crearemos objetos como pueden ser dispositivos de planta, en los que implementamos su modelo matemático, haciendo uso de los métodos correspondientes. Tenemos unidades funcionales, pero por sí solas no pueden hacer nada. Están aisladas.

Si tomamos el ejemplo del sistema mecánico del amortiguador, el generador de señales y el generador de número aleatorios, necesitamos relacionarlos, y que todos estos elementos tengan la posibilidad de transferir datos. Es más; si además de varios dispositivos de planta, tenemos varios controladores, es obvio que hemos de disponer de un sistema de comunicaciones, de varios canales.

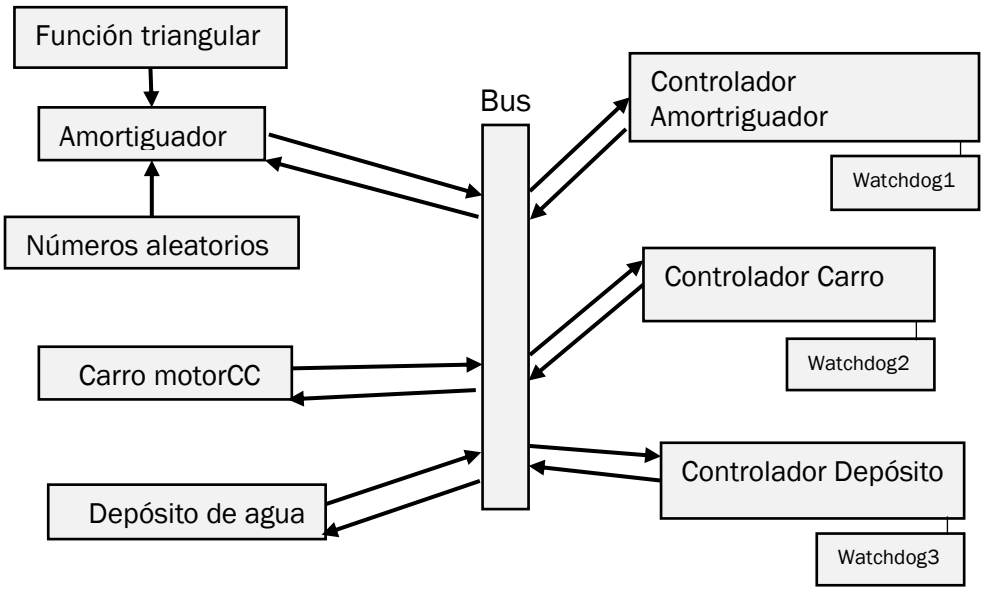


Figura 3.1

La conexión y la gestión de la concurrencia de hilos siguen estas tres líneas generales:

1) La entrada/salida de datos de los modelos de planta se dirigen a otras entidades que podrían ser otros dispositivos de planta, generadores de señal, números aleatorios, o también dispositivos de captación de datos, que permitan la comunicación con el controlador correspondiente. Hemos de establecer, por tanto, un canal de flujo de datos que unan pares de dispositivos. De esta forma surge la necesidad de los conectores, los puertos, y los buses que implementaremos en el presente capítulo.
2) El hilo de ejecución de cada dispositivo puede entrar en conflicto con hilos de otros dispositivos y producirse *data race*. Imaginemos que un hilo de ejecución está escribiendo una secuencia de datos de salida para que lo lea un dispositivo lector. Si, por planificación automática de tareas, se detiene

el proceso y el hilo que escribía se queda en espera, con la tarea a medias, el hilo lector, de activarse, leerá datos erróneos. Estos errores pueden aparecer de una forma impredecible, haciendo difícil la depuración.

El ejemplo expuesto ilustra el cuidado con el que hemos de programar las rutinas conectivas, compartición de datos y recursos comunes. A priori, dos elementos proporcionados por C++ se nos hacen indispensables: los **`std::mutex`** (objetos de exclusión mutua que sincronizan la concurrencia entre dos o más hilos) y datos del tipo **`std::atomic`**, que no pueden ser leídos o escritos de forma incompleta por un hilo de ejecución.

3) Interacción con el usuario. Entrada, grabación y análisis de datos. Es casi imposible concebir un sistema de control en el que el usuario no pueda influir o visualizar su comportamiento. Nos serviremos de archivos de datos que pueden ser mostrados con herramientas de visualización gráfica como GNU Plot. Para la entrada, la interacción por teclado es suficiente, aunque se deja la posibilidad, en un futuro, de que puedan aceptarse archivos de datos con los que configurar automáticamente los controladores, más otras prestaciones interesantes.

Para satisfacer esta interconexión se han ideado dos clases que son los "conectores" y los "conectables". Imaginemos cables de conexión entre dispositivos y sus clavijas para conectar. En el siguiente apartado se introducen conceptos muy relevantes y una introducción a la concurrencia de hilos.

3.1. Conexión de dispositivos

Veamos en qué se distinguen los conectores y los conectables:

- Los **conectores** son los propios dispositivos de conexión. Se derivan de dispositivos y son entidades pasivas (sin hilo propio).
- Los **conectables** son los dispositivos que se pueden conectar unos a otros, gracias a los conectores. Hacen uso de la clase **`Conectable`**.

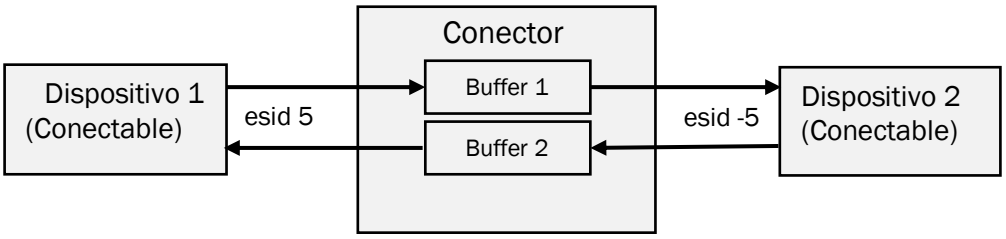


Figura 3.2

Las flechas indican el flujo de los datos entre cada dispositivo. Estas comunicaciones bidireccionales son estructuras que definimos previamente en los dispositivos, y que son tratadas como bloques de datos por conectores, discretizadores y buses. Cada tipo de estructura tiene una identificación numérica única que etiquetamos como "esid" (id de entrada/salida).

Hemos creado las clases que daban lugar a los dispositivos continuos, pero no tuvimos oportunidad de hacerlos conectables. Existen varias formas de solventarlo en C++, nosotros incluiremos dos nuevas clases: **`Conector`** y **`Conectable`**, en la jerarquía de clases, de tal forma que:

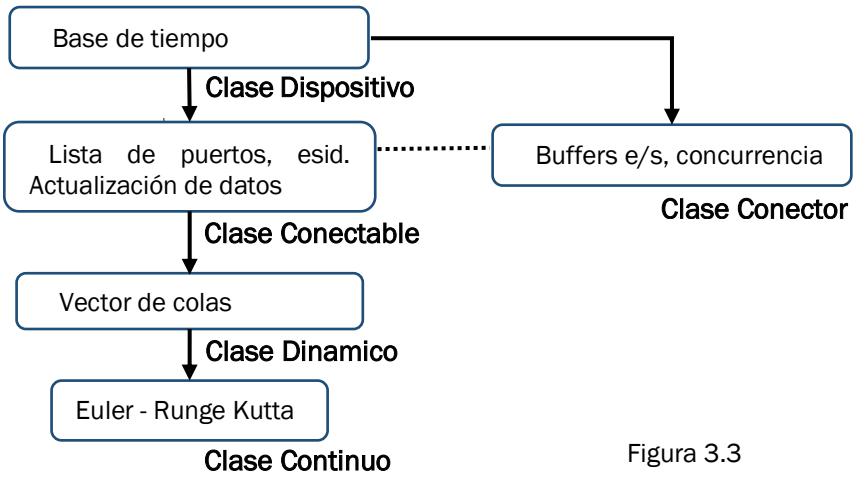


Figura 3.3

La línea discontinua indica que ambas clases tienen una relación bastante estrecha, al igual que un cable tiene relación con sus clavijas o extremos. Vamos a ir desgranando las nuevas características que incorporaremos al programa y que nos permitirá una comunicación completa entre los dispositivos que habíamos desarrollado en el anterior capítulo.

3.1.1. Elementos de conexión. Identificadores y puertos. Objetos conectables.

Se puede tener toda clase de bloques de datos para la comunicación, que se definen con las clásicas **`struct`** o estructuras en C.

Tomemos como ejemplo la comunicación que estableceremos entre planta y controlador, desde el lado del dispositivo de suspensión:

```
struct ES_1 { // Identificador de estructura esid = 1
          struct ENTRADA_ES_1 {
                float e1; // Fuerza del actuador
          } entrada;

          struct SALIDA_ES_1 {
                float s1; // Posiciones de chasis y rueda
                float s2;
                bool alarma; // Notificación de alarma
          } salida;
    };
```

Como es evidente, el dispositivo tiene como entrada la fuerza que se le dará al actuador hidráulico; y, como salida, los valores de las posiciones de chasis y rueda, pudiendo, además, transmitirse la señal de alarma.

Si tuviéramos que conectar con un cable esta suspensión mecánica, al otro lado recibiríamos los datos que salen de la suspensión. Tendríamos que

suministrarle los datos que van a la suspensión. Es decir, las entradas se convierten en salidas, y viceversa. La solución de notación para este cambio de perspectiva es definir una estructura complementaria con el valor opuesto. Si, por ejemplo, hemos conectado el amortiguador a un controlador, su esid pasa a ser -1 y tendríamos esta estructura:

```
struct ES_N1 { // Identificador de estructura esid = -1
          struct ENTRADA_ES_N1 {
                float e1; // Posiciones de chasis y rueda
                float e2;
                bool alarma; // Notificación de alarma
          } entrada;

          struct SALIDA_ES_N1 {
                float s1; // Fuerza del actuador
          } salida;

    };
```

Si lo comparamos con esid=1 vemos que las entradas y las salidas están intercambiadas. Añadiremos en la notación de la estructura la letra N, para indicar que es la estructura complementaria. Una vez definidas, es fácil averiguar su tamaño mediante **`sizeof()`** y así determinar el tamaño de los buffers de transferencia de datos en conectores, discretizadores, etc.

Cada dispositivo puede tener varias estructuras de entrada/salida. Por ejemplo, en este caso del amortiguador puede haber una señal que altura de calzada y otra de perturbación, por lo que no basta la comunicación que tiene con el controlador. Por tanto, la conectividad se hace flexible, adaptada según nuestras necesidades, mediante nodos. Con esta flexibilidad, surge el concepto de puerto.

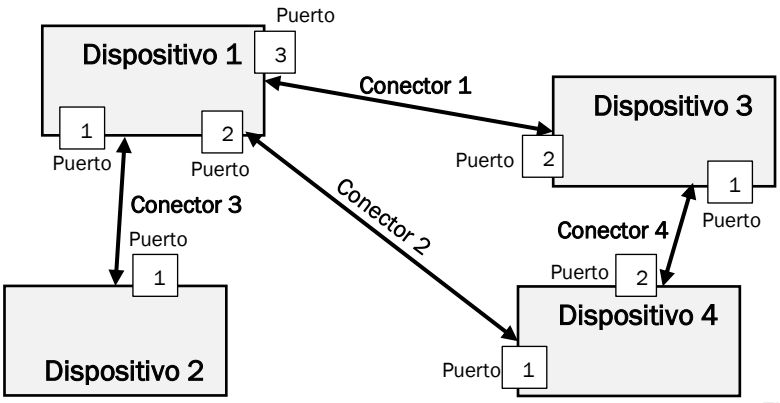


Figura 3.4

La asignación de puertos se hace con números naturales, configurándose estos en el código del dispositivo (por ejemplo, en su constructor), gracias al método **`nuevo_puerto()`** de la clase **`Conectable`**. Las características de cada conexión han de ser almacenadas utilizando un vector de estructura o una serie de vectores. Cada puerto tiene asociado una serie de parámetros que exponemos en la siguiente tabla:

Parámetros de conexión de cada puerto en **Conectable**	
`unsigned int puerto`	Número de puerto de la conexión
`int esid`	Identificación esid de la conexión
`bool asignado`	El puerto está asignado a un conector
`BYTE *entrada`	Puntero a los datos de entrada en el dispositivo
`BYTE *salida`	Puntero a los datos de salida en el dispositivo
`unsigned int lentrada`	Tamaño de datos de entrada
`unsigned int lsalida`	Tamaño de datos de salida
`bool solo_lectura`	El puerto es de solo lectura
`int err`	Código de retorno de la última E/S

Nota: BYTE es un alias de unsigned char, el cual suele ocupar un byte en C/C++

Por supuesto, estos parámetros se replican en cada puerto, de esta forma el dispositivo puede manejar cualquier tipo de conexiones. El método **`actualiza()`** de **`Conectable`**, en cada iteración del dispositivo realiza una lectura de los parámetros de cada puerto y llama al conector correspondiente, realizando el intercambio de datos.

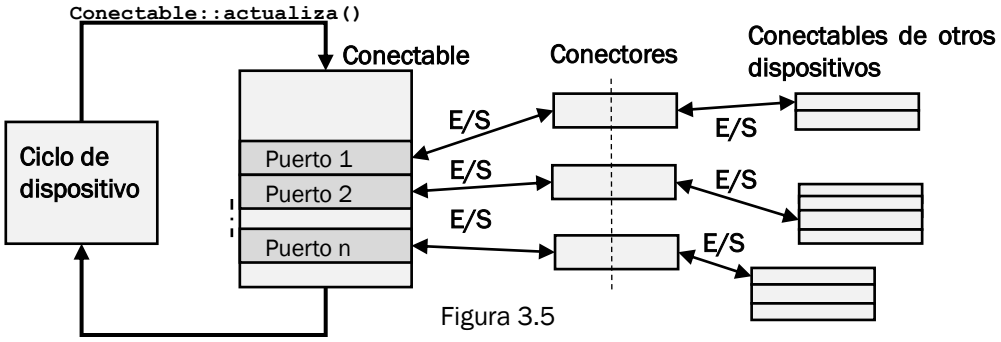


Figura 3.5

Almacenar los tamaños de datos es fundamental, puesto que los dispositivos que veremos más adelante sólo van a trabajar con paquetes de datos sin que conozcan su estructuración. Con conocer el tamaño pueden realizar su tarea de transmisión. La estructura anteriormente descrita se declara con **`std::vector`**.

`Conectable` suele ser de uso interno para las clases derivadas. Además de la estructura comentada anteriormente existen estos elementos:

Clase Conectable	

Variables	• Habilitación de comunicación E/S
	`bool comunicable;`
	• Código de retorno (error) más reciente. Se usa en discretizadores, cuando se tiene solo un puerto.
	`std::atomic<int> err_reciente {0};`

Métodos	• Constructor
	`Conectable(unsigned int t, int id);`
	• Modificar en un puerto el bit de solo lectura
	`int modificar_solo_lectura(unsigned int puerto, bool solo_lectura);`

	• Actualizador
	`int actualiza();`
	• Crear nuevo puerto
	`int nuevo_puerto(unsigned int puerto, int esid, BYTE *entrada, BYTE *salida, unsigned int lentrada, unsigned int lsalida);`
	• Conectar. Se elige conector y puerto. Es el único método accesible para los demás objetos y principal cuando se configure todo el sistema interconectado.
	`int conectar(Conector *conector, unsigned int puerto);`

Una vez que hemos determinado cómo conectar los dispositivos, nos centramos en los conectores que harán efectiva esa conexión. Ayudarán a introducirnos en aspectos muy interesantes de C++ y la concurrencia de hilos.

3.1.2. El conector. Sincronización E/S de dispositivos

Analizaremos el tránsito de datos por el conector.

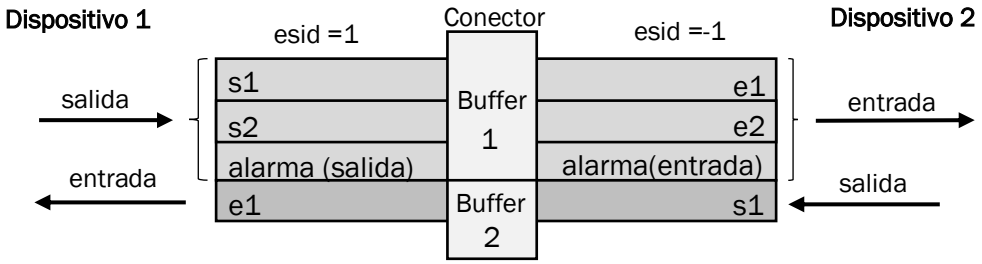


Figura 3.6

La nomenclatura usada en **`Conector`** establece como entrada todo flujo de datos que entra al conector, sea desde el dispositivo 1 o el dispositivo 2, desde el punto de vista del conector. El conector que hemos creado es más que un simple cable. También hace las funciones de espacio intermedio de memoria (buffer), protección y aviso.

1. El conector tiene dos buffers en los cuales ambos dispositivos leen y escriben.

El conector desconoce la estructura de los datos y copia dichos datos en bloque utilizando **`std::memcpy`**, desde el buffer al dispositivo o viceversa. Esta memoria se podría reservar de forma dinámica con **`std::malloc`** pero utilizaremos el operador new.

En vez de **`std::free`** utilizamos el operador **`delete`**. Al crear el conector llenamos los buffers inicialmente de ceros con **`std::memset`**. No hay inconveniente en emplearlo y, de hecho, se aconseja, porque utiliza un algoritmo rápido (en ensamblador) para realizar su tarea. El uso de **`new`** y **`delete`** se generaliza para cualquier objeto. En el caso de arrays de memoria no estructurada como la que aquí necesitamos, hacemos uso del tipo **`BYTE`** (**`unsigned char`**). Si queremos reservar memoria de tamaño **`tam`**, escribimos:

```
BYTE *buffer = new[tam];
```

Y si queremos liberar esta memoria, utilizamos **delete**.

```
delete buffer;
```

2. **Se habilita un control de flujo de datos y un protocolo entre los dos dispositivos** para que no interfieran al leer/escribir en los buffers. Además, es interesante que un dispositivo conozca si el otro leyó (al escribir éste) o escribió (en el momento que está leyendo).

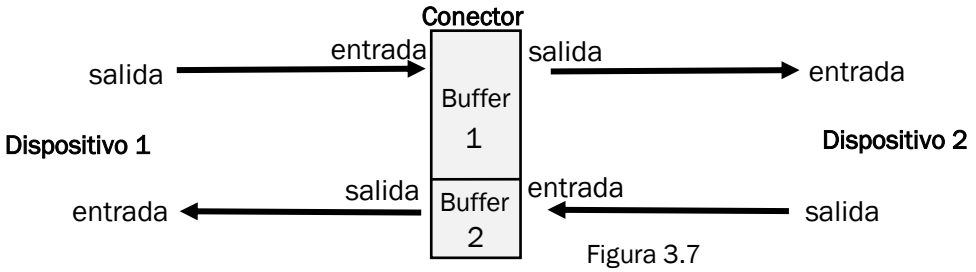


Figura 3.7

Los argumentos de los métodos del conector se refieren desde el punto de vista del conector. Es decir, la salida del dispositivo es la entrada del conector, y viceversa. Cada buffer E/S unidireccional (para cubrir la comunicación bidireccional) tiene asociado un bit llamado **escrito_leido**.

- Si **escrito_leido == false**, el dato fue leído por el dispositivo lector en ese buffer. Significa que el dispositivo que escribe puede aportar otro paquete de datos con la seguridad de que no se perderá el anterior.
- Si **escrito_leido == true**, el dato fue escrito por el dispositivo que escribe en ese buffer. El dispositivo que lea tiene la certeza de que el dato ha sido actualizado y puede copiarlo.

Cada vez que un dispositivo lea o escriba (da igual en qué dirección o buffer) asignará un valor adecuado en el bit.

Este mecanismo de arbitraje evita lecturas y escrituras innecesarias, además de que evita datos redundantes en la secuencia E/S de cada dispositivo.

Según el bit interno **escrito_leido**, se proporciona a los dispositivos un código de retorno o de salida:

Retorno	Significado
0	No se ha leído ni escrito
1	Se ha escrito con éxito el buffer
2	Se ha leído con éxito el buffer
3	Se ha leído y se ha escrito con éxito

Con este retorno el dispositivo sabrá si aceptar o no los datos de su memoria que debieron ser escritos por el método del conector. Así se conforma plenamente el mecanismo de intercambio. Resumimos en la siguiente tabla las variables y métodos más relevantes en la clase Conector:

Clase Conector	
Variables	• Parámetros del conector (para cada extremo). incluye punteros a buffers, bit de escrito-leído, id de dispositivos, esid, tamaño de datos, etc. `plantilla_conector conector;`

Métodos	• Constructor. Al ser un dispositivo pasivo, sin hilo propio, no pide frecuencia de muestreo
	`Conector(int id);`
	• Registro. Se registra un dispositivo en el conector. Esta llamada se suele hacer desde conectable.
	`int registrar(Conectable *disp, int esid, BYTE *entrada, BYTE *salida, unsigned int lentrada, unsigned int lsalida);`
	• Datos del complementario. Este método es utilizado por los discretizadores.
	`int datos_complementario(int *esid, unsigned int *lentrada, unsigned int *lsalida);`
	• Intercambio de E/S. Es el modo operativo habitual del conector, en el que cada dispositivo entrega y recoge mensajes.
	`int entradasalida(Conectable *conectable, bool solo_lectura);`

Se incluye un método, similar al visto en **`Conectable`**, que es de registro o alta. En este caso, para inscribir como máximo dos dispositivos (uno por cada extremo), de ahí que se hayan definido vectores estáticos de dos elementos. Estos métodos son llamados por **`Conectable::conectar()`**.

Se considera fundamental que se haga una comprobación en las conexiones que se efectúan, por si el usuario se equivocara. Hay que tener en cuenta que una mala configuración del dispositivo provocaría un error de segmentación, así como una corrupción de los datos.

- **Comprobación de tamaño de datos**. **`lentrada`** en un extremo ha de ser igual que **`lsalida`** en otro, y viceversa. Con esto garantizamos que no se escribirá en zonas equivocadas de la memoria
- **Comprobación de esid**. Si tenemos dos estructuras diferentes con tamaños de entrada y salida iguales, el conector logra transmitir los datos, pero serán ilegibles ya que cada dispositivo interpreta los datos de forma diferente.
- **Comprobación de ejecución**. No se puede conectar cuando se están ejecutando los dispositivos que hay en los extremos.

Al ser el conector un dispositivo crucial en el tránsito de datos entre dispositivos, se le puede dar un uso para volcar estos en un archivo. Para tal propósito no modificaremos esta clase, sino que crearemos una clase derivada.

3.1.3. Hilos concurrentes en el conector. La exclusión mutua y las variables atómicas

Compartir datos es delicado cuando estamos utilizando hilos, pues nunca sabemos cuándo un hilo está escribiendo y otro leyendo. Es decir, los datos se pueden corromper y se puede llegar a comportamientos imprevistos, muy difíciles de detectar. La sincronización en estas operaciones es fundamental.

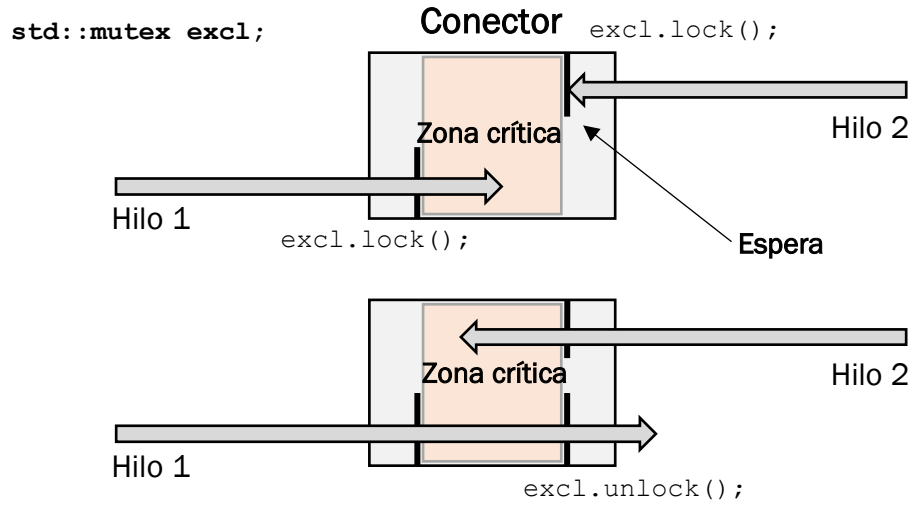


Figura 3.8

Como observamos en la figura, tenemos dos hilos que confluyen en un mismo método, el cual tiene una zona crítica susceptible al *data race*; o con instrucciones de un único proceso, como acceso crítico a periféricos hardware.

El hilo 1, al haber llegado primero, obtiene el uso exclusivo de los recursos gracias a la llamada **`excl.lock()`** que es un método de la variable **`std::mutex excl`**. Si, mientras está haciendo su tarea en la zona crítica, llega el hilo 2, éste también hará la llamada a **`excl.lock()`** pero lo encontrará ocupado, y por tanto, tendrá que esperar. Es decir, el mutex hace que el hilo 2 se quede pausado.

Una vez que el hilo 1 ha finalizado su tarea, hace una llamada a **`excl.unlock()`** y entonces el hilo 2 retoma su ejecución. Con esto conseguimos una adecuada sincronización de los hilos y evitamos el *data race*.

Existe otro mecanismo de concurrencia de hilos, mucho más transparente y rápido, al que daremos uso frecuentemente. En vez de restringir zonas de ejecución, restringe variables que pudieran ser compartidas, como los bits de estado de los dispositivos o algunas variables estadísticas. Son los datos **`atomic`**. Si queremos definir un dato **`std::atomic`** de tipo **`int`** inicializado a **`-3`** entonces escribiríamos:

```
std::atomic<int> dato {-3}; // Admisible a partir de C++11
```

Los datos declarados de esta forma pueden funcionar de forma normal como las demás variables, utilizando sus propias sobrecargas para los operadores `++`, `>=`, `==`, etc. Sin embargo, no pueden confluir dos hilos simultáneamente para lectura y escritura de ese dato. Uno de ellos ha de esperar (por lo general, el que llega después, mientras el otro está haciendo la operación de lectura/escritura.

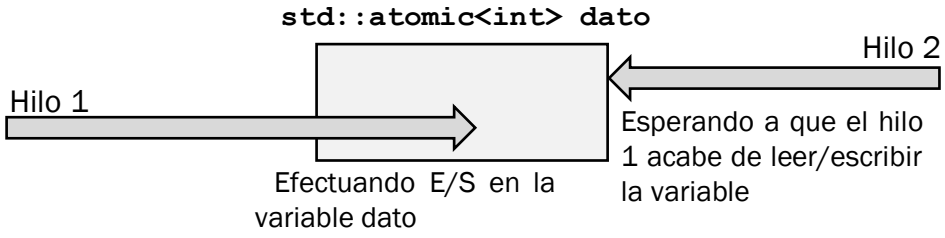


Figura 3.9

Existen otros elementos de sincronización interesantes como los semáforos, que utilizan numeración y conteo en su función de arbitraje.

3.2. Los dispositivos discretizadores

Estos dispositivos, en la práctica, toman "muestras" de los datos que entregan los dispositivos de planta, de una forma periódica. Los diferentes estados y variables de la planta son transmitidos a otro dispositivo, como podría ser un controlador. Sin embargo, aquí dan un paso más adelante y también ofrecerán información a su dispositivo asociado. Son, por tanto, bidireccionales.

Los discretizadores y los buses tienen una característica común: transmiten la información, desconociendo la estructura de ésta. Sólo manejarán, entradas, salidas y tamaño de los datos, en su formato binario. No necesitan saber más que la procedencia y el destino de los datos. Los llamaremos dispositivos intermediarios por ese papel que tienen. Son similares a los mensajeros.

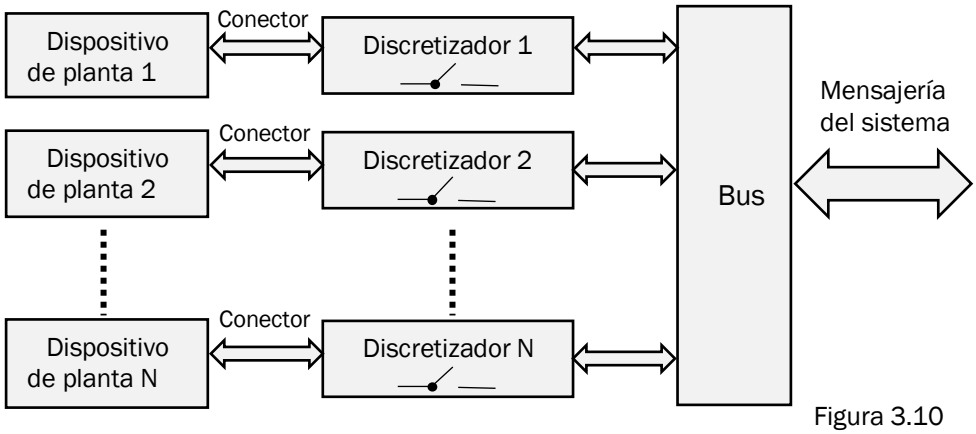


Figura 3.10

Ambos dispositivos estarán siempre asociados. De hecho, para crear un discretizador, necesitamos haber creado un bus previamente, y asociarlo.

El primer problema que se nos presenta es cómo utilizar el conector con el dispositivo con el que intercambiar datos. Recordemos que, al inscribirnos para el proceso conectable-conector, necesitamos conocer el esid y el tamaño de los datos de entrada/salida. Los dispositivos de planta y controladores lo tienen fácil porque tienen las mismas estructuras con las que trabajar. Las tienen en su zona de memoria y el tamaño se calcula con **`sizeof()`**.

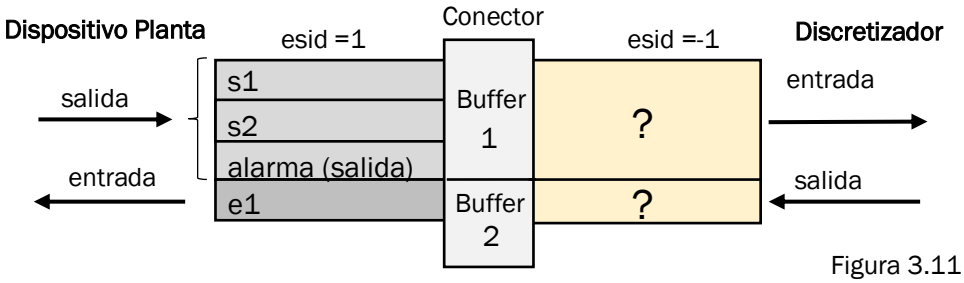


Figura 3.11

La solución propuesta es utilizar el conector para capturar los datos del otro extremo. Al iniciar el proceso, inscribimos el dispositivo de planta y éste proporcionará al conector los tamaños de datos de entrada y salida, más esid. El discretizador, después, llama al método **`datos_complementario()`** de la clase **`Conector`** para capturar esos parámetros e incorporarlos.

Mediante **`new`**, el discretizador habilita dinámicamente los buffers de memoria propios en los que almacenar el tránsito de datos, ya que a priori no sabe qué tamaño hubieran tenido. Finalmente, registra un puerto de E/S y lo conecta. Lo único que el discretizador no sabrá es cómo están estructurados internamente los bloques de datos de entrada y salida.

Clase Discretizador	

Variables	• Parámetros. Se transmiten como cabecera por el bus. Son los datos principales del discretizador como id, esid, tamaño de la cola, de los datos, frecuencia de muestreo, etc.
	`plantilla_datos_discretizador_sv parametros;`
	Buffers. Se reservarán de forma dinámica a partir de los tamaños de datos que proporcionó el conector
	`BYTE *al_bus, *al_dispositivo, *al_bus_c, *al_dispositivo_c;`

Métodos	• Constructor. Se necesita el puntero de bus, y proporcionar el tamaño de cola máximo que gestionará la mensajería
	`Discretizador(unsigned int t, int id, unsigned int lcola, Bus *bus);`
	• Crea el puerto, se asocia a un conector y crea los buffers necesarios
	`int asociar(Conector *conector);`
	• Rutina y actualizador
	`int actualiza();`
	`void rutina();`

Se ha visto conveniente incorporar un conteo para los mensajes transmitidos y por tanto una longitud máxima de cola o **`lcola`** que se utilizará en el bus.

Por lo general, se prefiere que la comunicación sea a tiempo real, sin apenas retardos, pero en la práctica no suele ser así. Con la longitud máxima de cola que especificamos y la frecuencia de muestreo del discretizador estamos indicando el retardo máximo R_{max} que podríamos tolerar en el sistema.

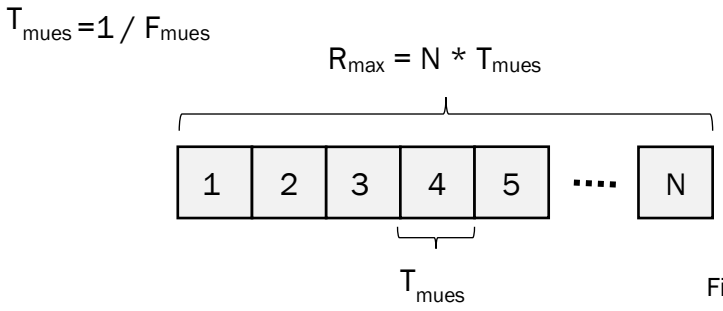


Figura 3.12

En esta sección de la comunicación, especialmente con la cola de mensajes del bus, es conveniente plantearse cómo funcionará el sistema con retrasos que podrían ocurrir utilizando una infraestructura externa como es la cola de mensajes. Según los ensayos efectuados, no suele haber problema, pero si no parametrizamos convenientemente las frecuencias de muestreo podría haber retrasos en la entrega de paquetes que comprometerían la estabilidad de los dispositivos de planta, cuando se cierra el bucle de control.

Una vez que el discretizador y el bus se conectan, comparten una estructura de datos fundamental en la E/S que servirá de cabecera en la transmisión por cola de mensajes, en **`plantilla_datos_discretizador_sv`**.

La nomenclatura que utilizaremos en las direcciones de transmisión cambia. Ahora nos referiremos a **`al_bus`** y **`al_dispositivo`** cuando queramos determinar una dirección, según este diagrama:

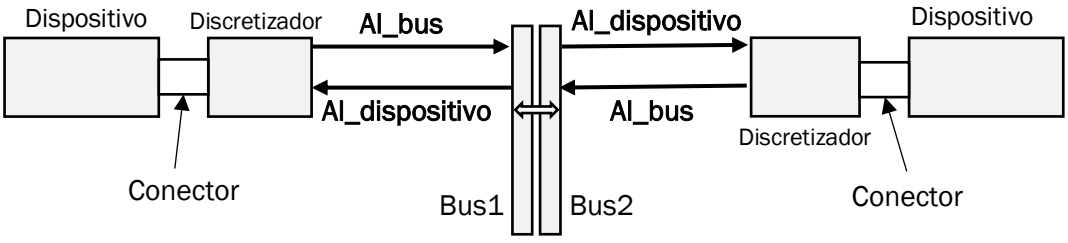


Figura 3.13

De esta forma, estemos en el discretizador-bus de la planta o en el discretizador-bus del controlador, no habrá lugar a equívocos con la notación del flujo de datos.

Como se ha visto, el discretizador es un elemento fundamental en la comunicación. Es por eso que, en caso de fallo de comunicación en el bus, por ejemplo, al no habilitarse un canal de mensajería, o por ser incompatible con el otro discretizador al otro lado del bus, habilitará su alarma, visible para el usuario, en la lista de dispositivos del menú interactivo.

3.3. Bus de comunicaciones. Cola de mensajes

Para completar la comunicación entre planta y controlador necesitamos una infraestructura que haga uso de la mensajería entre procesos. En muchos sistemas operativos existen modalidades de colas de mensajes, que podrían utilizarse para conectar el proceso que gestiona las plantas y el proceso que gestiona los controladores.

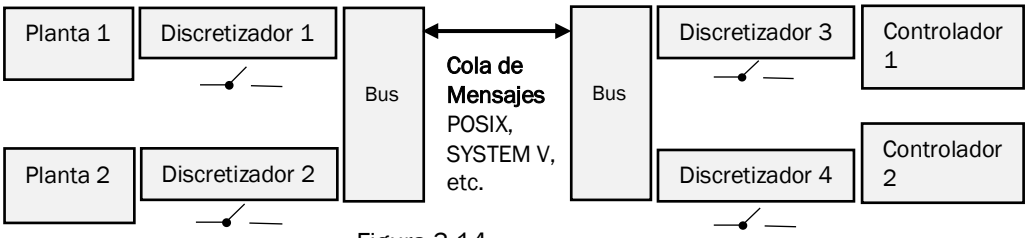


Figura 3.14

El ejemplo gráfico está simplificado a dos plantas/controladores, pero en realidad podemos comunicar infinidad de plantas y controladores utilizando el mismo par de buses y varias colas de mensajes. Lo flexibilizaremos por medio de vectores, como en otros dispositivos. Existen más formas de hacer comunicación entre procesos como las tuberías, pero éstas están más enfocadas al volcado masivo de datos y no tanto al envío de paquetes autónomos de información. Conviene recordar que los discretizadores han de estar sincronizados. No puede haber cuellos de botella ni información acumulada en la línea de transmisión de datos.

En este trabajo se ha escogido el sistema de mensajes de System V, que suele ser bastante popular en los entornos UNIX y que emplearemos en la implementación del programa. Describamos las características de la cola de mensajes System V:

- La identificación se hace mediante archivo y un número previamente seleccionado entre ambos puntos de comunicación, como puede ser la id de dispositivo de bus.
- La comunicación dispone de varios canales de colas. Es decir, entre los dos buses podremos comunicar una variedad de dispositivos. Esta comunicación de varios canales no tiene preferencias, como así podría ocurrir en otras colas implementadas por POSIX. Procuramos que sea una comunicación sencilla en el modo normal de operación, aunque es conveniente que se produzca un intercambio previo para determinar los parámetros de comunicación, tamaño del paquete de datos, etc.

El bus enlaza cada par de discretizadores cuyas id coincidan y, se encarga de proporcionar un canal de comunicación por cada discretizador registrado, que depende de su número de identificación y de su ubicación (en el lado de planta o en el lado de controlador). Este canal se vuelve de comunicación y recepción desde el momento en el que el discretizador que espera recibe la cabecera del discretizador que hay al otro lado.

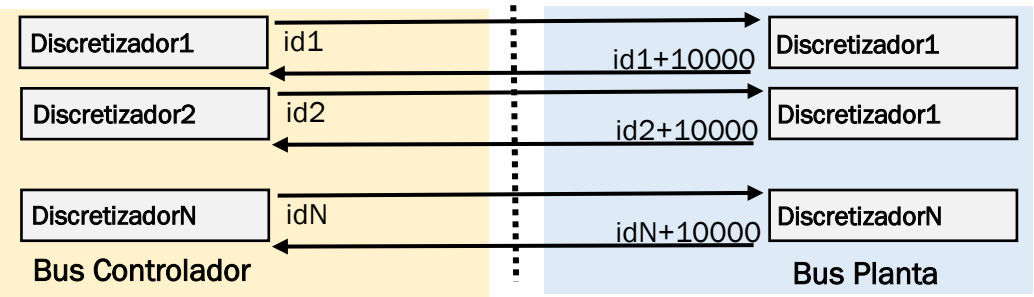


Figura 3.15

3.3.1. Etapas de la comunicación en el bus y operación

El intercambio de datos entre procesos independientes necesita utilizar un protocolo que garantice el inicio de las comunicaciones, su sincronización durante el intercambio de datos, y su finalización.

Etapa 1: Handshake e intercambio de datos de cabecera

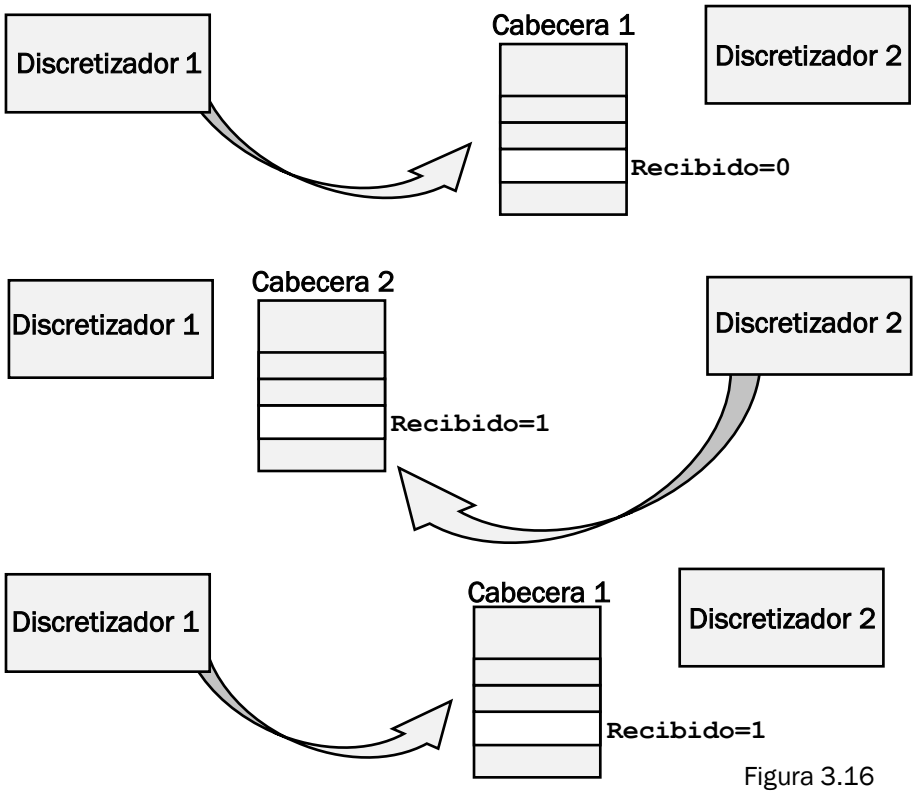


Figura 3.16

Para llegar a una primera comunicación, tanto los discretizadores del bus que esté en la planta como los del bus que esté en el controlador están obligados a emitir una primera cabecera de mensaje con **`recibido=0`**. Uno de los dos procesos, por lo general el primero en ejecutarse, es el que recibe la primera cabecera. Hay que tener en cuenta que en la cola puede haber paquetes corruptos o sin limpiar, por lo que las cabeceras tienen además una huella o *magic number* que hace reconocible su identificación.

El discretizador del proceso que recibe la primera cabecera, se queda con una copia y envía la suya con el bit de recibido como activado, enviándosela al otro proceso. Si el otro proceso recibe la cabecera, la copiará y volverá a enviar la suya propia, también con el bit de recibido activado.

De esta forma, los discretizadores asociados en cada canal conocen que lo siguiente que recibirán serán paquetes de datos.

Cada par de discretizadores han de realizar su *handshake* o intercambio de cabeceras. Con este intercambio se garantiza:

- Comprobar la compatibilidad de los paquetes de datos, a partir de sus esid, y de sus tamaños de entrada/salida. Si no fuesen compatibles, ambos discretizadores serán informados en el código de retorno del método de entrada/salida y se activarán sus alarmas.
- Leer otros datos de interés en la comunicación como son los tamaños de cola empleados y las frecuencias de muestreo. Si los tamaños máximos de colas difieren, se escogerá el tamaño más pequeño (se da preferencia al discretizador que necesita más "tiempo real" en su tamaño de cola). Si ambas frecuencias de muestreo difieren, está claro que la comunicación será asimétrica y que la frecuencia de muestreo más pequeña hará de cuello de botella; sin embargo, puede continuar la comunicación.

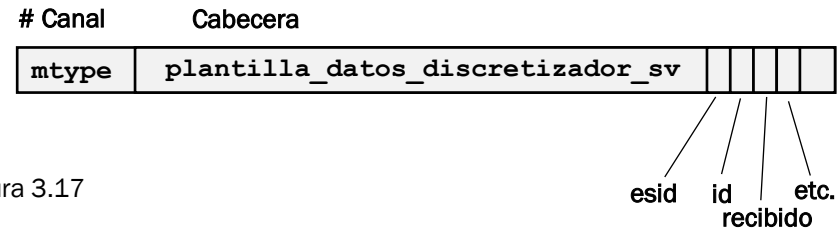


Figura 3.17

Se hará la comunicación de cola de mensajes habitual, con las mismas funciones que se utilizan en C. En el constructor del bus se habilitan las comunicaciones con **`ftok`** y **`msgget`**. Las id de ambos buses han de coincidir para ello. Después, se utiliza **`msgsnd`** y **`msgrcv`** para enviar y recibir mensajes. El formato estándar es el que se ha reflejado en la figura. La variable de tipo long mtype tiene el número de id del discretizador, y si el discretizador es de planta, se le suma su id+10000. De esa forma, se establece una comunicación full-duplex y de forma inequívoca, entre planta y controlador. Se recomienda que las id de los discretizadores nunca supere 10000 para evitar problemas.

Etapa 2: Operación habitual. Intercambio de datos.

Una vez que se han intercambiado las cabeceras, se ha determinado el tamaño máximo de cola y se han evitado incompatibilidades, los discretizadores envían/reciben mediante el bus los datos de sus dispositivos asociados. El bus sólo empezará a enviar datos cuando haya recibido la cabecera y sepa que el otro bus también recibió la cabecera.

Desde ese mismo instante, la estructura de envío por mensajería cambia totalmente y se muestra así:

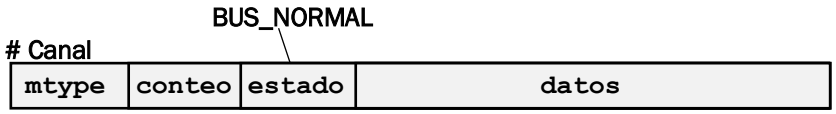


Figura 3.18

El estado indica que el bus está en su modo de operación habitual con el valor **BUS_NORMAL**. El conteo es lo que le queda de mensajes al emisor para dejar de emitir, si ya no recibe mensajes. Si recibe un mensaje, ese conteo aumenta en uno; si emite, desciende en uno. Es un mecanismo que evita saturar la cola de mensajes en caso de fallo súbito de alguno de los dos procesos. El límite superior de conteo es el tamaño máximo de la cola, acordada entre ambos discretizadores, tras el intercambio de cabeceras.

En todo momento, el método de entrada/salida del bus indicará al discretizador si se ha recibido, si se ha leído o si se ha leído/recibido, de la misma manera que lo hacían los conectores; y que de esta forma el discretizador decida si usar el conectable para leer/escribir, o solo para leer.

Etapa 3: Finalización.

En algún momento puede detenerse completamente la planta o el controlador, ya sea de forma voluntaria o por algún problema que hubiera. Se ha de garantizar que el otro bus sea informado, y por tanto los discretizadores que lo integran. La estructura enviada por mensaje es muy similar al funcionamiento normal. Simplemente se emite el estado **BUS_PARADA**.

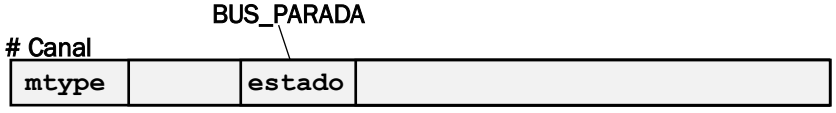


Figura 3.19

El resto de la estructura no se tiene en cuenta en la terminación. Los paquetes de datos de finalización son emitidos por el bus en su rutina de destructor e indican que el proceso finaliza. El discretizador que recibe esta cabecera borrará todos los datos del discretizador remoto y se pondrá en el estado inicial de handshake, emitiendo una cabecera, y a la espera de recibir otra.

Clase Bus	

Variables	• Parámetros y variables de operación, incluyendo cabeceras de los discretizadores al otro lado.
	`std::vector<plantilla_datos_discretizador_sv> parámetros, parametros_otro;`
	`std::vector<plantilla_operacion_bus> operacion;`
	• Identificación de cola. Coincidirán en los dos buses que se comuniquen.
	`int id_cola;`

Métodos	• Constructor. Se ha de proporcionar, además de la id, si el bus estará en el lado de los controladores o de la planta.

	`Bus(int id, tipo_proceso tipo);`
	• Registro de discretizador. Se proporcionan los punteros a sus buffers de trabajo entrada/salida, con espacio suficiente para albergar los paquetes de datos.
	`int registrar(Discretizador *discretizador, BYTE *al_bus_c, BYTE *al_dispositivo_c);`
	• Método de lectura y escritura, con el que los discretizadores realizan sus operaciones de comunicación.
	`int lecturaescritura(Discretizador *discretizador, bool solo_lectura, int *err_es);`

Al igual que los conectores, los buses también remiten información útil, sobre si se pudo leer, escribir o leer/escribir, de esta forma el discretizador evita enviar datos no actualizados por su conector, al dispositivo asociado, y se ahorra consumo innecesario de CPU.

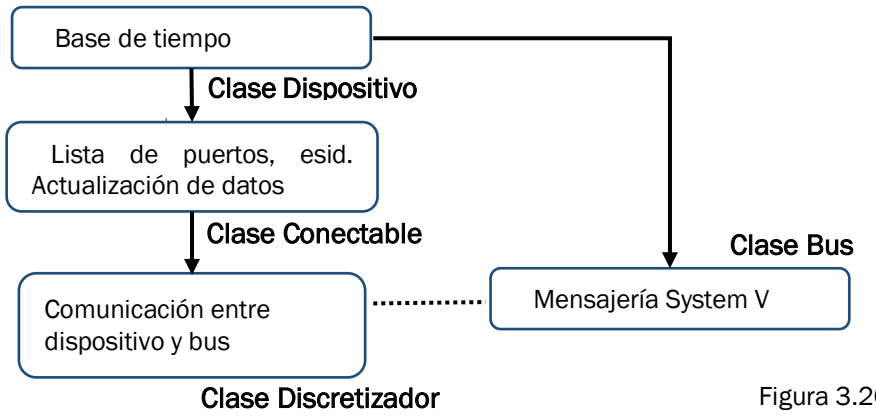


Figura 3.20

Se intenta garantizar una comunicación simétrica, que no es influida por el orden de arranque de los procesos, al ser ejecutados. Una vez finalizado un proceso, éste puede ser modificado, parametrizado y/o recompilado mientras el otro se queda a la espera, y entonces se vuelve a establecer la comunicación cuando se vuelve a ejecutar.

Los conteos de cola nos permiten fijar cual será el retardo máximo admisible, lo cual es muy importante en una comunicación de este tipo y que puede ser decisivo en el rendimiento de los controladores.

3.4. La cadena de actualizaciones

En casi todas las clases existe un método llamado **`actualiza()`**. Es un método básico que indica cómo se ha de comportar la clase durante su ciclo.

Junto con el control de ejecución de los hilos, estamos en una de las partes centrales del trabajo. Estos conceptos actúan de columna vertebral y es conveniente conocer su mecanismo de funcionamiento, pues nos será útil a la hora de realizar nuestros propios dispositivos o ampliar la funcionalidad.

Podemos imaginarnos **actualiza()** como una cadena de llamadas iterativas en la que cada clase llama a la actualización de su clase padre hasta llegar finalmente a **Dispositivo**, donde esta cadena alcanza su "cumbre" y donde existen los métodos más importantes del control de ejecución del hilo, comunes a todos los dispositivos.

El propósito de esta cadena es que cada clase realice su función principal, cada vez que se repite el ciclo de trabajo del dispositivo.

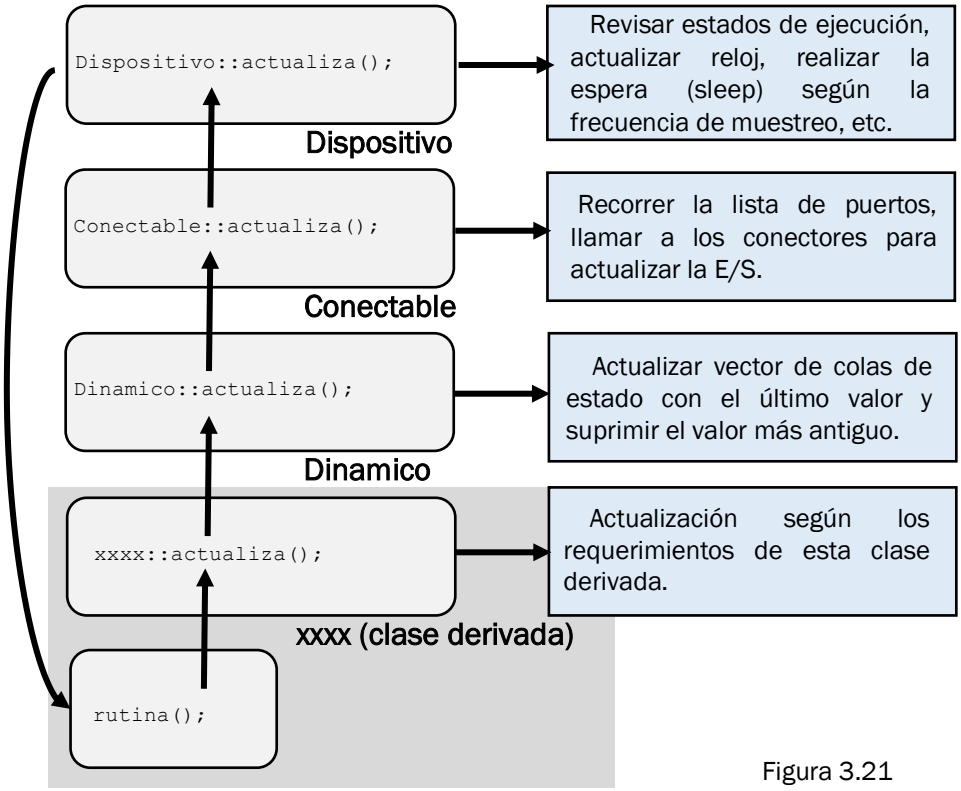


Figura 3.21

Si creamos nuestras propias clases ejecutables (o con derivaciones ejecutables), hemos de tener en cuenta llamar a la clase padre al actualizar, para que de esta forma sean efectivos los algoritmos de **Dinamico**, **Conectable**, **Dispositivo**.

Aunque el método de actualización está declarado como **virtual**, no se espera, de momento, que se utilice desde fuera o que tenga que estar sujeto al contexto del puntero del objeto que lo contiene. La cabecera de la cadena de actualizaciones es **rutina()** y una vez que la clase base **Dispositivo** ha terminado, se llama a la instrucción **return** de cada clase, volviendo el hilo de ejecución a la clase derivada que comenzó la cadena y repitiéndose el ciclo. En esta cadena, por supuesto, se pueden devolver números de error que serían útiles para la depuración del programa.

3.5. Ejecución de los hilos, rutinas y modos de operación

En una aplicación de esta índole es fundamental que el usuario tenga pleno control y pueda iniciar, pausar o interrumpir un dispositivo siempre que quiera. Recordemos que en la superclase **`Dispositivo`** habíamos declarado un método llamado **`virtual void rutina().`** Estamos ante un método polimórfico, y cada dispositivo que sea ejecutable ha de incorporarlo para que sea ejecutado cuando sea necesario. Este método va a ser la puerta de entrada y la puerta de salida del hilo. Mientras el hilo dure, realizará un ciclo constante de actualizaciones. Ésta es la rutina genérica de un hilo:

```
void Dispositivo::rutina() {

     while (Dispositivo::obtiene_ejecucion()) {
     Dispositivo::actualiza();
     }
     return;
}
```

Es decir, esto es un bucle de repetición **`while`** que depende del estado de la ejecución, proporcionado por el método **`obtiene_ejecución(),`** hasta que éste devuelve **`false`**. Para darle funcionalidad completa al flujo de ejecución hemos de declarar nuevas variables y características en la clase base **`Dispositivo`**.

Control de ejecución del hilo Dispositivo	
Variable	Descripción
`std::atomic<bool> ejecutable;`	Nos indica si el dispositivo es ejecutable true=sí ; false=no
`std::atomic<bool> ejecucion;`	Nos indica el estado de la ejecución true=ejecutándose; false=detenido
`std::atomic<bool> pausa;`	Nos indica si un dispositivo en ejecución está en un estado de pausada. true=pausado.
Método	Descripción
`int inicia();`	Inicia el dispositivo
`int pausa(bool pausar);`	Pausa o reanuda el dispositivo (true=pausar;false=reanudar)
`std::thread& acaba(int& error);`	Detiene el dispositivo (se proporciona referencia a la variable thread y así poder ejecutar join()
`void pausador(estados_pausador estado);`	Rutina privada del pausador. El argumento es un valor enumerado que puede ser RUTINA, BLOQUEO o DESBLOQUEO

La rutina de ejecución es polimórfica porque algunos dispositivos han de variar su funcionamiento. Por ejemplo, al detener al grabador de datos, éste vuelca todo lo recopilado a un archivo, tras el **`while`** y antes de acabar la rutina.

En el siguiente gráfico identificamos los puntos de entrada, salida y pausa del dispositivo, y los relacionamos con la cadena de actualizaciones.

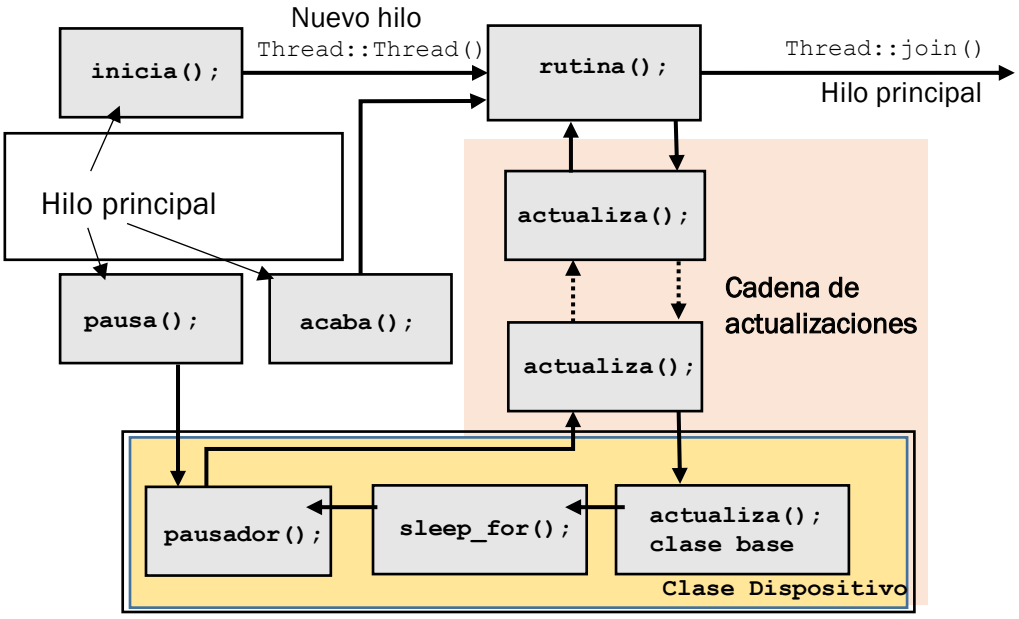


Figura 3.22

Por tanto, el hilo que desea actuar sobre la ejecución del dispositivo (llamado hilo principal) dispone de tres métodos en la clase **`Dispositivo`** que activarán/desactivarán los correspondientes bits y coordinarán la ejecución. Estos bits son **`std::atomic`** porque podría suceder que dos hilos o más concurren para tener el control, y esto podría causar *data race* y hacer que el sistema entre en estados de funcionamiento no deseados. Los datos de acceso atómico son adecuados para evitarlo.

Se hace imprescindible usar **`join()`** para que el hilo saliente se reunifique con el hilo que hizo la llamada de cierre, por eso devuelve la referencia al hilo.

La clase **`Dispositivo`**, además de la espera correspondiente por ciclo con **`sleep_for()`** también efectúa una comprobación de pausa que es parte del mecanismo que vamos a describir.

3.5.1. Mecanismo de pausa

No existe método específico en la clase **`std::thread`** que permita pausar un hilo. Es fácil iniciarlo, incorporando al constructor una función delta, como así hemos visto; y es fácil detenerlo con tan solo salir del bucle **`while`** de **`rutina()`** según el bit de ejecución, tras lo cual se reunifica con el hilo principal con **`join()`**; pero, ¿cómo pausar un hilo externamente, de forma indefinida?

Hemos visto que existen los objetos **`std::mutex`**, los cuales al llamar a su método **`lock()`**, impide que un hilo que concurre en el mismo método traspase esa instrucción, hasta que no se llama a **`unlock()`**. Este podría ser el

mecanismo más factible para pausar un hilo, hasta cuando queramos; y tan solo necesitamos diseñar el método apropiado.

La solución propuesta es un método de tres estados al que podrían acceder el hilo principal y el hilo del dispositivo, en cada ciclo. Los tres estados considerados son:

- **Bloqueo**. El hilo que entre con este estado, ejecutará un **`lock()`** y saldrá.
- **Desbloqueo**. El hilo que entre con este estado, ejecutará un **`unlock()`** y saldrá.
- **Rutina**. El hilo que entre con este estado, ejecutará un **`lock()`** y posteriormente un **`unlock()`**. Seguidamente, saldrá.

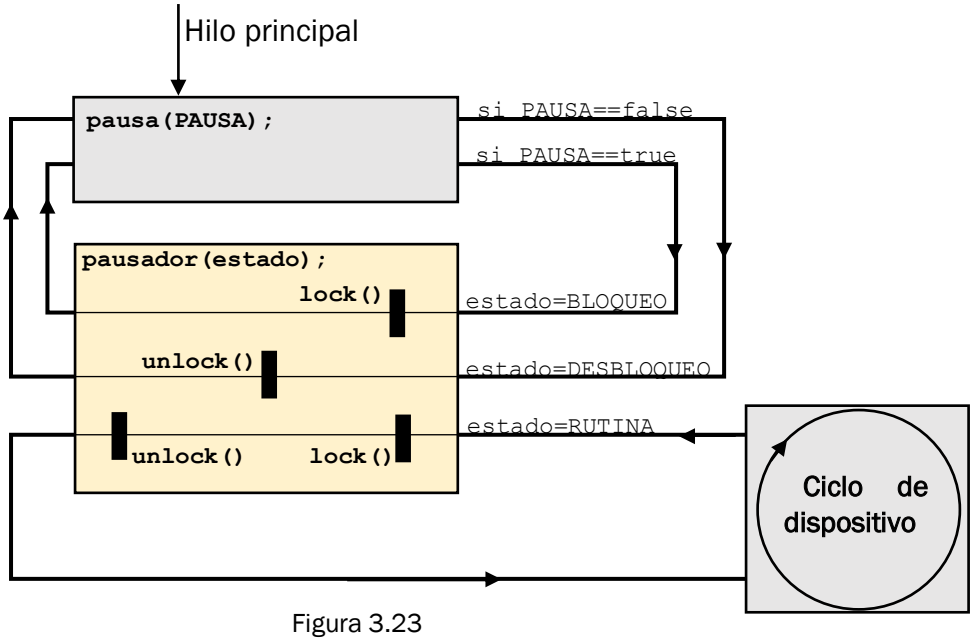


Figura 3.23

Si el hilo está ejecutándose y lo queremos pausar, llamamos al método pausa con **`pausa(true`**) y éste, a su vez, llama al método pausador con el estado **`BLOQUEO`**, ejecutando una instrucción **`lock()`**, para después salir. Cuando el ciclo del dispositivo entre en el método pausador con su estado habitual **`RUTINA`**, se parará en **`lock()`** y no podrá efectuar ninguna acción, a no ser que desde el hilo principal se llame a **`pausa(false)`**, con lo que entrará al pausador con **`DESBLOQUEO`**, ejecutando la instrucción **`unlock()`**. Entonces el hilo del dispositivo podrá continuar.

El funcionamiento normal del pausador es el hilo de dispositivo realizando un **`lock()`** y después un **`unlock()`**, con **`RUTINA`**.

Si, mientras el dispositivo está parado, se ordena su detención, el método **`acaba()`** llamará a **`pausa(false)`**, y una vez que el hilo del dispositivo se retome, cesará su ciclo, en cuanto haga la comprobación **`while`**, dentro de **`rutina()`**.

La pausa y la detención no difieren aparentemente porque el dispositivo continúa conservando su memoria. Lo podemos volver a iniciar desde el punto que se paró, exceptuando el grabador, que vuelca en la detención su contenido en un archivo. Sin embargo, internamente, al detener, el hilo desaparece. En pausa, el hilo sigue existiendo, aunque sin actividad. Se puede comprobar esta diferencia con los visores de procesos en Linux **`top`** y **`htop`**.

Otro mecanismo de pausa, habitual en todos los hilos de ejecución (pausa planificada, según la frecuencia de muestreo) está determinado por **`sleep_for()`**, como método de **`std::this_thread`**, el cual necesita una medida del tiempo, determinada por **`std::chrono`**.

3.5.2. Temporización y medida del tiempo

La programación multihilo, además de la sincronización, también tiene en consideración la medida del tiempo al dormir los hilos en tiempos planificados, necesarios, en nuestro caso, para que los dispositivos tengan un uso racional de los recursos; y en otros, por ejemplo, cuando queremos encender un aparato a determinada hora del día. Éste permanece dormido, a la espera.

`std::chrono` proporciona diversas funcionalidades. Nos fijaremos en dos:

- Punto temporal, que hace uso de uno de los relojes del sistema, o del reloj de alta resolución **`std::chrono::high_resolution::clock`**.
- Duración, que determinada por la plantilla **`std::clock::duration<>`** y que proporciona infinidad de conteos y resoluciones posibles. También se determina restando dos puntos temporales.

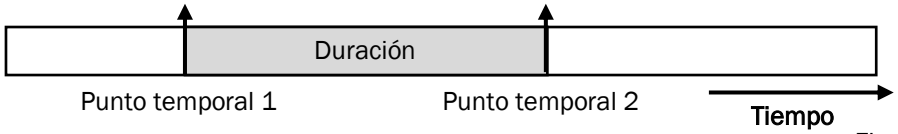


Figura 3.24

A partir de la frecuencia de muestreo se puede conocer el lapso teórico necesario para la espera del hilo. Su inversa, **`period`**, determinará la duración.

```
Dispositivo::period_m = std::chrono::duration<long int,
std::micro>((long int)(Dispositivo::period*1000000));
```

Hemos elegido una precisión de microsegundos, de ahí que se multiplique el período por un millón. Esta duración puede utilizarse para dormir el hilo ese tiempo, por cada período, haciendo uso de la instrucción **`std::this_thread::sleep_for(period_m - lapso)`**, siendo **`lapso`** la diferencia entre dos puntos temporales que determina el tiempo que se consume en cómputo, y que se ha de restar al período teórico.

Esta compensación **`(period_m - lapso)`** puede no funcionar si el reloj no es preciso o si se utiliza una máquina virtual. Es una característica fuertemente ligada al hardware.

3.6. Un ejemplo de conexión: Modelo de planta de un motor de corriente continua

Una vez que conocemos el flujo interno de datos, es momento de construir un ejemplo que nos permitirá combinar varios dispositivos de planta para hacer funcionar un motor expuesto a carga variable, como podría darse en la industria. De esta forma nos familiarizamos con los constructores de cada elemento y los métodos utilizados para interrelacionarlos.

El esquema de conexión es el siguiente (se indican números de los puertos):

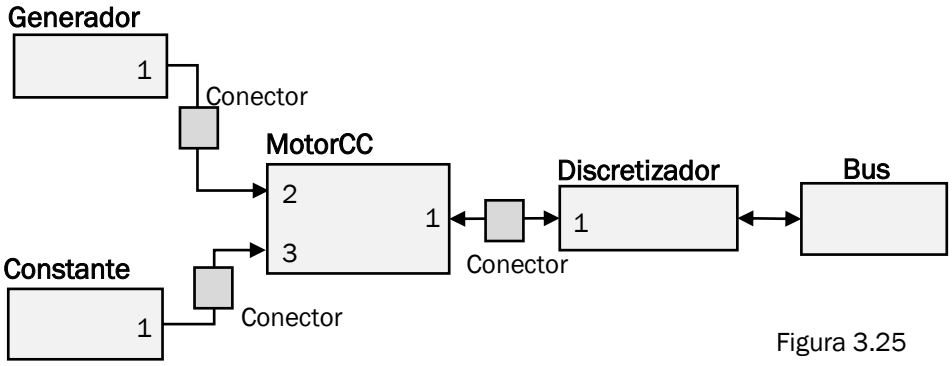


Figura 3.25

Motor CC (solo planta)	
Dispositivo	**Descripción**
Motor CC	Modelo físico de un motor de corriente continua, con momento de inercia propia y par externo variable. Se regula por voltaje.
Generador de funciones	Función cuadrada aplicada como par resistente
Generador de número constante	Voltaje proporcionado al motor. Se puede variar durante el funcionamiento y así crear escalones
Conector generador de funciones-motor	Conector para que ambos dispositivos se comuniquen
Conector generador de número constante-motor	Conector para que ambos dispositivos se comuniquen
Discretizador	Toma muestras del motor para transmitirlas al bus de comunicaciones
Conector motor-discretizador	Conector para que ambos dispositivos se comuniquen
Bus	Bus para comunicarse con otro proceso como podría ser un controlador.

El código se proporciona en *planta_motorcc_solo.cpp* y en la compilación con **`make`** se escribe como parámetro el objetivo **`planta_motorcc_solo`**, es decir, ejecutando **`make planta_motorcc_solo`**. En el anexo II hay más información.

Hay que tener en cuenta que, en el caso del motor, el puerto 3 enmascara la entrada del controlador del puerto 1. Al no haber controlador, no importa, pero hay que tenerlo en cuenta, según lo expuesto en el código fuente de motorcc.cpp. El puerto 2 del motor es el del par resistente que se opone al movimiento. Por tanto, seleccionamos el voltaje nosotros mismos (modificable en tiempo de ejecución), creamos un generador de onda cuadrada, parametrizamos el motor, y lo conectamos todo.

La creación de los objetos se hará mediante **new** y se trabajará con sus punteros llamando a los métodos con el operador **->**.

```
Constante *voltaje = new Constante(1,2001,100.,100,-100);

Periodica *par_resistente = new Periodica
                (10,2002,F_CUADRADA,200,200,0,400);

MotorCC *motor = new MotorCC
            (500,2003,10,0.1,0.3,220,30,1.9,0,0,0,0,0,20);
```

En el motor, además de los coeficientes referidos en su modelo, también especificamos sus condiciones iniciales y su corriente máxima de alarma. Como voltaje inicial, $100\,V$ y el par resistente es una señal cuadrada que se aplica frenando al motor en $400\,N \cdot m$, cada 20 segundos. El cálculo de los segundos de la señal es fácil, teniendo en cuenta que su frecuencia de muestreo es de $10\,Hz$ y el conteo ascendente/descendente es de 200. Con multiplicar el conteo por el período que es la inversa de la frecuencia, obtenemos los segundos.

Seguidamente creamos los conectores, el bus y el discretizador.

```
Conector *vol_mot = new Conector(6001); // De generador de voltaje
                                           a motor

Conector  *mot_dis  =  new  Conector(6002);  //Entre  motor  y
                                             discretizador

Conector *par_mot = new Conector(6003); // Entre par y motor

Bus *bus = new Bus(3001, PROCESO_PLANTA);

Discretizador *discret_mot = new Discretizador(50,4001,2, bus);
```

Hay que tener en cuenta que el constructor del discretizador necesita el puntero a bus, por eso lo creamos antes.

Realizamos las conexiones convencionales, entre dispositivos de planta. El método al que se llama es **registrar()**, y se halla en la clase **Conectable**. Los puertos ya han sido creados por los constructores de los dispositivos y tan solo hemos de especificar su número al hacer las conexiones, siguiendo el esquema anterior.

```
voltaje -> conectar(vol_mot,1); // voltaje-motor, extremo voltaje

motor -> conectar(vol_mot,3); // voltaje-motor, extremo del motor
```

```
par_resistente -> conectar(par_mot,1); // desde el generador de
                                          voltaje
motor -> conectar(par_mot,2); // desde el motor.
```

Como hemos visto en el apartado del discretizador, éste no conoce cómo van a ser estructurados los datos y necesita una asociación previa antes de conectarse en su único puerto que será el puerto 1.

```
motor -> conectar(mot_dis, 1); // Motor-discretizador
discret_mot -> asociar(mot_dis); // El discretizador capta los
                                    datos
discret_mot -> conectar(mot_dis, 1); // …y finalmente se conecta
```

Tan solo nos faltaría registrar estos elementos en el menú interactivo y configurar el grabador, para que podamos comprobar su funcionamiento. Esas clases de interacción y grabación las veremos más adelante:

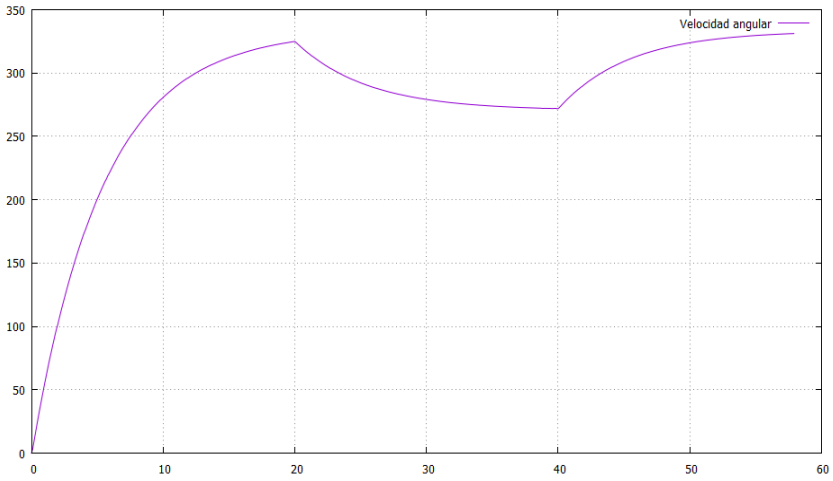


Figura 3.26

Se pueden elegir las id que queramos siempre que sean mayores que cero y menores de 10000. Cada dispositivo tendrá una id única, aunque, a efectos de comunicación, los buses de ambos procesos y los discretizadores asociados a uno y otro lado, deberán tener identificaciones iguales.

Para poner un poco de orden, es buena práctica que los agrupemos por el dígito de los millares. En todos los ensayos se ha utilizado esta numeración:

Numeración utilizada en id de dispositivos	
Dispositivo	**Numeración**
Controlador	1xxx
Dispositivo de planta o de proceso de señal	2xxx
Bus	3xxx
Discretizador	4xxx
Watchdog	5xxx
Conector	6xxx
Conector de recogida de datos	7xxx
Grabador	8xxx

Por supuesto, el usuario puede utilizar la que crea conveniente siempre y cuando los números id cumplan los requisitos descritos anteriormente.

4. Controladores

El control de un proceso es fundamental en la Automática. El objetivo de este capítulo es completar el lazo de control, aprovechando gran parte del trabajo realizado. Nos hemos hecho una idea de cómo funcionan las clases en la aplicación. La introducción del controlador tiene un lugar natural en la jerarquía de clases, puesto que estos dispositivos usan ecuaciones en diferencias y retardos. Por tanto, es incuestionable que tengan que derivarse de la clase **`Dinamico`**, el cual a su vez proporciona los métodos de un dispositivo conectable. Es decir, podremos conectar el controlador a otros controladores y también a los discretizadores que podrán comunicarse con el bus de forma bidireccional y así cerrar el ciclo planta-controlador.

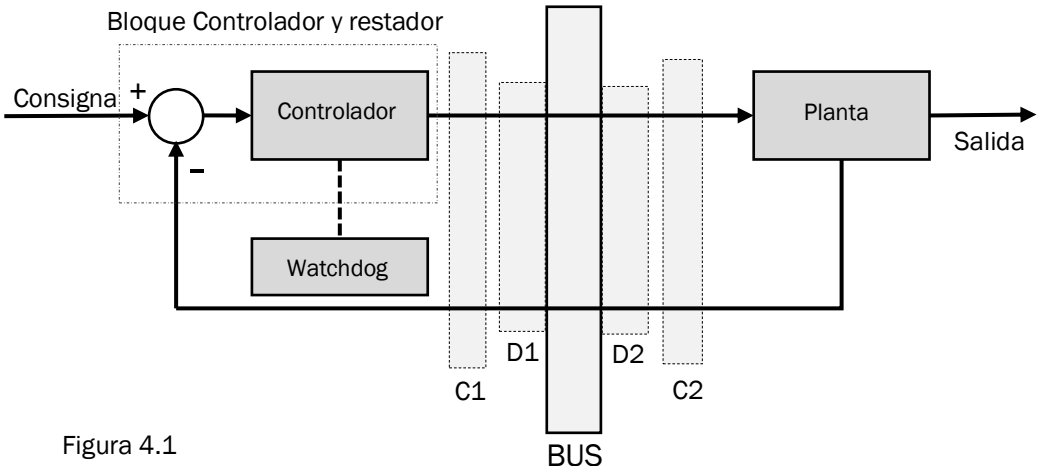


Figura 4.1

Para cerrar el bucle hacemos uso de D1 y D2, que son discretizadores de doble sentido. Los discretizadores se conectan con los dispositivos planta y controlador mediante los conectores C1 y C2. El bus actúa de comunicación interproceso (IPC) en el que se pueden establecer más lazos de control.

Se han integrado controlador y restador en el mismo bloque. Aunque dispongamos de un dispositivo matemático que es la combinación lineal, que se puede usar como restador $A - B = (1) \cdot A + (-1) \cdot B$ ve conveniente que la resta sea interna, en el propio dispositivo controlador.

En cuanto a diseño, mucho trabajo está hecho. Si nos ceñimos a controladores, éstas van a ser sus características más notables:

- **Usan modelos matemáticos diferentes**; de índole dinámica, como ocurre en dispositivos continuos, pero en otro dominio. Pasamos de las ecuaciones diferenciales de los dispositivos de planta a las ecuaciones en diferencias.
- **Tienen asociados watchdogs o perros guardianes**. La tarea de estos es de vigilancia, aviso y acción. A cada watchdog le podemos asociar uno o un conjunto de controladores. Cada controlador registra en el watchdog un

contador ascendente que es reseteado regularmente, cada vez que el controlador llama al método de refresco. Si al controlador le ocurriese algo (por ejemplo, un cuelgue), no llamaría al método, el contador llegaría al límite y se dispararía una alarma informando al usuario de que existe un problema.

Teniendo en cuenta estas dos características, ampliamos el programa e incluimos nuevas clases. En la jerarquía en la rama de dispositivos dinámicos añadimos **`Controlador`**.

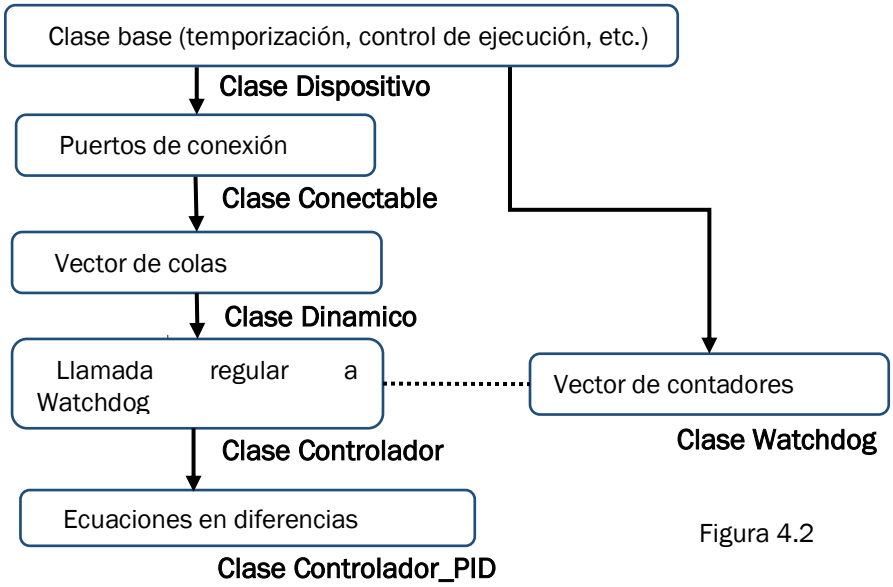


Figura 4.2

Por tanto, haremos uso de las características de los dispositivos, en general y de los dispositivos dinámicos (de retención de memoria), en particular. El presenta capítulo gira alrededor de estos dos importantes dispositivos: controladores y watchdogs.

4.1. Controladores

Los modelos con los que trabajaremos en los dispositivos cambian sustancialmente. En los dispositivos continuos de planta teníamos que conformar sistemas de ecuaciones diferenciales de coeficientes constantes (con alguna que otra excepción no lineal). En los controladores nos ayudaremos de la teoría de la transformada Z, y de su transformada inversa, la cual puede ser tratada en **`Dinamico`** como estados anteriores en sus vectores de cola.

4.1.1. Modelo matemático del controlador PID.

Cada variable de estado pasa a ser una sucesión numérica $\{x_k\}_{k\in\mathbb{N}}$ en la que cada elemento en el tiempo k se expresa funcionalmente como $x(k)$, $k \in \mathbb{N}$.

La transformada Z unilateral $\mathcal{Z}: x(k) \rightarrow [x(k)]$ se define como:

$$\mathcal{Z}[x(k)] = \sum_{k=0}^{\infty} x(k) z^{-k} \qquad (4.1)$$

La transformada de cualquier función $e(k)$ es denominada con su letra mayúscula $\mathcal{Z}[e(k)] = E(z)$. Como así ocurre en las transformadas de Laplace, también se puede definir la función de transferencia $G(z)$ entre una entrada $E(z)$ y una salida $U(z)$, de tal forma que $U(z) = G(z) \cdot E(z)$. (4.2)

PID se refiere a proporcional-integral-derivativo, determinado por tres coeficientes que llamaremos K_p, K_i y K_d. Es decir, el controlador realizará estas operaciones en la señal suministrada:

Proporcional. Es la operación más sencilla y hace uso de la propiedad lineal de la transformada Z. Si $u_p(k) = K_P \cdot e_p(k)$ entonces $U_p(z) = K_P \cdot E_p(z)$. (4.3)

Integral. Con la integral retenemos estados anteriores y, de alguna forma, los sumamos al estado actual. Aquí introducimos la variable tiempo t que es la variable de integración y que depende del período T de muestreo del controlador $t = kT$.

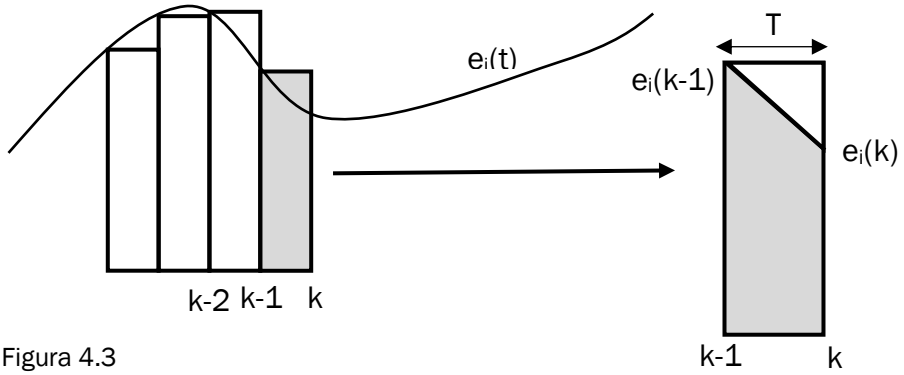


Figura 4.3

$$u_i(t) = K_i \int_0^t e_i(t)dt \tag{4.4}$$

Aproximando por integración trapezoidal, nos queda que:

$$u_i(k) = u_i(k-1) + K_i \cdot T \cdot \frac{e_i(k) + e_i(k-1)}{2} \tag{4.5}$$

Hacemos la transformación Z, sabiendo que:

$$\begin{gathered} \mathcal{Z}[e_i(k)] = E_i(z) \quad ; \quad \mathcal{Z}[u_i(k)] = U_i(z) \\ \mathcal{Z}[e_i(k-1)] = z^{-1}E_i(z) \quad ; \quad \mathcal{Z}[u_i(k-1)] = z^{-1}U_i(z) \end{gathered} \tag{4.6}$$

Por tanto, haciendo también uso de las propiedades lineales de la transformada tenemos:

$$U_i(z) = z^{-1}U_i(z) + K_i \cdot T \cdot \frac{E_i(z) + z^{-1}E_i(z)}{2} \tag{4.7}$$

Tan solo nos queda separar los términos para obtener la función de transferencia discreta de la integral.

$$U_i(z) = K_i \frac{T}{2} \cdot \frac{1 + z^{-1}}{1 - z^{-1}} E_i(z) \tag{4.8}$$

O lo que es lo mismo:

$$U_i(z) = K_i \frac{T}{2} \cdot \frac{z + 1}{z - 1} E_i(z) \tag{4.9}$$

Derivativa. Es la operación de diferenciación en el ámbito discreto.

$$u_d(t) = K_d \cdot [e_i(t)]' \tag{4.10}$$

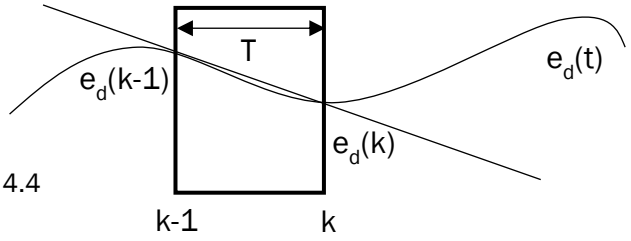


Figura 4.4

Por tanto, la pendiente es:

$$[e_i(k)]' = \frac{e_i(k) - e_i(k-1)}{T} \tag{4.11}$$

Tomamos transformadas Z:

$$U_d(z) = K_d \cdot \frac{1}{T} [E_i(z) - z^{-1} E_i(z)] \tag{4.12}$$

La función de transferencia es, en este caso:

$$U_d(z) = \frac{K_d}{T} \cdot \frac{z - 1}{z} E_i(z) \tag{4.13}$$

Sumando las tres operaciones **ya tenemos el modelo PID**.

$$\frac{U(z)}{E(z)} = K_P + K_i \cdot \frac{T}{2} \cdot \frac{z + 1}{z - 1} + K_d \cdot \frac{1}{T} \cdot \frac{z - 1}{z} \tag{4.14}$$

Tomando el denominador común en las fracciones y separando coeficientes según las potencias de z llegamos a la expresión compacta que nos proporcionará la transformada inversa de una forma directa.

$$\frac{U(z)}{E(z)} = \frac{az^2 + bz + c}{z^2 - z} = \frac{a + bz^{-1} + cz^{-2}}{1 - z^{-1}} \tag{4.15}$$

Los coeficientes a, b, c dependen de los parámetros:

$$a = K_P + K_i \frac{T}{2} + \frac{K_d}{T} \tag{4.16}$$

$$b = -K_P + K_i\frac{T}{2} - \frac{2K_d}{T}$$

$$c = \frac{K_d}{T}$$

La ecuación en diferencias se calcula fácilmente:

$$U(z)(1 - z^{-1}) = E(z)(a + bz^{-1} + cz^{-2})$$
$$u(k) = u(k-1) + ae(k) + be(k-1) + ce(k-2) \quad (4.17)$$

Por tanto, hemos de hacer uso de la clase **`Dinamico`**, con un vector de dos colas, de tamaño 2 en la cola de la variable $e(k)$ y de tamaño 1 en la cola de $u(k)$.

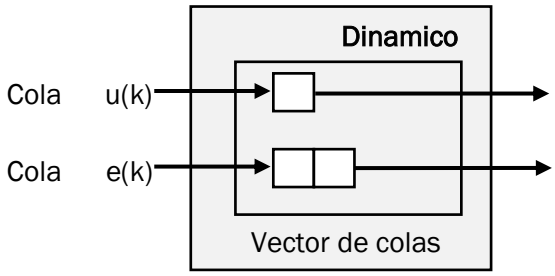


Figura 4.5

Podemos realizar muchos más sistemas discretos y filtros con infinidad de ecuaciones en diferencias. Hemos de conformar una clase base de controladores que proporcione los servicios básicos necesarios.

4.1.2. La clase controlador. Mensajería asíncrona.

Los controladores tienen en común que derivan de la clase **`Dinamico`** y que disponen de un sistema de vigilancia en el que, en cada ciclo, han de llamar a una función para constatar que siguen funcionando adecuadamente. Esta implementación necesita, por tanto, de un watchdog al que llamar, y que ha de estar ya disponible en el mismo constructor.

Aunque la clase, por sí misma, es sencilla, tiene unas implementaciones que la hacen interesante.

- **Método de detención forzada**. El método **`acaba()`** es el habitual para detener el hilo, pero se crea un método llamado **`acaba_forzado()`**, que en caso de mal funcionamiento, imposibilita un posterior inicio. El objeto deja de ser ejecutable.
- **Utilización de mensajería asíncrona** con los objetos **`std::promise`** y **`std::future`** para informar al objeto al que se delega la ejecución del hilo que éste ha acabado y así no hacerle esperar.

Tales características facilitan la gestión del controlador cuando éste comienza a funcionar mal, por ejemplo, al aplicar métodos de cálculo iterativos diseñados inadecuadamente, o datos desde la planta que ponen al dispositivo en estados

no previstos y que le conducen a un mal funcionamiento.

Llegados a tal punto, el controlador no puede operar con normalidad y ha de ser finalizado de forma automática, si así se acuerda previamente, para no perjudicar al resto de los controladores (demandando excesivo uso de la CPU). La finalización no voluntaria conlleva una responsabilidad para el watchdog, que se ha de hacer cargo de la ejecución y de su correcta finalización.

Para tal propósito, el controlador contribuye de su parte, y si la rutina del hilo (con el bucle y la llamada a la cadena de actualizaciones) era genérica, hemos de modificarla añadiendo estas instrucciones antes de **`while`**:

```
Controlador::estado_ejecucion = std::promise<long unsigned int>();
Controlador::futuro_estado_ejecucion = estado_ejecucion.get_future();
Controlador::estado_ejecucion.set_value_at_thread_exit(CODIGO_FIN_HILO);
```

Aquí ponemos en marcha el mecanismo asíncrono, con **`std::promise estado_ejecucion`**, que recibe mensajes del hilo del controlador, y un objeto **`std::future futuro_estado_ejecucion`** que los recibirá. Ambos son declarados en el controlador y han de ser enlazados con **`get_future()`**, pero el objeto **`std::future`** se lo podemos proporcionar al hilo que deseemos, que en nuestro caso particular será para el watchdog. El método **`set_value_at_thread_exit(CODIGO_FIN_HILO)`** indica que el objeto **`std::future`** tendrá el código de fin de hilo cuando **`rutina()`** acabe.

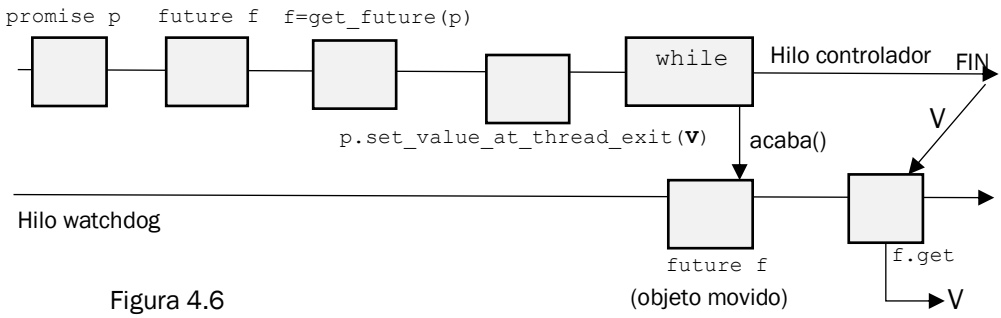


Figura 4.6

Es un método válido y alternativo que sustituye a las variables **`std::atomic`**, inmunes a cualquier *data race* por concurrencia de hilos. Estos estados se pasan por mensaje. Pero es más que una mensajería entre hilos. Al ser mensajería asíncrona actúa a modo de memoria, con un emisor que "hace la promesa" y un receptor que "la lee en un futuro". Se pueden traspasar todo tipo de datos, desde tipos básicos (**`int`**, **`char`**, **`bool`**, etc.) hasta estructuras.

Cada vez que el hilo se reinicia, hemos de destruir el objeto **`std::promise`** anterior. Aquí nos limitamos a reasignar con un nuevo constructor **`Controlador::estado_ejecucion = std::promise<long unsigned int>()`** puesto que el anterior hilo fue ya destruido, por lo que el destructor del antiguo **`std::promise`** será ejecutado al hacer la reasignación, evitando la excepción *promise already satisfied*, en el momento de obtener otro objeto **`std::future`**

con **get_future()**, ya que sólo se puede enlazar uno y **std::promise** aún lo retiente. Ha de ser eliminado con esa reasignación.

Clase Controlador	

Variables	• Variables de mensajería asíncrona promesa-futuro
	`std::promise<unsigned long int> estado_ejecucion;`
	`std::future<unsigned long int>`
	`futuro_estado_ejecucion;`
	• Watchdog asociado al controlador
	`Watchdog<T> *watchdog`

Métodos	• Constructores. Hay dos, dependiendo de si se quiere iniciar la clase Dinamico o no.
	`Controlador(unsigned int t, int id, const std::vector<unsigned int>& v, Watchdog<T> *watchdog);`
	`Controlador(unsigned int t, int id, Watchdog<T> *watchdog);`
	• Obtención de la referencia del objeto future para comunicación asíncrona
	`std::future<unsigned long int>& obtiene_estado();`
	• Detención forzada. Una vez detenido el dispositivo pierde su condición de ejecutable
	`std::thread& acaba_forzado(int& error); // Detención forzada`
	• Métodos propios de dispositivo ejecutable.
	`int actualiza();`
	`void rutina();`

En **actualiza()** nos aseguramos de hacer la llamada regular, por ciclo, al watchdog que se proporcionó al construir el controlador. Es una forma de decirle "que todo funciona bien".

```
if(Controlador<T>::watchdog != nullptr)
    Controlador<T>::watchdog -> sigo_operativo(this);
```

Si, por alguna razón no se hacen estas llamadas regulares, se da por hecho que el controlador está funcionando mal.

La implementación del modelo matemático del apartado anterior se realiza en la clase derivada Control_PID.

Clase Control_PID	

Variables	• Variables p,i,d y coeficientes a,b,c
	`T p,i,d,a,b,c;`
	• Booleano de modificación, que indica que el usuario modificó el objeto en propiedades()
	`std::atomic<bool> modificado;`
	• Señales de entrada y salida (entrada,x; salida, y)
	`T x,y;`

Métodos	• Método de cálculo.
	`T pid();`
	• Métodos de clase ejecutable
	`int actualiza();`
	`void rutina();`
	• Propiedades
	`void propiedades();`

4.2. Watchdogs

Muchos sistemas informáticos y microcontroladores utilizan frecuentemente estos dispositivos.

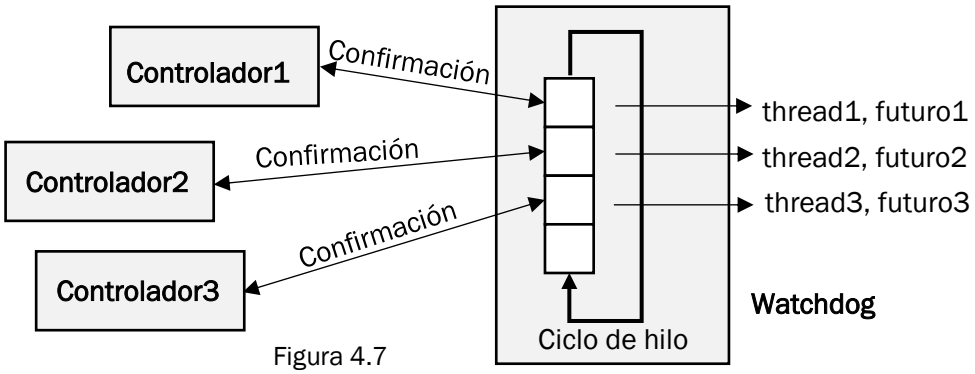


Figura 4.7

El dispositivo watchdog tiene un hilo propio, con una frecuencia de muestreo. Es un dispositivo que se limita a inspeccionar, cada cierto, tiempo el vector de contadores de todos los controladores que se han inscrito, en un ciclo constante. Cada controlador tiene su conteo máximo predefinido que se indica en el registro.

Si un controlador no llama a su rutina de verificación de funcionamiento en el watchdog, no reinicia el contador (no lo pone a cero) y el watchdog seguirá incrementando el conteo hasta llegar al límite máximo en el que:

- El watchdog pasa a estar en condición de alarma, visible en la lista de dispositivos, cuando hagamos uso del menú interactivo.
- Marca con un bit de alarma el controlador que falló, el cual será mostrado si el usuario accede a propiedades.
- El controlador, si así se quiso, es detenido automáticamente.

Es muy fácil comprender el funcionamiento básico del dispositivo. Se puede comprobar que funciona correctamente inscribiendo unos cuantos controladores y pausándolos cierto tiempo. Sus contadores llegarán al límite y se activará la alarma, llegando, en ocasiones, a cerrarse el controlador pausado.

Sin embargo, en cuanto a control de hilos, vamos a dar un paso adelante. Al estar los controladores bajo el control del watchdog, tendremos la posibilidad

de detenerlos y poder liberar sus recursos, sin intervención de la rutina principal.

4.2.1. Trasferencia del control del hilo y cierre.

Los hilos **std::thread** no se pueden copiar, sólo se pueden mover. Es decir, sólo pueden tener un manejador que por defecto pertenece a la rutina principal, que es la que llama a **inicia()** y, por tanto, es la que crea el hilo, recurriendo a la función lambda, como se ha visto.

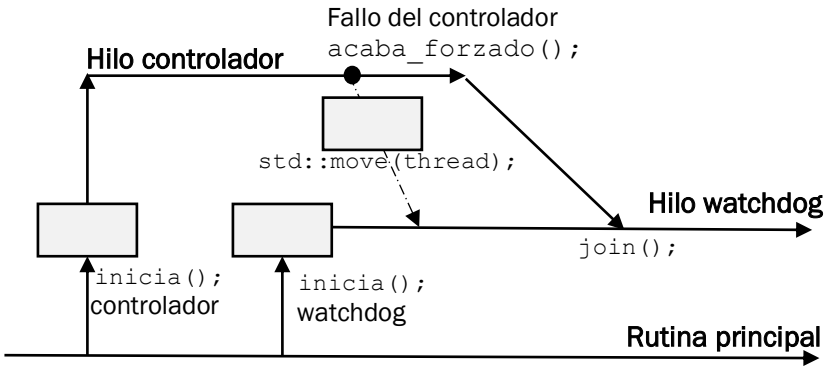


Figura 4.8

La referencia al objeto **std::thread** del controlador es argumento de retorno en el método **acaba()**, pero utilizar dicha referencia no es suficiente. Sería suficiente si hubiéramos invocado **acaba()** desde la rutina principal. En este caso hemos de transferir su titularidad, moverlo. Igualmente, también hemos de mover el elemento **std::future** que nos indicará cuándo ha finalizado el hilo completamente y así poder llamar a **join()** con la seguridad de que no provocará una excesiva demora, puesto que el watchdog podría estar prestando servicio a otros controladores.

Esta instrucción garantiza que todos los recursos del hilo quedan liberados. Si no ejecutásemos **join()** tras finalizar la rutina del dispositivo, al destruirlo (ya sea borrando el objeto que lo alberga con **delete**, o al salir de ámbito) habrá una excepción. Como el watchdog no posee el hilo del controlador, ya que éste fue iniciado en la rutina principal, hemos de hacer la transferencia. De lo contrario, no funcionará **join()** cuando lo ejecute, haciendo uso de la referencia.

Es decir, una vez que tenemos la referencia al hilo **std::thread& thread**, proporcionada por **acaba()**, usaremos **std::move**, ejecutándolo con el hilo del watchdog. En nuestro caso va a parar a un vector de estructuras de controladores a los que hay que detener porque están funcionando mal. Si tenemos en el watchdog una variable **std::thread thread_2**, entonces la instrucción de transferencia es:

```
std::thread thread_2 = std::move(thread1);
```

La transferencia de **std::future** (que no es copiable) es exactamente igual.

Mostramos las variables y métodos que conforman **`Watchdog`**:

Clase Watchdog	

Variables	• Estructuras con los datos de los controladores, punteros conteo, status hilos y futures de controladores a detener
	`std::vector<plantilla_controlador> controladores`
	`std::vector<plantilla_detencion> hilos_pendientes;`

Métodos	• Constructor del watchdog
	`Watchdog(unsigned int t, int id);`
	• Registro de un nuevo controlador a la lista. Se especifica límite de conteo y bit de detención
	`int registrar(Controlador<T> *controlador, unsigned int multiplo, bool detencion);`
	• Rutinas de dispositivo ejecutable
	`int actualiza();`
	`void rutina(); // Rutina del thread`
	• Método "checkpoint" al que han de llamar los dispositivos adscritos de forma regular
	`void sigo_operativo(Controlador<T> *controlador);`
	• Revisar disponibilidad del mensaje future
	`int futuro_preparado(std::future<long unsigned int> const& futuro);`

4.2.2. Gestión de excepciones

El sistema de excepciones no sólo informa al usuario de que algo va mal. Son rutinas esenciales encaminadas a restablecen el normal funcionamiento del programa si así se desea, según las excepciones que se encuentran, ya que muchas de estas se pueden retomar sin necesidad de que se interrumpa el programa, permitiendo reajustar sobre la marcha.

El manejo de vectores y buffers de memoria nos expone a la habitual excepción *segmentation fault* cuando nos equivocamos al utilizar vectores o zonas de memoria; pero con controladores, watchdogs y gestión de hilos de ejecución pueden ocurrir más excepciones que interrumpen el programa de forma abrupta:

- Excepción por intentar leer un **`std::future`** sin que éste haya sido inicializado desde **`std::promise`** con **`get_future()`**. No se dará el caso, porque usamos dicha llamada en cada inicio de rutina. Sin embargo, afrontaremos un caso de excepción, por descuido al plantear la rutina del dispositivo.
- Excepción por intentar conectar dos **`std::future`** con el mismo **`std::promise`**, haciendo uso de **`get_future()`**. La reasignación con nuevo constructor que hicimos en Controlador evita el problema.
- Excepción por intentar escribir dos valores **`std::promise`** consecutivos sin que hayan sido leidos por un **`std::future`**. No se dará el caso aquí porque sólo asignamos un valor por cada ejecución de rutina del controlador.

- Excepción por intentar destruir un hilo que no fue reunificado con **join()**. Lo intentamos evitar con watchdog, al detener el hilo, y evitando que el controlador dañado se vuelva a ejecutar.

Las clases de Controlador necesitan inicializar los **std::future**, pero podría ocurrir que quien está realizando sus propios controladores, con sus propias rutinas tenga el descuido de no invocar **get_future()**.

De no hacerse así, y como el watchdog no realiza verificación con **valid()**, éste probará a revisar si hay un valor con **wait_for()** y entonces saltará la excepción, al no existir objeto **std::future** asociado. Ocurre en el método **futuro_preparado()** y ahí es donde vamos a crear nuestro manejador de excepciones. Se utiliza **try** (intento) y **catch** (captura).

```
int resultado;

try {

      resultado = futuro.wait_for(std::chrono::seconds(0)) ==
         std::future_status::ready ? 1 : 0;

} // Zona de código bajo prueba de excepción


catch (std::future_error& e){  // Rutina manejadora

       std::cerr << "\n Se ha interceptado una excepción con std::future.
Mensaje: " << e.what() << "\n";

       resultado = -1; // Código de error por haber excepción
```

Observemos que cuando queremos esperar el valor válido con **wait_for()** indicamos que queremos esperar cero segundos. Si no está el valor disponible, no esperamos; al igual que no queríamos esperar en la cola de mensajes al utilizar **msgrcv()**.

Cuando no hay manejador de excepciones, el programa termina de forma abrupta, pero si colocamos este manejador -que actuará en caso de descuido del programador- informará al usuario de que algo ha ido mal y además se podrá retomar el control del programa, evitando su interrupción.

El manejador, al devolver un código de retorno especial en caso de excepción, se lo indicará a la rutina que lo llamó y ésta cerrará el hilo del controlador dañado, sin esperar a la señal de cierre de hilo, pues sabe que si invoca **get()** volverá a ocurrir otra excepción. Hemos gestionado la excepción convenientemente.

Es aconsejable colocar manejadores de excepciones try-catch similares, más si hay partes críticas que no deberían ser interrumpidas. Por ejemplo, imaginemos que watchdog por alguna razón falla además de los controladores y no concluye el hilo en la orden de cierre. Entonces no podrá reunificar los controladores dañados con **join()**, y en la salida de programa, donde se destruyen los objetos, los hilos no reunificados mostrarán la excepción. No

sería perjudicial para la funcionalidad del proceso, pero es una situación que se tiene que evitar, en todo lo posible.

Las excepciones, lejos de ser eventos graves que interrumpen el programa, son mecanismos avanzados que permiten retomar el programa, corregir el fallo e informar; siempre y cuando las interceptemos en las partes del código susceptibles a fallos. Otra herramienta para determinar el origen de los fallos es la herramienta de depuración **`gdb`**, que trataremos en el anexo II.

Si queremos probar el mecanismo de excepciones tan solo tenemos que comentar la línea **`estado_ejecucion.get_future()`** en los controladores utilizados por el proceso de control y comprobar qué ocurre cuando los pausamos hasta dispararse el watchdog.

4.3. Lazo de control del motor de corriente continua

Llegados a este punto ya disponemos de todas los dispositivos y herramientas para cerrar el bucle de control. Aunque aún quedan aspectos como la interacción y la captación/registro de las señales de los dispositivos, tenemos formalmente un bucle de control operativo. En el anterior capítulo habíamos modelado e implementado un motor de corriente continua al que le podíamos someter a diferentes voltajes para comprobar su funcionamiento; en transitorio y estacionario. Vamos a completar el esquema con un proceso de control donde incluimos bus de control, discretizador de control, controlador PID del motor, watchdog y dispositivo generador de números constantes, entregando la consigna de funcionamiento.

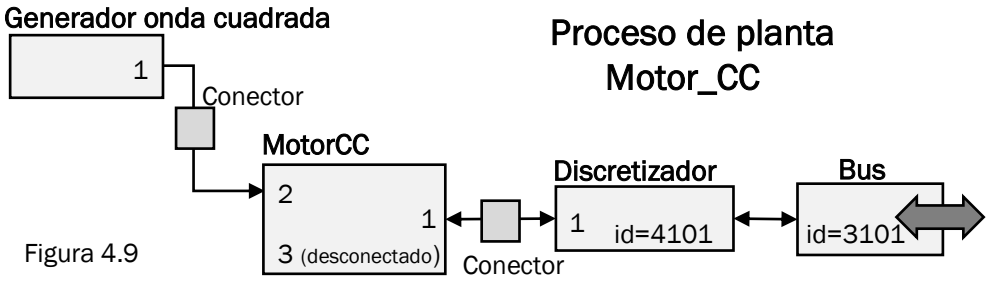


Figura 4.9

Al desconectar el puerto 3 del dispositivo motor CC en la planta (voltaje desde una fuente externa) garantizamos que éste leerá en exclusiva el puerto 1 que le comunica con el proceso de control, siendo entregado el voltaje desde el controlador, con la finalidad de regular automáticamente su régimen de giro.

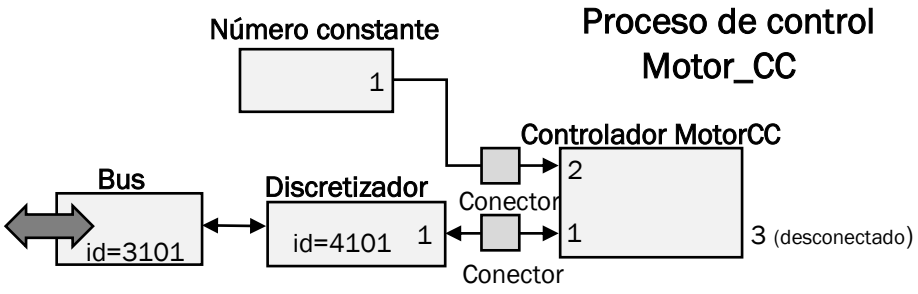


Figura 4.10

Ambos buses y discretizadores asociados se conectarán automáticamente por tener sus id iguales, en el momento en el que los dos procesos estén ejecutándose simultáneamente. El controlador del motor CC tiene internamente esta conexión:

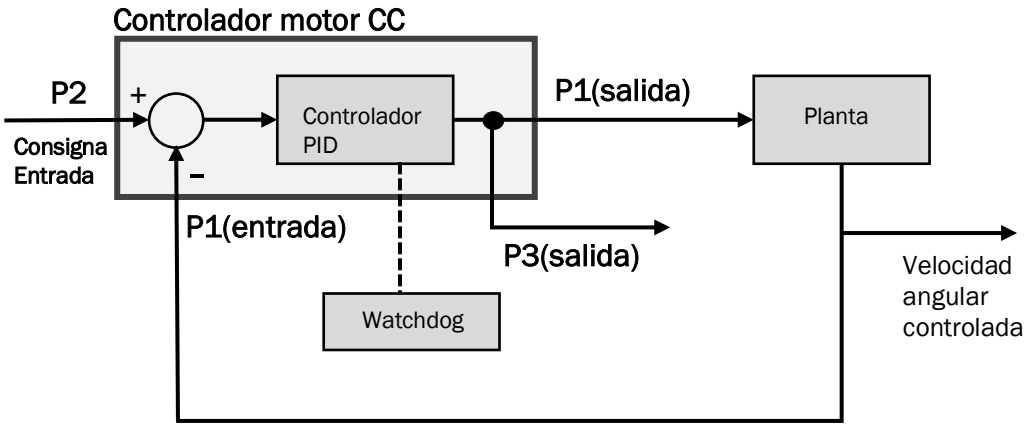


Figura 4.11

- El puerto 1 (P1) es la conexión principal del controlador, de entrada/salida, y que se conectará con la planta por medio de los discretizadores y los buses.
- El puerto 2 (P2) es la entrada de la consigna o valor de referencia del controlador.
- El puerto 3 (P3) es una réplica de la salida del puerto 1 y sirve para que se pueda conectar este controlador con otro, utilizando configuración en cascada y lazos de control más complejos. Aquí lo tendremos desconectado.

Internamente, el controlador incluye el restador, para simplificar el esquema de conexiones, ya que por lo general se hace uso de dicho restador y tan solo ocupa una instrucción en el código. Por supuesto, no puede faltar el watchdog, que se asociará al controlador sin necesidad de conectores, mediante registro.

La implementación planta-control aquí descrita está en los archivos planta_motorcc.cpp y control_motorcc.cpp, que generarán los dos procesos ejecutables. Los podemos construir con **`make planta_motorcc`** y **`make control_motorcc`**.

Las conexiones de planta las conocemos y aquí tan solo nos aseguraremos de tener el puerto 3 del modelo físico del motor desconectado para delegar todo el control al puerto 1. Nos centraremos en las instrucciones del proceso de control.

Creamos el objeto watchdog:

```
Watchdog<float> *watchdog_control_motorCC = new Watchdog
  <float>(10,5101);
```

Además de su id (5101) establecemos su frecuencia de muestreo. Aquí es de $10\ Hz$. Es decir, realiza su ciclo de comprobación y conteo cada décima de segundo. También hay que percatarse que el tipo numérico del controlador asociado va a ser **float**.

La declaración del objeto controlador es:

```
Control_motorCC *control_motorCC = new Control_motorCC(50,1101,
watchdog_control_motorCC,1,1.5,0, false);
```

La frecuencia de muestreo es de $50\ Hz$. Se ha de pasar al constructor el watchdog y los tres parámetros PID iniciales, que posteriormente podremos cambiar sobre la marcha al hacer los ensayos.

La siguiente instrucción es importante. Aquí es donde registramos el controlador en el watchdog. Recordemos que en el mismo watchdog podemos registrar más controladores.

```
watchdog_control_motorCC -> registrar(control_motorCC, 20, true);
```

Hemos establecido un conteo de 20 y **true** significa que, si hubiese un problema, el watchdog toma el control del hilo y lo cierra. Teniendo en cuenta que el watchdog tiene frecuencia de muestreo de $10\ Hz$ y que el conteo máximo (llamado también múltiplo) es 20, entonces se ve fácilmente que si transcurren 2 segundos sin ninguna llamada del controlador al método **sigo_operativo()**, se supondrá que éste funciona mal y saltará la alarma.

El resto de dispositivos que completan el proceso de control ya los conocemos (generador de números constantes, conectores, discretizador y bus).

```
Constante *consigna = new Constante(1,2101,0,-1000,1000);

Bus *bus = new Bus(3101, PROCESO_CONTROLADOR);

Conector *control_dis = new Conector(6101); // Entre controlador
y discretizador

Conector *consig_control = new Conector(6102); // Entre consigna
y controlador
```

Hay que observar que ahora el bus está configurado como **PROCESO_CONTROLADOR**, en contraposición al **PROCESO_PLANTA** que hemos definido en el bus de **planta_motorcc.cpp**.

Las conexiones se realizan también exactamente igual que en la planta. Los discretizadores también necesitarán "capturar" los datos del puerto del dispositivo a los que se asocian.

```
control_motorCC -> conectar(control_dis,1); // Discret-control

discret_mot -> asociar(control_dis); // Captura datos de puerto

discret_mot -> conectar(control_dis, 1);
```

```
consigna -> conectar(consig_control, 1); // Consigna-control
control_motorCC -> conectar(consig_control, 2);
```

Tan solo queda compilar, ejecutar ambos procesos de planta y controlador; y a su vez iniciar la ejecución de los hilos de cada dispositivo activo con el método `inicia()`. Los conectores y el bus, por supuesto, no es necesario que se inicien.

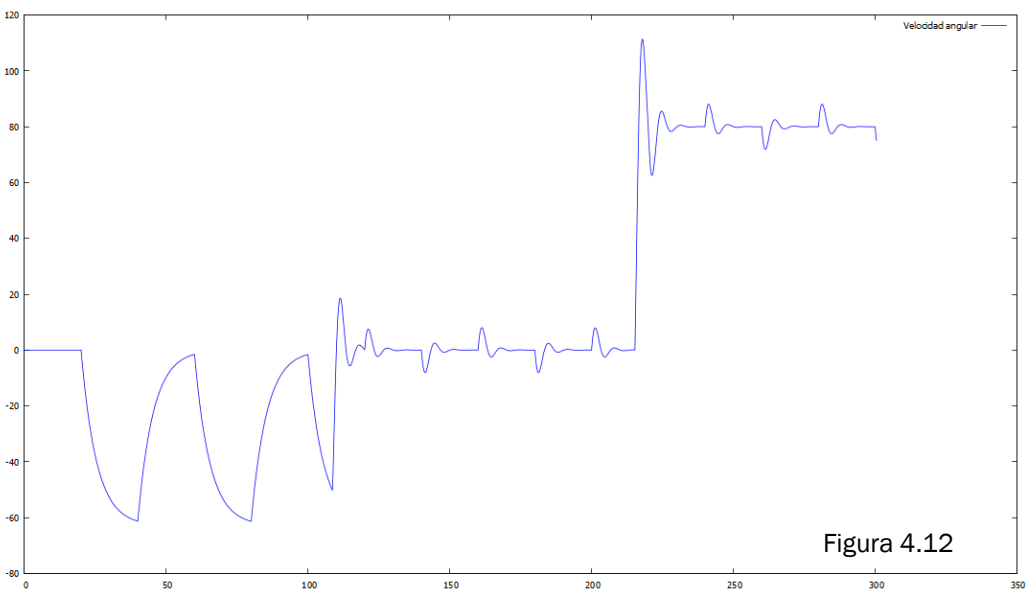


Figura 4.12

El procedimiento seguido para obtener este comportamiento ha sido:

- Exponer inicialmente el modelo del motor, con el controlador apagado, a un par en contra, conformando una onda cuadrada, de período 40 segundos, gracias al generador de onda que hemos conectado en el puerto 2 del dispositivo físico de planta. El par oscila entre $0\ N \cdot m$ y $400\ N \cdot m$. Al principio, el motor está a merced de ese par, mostrando únicamente su inercia, ya que no tiene voltaje aplicado, por eso la velocidad angular del giro es negativa. Después, al cesar el par, en la mitad del ciclo, se para el motor porque éste tiene rozamiento y actúa el circuito eléctrico también como freno (se considera la entrada de voltaje como un cortocircuito).
- A los 110 segundos activamos el controlador con consigna $0\ rads/s$ de velocidad. Parámetros PID: $P = 1, I = 1.5$ y $D = 0$ (no hay control diferencial). El motor tiende a estabilizarse en el punto de velocidad angular nula y resiste activamente el par con pequeñas variaciones de velocidad angular transitorias.
- A los 220 segundos hay un escalón en la velocidad porque cambiamos la consigna del controlador a $80\ rads/s$. Se puede observar la transición con una velocidad angular que rebasa puntualmente los $100\ rads/s$.

En definitiva, hemos conseguido unir los dos procesos (planta y controlador) para cerrar el lazo de control y que estos establezcan una comunicación.

4.4. Escalabilidad

El concepto de escalabilidad ha sido aplicado desde el principio. Con la facilidad del uso de vectores, colas, arrays, etc. gracias a la biblioteca STL, los conectables permiten tener muchos puertos en los dispositivos, los buses permiten tener muchos pares de discretizadores; y a la vez, cada planta y controlador permiten varios buses operando. Se pueden declarar varios motores, amotiguadores, etc. en el mismo proceso; y toda clase de dispositivos gracias a la facilidad con la que se crean los objetos a partir de las clases en C++.

Los watchdogs han sido diseñados para dar soporte a varios controladores. En el capítulo siguiente veremos que un conector de recogida puede tener múltiples sondas, y que un grabador puede tener múltiples conectores de recogida asociados.

Gracias a la programación orientada a objetos, que permite la construcción inmediata de dispositivos de toda índole, y al fácil manejo de estructuras complejas y la gestión de hilos, somos capaces de diseñar procesos de planta y de controladores interconectados, dispuestos en cualquier topología. El empleo de este lenguaje de programación multiparadigma en la ingeniería de sistemas puede ser muy interesante.

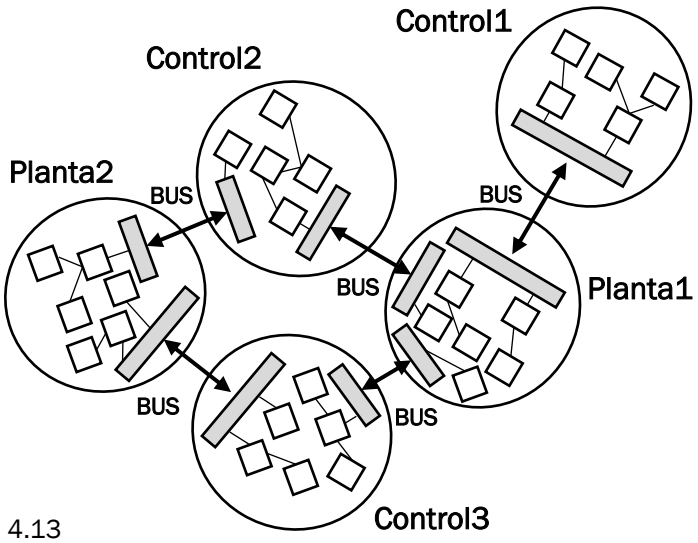


Figura 4.13

También se permite que coexistan dispositivos de planta y de control en el mismo proceso.

Como ejemplo de escalabilidad, integraremos las plantas de amortiguador y motor en un mismo proceso de planta. Y con los controladores haremos lo mismo en el proceso de control.

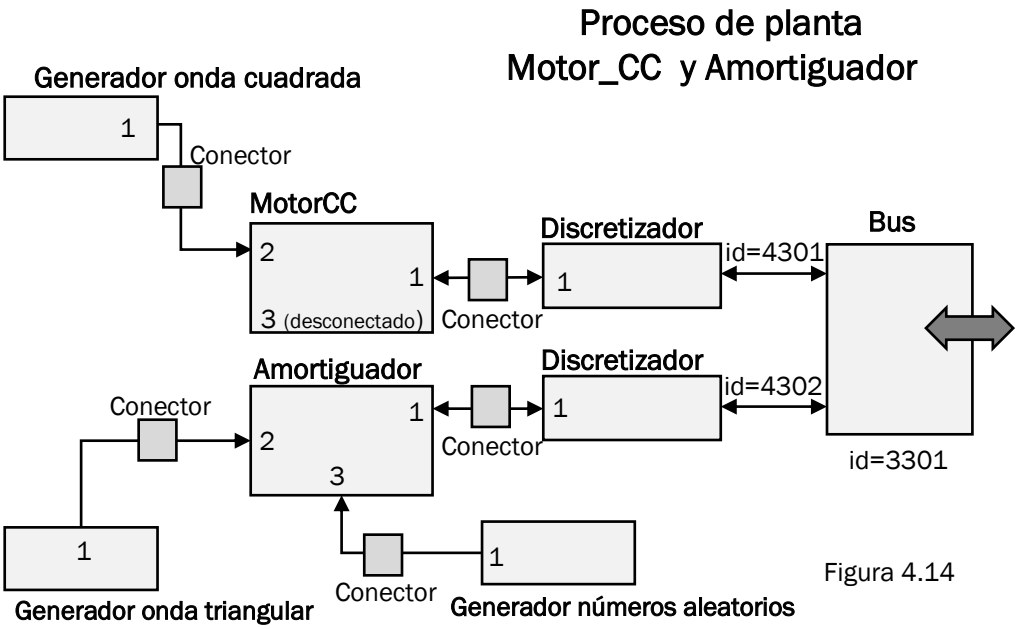


Figura 4.14

El amortiguador usa el puerto principal 1 que transmite posición de chasis, rueda y alarma, recibiendo la fuerza del actuador, desde el controlador. El puerto 2 recibe el perfil de la calzada por la que trascurre la rueda y por el puerto 3 (sin enmascarar a ningún otro) una señal aleatoria que se suma a la fuerza del actuador internamente, ya que estos dispositivos no suelen ser muy precisos. Se procura hacer la simulación más realista gracias a ese componente estocástico.

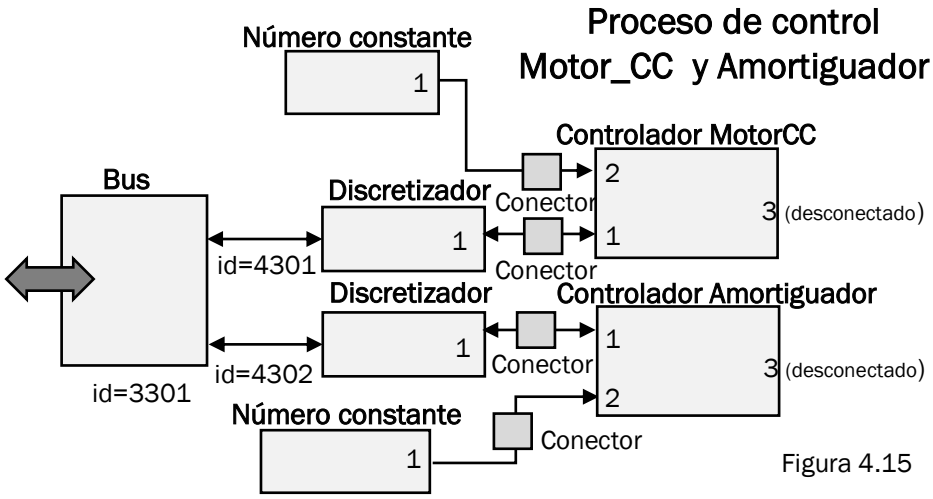


Figura 4.15

Internamente, el controlador del amortiguador también tiene el restador que cierra el lazo, y una salida en el puerto 3 que es copia exacta de la salida por el puerto 1. No entraremos en detalles de cómo se conecta toda la disposición, pues en el modelo físico del amortiguador los pasos de conexión con sus dispositivos generadores es el mismo que hemos seguido con el motor, con la salvedad de que son dos los generadores asociados:

```
 Aleatorio *perturbacion_actuador = new Aleatorio
(30,2201,419,577,129,899,1002,-18,18);

 Periodica *calzada = new Periodica
(100,2202,F_TRIANGULAR,300,300,-0.4,0.4);

 Amortiguador *amortiguador = new Amortiguador
(500,2203,250,20,15000,150000,3150,0.,0.,0.,0.,1,9500,10000);
```

Los dos discretizadores, ahora, los asociamos al mismo bus en sus constructores:

```
 Bus *bus = new Bus(3301, PROCESO_PLANTA);

 Discretizador *discret_mot = new Discretizador (50,4301,2, bus);

 Discretizador *discret_amo = new Discretizador (50,4302,2, bus);
```

En el proceso de control, las conexiones son similares a las que vimos con el bucle de control del motor, aunque ahora tenemos dos bucles y dos discretizadores, también conectados con el mismo bus. Utilizaremos un solo watchdog para ambos controladores:

```
 Watchdog<float> *watchdog_control = new Watchdog<float> (10,5301);

 Control_motorCC *control_motorCC = new Control_motorCC (50,1301,
watchdog_control,1,1.5,0, false);

 Control_amortiguador *control_amort = new Control_amortiguador
(50,1302, watchdog_control,19190,20320,0);

 watchdog_control -> registrar(control_motorCC, 20, true);

 watchdog_control -> registrar(control_amort, 20, true);
```

Obsérvese que se utiliza también como límite 2 segundos en el funcionamiento de los controladores para no ser interrumpidos por el watchdog. También podemos utilizar un watchdog por controlador.

Los archivos planta_motamo.cpp y control_motamo.cpp tienen todas las instrucciones necesarias para realizar este ensayo. Se compilan con **`make planta_motamo control_motamo`**. Se utilizan los conectores **`Conector`**, que ya conocemos y hemos descrito. Sin embargo, se han preparado unos conectores derivados que llamaremos **`Conector_rec`**, estos últimos se comportan igual que los que hemos visto, pero permitirán grabar datos mediante sondas, para analizar el sistema y su comportamiento, en el dominio del tiempo. Por defecto están comentados en el código, y será fácil activarlos. En el siguiente capítulo, describiremos su funcionamiento.

5. Interactuación con el usuario y grabación de datos

Una aplicación de estas características no puede ser "hermética". Cuando programamos un controlador, es interesante conocer su comportamiento en diferentes condiciones, visualizar sus variables de entrada y salida para poder analizarlas. Igualmente, si ensayamos un modelo simulado de planta, es conveniente que sepamos si es el adecuado, si sus valores están dentro de unos márgenes, si se comporta según lo esperado, etc.

Es decir, por nuestros procesos planta y controlador trascurre mucha información interna que hemos de retener y visualizar.

Por otra parte, es conveniente tener la posibilidad de encender, apagar, pausar dispositivos y reconfigurarlos mientras el programa funciona. Los ensayos han sido posibles gracias a este soporte que vamos a explicar. La clase **`Interactivo`**, que alberga el menú, los métodos polimórficos de propiedades; así como las clases **`Grabador`** y **`Conector_rec`** satisfacen las características requeridas.

5.1. La clase Interactivo

Durante este trabajo hemos ido derivando clases a partir de **`Dispositivo`**. Esta clase que gestionará el menú es independiente. No necesitamos un hilo autónomo, ni una frecuencia de muestreo, ni poder activar alarmas. La nueva clase **`Interactivo`**, que no deriva de ninguna otra, actúa inicialmente como un contenedor de punteros a dispositivos en el que se llaman a los métodos de los objetos ahí contenidos, según la elección del usuario en el menú.

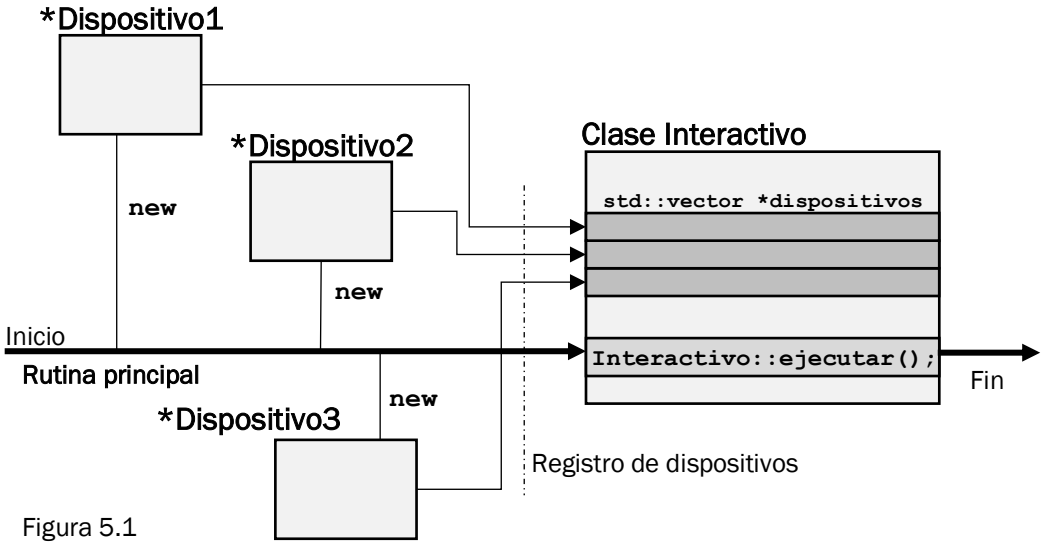


Figura 5.1

Gracias al uso de vectores podremos realizar la misma operación en todos los dispositivos. Con solo introducir una orden, por ejemplo, se inician, o se apagan en su totalidad.

Los objetos de dispositivo son creados de forma dinámica con el operador **`new`** y después de pasan los punteros a **`Interactivo`**. Esta es la razón de por qué creábamos las instancias de esta manera, con punteros y memoria dinámica. Una vez que el menú tiene toda la colección de punteros, puede hacer uso de características potentes como el polimorfismo. No necesita saber qué tipo de dispositivo está llamando para realizar las operaciones necesarias.

5.1.1. Menú del primer nivel (general)

Es el menú general. El que se inicia al arrancar el proceso, configurar los dispositivos, registrarlos en el menú e iniciarlo con **`ejecutar()`**. Se ha procurado tener una vista lo más completa posible de los estados de los dispositivos, sus tiempos de ejecución, pausa, alarma, identificación y etiqueta.

```
MENU GENERAL ---- LISTA DE DISPOSITIVOS ----E1=Ejecutable E2=Ejecutandose PA=Pausa AL=ALARMA

Opción  ID Muestreo Hz          E1 E2 PA AL     Tiempo seg.     Nombre

  1)    2102    10              SI SI NO NO     2.9     Par_resistente
  2)    2103    500             SI SI NO NO     2.5     Motor
  3)    7102    0               NO NO NO NO     0       conector par_mot
  4)    7101    0               NO NO NO NO     0       conector mot_dis
  5)    4101    50              SI SI NO NO     2.86    Discretizador motor
  6)    3101    0               NO NO NO NO     0       Bus
  7)    8101    10              SI SI NO NO     2.9     Grabador


  8) Detener todos los dispositivos
  9) Pausar todos los dispositivos
  10) Actualizar lista

  0) Salir del programa

Seleccione opción (puede ser un dispositivo o un comando general ->
```

Figura 5.2

Los dispositivos pasivos como el bus y los conectores no son ejecutables y, por tanto, tienen muestreo de $0\,Hz$. Para hacer más identificables los dispositivos, se les asocia una etiqueta alfanumérica al registrarles: su nombre.

Ya tenemos algunos elementos a tener en cuenta para conformar la clase del menú interactivo:

- Vector de punteros de dispositivos, al que iremos añadiendo más elementos, como estado de cada dispositivo, haciendo uso de los métodos base de Dispositivo como **`obtiene_ejecucion()`**, **`obtiene_pausa()`**, etc.
- Vector de etiquetas **`std::string`**. Es un vector de cadenas de texto en las que identificamos cada dispositivo con una etiqueta para que sean fáciles de distinguir en el menú. El manejo de las cadenas de texto en C++ mejora notablemente con respecto a las funciones C de **`#include string.h`** como **`strcpy()`**, **`strcmp()`**, etc.

- El método de registro, en el que damos de "alta" un dispositivo en el menú interactivo. Nos vale cualquier tipo de dispositivo, sea ejecutable o no. Se utilizará como puntero el puntero base **`*Dispositivo`**, para hacer uso del polimorfismo
- El método **`ejecutar()`**, que es la puerta de entrada al menú. Como opción podemos indicar al menú si al salir queremos destruir los dispositivos o no.
- El método de menú general (de primer nivel). Nos muestra un listado de los dispositivos, inspecciona el vector de estados de los dispositivos, hace recuento, y de forma contextual nos ofrece opciones generales como iniciar todos los dispositivos (que estuvieran parados), pausar todos los dispositivos (que estuvieran funcionando), reanudar (para los dispositivos que estuvieran parados) o detener todos los dispositivos (que estaban funcionando).

El método de menú general nos permite actualizar la lista de estado de dispositivos, salir del programa o seleccionar un dispositivo específico, con el que pasamos al menú de segundo nivel, para interaccionar con dicho dispositivo, iniciarlo, pausarlo, retomarlo o detenerlo; o visualizar sus propiedades.

Esta es la rutina que podríamos utilizar para iniciar todos los dispositivos que hay en el vector de punteros **`std::vector<Dispositivo *> dispositivos`**.

```
for(unsigned int i = 0; i < dispositivos.size(); i++)
     dispositivos.at(i) -> inicia();
```

5.1.2. Cadenas de texto y flujos de entrada/salida

Merece la pena profundizar en el manejo de las cadenas **`str::string`**, puesto que C tradicionalmente ha hecho bastante engorroso el manejo de cadenas, y más si éstas cambian de longitud. Su manejo es muy fácil en C++ y las utilizaremos en este capítulo. Se ha de insertar la instrucción **`#include <string>`**

- Declaración de cadenas mediante asignación directa.

```
std::string texto = "Hola"; std::string texto2 = " qué tal";
```

- Concatenación, usando el operador suma "+".

```
std::string texto3 = texto2+texto3; → texto3 = "Hola qué tal".
```

- Métodos propios en el objeto de cadena, como buscar una cadena en el texto y marcar su posición (con base en 0).

```
unsigned int i = texto3.find("la"); → i = 2
```

- Convertir de entero a cadena.

```
int i = 46;
std::string texto = std::to_string(i); → texto="46"
```

Estas cadenas pueden usarse en unas entidades que llamaremos flujos de entrada y salida. Éstas sustituyen a **printf(), scanf()** y las funciones de E/S de archivo. Las declararemos con **#include <iostream>** e **#include <fstream>**.

La utilización de los flujos de datos se asemeja a una tubería o cola donde vamos extrayendo, o introduciendo datos, utilizando los operadores << y >>.

```
std:cout << "Hola, mundo"; // Mostrar mensaje de "Hola, mundo"

std:cin >> int opcion; // El número introducido por teclado a variable opción
```

De forma similar a las cadenas **std::string**, también se pueden concatenar las salidas al flujo, como esta, utilizando la variable **int cuenta;**

```
std::cout << "La cuenta es: " << cuenta << " iteraciones\n";
```

En algunas ocasiones muy críticas, como, por ejemplo, la imposibilidad de poder abrir una cola de mensajes en el bus, recurrimos a la salida de errores estándar **std::cerr**:

```
std::cerr << "ERROR: No se pudo obtener id de la cola de mensajes del bus";
```

El manejo de flujos de entrada/salida y de cadenas de caracteres es muy habitual en el menú y en las propiedades de los dispositivos.

5.1.3. Menú de segundo nivel (dispositivo). Polimorfismo y destructores virtuales

Hemos visto en el menú general que con tan solo invocar los métodos que todos los dispositivos tienen en su clase base como **inicia(), pausa(),** etc. se pueden enviar comandos generales a todos los hilos del proceso. Sin embargo, hemos visto que cada dispositivo que deriva de la clase base **Dispositivo** tiene unas características muy diferentes. Las variables contenidas en **Conector** son muy diferentes a las que hay en **Bus**, o en la clase de generación de números constantes **Constante**.

Sin embargo, todas ellas tienen un método llamado **propiedades()**. Un método común que se puede llamar, independientemente de qué clase sea, pero que de alguna forma tiene que saber a qué clase nos estamos refiriendo. Estamos utilizando un vector de punteros a **Dispositivo** y, en un principio serían indistinguibles, pero C++ consigue distinguirlos gracias a los métodos virtuales.

```
Dispositivo *conector1 = new Conector;
Dispositivo *bus1 = new Bus;
bus1 -> propiedades(); // Llama a propiedades de Bus
conector1 -> propiedades(); // Llama a propiedades del conector.
```

Tan sólo hay que declarar **`propiedades()`** así, en la clase base:

```
virtual void propiedades();
```

Esta característica es muy potente, porque el menú interactivo, si quiere conocer las propiedades de un dispositivo, que será la primera opción en el menú, tan sólo tiene que llamar al método **`propiedades()`**, sin preocuparse de nada más. Todas las propiedades las visualizamos en contexto gracias a que las hemos ido incorporando en cada uno de los dispositivos. Algunas, incluso, permiten la entrada de datos, para modificar sus parámetros y así efectuar ensayos sobre la marcha, mientras funciona el proceso, como, por ejemplo, cambiar el valor de una consigna del controlador, o sus parámetros PID.

Creamos otro método virtual llamado **`propiedades_base()`**, en la que, como información extra, podremos visualizar propiedades no del todo bien accesibles como las de dispositivo, conectable o vectores dinámicos, ya que nunca vamos a declarar un objeto de estas clases. Siempre será de alguna clase derivada.

La implementación de este menú está en **`menu_dispositivo(unsigned int dispositivo)`** Al que enviamos el índice del dispositivo en el vector de dispositivos. Si queremos conocer sus propiedades, es sencillo:

```
dispositivos.at(dispositivo) -> propiedades();
```

```
MENU DE DISPOSITIVO ------------ ID=4301  Nombre: Discretizador motor-----

  1) Propiedades de este dispositivo

  2) Detener el dispositivo
  3) Pausar el dispositivo

  0) Salir del menu de dispositivo

Seleccione opción de dispositivo -> 1


-------- PROPIEDADES DEL DISCRETIZADOR -----------------------------


Parámetros del discretizador
        id:                                     4301
        esid:                                   -3
        Tamaño de cola:                         2
        Frecuencia de muestreo:                 50
        esid:                                   -3
        Tamaño de paquete hacia el dispositivo: 4
        Tamaño de paquete hacia el bus:         16

Error del Bus:                          0
Codigo de retorno del Bus:              -5
Escribe a dispositivo:          NO
Condición de alarma:            NO
```

Figura 5.3

Este menú de segundo nivel además proporciona iniciación, pausa, reanudación y parada específica del dispositivo, según el contexto. Si no es ejecutable, por ejemplo, no mostrará opciones de ejecución. En este ejemplo vemos el menú de dispositivo de un discretizador (ejecutable) y sus propiedades.

Algunos dispositivos también tienen un pequeño menú, para introducir datos, cuando accedemos a sus propiedades, como así pasa en los controladores PID.

```
MENU DE DISPOSITIVO ------------ ID=1302  Nombre: Controlador amortiguador-----

  1) Propiedades de este dispositivo

  2) Iniciar el dispositivo

  0) Salir del menu de dispositivo

Seleccione opción de dispositivo -> 1


-------- PROPIEDADES DEL CONTROLADOR PID ----------------------------

                P:      19190
                I:      20320
                D:      0

        1) Modificar valor P
        2) Modificar valor I
        3) Modificar valor D

        0) Salir

------------------------------------------------------------------------
Seleccione opción ->
```

Figura 5.4

Con el sistema de menús propuesto se procura tener una interactividad lo más completa posible. De esta forma, podemos realizar varios ensayos en una sesión y, sobre todo, comprobar el funcionamiento del proceso, sus dispositivos, el estado de conexión del bus, etc.

La terminación de los objetos, mediante los destructores, es tan vital como su construcción. En las clases como Conector o Discretizador se utilizan espacios de memoria reservados dinámicamente que han de ser liberados. No son espacios muy grandes, pero es buena práctica en programación que, si reservamos memoria, tengamos de liberarla. La función de los destructores es clara: si vamos a destruir el objeto, previamente realizamos unas operaciones necesarias, y entonces ya se puede destruir el objeto.

Como aquí estamos utilizando punteros de la clase base, la destrucción del dispositivo número i, ubicado en el vector dispositivos, con **`delete dispositivos.at(i)`** requiere utilizar destructores virtuales.

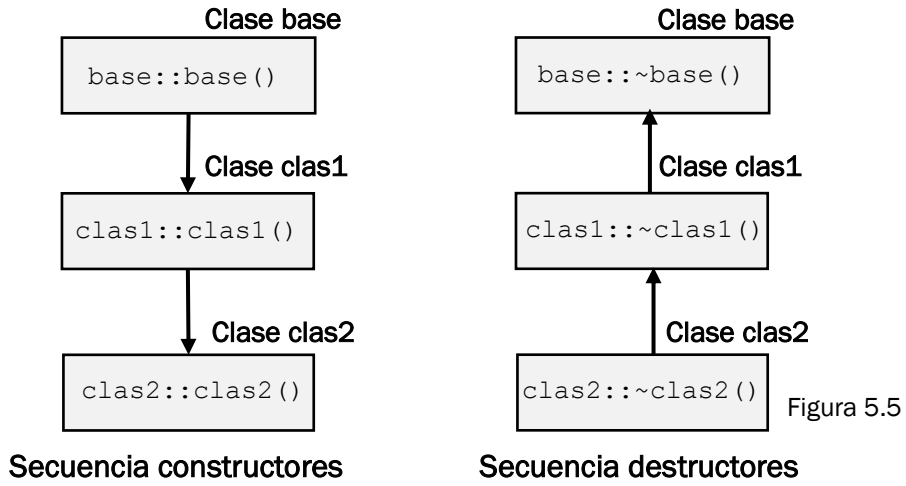


Figura 5.5

La figura ilustra que el primer constructor que se invoca al crear un objeto es el de la clase base; y el último, el de la última clase derivada. Con los

destructores pasa lo contrario y el primer destructor invocado es el de la última clase derivada, hasta llegar a la clase base.

Es también importante, al detener un dispositivo, obtener su hilo y llamar al método **`join()`**. Como aquí quien inicia y detiene los hilos es la rutina principal no debería haber mayor problema. No es necesario utilizar **`std::future`** en los controladores, ya que no nos importará esperar un poco. En ocasiones, al cerrar dispositivos con poca frecuencia de muestreo notamos, al salir del proceso, que hay una latencia y es porque se está esperando a su cierre.

No moveremos el hilo al obtener su referencia en **`acaba()`** como así pasaba con **`Watchdog`**. Declaramos directamente su referencia al invocar el método de detención y la usamos con **`join()`**.

```
int err;
std::thread& hilo = acaba(err);
hilo.join(); // Si el dispositivo es lento, tardará un poco
```

5.2. El conector de recogida de datos

Esta característica es esencial para guardar y analizar los datos y variables de los dispositivos. Nos permitirá consolidar aspectos interesantes de la herencia y el polimorfismo. También conoceremos otra estructura de datos, sin contenedor ni acceso por índice como son las colas estándar **`std::queue`**. Finalmente, combinaremos la concatenación **`std::string`** con la grabación de datos a archivos, al más puro estilo C++, usando **`std::ofstream`**.

Ante esta situación, nos planteamos de dónde extraer los datos. Podría ser de un dispositivo, o del mismo bus, pero como prácticamente todos los elementos han de unirse por conectores, podríamos colocar unas "sondas" en los conectores, y extraer su información, para así acceder a los datos del sistema. Tan sólo necesitamos tener a mano la estructura de los paquetes de datos, y los conocemos, pues se ubican en el archivo estructuras_datos.h.

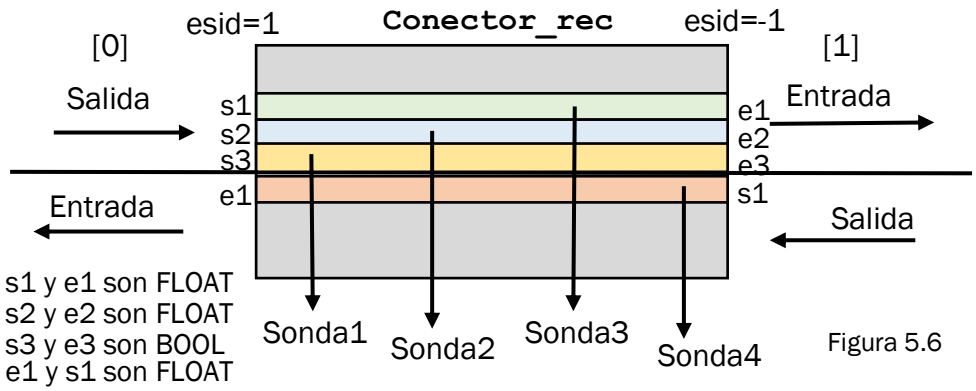


Figura 5.6

Cada conector tiene varias líneas de datos y cada una podría ser "pinchada" por un elemento que hemos llamado sonda. Por cada conector podemos crear varias sondas, tanto en entradas como en salidas. La referencia la vamos a

obtener respecto al primer elemento que habíamos conectado. Creamos un bit llamado `io` que será `false` si recogemos datos de la salida del dispositivo que se conectó primero y que está marcado por [0]; en cambio, si queremos escoger los datos que llegan a este mismo dispositivo, deberemos marcar `io` con `true`.

Por tanto, según el gráfico anterior, la sonda 1, la sonda 2 y la sonda 3 son `false`. La sonda 4, en cambio, es `true`.

Una vez determinado el sentido, hemos de determinar la posición y para ello hemos de conocer la estructura de datos, que en este caso es esid=±1. La posición de la sonda 3 es fácil de determinar según un puntero que empezaría en la estructura de salida del dispositivo [0]; y es la posición cero.

En cambio, la sonda 2 está en el segundo dato, así que hemos de conocer el tamaño del primero, y como ese tamaño es `float`, entonces `pos = sizeof(float)`.

La sonda 4 está en la otra dirección con `io=true`, así que sería la posición 0. Y la sonda 1 estaría en una posición `pos = 2*sizeof(float)`.

Por último, está el tipo de datos a captar, que la sonda ha de reconocer para que sea codificado convenientemente.

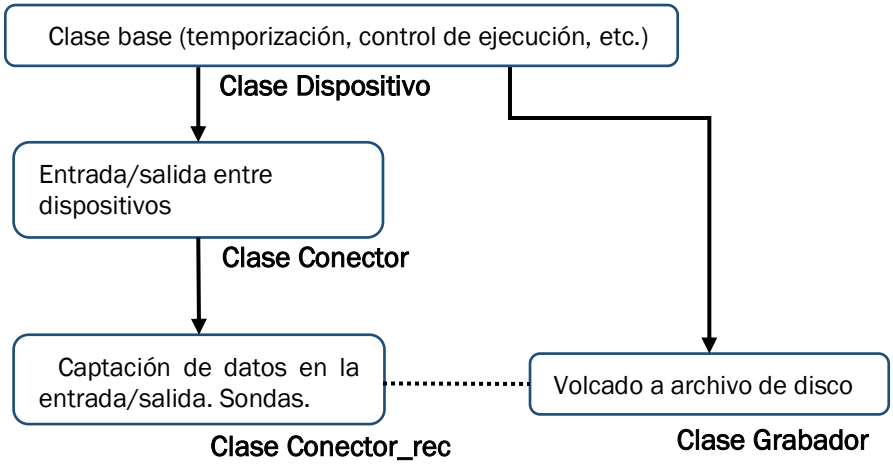


Figura 5.7

Para crear esta nueva clase `Conector_rec` tan solo necesitaremos modificar algunas rutinas para crear buffers de copia. El resto de la funcionalidad permanece intacta y, por tanto, hacemos uso de la herencia desde `Conector`.

Es decir, para los dispositivos, seguirá siendo un conector, sin embargo, a la hora de registrar las conexiones, tendremos nuestro propio método que llamará al de la clase padre y realizará tareas propias para facilitar una copia de los datos desde el grabador.

La utilización de buffers para las sondas es por motivos de rapidez. Se pretende perjudicar lo más mínimo la labor de entrada/salida de los

dispositivos en la exclusión mutua (donde ahora concurrirán tres hilos independientes). El copiador de zonas de memoria **`std::memcpy`** se caracteriza por ser un algoritmo rápido. De nuevo, conviene recordar que, al utilizar punteros a zonas de memoria, hay que tener mucho cuidado diseñando los algoritmos para que no salten excepciones por lectura/escritura en zonas fuera de rango. De hecho, se han habilitado mecanismos y comprobaciones para que, en caso de equivocación del usuario configurando las sondas, no pueda ocurrir esto.

Nos vemos en la obligación de hacer **`virtual`** la función **`registrar()`** pues habrá dos rutinas diferentes y los conectores se utilizan, como los demás objetos, por punteros. La ventaja de esto es que las conexiones se harán igual. Misma función y mismos argumentos.

Clase Conector_rec	

Variables	• Buffers intermedios para la recogida de datos
	`BYTE *buffer_sondeo_entrada = nullptr;`
	`BYTE *buffer_sondeo_salida = nullptr;`
	• Estructura de los datos de las sondas
	`std::vector<plantilla_datos_rec> datos;`

Métodos	• Constructor. Hay que incluir el puntero al grabador
	`Conector_rec(int id, Grabador *grabador);`
	• Creación de sondas para el conector
	`int crea_sonda(bool io, unsigned int pos, tipo_datos_rec tipo, std::string etiqueta);`
	• Copia de datos del conector_rec para el grabador
	`const std::vector<plantilla_datos_rec>& obtener_datos();`
	• Orden de recolección de datos. Se copian los buffers del conector a los buffers intermedios
	`void recoleccion();`

El resto de métodos son los de la clase base **`Conector`**, con algunas modificaciones, en especial al registrar los dispositivos, donde tiene que averiguar qué tamaño tendrán los buffers intermedios. Debido a esta modificación de la clase derivada, se declara **`registrar()`** como virtual. El destructor de la clase ha de liberar los buffers intermedios.

La orden de recolección es llamada por el grabador e indica al conector de recogida que copie los datos que hay almacenados en ese momento en su buffer.

Los datos de las sondas (posición, dirección de los datos, longitud de datos, etiqueta identificable en **`std::string`**, puntero al buffer y puntero al conector) son traspasadas al grabador gracias a la función **`obtener_datos()`**.

Por ejemplo, en el conector entre el amortiguador y su discretizador queremos crear una sonda, para que capte la posición del chasis, ubicada, según en la estructura de datos esid=1, en **`salida.s1`**. Dato del tipo **`float`**. Primero se crea **`Conector_rec`**, asociado a un grabador que llamaremos **`grabador`**.

```
Conector_rec *amo_dis = new Conector_rec(7201, grabador);
```

Los registros (conexiones) de los dispositivos se hacen exactamente igual que con un conector. Internamente es otro procedimiento, pero al estar declarado como **`virtual`**, se garantiza que los conectables de los dispositivos llaman al método modificado **`registrar()`** de **`Grabador_rec`**.

```
amortiguador -> conectar(amo_dis, 1);
discret_amo -> asociar(amo_dis);
discret_amo -> conectar(amo_dis, 1);
```

Y se crea la sonda propuesta:

```
amo_dis -> crea_sonda(false, 0, FLOAT, "Pos_chasis");
```

Si quisiéramos capturar lo que recibe el amortiguador que es la fuerza al actuador hidráulico, desde el controlador, hay que tener en cuenta que es la dirección contraria, con **`io=true`**. Cuando se habla de "dirección" contraria es la que sale del segundo dispositivo que se registró; en este caso, el discretizador.

```
amo_dis -> crea_sonda(true, 0, FLOAT, "Actuador");
```

Ya tenemos los conectores preparados para capturar los datos. Nos falta un último elemento fundamental que es el grabador, con el que recogeremos dichos datos de forma periódica y los volcaremos en un archivo legible.

5.3. El grabador

Una vez que hemos configurado los conectores de recogida de datos, tenemos una colección de vectores (sondas) asociadas a un grabador, que es el referido en sus constructores.

Por tanto, al iniciarse la rutina del grabador, llama a **`obtener_datos()`** de cada uno de los miembros de su lista de conectores y confecciona un listado de sondas con las que trabajar.

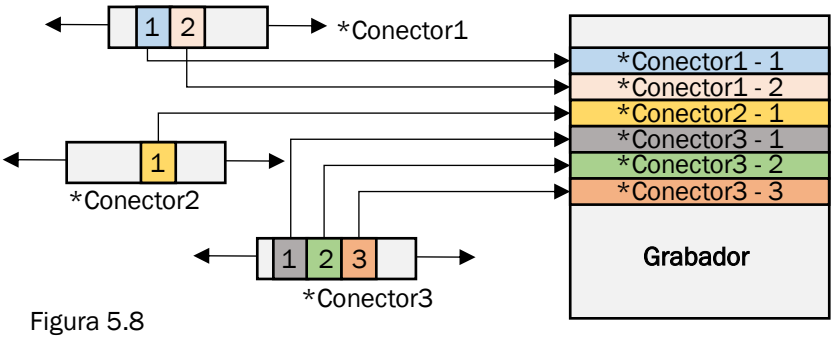


Figura 5.8

Una vez que tenemos las sondas de uno o varios conectores almacenadas en el grabador, disponemos de un acceso directo a los buffers de memoria de dichos conectores para que regularmente el grabador los vaya guardando.

Se podría utilizar directamente una salida a archivo para guardar, pero hay que tener en cuenta que el acceso a disco suele ser muy lento y no es práctico utilizarlo mientras se recogen los datos. Es mucho más rápida la memoria RAM para este cometido y podríamos utilizar el acceso a disco cuando tengamos la grabación completa en la memoria. De esta forma no perjudicamos el rendimiento global del sistema mientras tomamos los datos.

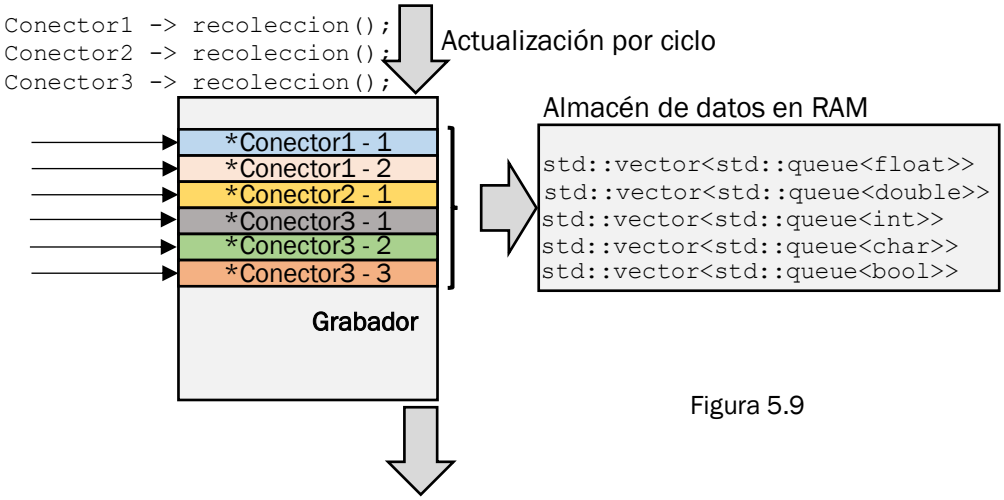


Figura 5.9

La organización de la memoria también es primordial. Se ha elegido, como lo más óptimo en la organización de los datos, los vectores de colas de los tipos de datos más comunes: **`float, double, int, char y bool`**.

Se trata de vectores porque podría ocurrir, por ejemplo, que hay dos o más sondas de tipo **`float`**, en el mismo conector o en diferentes conectores.

En cada ciclo del grabador, se llama a los métodos de recolección de cada conector para que estos hagan la copia rápida. Después, el grabador recoge directamente la información con los punteros a esas copias y la distribuye, según el tipo a los vectores de colas. Estas operaciones en RAM permiten más frecuencia de muestreo en el grabador, aunque menos cantidad de datos a recopilar por vez.

5.3.1. Colas del grabador

La cola **`std::queue`** es una estructura de datos básica y muy recurrida en sistemas. Por ejemplo, la mensajería del bus utiliza colas. También las hemos usado en la clase **`Dinamico`**, aunque con indexado (colas dobles **`std::deque`**).

Aquí no necesitamos indexación sino almacenado y recuperación. En el ciclo de grabación realizaremos el almacenado con **`cola.push(dato)`**, hasta que se finalice la grabación o hasta que se supere el límite de datos.

A la hora de recuperar esos datos y volcarlos a archivo, se utilizará la función inversa que será **`dato = cola.pop()`**. Las colas pueden crecer de una forma indefinida y su ventaja es que, al ser más sencillas, son más eficientes que las entidades indexadas como vectores y colas dobles. No necesitamos acceder a miembros específicos de la cola sino grabar en bloque, y después recuperar.

5.3.2. Grabación a archivo de datos

Si al iniciar el dispositivo, éste se pone a grabar de los conectores que tenga asociados, al detenerlo y finalizarlo, ubicaremos en **`rutina()`** una llamada al método privado de volcado de archivo. Es una particularidad que hace diferente a esta clase.

Al ser el volcado a archivo la etapa más lenta, es frecuente que tenga que esperar la rutina principal cuando se detiene con **`join()`**, sobre todo si son muchos datos a volcar. Hay que tener en cuenta que es un volcado en un archivo de texto, legible, y que han de convertirse todos los datos de binario a carácter.

Las funciones empleadas en C++ difieren significativamente de las incluidas en **`<file.h>`** como **`fopen()`**, **`fclose()`**, **`fprintf()`**, etc, cuando se hace entrada/salida en archivos. También, como vimos en la entrada/salida de pantalla, se basan en flujos y el archivo de cabecera que emplearemos será **`<fstream>`**.

Cada archivo se maneja con un objeto. Para empezar a guardar datos empleamos:

```
std::ofstream archivo_salida;

std::string nombre = "Archivo.dat";

archivo_salida.open(nombre, std::ios::out);
```

Esto es el equivalente a **`fopen()`**, pero empleando métodos de la clase **`std::ofstream`**. Se ha utilizado el flag **`ios::out`** para indicar que sólo queremos emplear un archivo de salida (escribir a disco).

Aunque se ha indicado como nombre de archivo "Archivo.dat", en realidad usaremos concatenaciones de cadena que especificarán el número de identificación id del grabador, el tipo de proceso (planta o controlador), y el número de grabación de la sesión (si así lo deseamos). Construir el nombre de archivo se realiza fácilmente, gracias a la concatenación.

```
std::string proceso = "PLANTA";

nombre = proceso + "_ID" +
     std::to_string(Dispositivo::obtiene_id()) + ".dat";
```

Si id=8101, entonces el nombre de archivo será PLANTA_ID8101.dat.

El resto del proceso de volcado son bucles que reproducen exactamente los bucles de grabación, pero a la inversa. En vez de añadir con **`push()`**, se van extrayendo los valores de las colas con **`pop()`**, según la estructura seguida. Se adjunta el tiempo de cada iteración, en pasos equivalentes al período de muestreo y que se obtiene con el método **`obtiene_period()`** de **`Dispositivo`**. También se enumeran las etiquetas de las sondas.

La forma de ir guardando los datos en el archivo es mediante flujo, de tal forma que si hemos declarado **`std::ofstream archivo_salida`** y lo hemos abierto con un nombre de archivo, entonces para volcar toda la colección de etiquetas que tenemos en el vector de estructuras **`datos`** escribimos:

```
for(unsigned int i=0 ; i < datos.size() ; i++)
        archivo_salida << "\t\t" << datos[i].etiqueta;
```

El cierre se hace con **`archivo_salida.close()`**. Una vez que se ha completado toda la tarea con los datos almacenados, entonces ya se puede salir de la rutina del hilo y finalizar.

Clase Grabador	

Variables	• Vector de punteros de conectores
	`std::vector<Conector_rec *> conectores;`
	• Tipo del proceso, si planta o control (para el nombre de archivo de salida
	`tipo_proceso tipo;`
	• Vectores de colas
	`std::vector<std::queue<float>> datos_float;`
	`std::vector<std::queue<double>> datos_double;`
	`std::vector<std::queue<int>> datos_int;`
	`std::vector<std::queue<char>> datos_char;`
	`std::vector<std::queue<bool>> datos_bool;`
	• Límite de almacenamiento especificado por el usuario
	`std::atomic<unsigned int> kbs;`
	• Vector de las sondas de los conectores (posición, tipo de datos, etiquetas)
	`std::vector<plantilla_datos_rec> datos;`

Métodos	• Constructor, donde establecemos id, frecuencia de muestreo, conteo en el nombre del archivo por sesión de grabación y límite de almacenamiento
	`Grabador(unsigned int t, int id, tipo_proceso tipo, bool diferentes, unsigned int kbs);`
	• Rutina y actualización
	`void rutina();`
	`int actualiza();`
	• Registrar un conector al grabador (invocado por el constructor de Conector_rec
	`int registrar(Conector_rec *conector);`
	• Volcado a archivo tras dar orden de detención
	`void archivo();`

IMPORTANTE: Si el límite de almacenamiento proporcionado es cero, el dispositivo grabará indefinidamente, llegando a ocupar toda la memoria RAM si no se detiene.

La operación con este dispositivo es muy sencilla. En el mismo momento que éste comienza a ejecutarse con **`inicia()`**, recopila las sondas y vuelca su información a la memoria interna. Estará constantemente grabando, a no ser que se detenga con **`acaba()`** o alcance el límite máximo establecido por el usuario (en kilobytes). Al volver a arrancar iniciará de nuevo la grabación, en el

archivo anterior, o en otro nuevo, dependiendo del bit **`diferentes`** que se le proporcionó en el constructor. Así podemos disponer de varias sesiones de grabación en varios archivos. Si el dispositivo se pausa, pausará la grabación.

Cuando arrancamos todos los dispositivos en el menú, con *iniciar todos*, arrancamos también la grabadora. Esta situación es idónea para reflejar los transitorios que pudiera haber, en el arranque del sistema.

5.4. Registro de los grabadores y sondas. Creación del menú

Nos serviremos del modelo del depósito (sin controlador) para describir la creación del menú, conectores de recogida y grabadores. Se ha escrito en planta_deposito_solo.cpp.

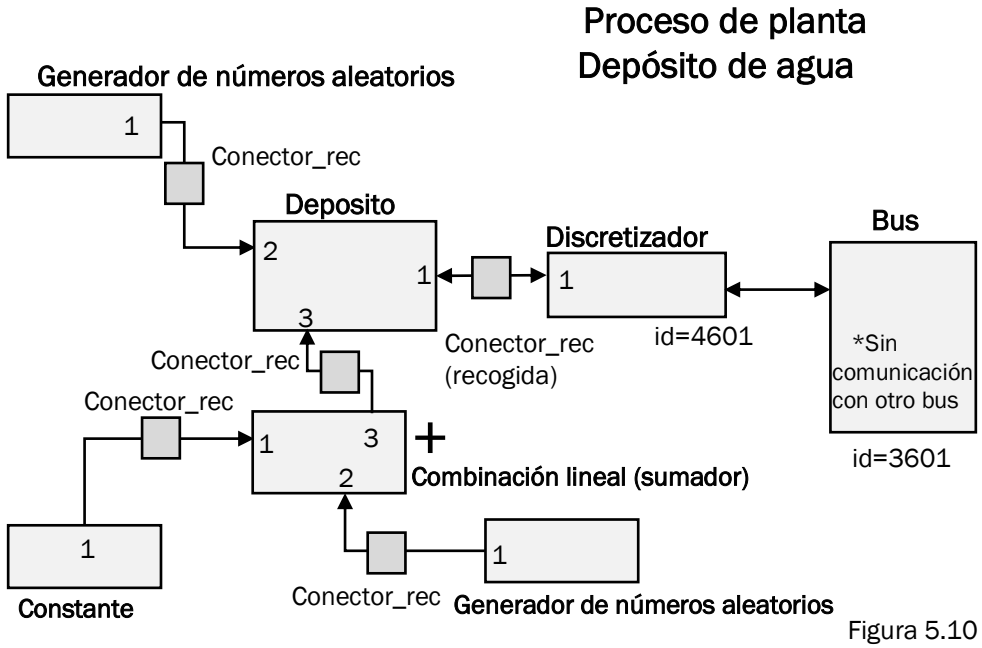


Figura 5.10

En el modelo se han introducido series aleatorias, tanto en el coeficiente de descarga (régimen turbulento) como en la entrada del caudal, dentro de unos márgenes, para hacerlo más realista.

Crearemos dos grabadores sin límite de grabación. Es recomendable proporcionarles valores como **`500`** (500 Kb), pero como el depósito es grande, tardará bastantes minutos en alcanzar su régimen estacionario que equilibre el caudal que entra con el caudal que sale. Les proporcionamos el valor cero, indicándoles que pueden grabar sin límites:

```
 Grabador *grabador1 = new Grabador(10,8601, PROCESO_PLANTA, false,
0);

 Grabador *grabador2 = new Grabador(10,8602, PROCESO_PLANTA, false,
0);
```

Se declaran los objetos de dispositivos físicos de planta:

```
 Aleatorio *coef_des = new Aleatorio
(20,2601,419,577,129,899,1002,0.79,0.81);

 Constante *caudal_entrada = new Constante(1,2602,0.044,100,0);
   // Inicialmente 0.044 (En m^3 / s) -> 44 L /s

 Aleatorio *fluctuacion_caudal = new Aleatorio
(5,2603,193,577,599,774,1002,0.001,-0.001);

 Combinacion *suma = new Combinacion(5,2604,1,1);

 Deposito *deposito = new Deposito(200,2603,0.01,2,0.1,0.1,0,0.8);
 // Abertura en el sumidero de 0.01 m^2 (cuadrado de 10 cm x 10 cm)
```

Las fluctuaciones aleatorias son de $\pm 1\ L/s$ en el caudal y ± 0.01 en el coeficiente de descarga. El objeto de combinación lineal está operando como un sumador.

Ahora, cuando hagamos uso de **Conector_rec**, varía su constructor, y además del id hay que suministrarles el grabador al que van asociados.

```
      Conector_rec *dep_dis = new Conector_rec(7601, grabador1); //
Entre deposito y discretizador

      Conector_rec *cau_sum = new Conector_rec(7602, grabador2);
//Entre caudal y suma

      Conector_rec *flu_sum = new Conector_rec(7603, grabador2);
//Entre fluctuación y suma

      Conector_rec *sum_dep = new Conector_rec(7604, grabador2);
//Entre suma y deposito

      Conector_rec *coef_dep = new Conector_rec(7605, grabador2);
// Entre coef_descarga y deposito
```

En cuanto a conexiones con los dispositivos, estos conectores de recogida de datos funcionan exactamente igual que un conector normal, gracias a que se ha hecho virtual la función de registro, se llama al método de la clase derivada, y así se habilitan los buffers de copia. En el código fuente están todas las conexiones, y aquí mostraremos la que se hace con el discretizador, el cual requiere una asociación previa.

```
 deposito -> conectar(dep_dis, 1); // Motor-discretizador

 discret_dep -> asociar(dep_dis);

 discret_dep -> conectar(dep_dis, 1); // Tras asociar, conectar
```

Gracias al polimorfismo dinámico y los métodos virtuales, el usuario, a la hora de utilizar un **Conector** y un **Conector_rec**, puede utilizar los punteros de los objetos base y objetos derivados. En tiempo de ejecución, el programa llamará a la función adecuada, según el tipo exacto de puntero. Con este mecanismo, se hace muy fácil emplear estos nuevos conectores, sin necesidad de definir funciones con otros nombres, que harían más difícil la programación.

Estos conectores de recogida son diferentes, y se pueden definir unas sondas que emplearemos para capturar los datos que transcurren por ellos:

```
// Grupo Grabador 1

dep_dis -> crea_sonda(false, 0, FLOAT, "Altura");

dep_dis -> crea_sonda(false, 0+sizeof(float), FLOAT, "Caudal salida");  // Grupo Grabador 1

dep_dis -> crea_sonda(false, 0+2*sizeof(float), BOOL, "Alarma");

// Grupo Grabador 2

sum_dep -> crea_sonda(false, 0, FLOAT, "Caudal total entrada");

cau_sum -> crea_sonda(false, 0, FLOAT, "Caudal fuente");

flu_sum -> crea_sonda(false, 0, FLOAT, "Caudal fluctuación");

coef_dep -> crea_sonda(false, 0, FLOAT, "ks descarga");
```

Si conocemos las estructuras de los datos **ES_1** y **ES_2** (en el archivo estructuras_datos.h), nos será relativamente fácil dar con las posiciones que hay que poner en las sondas, con la ayuda de **sizeof()**. También especificamos los tipos de datos a captar y una etiqueta para identificar su columna en el archivo de salida.

Con esta información, cada grabador sabrá qué datos grabar y cómo disponerlos en el archivo de salida.

El objeto del menú es el más sencillo de manejar. Lo declaramos de esta forma: **Interactivo menu;**

Y añadimos, uno a uno, todos los dispositivos que hemos ido declarando gracias a su método **registro()**. Por ejemplo, para el depósito: **menu.registro(deposito, "Depósito");**

Una vez añadidos todos los objetos, ejecutamos el menú:

```
menu.ejecutar(true);
```

El bit true indica que se destruyan todos los objetos de los dispositivos registrados, al decidir el usuario que se salga del menú.

Clase Interactivo	
Variables	• Datos de cada uno de los dispositivos `std::vector<Dispositivo *> dispositivos, std::vector<std::string> nombres, std::vector<bool> ejecutable, ejecución, pausa, alarma;`

Métodos	• Sin constructor
	• Registrar dispositivo
	`int registro(Dispositivo *dispositivo, std::string nombre);`
	• Ejecutar menú
	`int ejecutar(bool borrar);`

En el momento en el que se ejecuta el menú, el usuario hará uso de éste. Se pueden iniciar, detener, pausar o retomar hilos, en bloque o uno por uno. También puede visualizar y modificar propiedades.

El uso del grabador es muy fácil. Si se ejecuta, empieza a grabar, y si se detiene, vuelca esa grabación en un archivo. En nuestro ejemplo del depósito se crearán dos archivos. El formato del archivo de salida es de texto, editable. Aquí está la posible salida de las primeras líneas del grabador 1:

```
# PLANTA_ID8601.dat
#Tiempo    Altura            Caudal salida     Alarma
0          0.1               -0                0
0.1        0.10024           0.0114333         0
0.2        0.100182          0.0114445         0
0.3        0.100443          0.0110137         0
0.4        0.100789          0.0109973         0
0.5        0.102376          0.0112856         0
0.6        0.104207          0.0115031         0
```

Hay que tener en cuenta que no siempre coinciden las tabulaciones en el archivo de salida. El uso de alguna aplicación para generar gráficas resuelve este problema y presenta los resultados de los análisis y simulaciones, de una forma más clara.

5.5. Visualización gráfica de los resultados. GNU Plot.

El archivo de salida, aunque nos aporta datos interesantes sobre el comportamiento del sistema, no nos permite conocer a primera vista valores máximos, mínimos, naturaleza del transitorio, del estacionario, frecuencia, etc. Es preferible hacerlo de una manera gráfica, con variables respecto del tiempo.

Para las gráficas, haremos uso de una herramienta útil, escrita en código libre, como es GNU Plot. Tras instalarla en nuestro equipo nos ofrece una consola que se puede ver así:

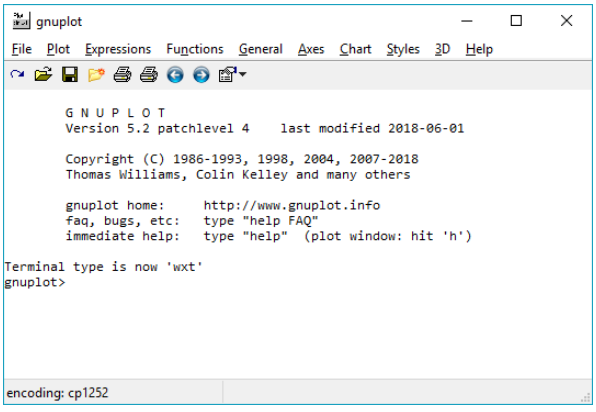


Figura 5.11

Existe abundante documentación sobre GNU Plot y diversas formas de mostrar las gráficas. Nos serviremos de la presentación clásica, uniendo puntos con líneas, y presentando un valor respecto a otro.

La descarga y la consulta de documentación de este programa está en http://gnuplot.info/. En nuestro caso, nos basta con especificar el archivo de salida y elegir la línea (o líneas a graficar).

Primero, elegimos el directorio de trabajo, en file -> change directory. Pretendemos visualizar la altura con respecto al tiempo, de color azul e interpolando líneas entre los puntos. Así que, una vez en nuestro directorio de trabajo, ejecutamos esta línea en la consola de GNU Plot:

```
> plot "PLANTA_ID8601.dat" using 1:2 title 'Altura' with lines lt rgb "blue"
```

Con **`using 1:2`** indicamos que queremos representar la columna 2 (altura del agua en el depósito) respecto a la columna 1 (tiempo).

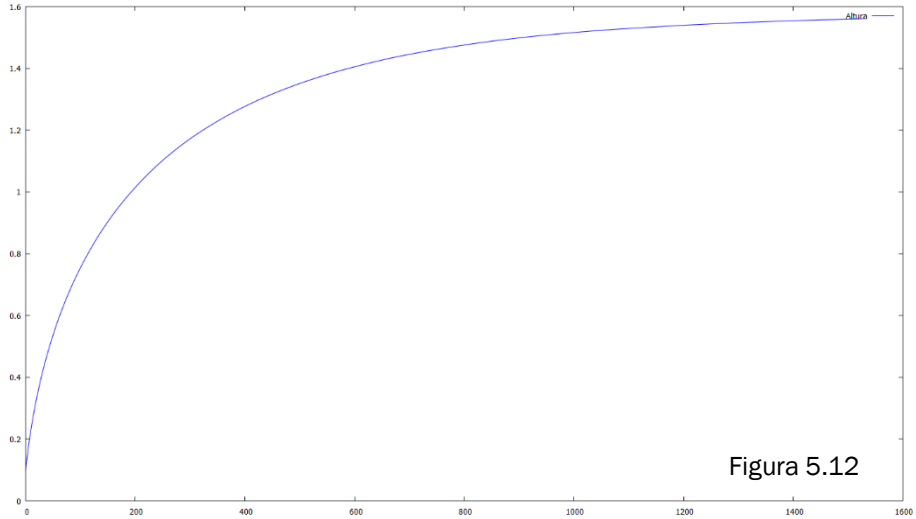


Figura 5.12

Como era de esperar, el depósito alcanza una altura estacionaria, pues el caudal de entrada es constante. Con **`using 1:3`** obtenemos el caudal de salida:

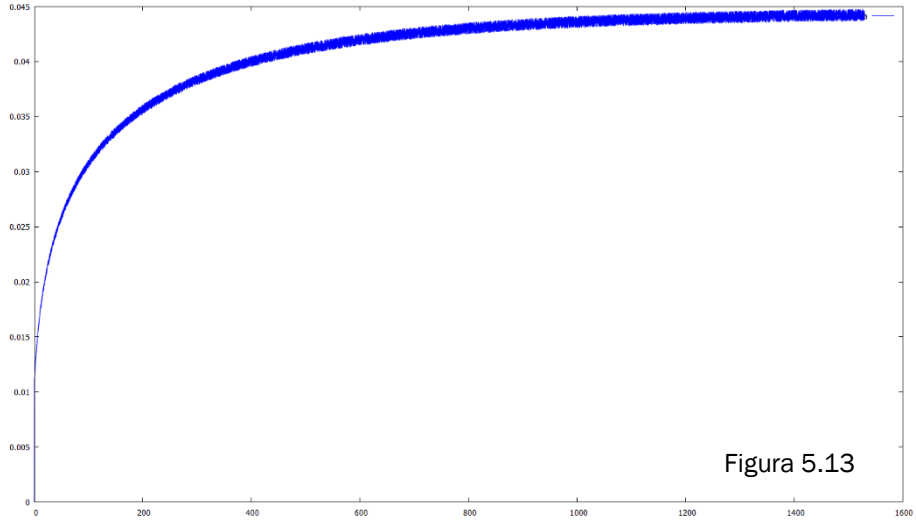


Figura 5.13

La componente estocástica del caudal de salida (debido al coeficiente de descarga) queda evidente en la gráfica.

También se puede usar GNU Plot para hacer comparativas entre variables, incluso si éstas se ubican en archivos diferentes. La primera variable del grabador 2 es el caudal de entrada que entra al depósito, y que es la salida del sumador, que la suma del caudal promedio y la componente estocástica uniforme de $\pm 1\ L/s$. Para hacer la comparativa escribimos:

```
> plot "PLANTA_ID8602.dat" using 1:4 title 'Caudal de Entrada' with lines lt rgb "blue", "PLANTA_ID8601.dat" using 1:3 title 'Caudal de Salida' with lines lt rgb "black"
```

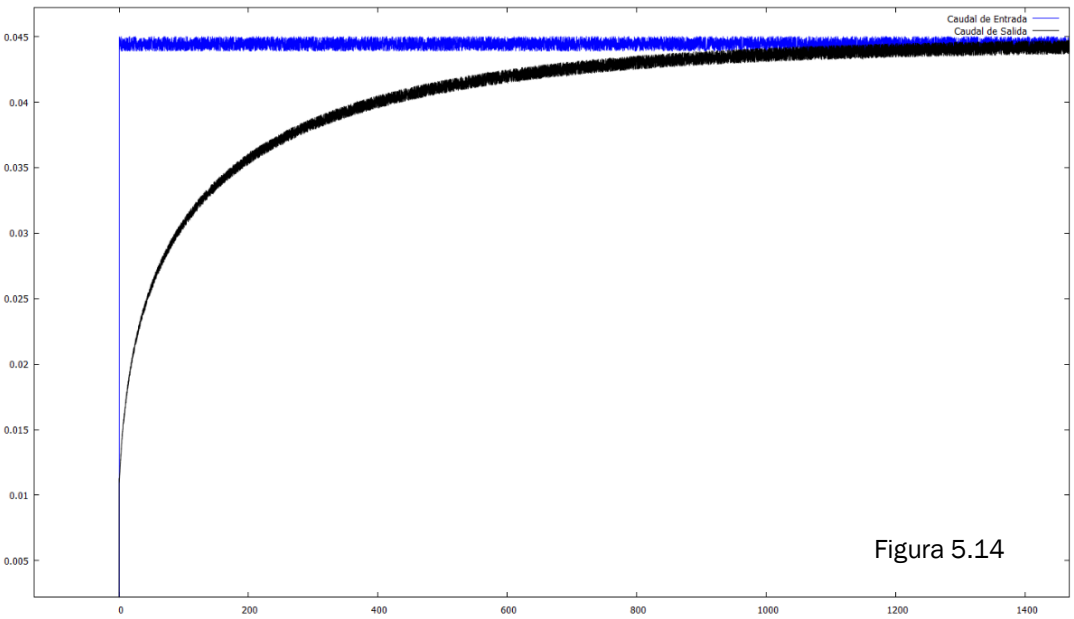


Figura 5.14

Cuando se alcanza el régimen estacionario, los caudales de entrada y salida han de coincidir, por principio de la conservación de la masa, como así se refleja en la figura.

Las plantas sin controlador como planta_motorcc_solo.cpp, planta_carro_solo.cpp y planta_deposito_solo.cpp se han utilizado para ensayar los modelos físicos, verificando su comportamiento, según las condiciones a las que se expongan. Una vez verificados (por si estos reflejan errores de programación), se ponen a prueba con los controladores.

El generador aleatorio, el generador de número constante y el sumador se pueden sustituir por un generador aleatorio, pero se ha preferido utilizar estos tres elementos porque es más fácil cambiar el promedio (caudal promedio de la constante) que los límites del generador de números. Cualquier propiedad se puede cambiar en tiempo de ejecución, haciendo uso de las propiedades del objeto, en el menú.

6. Control y comunicación con dispositivos reales

Una vez que conocemos todo lo que nos aporta C++ en cuanto a sistemas, simulación y control, se puede dar otro paso.

Recientemente han surgido en el mercado múltiples microcomputadores que podrían tener aplicaciones de interés. Por lo general contienen procesadores rápidos, y en ocasiones permiten procesos multihilo. Nos centraremos en dos dispositivos célebres en el día de hoy: Arduino y Raspberry.

Una característica deseable, cuando concebimos el control en un sistema físico, es que el control sea de respuesta rápida, principalmente cuando se manejan procesos en tiempo real. Primero, buscaremos limitaciones de nuestra implementación, y una forma de acotar el tiempo de respuesta.

Después conoceremos cómo podríamos utilizar los controladores que hemos desarrollado, y qué modificaciones tendríamos que hacer para que sea posible.

6.1. Control en tiempo real

En el ámbito industrial, bastantes veces el control de una máquina ha de ser de respuesta rápida. Definiremos la respuesta como el tiempo que tarda la máquina en recibir una orden desde el momento que cambia de estado, teniendo que trascurrir la información de los sensores al controlador, ser procesada por el controlador, y devuelta de nuevo a la máquina, como orden al actuador correspondiente. Es deseable que este tiempo de respuesta sea el mínimo posible; o que al menos podamos estimarlo convenientemente a la hora de diseñar el bucle de control.

6.1.1. Retardo del bucle de control

El tiempo de retardo es primordial para estimar el grado de respuesta del sistema, y nos dirá si es adecuado o no para determinada aplicación. Hay que tener en cuenta que las respuestas más lentas no son adecuadas en plantas que requieren un control minucioso a alta frecuencia. En uno de los ensayos del anexo III realizaremos una comparativa en función de esa respuesta, en la simulación del amortiguador que mostrará la relevancia del retardo. Los resultados simulados, por supuesto, son extrapolables a dispositivos reales.

Elegimos una respuesta del controlador de, por ejemplo, 30 milisegundos y proponemos estas tres vías de trabajo:

- **Utilizar nuestro sistema habitual**, con bus intermedio, conexión entre discretizadores y dispositivos. Con esta disposición contaremos con dos procesos, posibilidades de escalabilidad y las ventajas enumeradas en capítulos anteriores. Tan sólo habría que modificar el discretizador conectado

a la planta (HW1), sustituyendo la conexión con **`Conectable`** por la entrada/salida a dispositivo físico. Además, en caso de fallar la toma de datos, podríamos modificarla, resetear la toma de datos entrada/salida, y reiniciar el proceso de planta sin que afectara el proceso controlador.

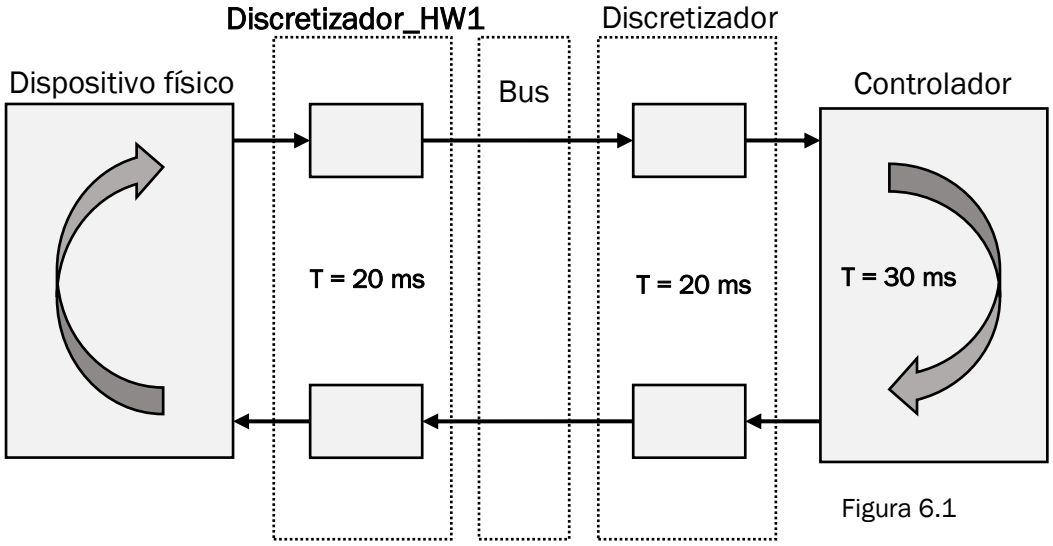


Figura 6.1

La limitación es la lentitud de respuesta, y la necesidad de que el microcontrolador sea de procesador multihilo. La peor respuesta posible en este bucle sería $T_{ret} = 20ms + 20\ ms + 30ms + 20ms + 20ms = 110ms$, a tener en cuenta, por ejemplo, para la respuesta en frecuencia del controlador.

Esta disposición, de respuesta lenta y de alta flexibilidad (multiproceso, multihilo…). Es la más adecuada para los ordenadores personales equipados con tarjetas de adquisición de datos, y Raspberry Pi.

- **Utilizar un sistema más reducido, de un proceso** (sin bus), con dos hilos por bucle de control, consistente en controlador y discretizador modificado.

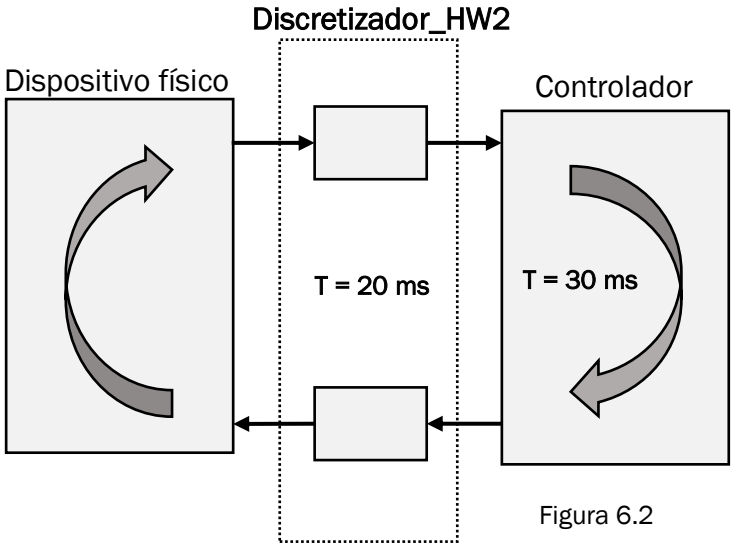


Figura 6.2

Al prescindir de elementos que contribuyen al retardo, mejoramos la respuesta del sistema. En este caso, la peor respuesta que puede haber es $T_{ret} = 20ms + 30ms + 20ms = 70ms$.

El discretizador es un hilo separado del controlador. Se hace uso de la clase **`Conectable`** y de un conector, pero se prescinde del bus. La modificación, por tanto, es diferente a los discretizadores de planta del punto anterior y se deberá implementar una clase diferente. Los microcontroladores que lo usarán también han de soportar multihilo, como son los ordenadores personales y Raspberry Pi.

- **El esquema más reducido, de un proceso y de un hilo**. No se usa ni bus, ni conector. El flujo de programa del controlador, en su ciclo iterativo (que hemos insistido en llamar *ciclo de actualizaciones*), ha de funcionar como controlador y como puerto de entrada/salida.

Es la configuración más idónea para los Arduino, pues no suelen soportar multihilo. Aunque se puede emular un *task scheduler* muy primitivo en **`loop()`** para tener multitarea, no nos sirve la clase **`std::thread`**. Por tanto, hemos de implementar un esquema mucho más básico que los anteriores:

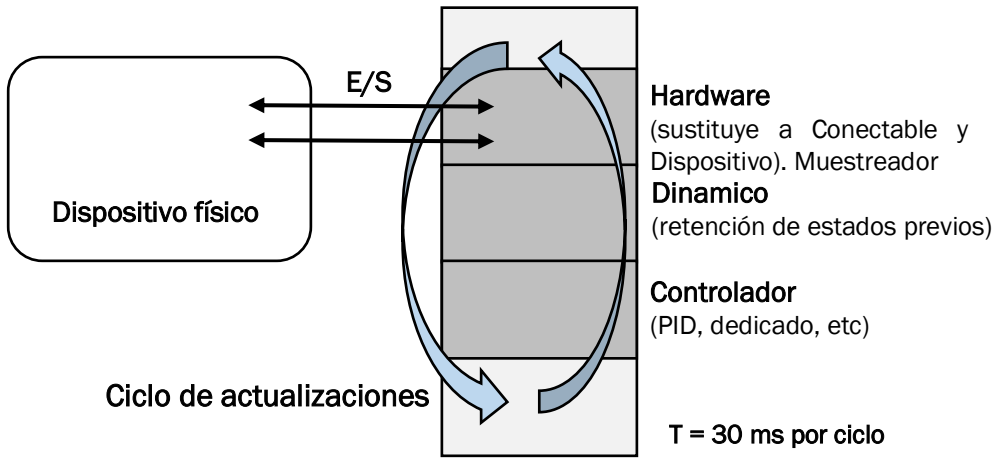


Figura 6.3

Si integramos el control y la comunicación en un hilo único que cumpla el ciclo de actualizaciones, el retardo máximo será el período del controlador, y en este caso, la peor respuesta sería de tan solo $T_{ret} = 30ms$.

En estos esquemas se ha considerado que los medidores, actuadores, acopladores ópticos, lógica CMOS (y TTL) asociada, cables de transmisión y bus no tienen retardo, aunque se sabe que en la práctica no es así.

Como se ha dicho, éste es el esquema más idóneo para microcontroladores con pocos recursos de memoria como los Arduino. El controlador, de tener un watchdog, es preferible que esté implementado como hardware, realizando

una comprobación periódica de que el controlador evoluciona o que responde a una señal.

Otra limitación importante es la asignación de memoria para ubicar objetos dinámicos como los que hemos empleado. Los vectores **`std::vector`**, colas **`std::queue`** y colas iteradas **`std::deque`** dejan de tener sentido, ya que estamos utilizando microcontroladores de muy poca memoria. Por tanto, deberemos utilizar objetos estáticos, simples (arrays C, por ejemplo), con los que la asignación es fija. Si queremos implementar, por ejemplo, un controlador PID sabemos que se necesita una cola de un elemento y otra de dos elementos. Reservamos previamente esa memoria.

La efectividad del control, por tanto, recae en la complejidad del sistema (varios procesos, un proceso, un solo hilo); y en el período de muestreo, siendo los Arduino los más rápidos, mientras que Raspberry Pi y PC son más versátiles.

6.1.2. Muestreo e interrupciones

Las interrupciones son mecanismos de acción asíncrona que recibe el microprocesador a través de un dispositivo hardware y que requiere su atención inmediata. Los microprocesadores permiten recibir las interrupciones en un orden, según su importancia. Por ejemplo, la interrupción de un puerto serie recibiendo una información es menos importante que la interrupción de un evento más grave como es un error de paridad en la memoria.

En el contexto de rutina, una interrupción hace que la rutina abandone la ejecución de sus instrucciones por un momento y sea llamada una subrutina que procese esa interrupción. Por ejemplo, si cambia el nivel de un pin digital de entrada, por disparo de flanco (ascendente o descendente), se entra en la rutina de interrupción y se actualizan estados.

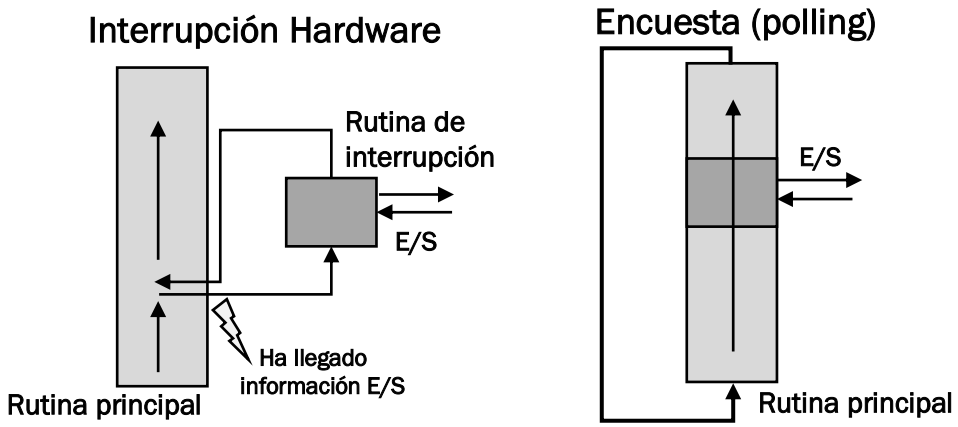


Figura 6.4

Esto llega a ser más eficiente que estar monitorizando el puerto cada cierto tiempo (encuesta, muestreo o *polling*), pues se ahorran instrucciones de lectura innecesarias.

La limitación es que la rutina de tratamiento de interrupción ha de tener el menor número de instrucciones posible, puesto que se está deteniendo el curso normal de la ejecución. Eso tiene un impacto negativo en el rendimiento del sistema.

Hay mecanismos de interrupción tanto en Raspberry como en Arduino. En este último, los puertos asociados con interrupciones cambian según el modelo. Aunque en cuestión de software hemos hecho muestreo, pues los dispositivos eran virtuales, las comunicaciones físicas con el mundo real están asociadas a interrupciones y eventos asíncronos que permiten un mejor rendimiento, principalmente cuando se está gestionando una gran cantidad de datos.

6.2. Control con Raspberry Pi y PC. Discretizadores modificados

Conocemos las características de un PC, que suelen ser más que suficientes para actuar como controladores de procesos, aunque no son los más empleados. Otra elección son los dispositivos Raspberry Pi, con procesadores ARM de buenas prestaciones que soportan la multitarea. También tienen la ventaja de ser dispositivos de placa, simples y con gran conectividad (puertos USB, bus I^2C, ethernet, GPIO). Se ejecutan con una versión de Debian: Raspbian. Debian es el sistema operativo con el que hemos compilado y ensayado el sistema.

Por tanto, estamos ante el dispositivo más idóneo para realizar las dos adaptaciones presentadas en el apartado anterior: El esquema completo; y el esquema de un proceso, sin bus. Los discretizadores, que tan solo realizaban intercambios de datos de un dispositivo virtual a otro, pasan a ser discretizadores reales, puesto que se comunican con dispositivos físicos, a través de puertos de entrada/salida.

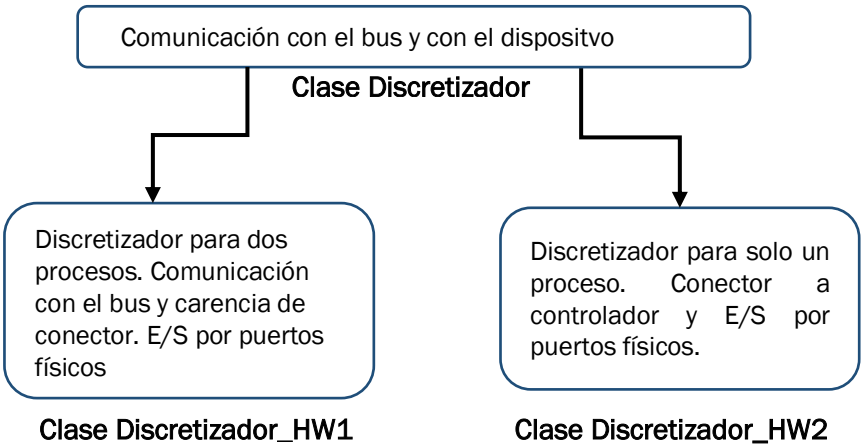


Figura 6.5

Cada tipo de discretizador cuenta, por tanto, con unas peculiaridades que requerirán mínimas modificaciones en la clase discretizador. En el resto, se

utilizará el mecanismo de la herencia (puesto que uno de los discretizadores hace uso del bus y otro del conector con el controlador).

6.2.1. Entrada / Salida

Por lo general, el kernel del sistema operativo gestiona los dispositivos, mediante los controladores de dispositivo y habilita una zona de memoria para el intercambio E/S que en el caso de Linux son archivos, ubicados en diferentes localizaciones.

- **Lectura y escritura directa de los archivos del sistema**. Necesitan que nuestro proceso tenga privilegio de superusuario o root, puesto que estamos utilizando directamente acceso a partes críticas del sistema.
- **Bibliotecas de funciones**. Hacen de intermediarias y proveen de un interfaz adecuado para que el programador construya las rutinas de entrada salida, mediante llamadas a funciones. Algunas no necesitan que el proceso sea root.

En Raspberry se pueden conectar muchas interfaces. La desventaja de los puertos GPIO integrados en la placa es que son puramente digitales y si queremos codificar 256 estados para representar de forma más o menos precisa el voltaje variable de un motor a controlar, necesitamos 8 pines.

Sin embargo, existen más formas de comunicación, adecuadas para dispositivos analógicos. Una de las más interesantes hace uso de I^2C, que es un sistema de bus universal ampliamente utilizado, y en el que se pueden conectar multitud de convertidores A/D, D/A de varios fabricantes. Este bus hace uso de cuatro de los pines de GPIO, mediante comunicación serie.

6.2.2. Discretizador modificado HW1

El control de un dispositivo físico o de planta se puede realizar mediante puertos en el proceso de planta. El controlador es un proceso aparte, aislado, y se comunicaría con la planta mediante bus y cola de mensajes. El elemento más susceptible de ser ampliado y adaptado a una interfaz con el mundo físico es el discretizador. Por tanto, creamos una clase derivada llamada **`Discretizador_HW1`**, con estas propiedades:

- **No tiene conector**. Pero sí tiene bus. Al llamar a la superclase **`Discretizador`** no proporcionará conector alguno. En su defecto, deberá proporcionar una **`esid`** y longitudes de paquete para que siga funcionando el bus asociado normalmente.
- **El conector es sustituido por la interfaz de entrada/salida que se implementará en un nuevo método virtual llamado `hardware()`**. La cadena de actualizaciones llamará a este método en cada iteración y cada clase derivada proporcionará su método, según el contexto de operación.

Las clases derivadas que hagan uso de esta infraestructura deberán incorporar, cuando sea necesario, iniciaciones de los puertos E/S en su constructor y finalizaciones de dichos puertos en el destructor.

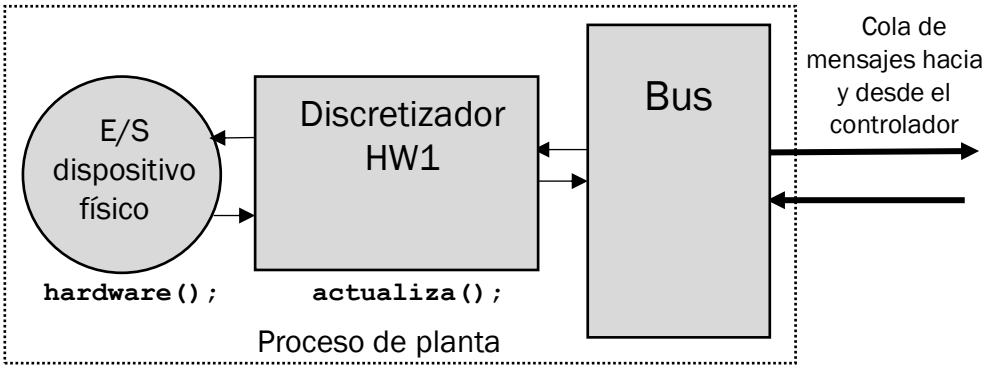


Figura 6.6

Se incorpora una nueva función o método en la clase **`Discretizador_HW1`** que es **`hardware()`**, y con la que se hará la entrada/salida mediante los puertos de comunicación. Esta clase hereda muchas de las características de **`Discretizador`** y no es necesario reescribir código, aunque sí realizar algunas modificaciones en la clase base para que tolere la ausencia de conectores y que capte los datos de conexión (esid, longitudes de paquete) desde la clase derivada.

6.2.3. Discretizador modificado HW2

Si conocemos bien el modo de operar del discretizador de clase HW1, no nos costará comprender el funcionamiento de **`Discretizador_HW2`**.

- **No tiene bus**. Pero sí tiene conector. Por tanto, se puede incorporar en el mismo proceso del controlador. Al llamar a la superclase **`Discretizador`**, no proporcionará bus. El conector ha de tener los mismos parámetros **`esid`** y longitudes de paquete que los suministrados al dispositivo en su constructor.
- **El bus es sustituido por la interfaz de entrada/salida que se implementa en el método virtual `hardware()`**, al igual que **`Discretizador_HW1`**. Esta interfaz depende del contexto y cada clase derivada aportará su método particular.

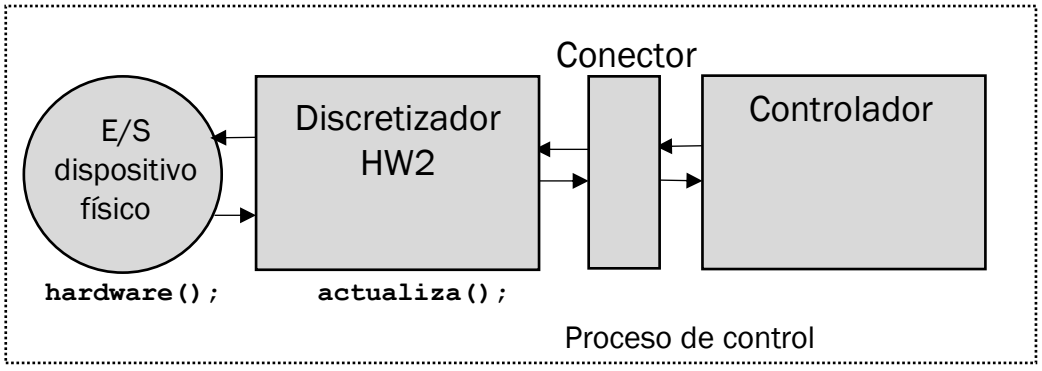


Figura 6.7

Como ocurre en **Discretizador_HW1**, también puede que haya que iniciar los puertos y finalizarlos, así que se puede hacer uso de rutinas específicas en el constructor y destructor. También hay que implementar una rutina **hardware()** para la entrada/salida.

Como se ha apuntado en 6.1.1, este discretizador modificado es más rápido que **Discretizador_HW1**, aunque se hará todo el control en el mismo proceso, con las desventajas que ello pudiera tener.

La clase base **Discretizador** también ha de ser modificada para tolerar la inexistencia del bus, el cual es sustituido por la rutina E/S **hardware()**.

6.2.4. Modificaciones efectuadas en la clase Discretizador y clases derivadas

A lo largo del trabajo se han mostrado las ventajas que proporciona la herencia en la programación orientada a objetos. De la infraestructura de un controlador, por ejemplo, se pueden derivar controladores más específicos sin drásticos cambios en el código. También se ha visto un ejemplo claro con los conectores y los conectores de recogida.

La especialización de un objeto o dispositivo es la clave que justifica la herencia, y esto ocurre con los discretizadores HW.

Tan solo hay que asegurarse de que la clase base pueda configurarse sin necesidad de tener conector y bus simultáneamente. No se hizo virtual la función **asociar()**, pues difiere según la subclase y además no hace falta puntero a **Discretizador** al definir los discretizadores HW.

Las carencias de información (al asociarse el discretizador) las han de suministrar las clases derivadas HW, especialmente la clase HW2, que no tiene conector de referencia, pues que dichas clases conocen de antemano qué datos de entrada/salida van a utilizar para la entrada/salida por puerto.

Clase Discretizador (variables y métodos añadidos)	

Variables	• Esid y longitudes de paquetes (accesibles por las clases derivadas HW1 y HW2)
	`int esid_conexion = 0;`
	`unsigned int lal_bus_conexion = 0;`
	`unsigned int lal_dispositivo_conexion = 0;`

Métodos	• Hardware. Comunicación con el dispositivo físico `virtual void hardware();`

Por supuesto, estos añadidos no afectan las funcionalidades habituales en cuanto a simulación de sistemas. El comportamiento de **Discretizador** con un conector y bus es exactamente el mismo que el descrito en el apartado 3.2.

Clases Discretizador_HW1 y Discretizador_HW2	
Métodos	• Constructor en HW1 `Discretizador_HW1(unsigned int t, int id, int esid, unsigned int laplanta, unsigned int lacontrolador, unsigned int lcola, Bus *bus);`

	• Constructor en HW2
	`Discretizador_HW2(unsigned int t, int id, int esid, unsigned int laplanta, unsigned int lacontrolador);`
	• Asociar en HW1 (sin conector)
	`int asociar();`
	• Asociar en HW2 (con conector)
	`int asociar(Conector *conector);`
	• Actualiza
	`int actualiza();`
	• Método hardware. Entrada/salida de/hacia el dispositivo físico
	`virtual void hardware();`

El constructor de ambas clases HW1 y HW2 necesita también las longitudes de paquete, en el contexto de la rutina hardware.

Son las clases derivadas las que definirán estos paquetes, según **`estructuras_datos.h`** y así poder asociarse con el bus o con el conector, dependiendo del tipo de discretizador. Tan solo faltaría configurar la interfaz de E/S.

6.2.5. Convertidores A/D y D/A por I²C para Raspberry

Raspberry tiene diversas formas de comunicación con otros sistemas.

		Pin no.			
DC Power	3.3V	1	2	5V	DC Power
SDA1, I²C	GPIO 2	3	4	5V	DC Power
SCL1, I²C	GPIO 3	5	6	GND	
GPIO_GCLK	GPIO 4	7	8	GPIO 14	TXD0
	GND	9	10	GPIO 15	RXD0
GPIO_GEN0	GPIO 17	11	12	GPIO 18	GPIO_GEN1
GPIO_GEN2	GPIO 27	13	14	GND	
GPIO_GEN3	GPIO 22	15	16	GPIO 23	GPIO_GEN4
DC Power	3.3V	17	18	GPIO 24	GPIO_GEN5
SPI_MOSI	GPIO 10	19	20	GND	
SPI_MISO	GPIO 9	21	22	GPIO 25	GPIO_GEN6
SPI_CLK	GPIO 11	23	24	GPIO 8	SPI_CE0_N
	GND	25	26	GPIO 7	SPI_CE1_N
I²C ID EEPROM	DNC	27	28	DNC	I²C ID EEPROM
	GPIO 5	29	30	GND	
	GPIO 6	31	32	GPIO 12	
	GPIO 13	33	34	GND	
	GPIO 19	35	36	GPIO 16	
	GPIO 26	37	38	GPIO 20	
	GND	39	40	GPIO 21	

Diagrama de pines de Raspberry PI 3. https://randomnerdtutorials.com/getting-started-with-raspberry-pi/ Figura 6.8

Sin embargo, si queremos hacer un control preciso de un motor con solo la raspberry, por ejemplo, deberíamos recurrir a técnicas como la modulación de ancho de pulso (PWM) para transmitir las consignas al motor. Otra opción sería, por ejemplo, usar 6 bits de salida de GPIO que darían lugar a 64 voltajes diferentes en un convertidor externo digital/analógico.

Para la entrada del tacómetro, también habría que asignar algunos bits digitales en GPIO y realizar la conversión, analógico/digital, con un chip externo.

Se puede facilitar la conexión de Raspberry con los sistemas de planta utilizando placas externas convertidoras y polivalentes que se comunican con el bus I²C, ya que Raspberry reserva los pines 3 y 5 para SDA y SCL de I²C.

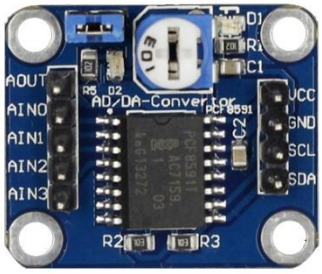


Figura 6.9

PCF8591. Módulo convertidor A/D y D/A de 8 bits con interfaz I²C
http://wiki.sunfounder.cc/index.php?title=PCF8591_8-bit_A/D_and_D/A_converter_Module

Una vez elegido el interfaz, se implementan las llamadas a las funciones en las clases derivadas de **`Discretizador_HW1`** y **`Discretizador_HW2`**, haciendo uso de las bibliotecas adecuadas, que se instalarían en el sistema operativo raspbian y se incluirían en la declaración del código fuente, tal como se muestra en las implementaciones con el control PID del motor de corriente continua **`discretizador_motorcc_hw1.cpp`**, **`discretizador_motorcc_hw2.cpp`**.

```
#include <wiringPi.h>
#include <pcf8591.h>
```

Aun utilizando bibliotecas, para asegurar una comunicación con el hardware, el proceso donde se aloja el discretizador ha de ser ejecutado en root o superusuario.

En los constructores se inicia el hardware.

```
wiringPiSetup(); // Se inicia configuración

pcf8591Setup (PCF, 0x48); // Configurar PCF8591 en el pin 120, y en la dirección 0x48

pinMode (PIN_ALARMA, INPUT); // Configurar pin digital GPIO4 como entrada de alarma
```

Se ha reservado un bit de entrada para la alarma. Si hubiera alarma, el controlador se desactiva.

En las rutinas **`hardware()`** se ubica la adquisición y el envío de datos a GPIO y al convertidor A/D y D/A mediante bus I²C. Con el motor de corriente continua la entrada al controlador es la velocidad angular del motor (analógico codificado en un entero de 8 bits) más la señal de alarma (digital). El controlador responde con el voltaje, que se envía como entero de 8 bits a la alimentación del motor. Esta rutina, por tanto, también realiza las conversiones numéricas. Tomemos como ejemplo el discretizador HW1:

```
 int tac_i = analogRead(PCF + 0); // Se lee puerto analógico de entrada, placa PCF8591
```

```
 // Sabemos sus límites en coma flotante con TAC_MAX y TAC_MIN, por
tanto, convertimos a valor decimal tac_f

float tac_f = TAC_MIN + (TAC_MAX - TAC_MIN) * ((float)tac_i /
(float)ESCALA_ADC);

Discretizador_motorcc_HW1::salida_es_3->s1   =   tac_f;   //   Al
controlador
```

Seguidamente, se lee del controlador el valor del voltaje y se acondiciona para enviarlo al convertidor D/A:

```
float vol_f = Discretizador_motorcc_HW1::entrada_es_3->e1;
 // Comprobamos límites que no se pueden sobrepasar
vol_f = vol_f > MAX_VOL ? MAX_VOL : vol_f;
vol_f = vol_f < MIN_VOL ? MIN_VOL : vol_f;
     // Conversión a entero
vol_f = (vol_f - MIN_VOL) / (MAX_VOL - MIN_VOL);
vol_i = (int)((float)ESCALA_DAC * vol_f);
analogWrite (PCF + 0, vol_i); // Escribimos en el puerto de salida
```

Finalmente, hay que leer la alarma, por si hubiera que alertar al controlador

```
bool alarma = digitalRead(PIN_ALARMA) == HIGH ? true : false;
Discretizador_motorcc_HW1::salida_es_3->alarma = alarma;
```

La rutina de E/S es igual en `Discretizador_HW2`.

6.3. Control con Arduino. El esquema reducido

Se han popularizado estos microcontroladores, ya que permiten una rápida puesta en funcionamiento y reprogramación, además cuentan con gran flexibilidad, diferentes formas de entrada/salida (binarios, convertidores D/A, A/D, etc.) que están integrados en la placa. El entorno de programación C++, con el que transferir las instrucciones al dispositivo, funciona en diversas plataformas.

Es, por tanto, el idóneo para implementar un esquema reducido de nuestros controladores. La simplificación del controlador se justifica por la limitada memoria de trabajo de un Arduino, que es una mínima fracción de la disponible en un PC o una Raspberry. Además, el procesador no soporta la multitarea.

La modificación es tan drástica que nos limitaremos a crear otra jerarquía de clases, en la cual basar nuestro trabajo.

6.3.1. Jerarquía de clases en Arduino

Para nuestra aplicación específica (controlador PID para motor CC) nos servimos de conceptos empleados en capítulos anteriores, pero sin la posibilidad de conexión, bus, grabación, watchdog por software o cambio de flujo de ejecución en tiempo real.

Clase Hardware	

Variables	• Canales analógicos de entrada, salida y consigna del motor
	`unsigned int canalE, canalS, canal_consigna;`
	• Canal digital de la alarma del motor
	`unsigned int canal_alarma`
	• Valores de entrada, salida, consigna y alarma
	`float entrada, salida, consigna;`
	`bool alarma = false;`

Métodos	• Constructor
	`void Hardware(unsigned int canalE, unsigned int canalS, unsigned int canal_consigna, unsigned int canal_alarma);`
	• Actualización (cadena de actualizaciones)
	`int actualiza();`
	• Conocer si hay alarma
	`bool estado_alarma();`

Clase Dinamico	

Variables	• Cuatro colas (arrays) de memoria estática y sus longitudes
	`float cola1[4], cola2[4], cola3[4], cola4[4];`
	`unsigned int longitudes[4]`

Métodos	• Constructor
	`Dinamico(unsigned int canalE, unsigned int canalS, unsigned int canal_consigna, unsigned int canal_alarma, unsigned int longitud1, unsigned int longitud2, unsigned int longitud3, unsigned int longitud4);`
	• Actualización (cadena de actualizaciones)
	`int actualiza();`

Clase Controlador_PID	

Métodos	• Constructor
	`Controlador_PID(unsigned int canalE, unsigned int canalS, unsigned int canal_consigna, unsigned int canal_alarma, float p, float i, float d);`
	• Actualización (cadena de actualizaciones)
	`int actualiza();`

Los Arduino más básicos tienen seis canales analógicos y más de ocho canales digitales.

Se pueden asignar tres canales por controlador PID de motor CC (analógicos), más un canal de alarma (digital) por controlador, siguiendo este esquema:

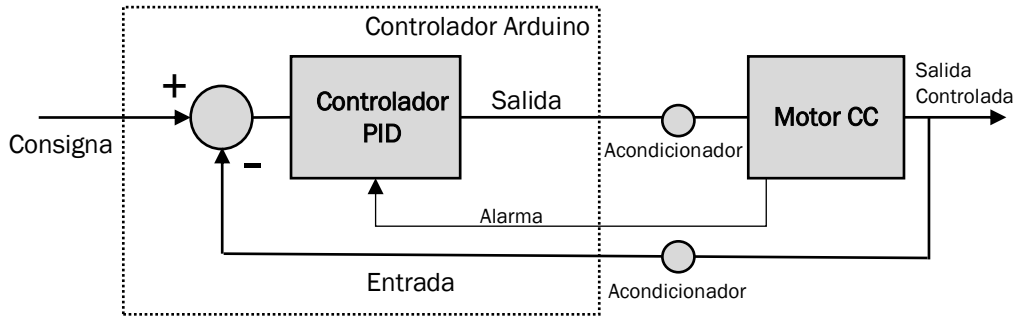


Figura 6.10

Por tanto, se puede controlar como mínimo dos motores con consignas externas, como entradas analógicas, con el mismo microcontrolador.

Las señales de entrada y salida son susceptibles de ser reacondicionadas para convertirse en voltajes válidos, hacia el motor o desde el tacómetro que mide la velocidad de giro.

Arduino dispone de un watchdog por hardware y tan solo hay que configurar algunos registros internos, de tal forma que, si el programa no responde, el dispositivo se reseteará automáticamente.

Otro mecanismo de protección del bucle motor-controlador es la alarma. Se dispone de un detector en el motor (térmico, amperímetro, etc.) que habilitará el estado 1 o **`high`** en el pin de alarma, cuando hubiera algún problema. Si se llega a esta situación el controlador entregará permanentemente una salida nula (voltaje cero), desactivando la fuente de suministro de voltaje del motor dañado.

Obsérvese también la gran simplificación en cuanto a tipos de datos. Ya no se utilizan colas y vectores de la biblioteca STL al resultar muy pesados para un procesador como Atmel y al consumir una notable cantidad de memoria en un sistema embebido de pocos kilobytes de capacidad de almacenamiento.

En su defecto, optamos por estructuras y básicas del lenguaje C como los vectores (arrays) estáticos. Cuando se programe en Arduino, se valora la simplicidad, aunque por supuesto haremos uso de la potencia que proporciona C++.

6.3.2. Compilación con Arduino

Un ejemplo, que tiene como finalidad controlar motores CC, se halla en base_arduino.h y base_arduino.cpp, donde se ha programado el controlador PID, en el directorio /microcontroladores. El programa principal que habilita los controladores y realiza el bucle se halla en el directorio principal del trabajo y es control_motorcc_arduino.cpp.

Sin embargo, el usuario también ha de compilar las bibliotecas necesarias, y ese es el propósito del directorio /microcontroladores/arduino_libcore, que hace uso del paquete arduino-core.

Enlazando todos los elementos anteriores obtenemos un archivo **`.hex`** o volcado binario que se podrá grabar en la EPROM del Arduino.

El código fuente que se acompaña con el trabajo se ha programado y verificado con el sistema operativo Linux Debian, sin haber errores durante la compilación, utilizando los repositorios oficiales de la distribución. No se garantiza que así sea en otras distribuciones o con futuras versiones del paquete Arduino-core, y quizá hubiera que cambiar parámetros como los directorios de los archivos fuente en Makefile, cuando ocurriera algún error.

Los paquetes instalados en Debian para que se compilaran los módulos de Arduino con éxito son los siguientes:

- gcc-avr
- binutils-avr
- arduino-core

La programación en línea de comandos de Arduino tan solo tiene el añadido de tener que compilar la biblioteca **`libcore.a`**. Esta biblioteca se compila y aloja en /microcontroladores/Arduino_libcore.

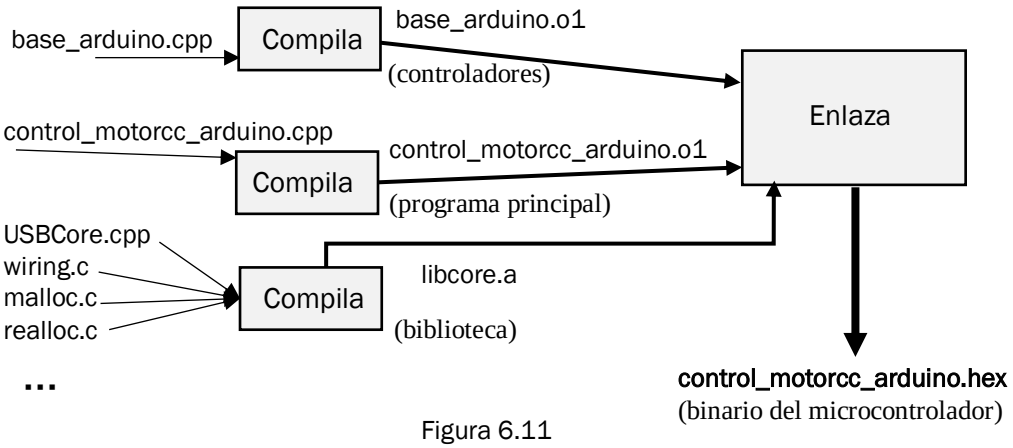


Figura 6.11

El modo de compilación difiere del resto de los ensayos, realizados para PC y Raspberry. Se emplea una variante del compilador gcc llamada avr-gcc. En realidad, estamos empleando un compilador cruzado o *cross compiler* que compila desde una máquina de procesador Intel x86 (o ARM, en Raspberry) y que genera un código ejetutable por un procesador Atmel de 8 bits de arquitectura RISC.

Este guión de compilación especial para los Arduino se refleja en los makefile y tan solo habrá que habilitar flags especiales. Recordemos que estas arquitecturas de sistemas embebidos simples tienen rutinas fijas como **`setup()`** y **`loop()`**, que se pueden ubicar en lugares de la memoria específicos mediante el linker o enlazador.

6.3.3. Funcionamiento del controlador en Arduino

La ejecución, si antes se fundamentaba en hilos, ahora es un ciclo continuo, bucle o **`loop()`** que realiza llamadas de forma constante a los diferentes controladores implementados, hasta que se apaga el dispositivo o éste se detiene. Esto es lo más cerca que se puede estar de la multitarea.

Como iniciación en Arduino está la rutina **`setup()`**, y se ejecuta nada más encender el microcontrolador. Puede servir para preparar adecuadamente el dispositivo, antes de empezar el ciclo **`loop()`**.

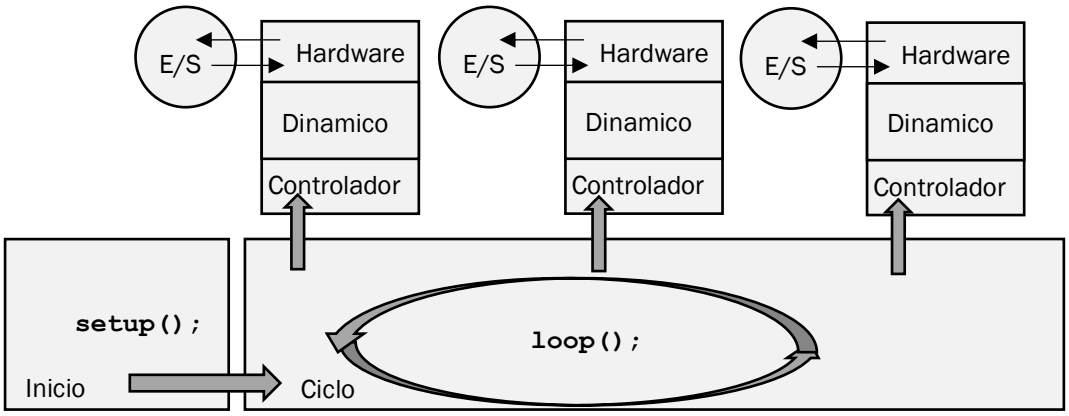


Figura 6.12

Utilizaremos C++ para crear diferentes objetos cuyas cadenas de actualización Hardware → Dinamico → Controlador son llamadas en cada ciclo.

Por ejemplo, si queremos emplear dos controladores (mediante sus cadenas de actualizaciones) tan solo hemos de invocar sus respectivas rutinas en el ciclo **`loop()`** y así funcionarán a la vez.

```
Controlador_PID controlador1(A0, 0, A1, 0, 1.2, 1, 0, 0.1);
Controlador_PID controlador2(A2, 2, A3, 1, 2, 3, 0, 0.01);
void loop() { // Bucle de operación
     controlador1.actualiza();
     controlador2.actualiza();
}
```

6.4. Estándares de comunicación en la industria

Aunque el bus de comunicaciones por cola de mensajería en este trabajo tiene un alcance a nivel del sistema operativo, se puede expandir para utilizar sockets de comunicaciones entre máquinas que podrían estar alejadas entre sí, mediante el protocolo TCP/IP u otros que sean capaces de transmitir mensajes, preferentemente en tiempo real (en procesos críticos o de respuesta rápida).

Es decir, el dispositivo Bus puede ser sustituido por implementaciones más flexibles y potentes que se han consolidado en los entornos industriales, a lo largo de años de experiencia y mejora.

Una referencia base a tomar es la pirámide de la automatización, donde se reflejan los diferentes niveles de comunicación que existe en el ámbito industrial y empresarial, en relación con los procesos que se gestionan.

Pirámide de la automatización http://victor-fuzzylogic.blogspot.com/2013/10/buses-de-campo.html Figura 6.13

Hasta ahora, nos hemos movido por los niveles inferiores, que conforman el nivel de campo y de célula. Por una parte, el nivel de campo satisface las comunicaciones -en tiempo real- entre innumerables dispositivos actuadores, detectores y automatismos básicos que conforman cualquier entorno industrial; por otra, los controladores propuestos cubren el nivel de célula, realizando en la medida de lo posible un acopio de información que es enviada a los niveles superiores para ser analizados.

- **Nivel de campo**. Sensores, actuadores, sistemas básicos de control que no toman decisiones y envían/reciben señales continuamente.
- **Nivel de célula**. Controladores PID, control numérico, robots, con toma de decisiones sobre la marcha, a muy corto plazo.
- **Nivel de proceso (SCADA)**. Se tiene en cuenta la información y se toman decisiones en un determinado proceso, como podría ser el calentado, fundido, moldeado y templado de una pieza de acero; o un proceso de depuración de aguas residuales, que requiere varias etapas (filtrado, decantación, tratamiento aerobio, tratamiento anaerobio, etc.)
- **Nivel de planta**. La acumulación de información se vuelve importante y se toman decisiones globales que afectan a los procesos, en plazos medios.
- **Nivel corporativo**. Decisiones a largo plazo y planificación empresarial.

Todo ello conlleva un tratamiento de la información entre los dispositivos, en virtud del nivel de información y decisión, pues no es lo mismo comandar un montón de dispositivos pequeños desde una célula (a semejanza del bus planta-controlador que se ha programado en este trabajo) que reflejar el estado de un proceso en un panel SCADA, o filtrar los datos para realizar estadísticas en un contexto contable.

6.4.1. OPC Foundation

OPC Foundation satisface la informatización de la industria desde una perspectiva cliente-servidor. A mediados de los años 90 varias compañías definieron una línea de actuación que hiciera uso de las tecnologías informáticas emergentes como *Plug and Play*, y así evitar los problemas de instalación de drivers para los módulos hardware de entrada/salida.

A su vez, también definió un protocolo para la comunicación de dispositivos muy dispares entre sí (como podría ser un termopar y un motor eléctrico de corriente continua). Se posibilitaba la compartición de datos en capas abstractas de controladores, actuadores, sensores, etc. y se utilizaba el modelo cliente-servidor para comunicar todos estos dispositivos con servidores centrales que hicieran de nodos de conexión.

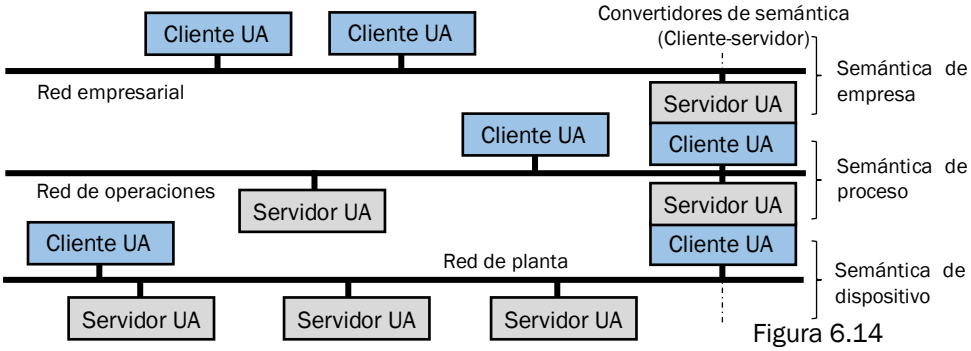


Figura 6.14

El resultado es una red de gran flexibilidad, versatilidad, escalabilidad, con un estándar que salva las diferencias entre fabricantes a la hora de diseñar el hardware.

En OPC Foundation hay varias líneas de actuación:

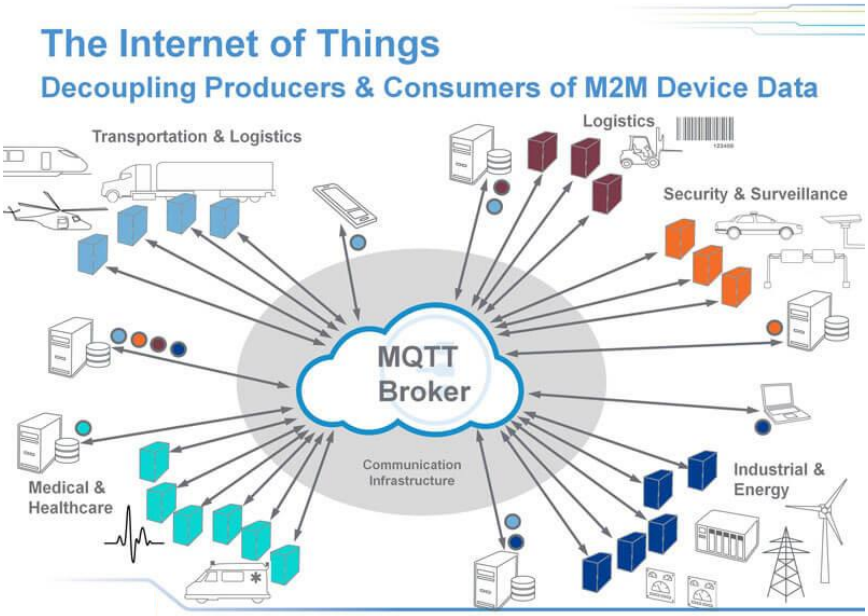
- **Acceso a datos (OPC-DA)**, como línea inicial, en el que se usa el concepto *Plug And Play*, además del modelo Cliente-Servidor.
- **Alarmas y eventos (OPC-A&E)**. Gestión de eventos asíncronos y alarmas.
- **Acceso a datos históricos (OPC-HDA)**, donde se almacenan los datos de interés que ayudan a analizar el comportamiento y evolución de un proceso o factoría, susceptibles a tratamiento estadístico y toma de decisiones.

La interfaz de OPC ha cambiado con el tiempo. Tras un tiempo funcionando, los desarrolladores se percataron de que OPC era demasiado dependiente de la tecnología DCOM de Windows y difícil de configurar en aspectos de telecomunicación. Esto dio lugar a una evolución con tecnología propia llamada OPC-DA, que es capaz de satisfacer la demanda clásica de los sistemas de control, y también permite una gestión en tiempo real.

Actualmente OPC permite que nos suscribamos a los elementos del sistema que nos interesan, nos permite definir elementos de seguridad, control, en un contexto propio de desarrollo, independiente de los requisitos propios del hardware y del sistema operativo.

6.4.2. MQTT

MQTT (Message Queuing Telemetry Transport) utiliza un modelo similar de suscripción de elementos dispares, mediante el modelo Cliente-Servidor; con gran economía de la información que se transmite, ahorrando gran cantidad de ancho de banda y de energía. El enfoque propuesto por MQTT es el Internet de las Cosas donde puede convivir grandes dispositivos con aparatos portátiles, los cuales se pueden comunicar con una estación llamada "broker", y que actúa a modo de servidor central.



MQTT e Internet de las Cosas.
https://aprendiendoarduino.wordpress.com/tag/mqtt-client/ Figura 6.15

El broker gestiona las publicaciones y suscripciones de cada dispositivo (dependiendo de si son de entrada o salida), mediante canales jerarquizados. De esta forma se consigue filtrar la información de forma efectiva, según el propósito de la aplicación.

7. Conclusiones

En el trascurso de este trabajo se han desvelado bastantes herramientas del lenguaje C++ que proporcionan utilidad en la Ingeniería de Sistemas, como estructuras de datos avanzadas y clases que definen código y datos de todos los elementos implicados, tanto dispositivos físicos como conectores, grabadores, controladores, buses, watchdogs y generadores numéricos; facilitando su creación, su mantenimiento, su depuración y su posible ampliación.

También se han aprovechado las características de los procesadores actuales multitarea, gracias al estándar C++11, que permite la creación de hilos, su gestión, su transferencia y su comunicación, gracias a clases como **`std::thread`**, **`std::mutex`**, **`std::future`**, entre otras.

El resultado ha sido la creación de procesos e hilos de planta y controladores que se comunican, cooperan, son autónomos y que resuelven diferentes situaciones. También pueden ser empleados de forma didáctica como introducción del lenguaje C++ en esta área de la ingeniería, tanto en la simulación de procesos industriales como en control de sistemas reales, con microcontroladores embebidos de uso común.

7.1. ¿Qué nos aporta el lenguaje C++ en el control de sistemas?

Es evidente que el paso necesario para aplicar C++ en la disciplina del control de sistemas es conocer el lenguaje en sí. Es un esfuerzo natural, si se parte de C, y se piensa en un superconjunto de instrucciones que trasciende el paradigma de programación procedimental al que muchas personas están acostumbradas.

De los procedimientos a los objetos

Las rutinas y su clasificación en bibliotecas ha sido clave para disponer de un orden creciente de complejidad que ha resuelto satisfactoriamente muchos problemas, en la historia de la informática. Como se ha expuesto en 2.1, podríamos partir de operaciones elementales como podría ser la suma aritmética y particularizamos, realizando operaciones cada vez más complejas como la resta, multiplicación, división, raíz cuadrada, o cálculo de logaritmos.

A partir de la rutina de dibujo de una línea también podemos realizar figuras más complejas como triángulos, cuadrados, polígonos, e incluso polígonos de “muchos lados”, que se aproximan a las circunferencias.

Esto nos permitiría confeccionar el programa que hemos ido desarrollando a lo largo de estas páginas, con tan solo usar lenguaje procedimental, sin

embargo, gastaríamos mucho esfuerzo, y estaríamos expuestos a errores de programación, sobre todo a la hora de identificar las rutinas, sus parámetros y el manejo de las variables en memoria. Puede llegar a ser un asunto complicado, sobre todo si hemos de gestionar decenas de entidades diferentes.

Utilizando la programación orientada a objetos nos aseguramos de que cada objeto utiliza una "plantilla" de clase, con procedimientos y memoria asociada a ese objeto en exclusiva, siendo tarea del programador permitir que los datos se puedan compartir entre objetos. De esta forma se asegura un orden e integridad de datos y rutinas.

Si partimos de una clase llamada "bicicleta", podemos almacenar datos como su ubicación geográfica, velocidad, estado de sus neumáticos, velocidad de giro de las ruedas, etc. Para cada bicicleta que declaremos (como objeto) y también procedimientos para éstas como puede ser pedalear, frenar, encender la luz de noche.

Si tenemos otra clase llamada "motorcc" (hemos definido precisamente una clase así que modela un motor; intensidad, velocidad angular, voltaje de entrada, etc.) podríamos ir más allá y acoplarlo a nuestra bicicleta, creando una clase derivada nueva llamada "bicicleta_motorizada", en la que, además de las propiedades de una bicicleta clásica, tenemos las propiedades de un motor eléctrico, que permitirá que el ciclista no tenga que pedalear siempre.

Esta nueva forma de trabajar con código y datos nos va a ahorrar quebraderos de cabeza. Ahora, más centrados en la Ingeniería de Sistemas, pensemos en entidades generales de clase **`Dispositivo`**, que usaremos en un entorno industrial (con estados, como alarma por mal funcionamiento) y con características discretas, como frecuencia de muestreo. A partir de ahí crearemos tipos de dispositivos como motores eléctricos, controladores PID, conectores para transmitir datos o grabadores para guardar los datos.

Es este salto esencial entre el procedimiento y el objeto lo que diferencia C y C++ y nos permite una creación más intuitiva de programas de cierta complejidad, como el propuesto en este trabajo.

Del proceso de un solo hilo a la multitarea

Imaginemos qué ocurriría si realizásemos el programa con un solo hilo de ejecución. Tendríamos que realizar llamadas periódicas a cada entidad desde una rutina que gestiona las demás tareas, con la posibilidad de fallos si ampliamos el programa y conectamos más entidades o dispositivos.

La gestión de ejecución y ocupación de CPU de cada rutina también sería complicada. Además, no es lo que ocurriría en un sistema real. En una situación cotidiana tenemos un motor, un amortiguador y un controlador PID

funcionando independientemente, aunque estén intercambiando datos entre ellos. La idea, al crear la clase base **`Dispositivo`**, ha sido dotarlo de un hilo propio de ejecución, y rutinas asociadas para arrancarlo, pausarlo o detenerlo. El trabajo pesado sería realizado por el sistema operativo y el microprocesador.

El programador de sistemas tan solo se ha de ocupar de las entidades y su relación, sabiendo que éstas funcionarán de una forma autónoma.

Esto también añade responsabilidades, tal como se ha visto en 3.1.3. Se ha de evitar corrupción de datos y funcionamientos anómalos debido al *data race* que se produce cuando dos hilos escriben simultáneamente en la misma zona de memoria, sin un arbitraje o cierta sincronización. Para ello se utilizan datos atómicos **`std::atomic`** y mecanismos de exclusión mutua **`std::mutex`**.

La delegación del control de hilos también puede ser posible, por ejemplo, entre controlador y watchdog. Si ocurriera un error en el controlador, el watchdog cerraría hilo del controlador defectuoso, impidiendo que afecte al rendimiento del sistema.

Estructuras avanzadas de datos. Biblioteca STL

Con C se suelen manejar vectores y listas, en las que frecuentemente el usuario ha de programar rutinas que realicen operaciones sobre estos. También llega a requerir tiempo de programación, sobre todo si se utiliza asignación dinámica de memoria.

Los contenedores STL y las estructuras avanzadas están integradas en C++, con su correspondiente biblioteca. Gracias a ellos creamos vectores **`std::vector`** muy fácilmente, accediendo a sus elementos uno a uno. También podemos añadir elementos, o borrarlos.

Se han utilizado también colas simples para las grabadoras **`std::queue`** y colas indexadas **`std::deque`**, para los dispositivos dinámicos que almacenan en la memoria los estados anteriores del dispositivo.

Se ha constatado una mejora utilizando las cadenas de caracteres, con un contenedor propio **`std::string`**, con métodos que permiten concatenar cadenas, buscar texto, sustituirlo, etc.

Todos estos contenedores y plantillas hacen uso de la memoria dinámica y han resultado de gran utilidad para agilizar la escritura del programa.

Encapsulamiento, herencia y polimorfismo

El encapsulamiento permite proteger los datos y hacer accesibles los que nosotros elijamos. La herencia permite reaprovechar características de los objetos para crear otros nuevos. Por ejemplo, habíamos creado los conectores

con la clase **`Conector`**, surgió la posibilidad de ampliar sus características dotando a los conectores de comunicación con los dispositivos grabadores, mediante capturas regulares de sus buffers.

Se puede ir ampliando el proyecto reutilizando convenientemente las piezas ya creadas, y sin los inconvenientes de la programación procedimental, ya que además de heredarse código, se heredan estructuras de datos.

El polimorfismo es también un concepto muy ligado a la realidad, en el sentido de que si queremos averiguar las propiedades de un dispositivo cualquiera, se puede utilizar el mismo método, llamado **`propiedades()`**, declarado como virtual, y por tanto personalizable para cada clase. Aunque utilicemos un puntero a la clase base. C++ averiguará a qué clase nos estamos refiriendo.

No lo hemos aplicado en este trabajo, pero el polimorfismo también se puede aplicar en operaciones, de tal forma que internamente el programa puede detectar cuándo estamos sumando números escalares, o vectores, y elegir el algoritmo adecuado, según el contexto de la variable de entrada.

El encapsulamiento, la herencia y el polimorfismo son tres características esenciales del C++ que nos permitirán escribir programas claros, extensibles y fáciles de revisar para ser depurados, en caso de errores. Es por ello que C++ es mucho más adecuado en proyectos grandes, a pesar de que el programador ha de hacer un esfuerzo inicial generalizando las clases que va a utilizar. Después puede disponer de una jerarquía adecuada que cubra sus necesidades.

Control detallado de los procesos de control y simulación

Una vez que hemos declarado los objetos, se pueden dar de alta en un menú, en un conector, en un bus, con gran flexibilidad, fiabilidad y escalabilidad, de acuerdo con el manifiesto de los sistemas reactivos. Eso es garantía de funcionamiento del sistema, independientemente de lo grande que sea éste.

Dotar a los procesos de planta y controlador de menús interactivos permite que el usuario cambie parámetros en tiempo real y que actúe sobre los hilos de ejecución, desde la misma consola de comandos.

También tendremos la posibilidad de registrar la actividad de los dispositivos físicos y controladores, grabando la evolución de sus variables internas. Gracias a modificaciones en los conectores, al uso de grabadores, como dispositivos que recopilan datos mediante colas, y al uso de los métodos de E/S para archivos en C++, completamos el estudio de los sistemas, recibiendo la información necesaria para modificarlos, mejorar los modelos físicos, y controladores.

C++ también es adecuado en microcontroladores

La evolución de un lenguaje no lo hace necesariamente más complejo, ni crea programas más pesados, sino que facilita la tarea de programación. Aunque C++ sea adecuado en grandes proyectos, también se ha adoptado en los microcontroladores Arduino, pudiéndose utilizar sus características notables como encapsulamiento, herencia y polimorfismo, para generar rutinas de unos pocos KB de peso. Esto permite elegir qué controlador queremos utilizar de nuestra biblioteca, modificarlo, y compilarlo al instante.

Los lenguajes de propósito general basados en objetos aportan grandes ventajas en el campo de la Ingeniería de Sistemas y el control. C++ es un lenguaje maduro, consolidado y que está siendo revisado regularmente para añadirle nuevas características.

7.2. Líneas futuras

El trabajo no se detiene aquí. De hecho, el lector puede que se plantee realizar mejoras, o quizá haya creído mejor acometer el diseño de los sistemas de otra forma, con otro tipo de dispositivos, con otras clases, una vez conocidas las herramientas de C++ para la creación de sistemas y controladores.

Tan solo se ha presentado una propuesta que demuestra la capacidad de C++, y que expone sus posibilidades, de una forma quizá más didáctica que aplicada. Por supuesto, también cabe la posibilidad de emplear los controladores con los discretizadores HW1 y HW2, o el esquema de Arduino, para realizar un control automático en dispositivos específicos. Las posibilidades son amplias.

A pesar de que se considera que se ha abarcado muchas propiedades del C++, fácilmente accesibles para quien ha utilizado como punto de partida C y quiere profundizar en la programación orientada a objetos y al multiproceso, se pueden alcanzar metas más ambiciosas, partiendo de los controladores propuestos, o con el fin de simular sistemas físicos, en etapas de diseño y planificación.

- **Interfaz gráfica**. La única interfaz gráfica que se ha utilizado es GNU Plot, a partir de los archivos de salida. Se realiza de una forma no interactiva. Utilizando convenientemente las rutinas y clases de cualquier marco de trabajo (framework) para ventanas gráficas. Por ejemplo, se puede utilizar en Windows el célebre C++ Builder; y en Linux, las herramientas proporcionadas por GTK+ o QT.
- **Comunicación entre procesos mediante sockets**. Hemos empleado la cola de mensajes estándar del sistema UNIX, pero existen más formas de comunicación, como pueden ser tuberías (pipes), para flujos de información; o mediante sockets de internet, si queremos, por ejemplo, controlar una planta utilizando TCP/IP, desde el ordenador de mando. Tan solo hay que modificar la clase Bus y añadirle subclases especializadas que cambien la

rutina de comunicación; ya sea en un simple punto a punto, o en redes distribuidas, según el contexto del Internet de las Cosas (IoT).

- **Análisis del sistema en tiempo real**. Hemos visto que las grabadoras van almacenando en la memoria la evolución del sistema, interceptando los datos en los conectores de recogida. Después, al detener su hilo, lo vuelcan todo a un archivo. Con esta mejora, se podría visualizar en pantalla lo que lleva captando la grabadora y tomar las medidas correctivas oportunas durante los ensayos que se realicen.
- **Flexibilización de las simulaciones y paso variable**. Hemos visto en un modelo que el paso determina la convergencia de la simulación. En esencia, los modelos no lineales tienen mejor comportamiento en unos puntos que en otros, por eso se utiliza paso variable. También nos hemos restringido a integraciones numéricas por el método de Euler y Runge Kutta de cuarto orden, ambos de un solo paso, los cuales solo analizan el estado anterior para calcular el siguiente. De una forma similar, y variando las rutinas de la clase `Dinamico` y `Continuo`, se pueden escribir más algoritmos de integración que sean multipaso, como los métodos de Adams-Bashforth y Adams-Moulton.
- **Controladores en cascada**. Algunos controladores utilizados en este trabajo replican su salida de control en otra línea que no se ha utilizado. Esta característica se podría utilizar de forma ventajosa para realizar control en bucles anidados, o para procesos en línea con varias variables.
- **Mejora de las conexiones entre dispositivos**. Por el momento, se han conectado dispositivos de dos en dos, pero se podrían integrar conexiones con operaciones básicas, como, por ejemplo, habilitar un restador entre tres dispositivos (señal del sensor, consigna y controlador) que haga la operación matemática y haga a su vez de conector.

Como se puede ver, se pueden hacer todo tipo de mejoras. Una vez que se tiene una base en la programación C++ de controladores, es fácil expandir la aplicación y utilizar las características base para crear nuevas características.

Bibliografía

Bonnet, S. (27 de Febrero de 2015). *Digital Dimension*. Obtenido de Digital Dimension: http://www.digitaldimension.solutions/es/blog-es/opinion-de-expertos/2015/02/mqtt-un-protocolo-especifico-para-el-internet-de-las-cosas/

Burden, R., & Faires, J. (1998). *Análisis numérico.* México D.F.: Thompson editores.

Butcher, J. C. (2003). *Numerical Methods for Ordinary Differential Equations.* New Yotk: John Wiley & Sons.

Colaboradores del Proyecto Arduino. (s.f.). *Arduino - Language Reference*. Obtenido de Arduino: https://www.arduino.cc/reference/en/

Colaboradores del Proyecto Raspberry. (s.f.). *Raspberry Pi Documentation*. Obtenido de Raspberry Pi Official Site: https://www.raspberrypi.org/documentation/

de Burgos Román, J. (2008). *Cálculo infinitesimal de varias variables.* McGraw-Hill Interamericana de España S.L.

Elektronika, A. –L. (4 de Octubre de 2016). *OPC: Desde el clásico al nuevo OPC-UA*. Obtenido de OPC: Desde el clásico al nuevo OPC-UA: https://larraioz.com/articulos/opc-desde-el-clasico-al-nuevo-opc-ua

Francisco García Mora, Jorge Sierra Acosta, María Virginia Guzmán Ibarra. (2007). *Sistemas de simulación para administración e ingeniería.* México D.F.: Grupo Editorial Patria.

Joyanes Aguilar, L., Sánchez García , L., & Zahonero Martínez, I. (2007). *Estructura de datos en C++.* Aravaca, Madrid: McGraw-Hill Interamericana.

Kormanyos, C. M. (2013). *Real-Time C++.* Berlin.

Ogata, K. (1995). *Sistemas de control en tiempo discreto.* México: Prentice-Hall Hispanoamericana S.A.

Ogata, K. (2003). *Ingeniería de control moderna.* Madrid: Pearson Educación S.A.

Phillips, C. L. (1987). *Sistemas de control digital.* New Jersey: Prentice-Hall.

Stroustrup, B. (2002). *El lenguaje de programación C++. Edición especial.* Madrid: Addison Wesley.

Stroustrup, B. (2014). *A Tour of C++.* Addison-Wesley.

Tojeiro Calaza, G. (2014). *Taller de Arduino.* Barcelona: Marcombo. Ediciones técnicas.

White, F. M. (2009). *Fluid Mechanics* (7th ed.). New York: Mcgraw-Hill series in mechanical engineering.

Williams, A. (2012). *Concurrency in action. Practical multithreading.* New York: Manning Publications Co. .

Referencias en figuras y ecuaciones

Figuras

Fig. 6.8 - Diagrama de pines Raspberry PI 3. https://randomnerdtutorials.com/getting-started-with-raspberry-pi/

Fig. 6.9 - PCF8591. Módulo convertidor A/D y D/A de 8 bits con interfaz I^2C. http://wiki.sunfounder.cc/index.php?title=PCF8591_8-bit_A/D_and_D/A_converter_Module

Fig 6.13 - Pirámide de la automatización. http://victor-fuzzylogic.blogspot.com/2013/10/buses-de-campo.html

Fig 6.15 - MQTT e Internet de las Cosas. https://aprendiendoarduino.wordpress.com/tag/mqtt-client/

El resto de figuras han sido elaboradas por el autor de este trabajo.

Ecuaciones

Ec. (2.1). Cálculo de una raíz cuadrada. Se aplica la fórmula del método de Newton-Raphson. (Burden & Faires, 1998).

Ec. (2.2). Expresión de una combinación lineal con parámetros λ y μ.

Ec. (2.3) y (2.4). Método congruencial aditivo con varias semillas para la generación de una secuencia de números pseudoaleatorios (Francisco García Mora, Jorge Sierra Acosta, María Virginia Guzmán Ibarra, 2007).

Ec. (2.5), (2.6), (2.7), (2.8), (2.9). Sistema de ecuaciones diferenciales, método de Euler y de Runge Kutta orden cuatro. (Burden & Faires, 1998).

Ec. (2.10), (2.11). Modelo físico de un motor eléctrico acoplado a una masa, con rozamiento en el eje.

Ec. (2.12), (2.13). Modelo físico de un amortiguador de vehículo de doble masa (chasis y rueda).

Para más referencias sobre los modelos físicos se puede consultar (Ogata, Ingeniería de control moderna, 2003) y (Phillips, 1987).

Los modelos físicos utilizan los siguientes principios: En electricidad, leyes de Kirchhoff; en mecánica, leyes de Newton (equilibrio de fuerzas); en hidrodinámica, conservación de la masa y de la energía.

Ec. (2.14). Relación entre velocidad lineal y angular, relación entre desplazamiento lineal y angular; **Ec. (2.15), (2.16), (2.17)**. Modificación del modelo físico del motor eléctrico, adaptado a un carro de desplazamiento lineal.

Ec. (2.18), Ec. (2.19). Principio de la conservación de la masa en fluidos y ecuación de la conservación de la energía de Bernoulli en fluidos (White, 2009); **Ec. (2.20), (2.22), (2.23), (2.26)**. Modelo del depósito de agua con perfil definido en **Ec. (2.21)**; **Ec. (2.24), (2.25)** Linealización de una ecuación diferencial mediante Teorema de Taylor de varias variables (de Burgos Román, 2008); **Ec. (2.27), (2.28).** Criterio de estabilidad en resolución de ecuaciones numéricas (Butcher, 2003); **Ec. (2.29), (2.30), (2.31)**. Determinación del paso mínimo en el modelo del depósito.

Ec. (4.1), (4.2), (4.6). Propiedades fundamentales de la Transformada Z (Ogata, Sistemas de control en tiempo discreto, 1995).

Ec. (4.4), (4.5), (4.7), (4.8), (4.9), (4.10), (4.11), (4.12), (4.13), (4.14), (4.15), (4.16), (4.17). Cálculo de la ecuación en diferencias de un filtro PID, con integración trapezoidal (Phillips, 1987).

Anexo I. Jerarquía de clases

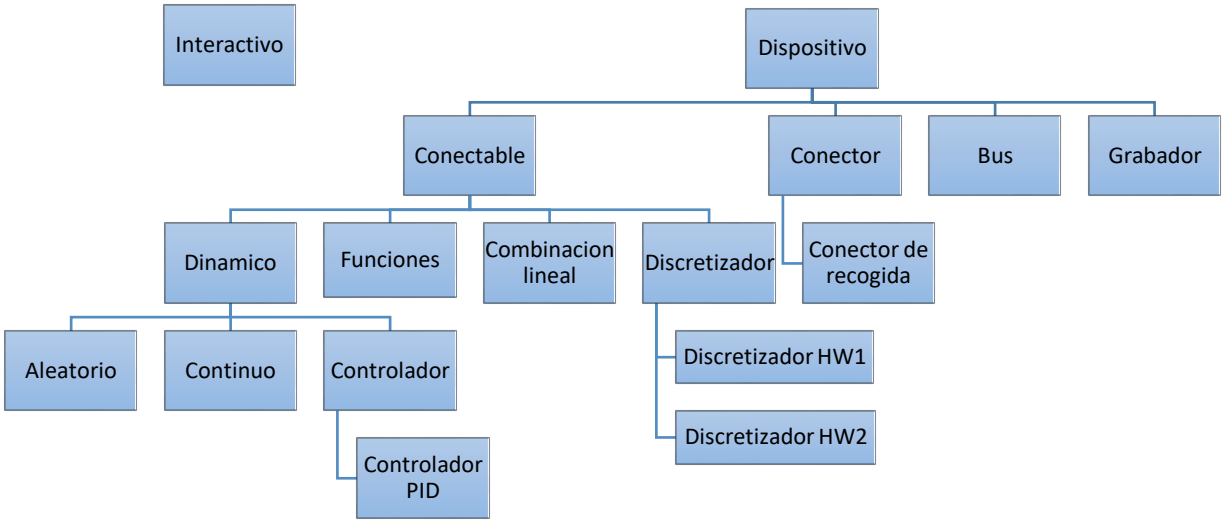


Figura 8.1

De **`Continuo`**, **`Controlador`**, y **`Controlador_PID`** derivan estas clases que son los ejemplos de implementación propuestos:

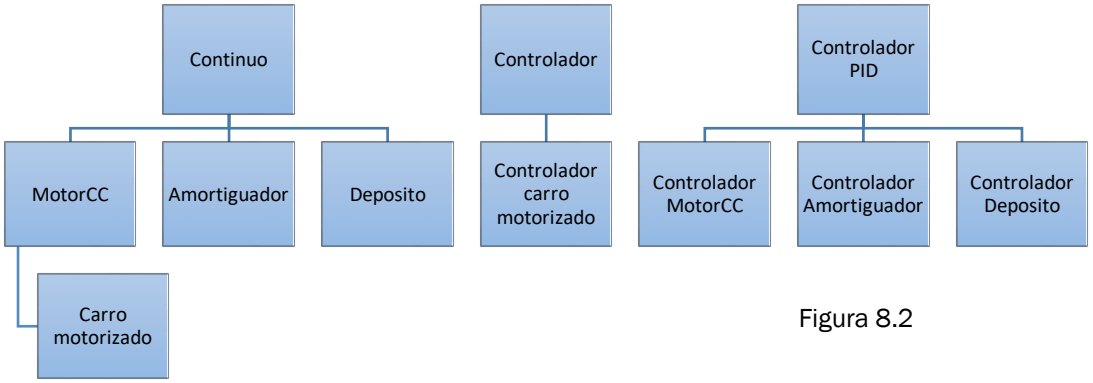


Figura 8.2

La clase **`Interactivo`** es independiente. El resto de clases son derivadas de **`Dispositivo`**.

Existen clases con un estrecho vínculo. Por lo general una necesita de otra para desarrollar una tarea. Así lo hemos visto a lo largo de los capítulos.

Clases relacionadas		
Tarea	**Clase 1**	**Clase 2**
Conexión de dispositivos	Conectable	Conector
Conexión de procesos	Discretizador	Bus
Control	Controlador	Watchdog
Grabación	Grabador	Conector de recogida

Jerarquía de clases en los discretizadores HW1 y HW2. Controlador Arduino y el esquema reducido

- Todas las clases derivadas de **Discretizador_HW1** y **Discretizador_HW2** implementan sus propios métodos hardware, constructores y destructores, con la finalidad de ajustar los interfaces de E/S (registros, relojes, etc.). Los ejemplos suministrados (haciendo uso de la estructura de datos **`esid 3`** del motor CC) aportan comentarios que facilitan la comprensión de esas rutinas y el usuario es libre de implementarlas según el contexto, sea un equipo de escritorio (PC) o Raspberry.

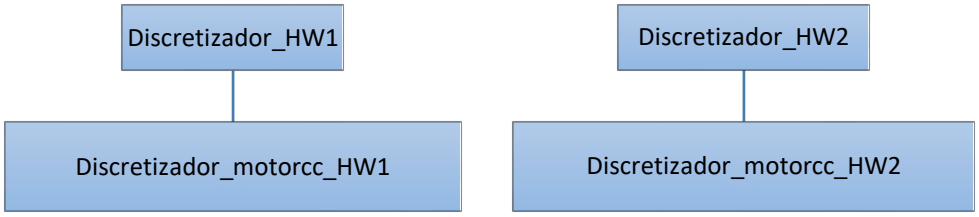


Figura 8.3

- **Arduino** utiliza un procesador básico AVR Atmel, de arquitectura RISC 8 bits y muy poca memoria RAM. El programa de control se introduce "quemando" o "flasheando" la EPROM. La notable redución de prestaciones en la CPU nos obliga a utilizar una jerarquía de clases diferente y con muchas menos funcionalidades. Con la siguiente disposición jerárquica se procura disponer de almacenamiento de estados (Dinamico), para ser utilizado por toda la diversidad de controladores que puede albergar un controlador, incluyendo PID.

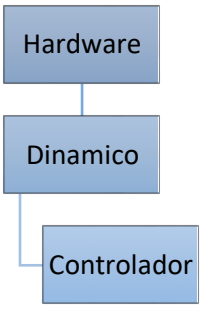


Figura 8.4

A pesar de las limitaciones de la CPU, los microcontroladores Arduino tienen excelentes puertos de comunicaciones y una colección muy interesante de temporizadores y convertidores A/D y D/A en la misma placa que les permiten un control preciso de los procesos industriales.

Anexo II. Entorno de compilación, archivos fuente y depuración

El programa ha sido escrito en Windows utilizando el editor gratuito de código fuente multiarchivo Notepad++. Sin embargo, el entorno de compilación, ejecución y depuración es Linux, concretamente la distribución Debian 9, virtualizado con VirtualBox de Oracle, de descarga también gratuita.

Si el usuario no está familiarizado con Linux, es recomendable que sepa instalar el sistema operativo en un disco duro virtual o en una partición, también operar con algunos comandos básicos para la gestión y ejecución de archivos en shell. También ha de conocer aspectos de la máquina virtual y sobre todo el intercambio de archivos entre la zona de trabajo en Linux y la zona de trabajo en Windows, gracias a la extensión de VirtualBox llamada *Guest Additions*. Así será más cómo trabajar con el editor y el compilador.

En Linux (Debian) hemos de instalar los paquetes **`build-essential`**, **`avr-gcc`**, **`avr-binutils`** y **`arduino-core`**. Así dispondremos de la colección de compiladores (PC y Raspberry); incluyendo el *cross compiler* y las bibliotecas base para las placas Atmel de Arduino. Es muy recomendable instalar también el depurador **`gdb`**, que nos facilitará la labor en cuanto salta una excepción (muy común es *segmentation fault*) y queremos saber qué instrucción la provocó.

Se ha preparado el guión de compilación o makefile para compilar los modelos propuestos en el trabajo. Si queremos compilar todo, ejecutamos **`make`** en el directorio principal y se realizará la compilación automáticamente.

Con **`make clean`**, borramos todos los archivos objeto y los ejecutables. Es recomendable compilar de cero, principalmente si se hacen grandes cambios en el código fuente o si se cambia de máquina *host*.

El trabajo se reparte en tres directorios, según la función de los archivos:

Directorios del código fuente	
Directorio	**Tipos de archivo que contiene**
/ (directorio raiz)	• Archivos base • Ensayos con plantas y controladores
/sistemas (directorio de sistemas)	• Dispositivos de planta • Dispositivos generadores de señales • Dispositivos matemáticos • Dispositivos controladores
/microcontroladores (directorio de microcontroladores)	• Discretizadores de motor CC tipo HW1 y HW2 • Controlador Arduino

Lista de archivos fuente	
Archivos base en /	
base.h	Declaración de todas las clases base del trabajo, excepto sistemas de planta y controladores.
base.cpp	Implementación de las clase base Dispositivo, Dinamico, Continuo.
conectable.cpp	Implementación de la clase Conectable
conector.cpp	Implementación de la clase Conector y Conector_rec.
discretizador.cpp	Implementación de la clase Discretizador.
bus.cpp	Implementación de la clase Bus.
grabador.cpp	Implementación de la clase Grabador.
controlador.cpp	Implementación de la clase Controlador y Contolador_PID.
watchdog.cpp	Implementación de la clase Watchdog.
interactivo.cpp	Implementación de la clase Interactivo.
Generadores y funciones matemáticas en /sistemas	
constante.h	Declaración de la clase Constante.
constante.cpp	Implementación de Constante, generador de números constantes.
funciones_periodicas.h	Declaración de la clase Periodica.
funciones_periodicas.cpp	Implementación de la clase Periodica. Generador de funciones periódicas (triangulares y cuadradas).
aleatorio.h	Declaración de la clase Aleatorio.
aleatorio.cpp	Implementación de la clase Aleatorio. Generador de números aleatorios.
combinación_lineal.h	Declaración de la clase Combinacion.
combinación_lineal.cpp	Implementación de la clase Combinacion. Combinación lineal de dos entradas.
Modelos de dispositivos de planta en /sistemas	
motorcc.h	Declaración de la clase Motorcc.
motorcc.cpp	Implementación del motor de corriente continua.
amortiguador.h	Declaración de la clase Amortiguador.
amortiguador.cpp	Implementación de la suspensión de dos masas con actuador.
carro_motorizado.h	Declaración de la clase Carro.
carro_motorizado.cpp	Implementación del carro motorizado de dos detectores en los extremos.
deposito.h	Declaración de la clase Deposito.
deposito.cpp	Implementación del depósito de agua con caudal de entrada y sumidero de salida regulable.
Controladores en /sistemas	
controlador_motorcc.h	Declaración de la clase Control_motor.
controlador_motorcc.cpp	Controlador PID motor corriente continua.
controlador_amortiguador.h	Declaración de la clase Control_amortiguador.
controlador_amortiguador.cpp	Controlador PID suspensión.
controlador_carro_motorizado.h	Declaración de la clase Control_carro.
controlador_carro_motorizado.cpp	Controlador específico del carro motorizado. Parada temporizada en los extremos.
controlador_deposito.h	Declaración de la clase Control_deposito.
controlador_deposito.cpp	Control PID del depósito de agua con límite en la salida de caudal.
Ensayos de proceso de control y planta en /	
planta_motorcc.cpp	Planta del motor corriente continua.
control_motorcc.cpp	Control PID con consigna del motor de corriente contina.

planta_amort.cpp	Planta de la suspensión de dos masas.
control_amort.cpp	Control PID de la suspensión de dos masas.
planta_motamo.cpp	Planta de motorCC y suspensión combinados.
control_motamo.cpp	Control de motorCC y suspensión combinados.
planta_carro.cpp	Planta del carro motorizado.
control_carro.cpp	Control del carro motorizado con consigna de voltaje (velocidad) y tiempo de espera.
planta_deposito.cpp	Planta del depósito de agua con caudal de entrada y sumidero.
control_deposito.cpp	Control PID del depósito de agua y salida de caudal limitada (saturación).
planta_motorcc_solo.cpp	Ensayo del motor de corriente continua, sin controlador.
planta_carro_solo.cpp	Ensayo del carro motorizado, sin controlador.
planta_deposito_solo.cpp	Ensayo del depósito de agua, sin controlador.
Discretizadores HW1, HW2 y control Arduino en /microcontroladores	
discretizador_motorcc_hw1.h	Cabecera del discretizador sin conector HW1 para motor CC
discretizador_motorcc_hw1.cpp	Implementación en código del discretizador sin conector HW1 para motor CC
discretizador_motorcc_hw2.h	Cabecera del discretizador sin bus HW2 para motor CC
discretizador_motorcc_hw2.cpp	Implementación en código del discretizador sin bus HW2 para motor CC
motorcc_arduino.h	Cabecera de jerarquía C++ para el controlador Arduino
motorcc_arduino.cpp	Implementación para el controlador Arduino
Plantas y controladores HW1, HW2 y Arduino en /	
planta_motorcc_hw1.cpp	Planta para el motor CC. Implementación HW1
control_motorcc_hw1.cpp	Control para el motor CC. Implementación HW1
control_motorcc_hw2.cpp	Control para el motor CC. Implementación HW2
control_motorcc_arduino.cpp	Control para Arduino. Generaría el archivo binario .hex para la EPROM del microcontrolador

Objetivos de la compilación

El fichero **`Makefile`** tiene todas las instrucciones para compilar cualquiera de los ensayos. Si queremos compilar únicamente planta y controlador del depósito escribiríamos **`make planta_deposito control_deposito`**.

Hay dos fases en el proceso de compilación que no tiene que seguir necesariamente un orden fijo, pues se basa en la disponibilidad de los archivos objeto y las marcas de tiempo de los archivos fuente. Por una parte, cada archivo fuente es convertido a código objeto. Finalmente, se crean los ejecutables mediante en enlazador o *linker*, a partir de los archivos objeto.

Depuración

Esta etapa es fundamental en el desarrollo de cualquier software y en la que se emplea una parte importante del tiempo.

- Prevención

Inicialmente, es recomendable atender a las indicaciones del compilador, principalmente cuando éste indica mensajes de aviso. Se utiliza la bandera

-Wall en la entrada del compilador por este propósito. Con ello hacemos que el compilador sea meticuloso al avisar. Es frecuente cometer errores como:

```
if(contador = cuenta_maxima) break; // Finalizar bucle
```

El compilador nos indicará, mediante aviso (sin error) que estamos haciendo una asignación dentro del condicional, y que no usamos el operador **==**. Es decir, en ejecución saldremos del bucle (**break**) si la asignación es distinta de cero y no cuando el contador sea igual a la cuenta máxima. Esto producirá fallos en el programa, en ocasiones muy difíciles de detectar.

Gracias a los avisos, se indican fallos como éste (aunque no siempre ocurre así, sobre todo en expresiones complicadas); también se indican posibles punteros nulos en uso, variables que no se utilizan, variables no inicializadas, etc. y así, de esta forma preventiva, evitamos gastar tiempo depurando.

- Errores lógicos

Aunque puede resultar farragoso corregir los errores sintácticos que nos indica el compilador, los errores lógicos son, sin duda, mucho más difíciles de detectar, una vez que el compilador ha dado su visto bueno. El resultado es un mal funcionamiento del programa debido a algún descuido del programador, por realizar mal un bucle, no actualizar debidamente una variable y un largo etcétera.

Existen varios métodos que permiten al programador revisar si las variables evolucionan según lo previsto.

❖ Depurador gdb. Se hace imprescindible compilar con la opción **-g** para ello. Tan solo hay que añadirla en la variable **CCFLAGS** de los tres makefile presentes en el directorio principal, /sistemas y /microcontroladores.

Con ello, ejecutamos **gdb** escribiendo **gdb archivo_ejecutable** y hacemos uso de las diferentes opciones de ejecución paso a paso, *breakpoints* (detención en el inicio de una función) con **b funcion**, y visualizamos el valor instantáneo de la variable con **p variable**. Muchas más características se pueden encontrar en la documentación de GNU GDB.

❖ Añadir **std::cout** en diferentes partes del código susceptibles a tener los fallos lógicos, y que nos indican el valor de una o varias variables por pantalla, en tiempo de ejecución. Es un método más sencillo si no se quiere utilizar gdb.

- Excepciones

Son interrupciones del programa por eventos importantes, generalmente imposibles de revertir si no se detectan en el código y el programa no actúa, por lo que el resultado es la salida súbita del programa con algún mensaje de error. Utilizando vectores y arrays es común la excepción *segmentation fault* o

violación de acceso, pues el programa está accediendo/escribiendo en memoria que no le fue asignada.

Es interesante incluir las estructuras **try – catch** explicadas en 4.2 en lugares susceptibles a tener una excepción, como podría ser la lectura de un **std::future** no asignado. El programa gestiona la excepción, se da cuenta del error y actúa en consecuencia, sin que éste se interrumpa.

Sin embargo, en este apartado nos serviremos del depurador **gdb**. Es aconsejable -únicamente para este propósito- compilar con **-g** y así crear las marcas necesarias que localizarán mejor el lugar donde ocurrió el error.

La función **Dinamico::configura** crea el vector de colas para los dispositivos dinámicos. Al crear las colas, las rellena de cero mediante un bucle que depende del tamaño del vector **size()**.

```
for (i=0;i<v.size();i++)
```

Imaginemos que nos equivocamos, recorriendo el bucle un elemento que está fuera de los límites del vector:

```
for (i=0;i<v.size()+1;i++)
```

Compilamos, ejecutamos, por ejemplo **./planta_motorcc**. El error, nada más iniciar es el previsible:

```
Violación de segmento
```

La pregunta inmediata que se hace el programador es saber dónde ha ocurrido esta excepción. Nos servimos de **gdb** y ejecutamos **gdb planta_motorcc**.

Una vez dentro ejecutamos el comando **run**, en el que corre el programa de forma controlada. La ejecución se para debido a la excepción. Nos muestra este mensaje con información complementaria:

```
Program received signal SIGSEGV, Segmentation fault.
std::__fill_a<float*, float> (__value=@0xbffff05c: 0, __last=<optimized
out>, __first=0x0) at /usr/include/c++/6/bits/stl_algobase.h:696
696  *__first = __tmp;
```

Es un mensaje bastante críptico, pero que nos indica que algo ocurrió al hacer un relleno (**fill**) en una zona de memoria, mientras trascurría un altoritmo STL. Si aún no sabemos qué está ocurriendo, ejecutamos **backtrace** y así conocemos los pasos previos que dieron lugar a la excepción:

```
#3 std::deque<float, std::allocator<float> >::_M_fill_assign
(__val=@0xbffff05c: 0, __n=<optimized out>, this=<optimized out>) at
/usr/include/c++/6/bits/stl_deque.h:1930
#4  std::deque<float, std::allocator<float> >::assign
(__val=@0xbffff05c: 0, __n=<optimized out>, this=<optimized out>) at
/usr/include/c++/6/bits/stl_deque.h:1101
#5  Dinamico<float>::configura (this=0x43cb10, v=std::vector of length
3, capacity 3 = {...}) at base.cpp:283
```

```
 #6  0x0040ab64 in Continuo<float>::configura
(this=0x43cb10,a=std::vector of length 3, capacity 3 = {...},
b=std::vector of length 3, capacity 3 = {...})at base.cpp:374
 #7  0x004295bf in MotorCC::configura(float, float, float, float, float,
float, float, float, float, float, float, float) ()
 #8  0x00429968 in MotorCC::MotorCC(unsigned int, int, float, float, float,
float, float, float, float, float, float, float, float, float) ()
 #9  0x00402c58 in main () at planta_motorcc.cpp:25
```

La información proporcionada es mucho más precisa, puesto que además de la función **`Dinamico<float>::configura`**, nos localiza la instrucción en la línea **`base.cpp:283`** que es la inicialización de colas dobles rellenas de ceros con el método **`assign()`**, donde ha ocurrido la excepción. Por tanto, hemos de revisar los límites del bucle (que hemos modificado), o las asignaciones previas.

- Seguimiento en ejecución

Si instalamos paquetes como **`top`** ó **`htop`**, se puede hacer un seguimiento de los procesos y de los hilos que hay abiertos y en ejecución. Si se pausa un hilo, seguirá mostrándose, aunque sin actividad (sin gasto de CPU). Al finalizar el hilo, queda eliminado de la lista.

En la siguiente pantalla tenemos planta y controlador del motor de corriente continua, ejecutándose cada uno en un terminal diferente, dentro de la misma máquina. En un tercer terminal, abrimos **`htop`** para visualizar los hilos:

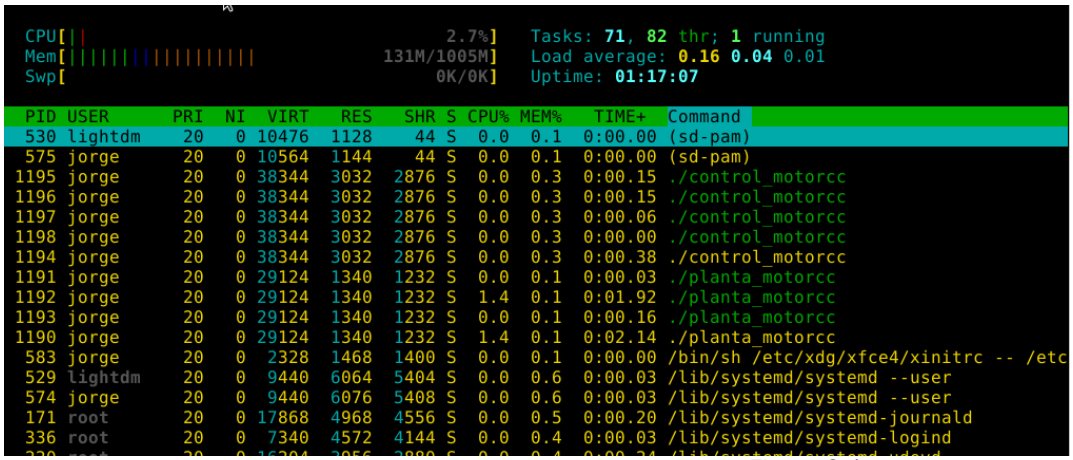


Figura 9.1

Cuando se crea un hilo, a éste se le asigna un código PID y depende del proceso principal (planta o control). Sin mucho indagar, vemos que un hilo en planta está consumiendo una cantidad de CPU mayor que los demás (mostrados como `0.0` por tener gasto mínimo). Es probablemente el hilo con el dispositivo de planta, realizando constantes cálculos del sistema de ecuaciones diferenciales, para simular el motor físico de corriente continua.

Los demás hilos representan grabadores, muestreadores, dispositivos generadores, controlador, watchdog, etc. Cada uno con su PID, asignación de memoria y nombre del proceso principal (mostrados en verde).

Instrucciones complementarias de compilación

Raspberry

Se siguen, exactamente, los mismos pasos anteriormente descritos y la operatividad de los multihilos es completa en su procesador ARM, sin modificación del código fuente. Si se quiere usar GPIO para E/S (I^2C + pines digitales) hay que instalar el paquete **`WiringPi`**. Después, simplemente, hay que adaptar las rutinas de hardware, ya sea para la modalidad HW1 o HW2. En PC (tarjeta de adquisición de datos) dependeremos del modelo de la tarjeta.

Arduino

Aunque no se ha reflejado en el listado de directorios, Arduino usa un directorio auxiliar para crear la biblioteca **`libcore.a`**, a partir de código fuente del proyecto Arduino. Esto se realiza en /microcontroladores/Arduino_libcore.

Antes de empezar...

Se recomienda tener el sistema preparado con los paquetes **`build-essential, avr-gcc, avr-binutils y arduino-core`** (o equivalentes, si no trabajamos con Debian), y ejecutar **`make clean`** en el directorio principal para eliminar los objetos que se crearon en este trabajo, así los reconstruimos de nuevo con **`make`**. Hay que tener en cuenta que la compilación es la más adecuada en la máquina donde se van a ejecutar los sistemas de planta y los controladores.

Si no se instalan los paquetes sugeridos, la compilación fallará en algún punto.

El trabajo ha sido probado, sin errores, con la versión Linux Debian 9.6 Stretch, 32 bits, que es la más actualizada, a fecha de diciembre de 2018.

También ha sido compilado y probado, sin errores, en una Raspberry, con sistema operativo Raspbian.

Por supuesto, también se pueden utilizar máquinas y sistemas operativos de 64 bits, sin modificación de código.

En futuras versiones del sistema operativo podría cambiar el paquete **`arduino-core`**, por lo que hay que tener previstos estos cambios y realizar las modificaciones oportunas, en el guion de compilación o en algún archivo fuente del controlador de Arduino.

Esta información para la compilación se halla también en el archivo **README**.

Anexo III. Ensayos

Este apartado muestra algunos ensayos efectuados con los modelos físicos y controladores.

- <u>Retardos en el bucle de control y frecuencia de muestreo limitada</u>

Pueden llegar a ser decisivos en cualquier sistema automático.

Los retardos admisibles suelen estar en función de la velocidad de los cambios en la planta a controlar. De forma similar, según el Teorema del muestreo de Nyquist, si se necesita una respuesta rápida de alta frecuencia ω_1, es inviable muestrear con una frecuencia $\omega_2 < \omega_1/2$. Tomemos ejemplo del sistema de suspensiones controlado por PID y variemos la frecuencia de muestreo del discretizador de la suspensión.

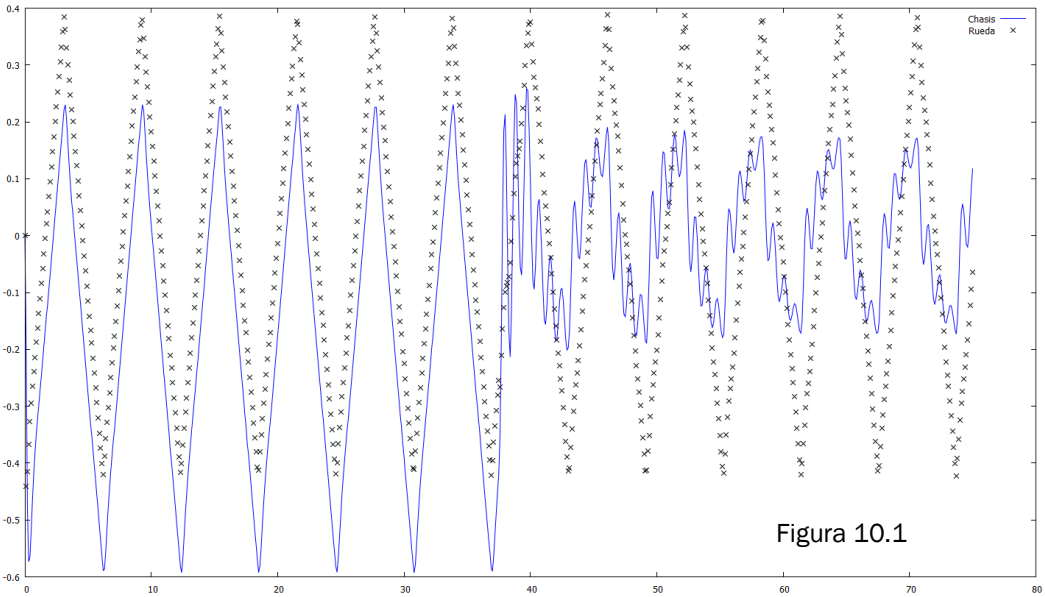


Figura 10.1

A los 40 segundos es cuando iniciamos el controlador PID con:

$P = 19190, I = 20320, D = 0.$

Tanto los discretizadores como el controlador están a 50 Hz, mientras que la suspensión se simula a 500Hz.

La figura refleja que una vez que se pone en marcha el controlador (tardando algunos ciclos de discretizador para que el bus haga handshake), éste comienza la regulación del actuador hidráulico, de tal forma que si la calzada tiene un contorno triangular, ésta se recorta drásticamente. Hay que tener en cuenta que la consigna se ha establecido en cero, aunque lo habitual es tenerla en una cantidad algo menor que cero ya que, en reposo, la masa suspendida, que es elevada, comprime el resorte de la suspensión y su altura baja.

Disminuyamos la frecuencia de muestreo de los discretizadores a 20 Hz.

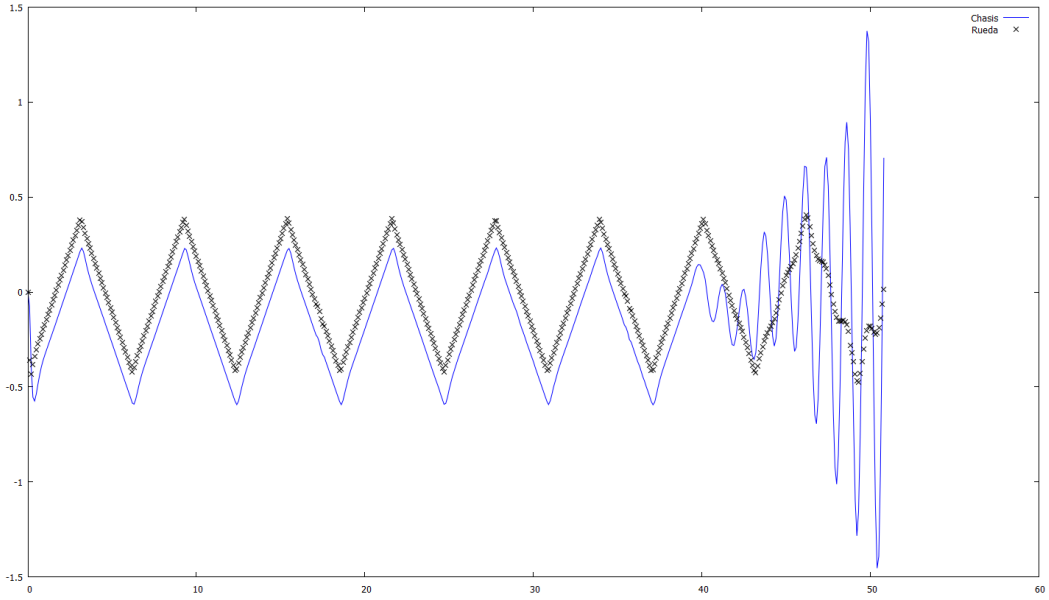


Figura 10.2

El controlador se ha activado, como en la ocasión anterior, a los 40 segundos, con los mismos parámetros PID y misma frecuencia de controlador. Al ser la frecuencia de muestreo tan baja, el controlador no logra responder adecuadamente a los cambios de la altura de calzada, principalmente cuando éstos son bruscos (picos y valles de la señal triangular) y el sistema es inestable, con variables de estado que tienden a infinito.

Con límites sería mucho más realista, ya que el actuador hidráulico no suministra siempre la fuerza que deseamos. Si la fuerza máxima es de $10000\ N$, entonces:

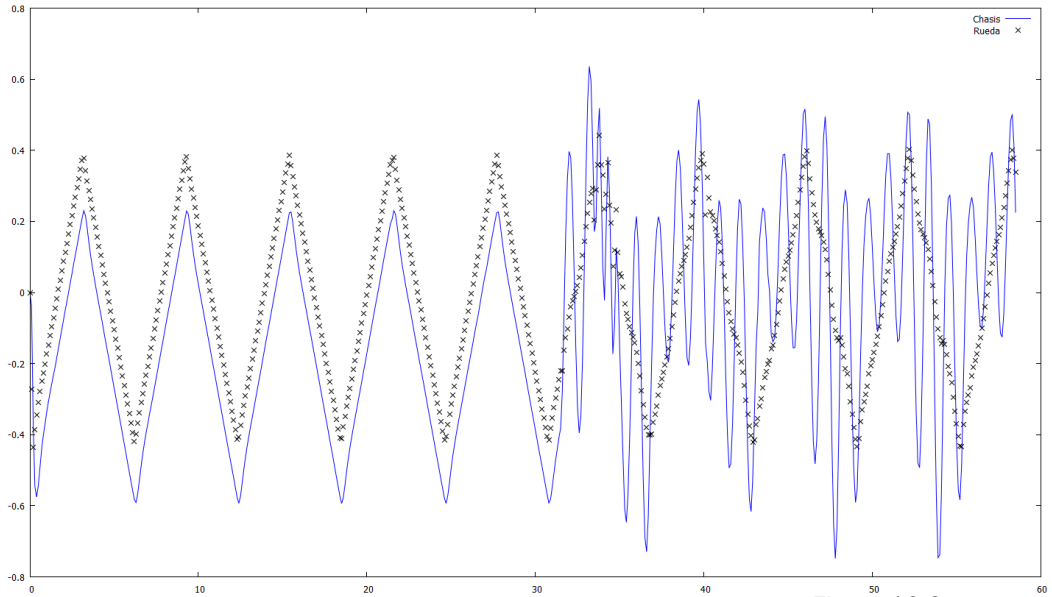


Figura 10.3

La respuesta está en unos límites finitos, pero continúa siendo inadecuada.

La medida del actuador en el conector sobrepasa el valor máximo de $10000\ N$ predeterminado. El controlador puede ser reprogramado para que no consigne estos valores, pero se ha preferido poner el límite en el dispositivo de planta.

- Control del depósito de agua

Aunque la limitación en el control está en que el caudal es siempre positivo y no puede exceder de un valor, se puede realizar el control de una forma razonable, incluso con bajas frecuencias de muestreo. Al tratarse de un depósito de grandes dimensiones, se entiende que el nivel del agua no puede experimentar cambios de nivel de agua significativos en poco tiempo.

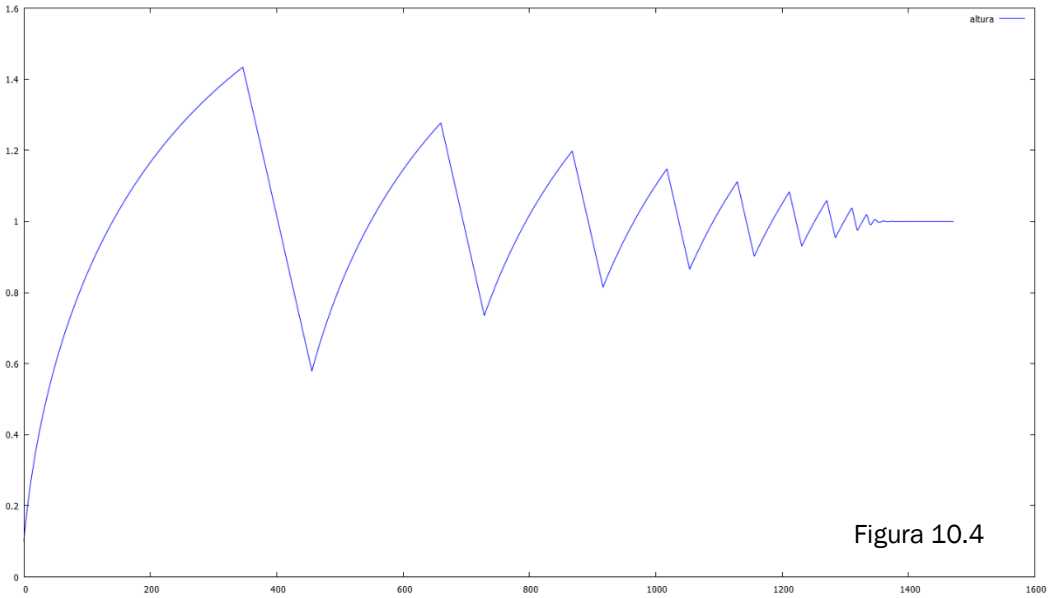


Figura 10.4

Los dos caudales (entrada y salida) tienen esta gráfica:

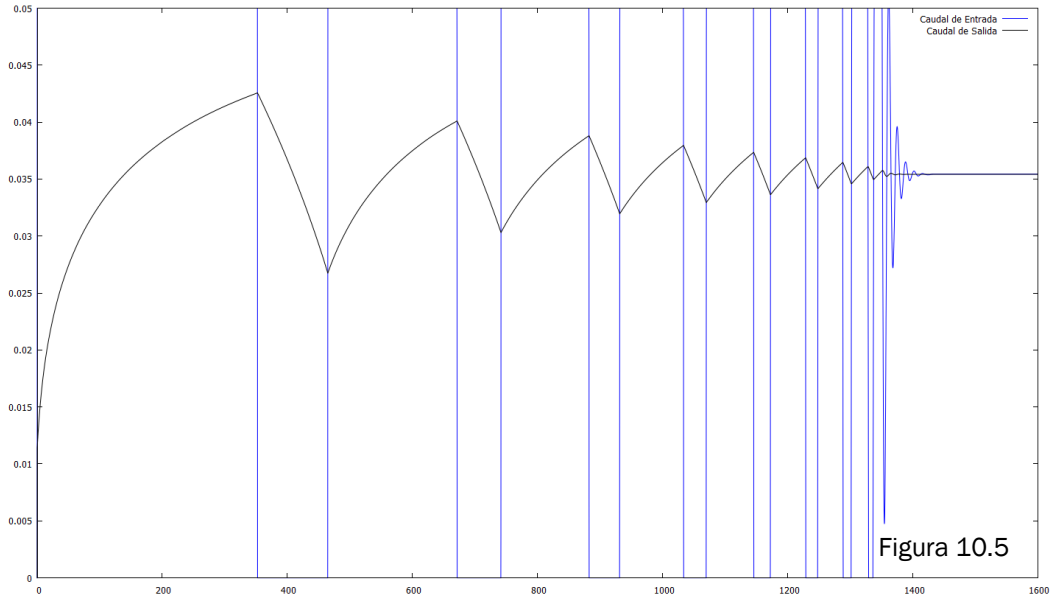


Figura 10.5

Se ha eliminado cualquier componente estocástica, siendo el coeficiente de descarga una constante (modificable para crear escalones).

El aspecto de estas gráficas, con oscilaciones abruptas se puede entender por la limitación del caudal de entrada, que no excede los $50\ L/s$. Ambos caudales tienden a ser de $\sim 35\ L/s$ El controlador PID tiene como parámetros $P = 1, I = 1, D = 0$; y se le ha dado la consigna de 1 metro de altura, la cual se consigue a los 1400 segundos de operación.

El llenado del depósito es irregular, debido al límite máximo de caudal impuesto (restricción) y a que tampoco se puede restar caudal de entrada.

Una vez estabilizada la altura y ambos caudales (régimen estacionario) se puede ensayar el lazo por medio de la constante de descarga, a semejanza de una válvula o grifo. Si cerrásemos la válvula la constante sería cero y se obtendría este transitorio:

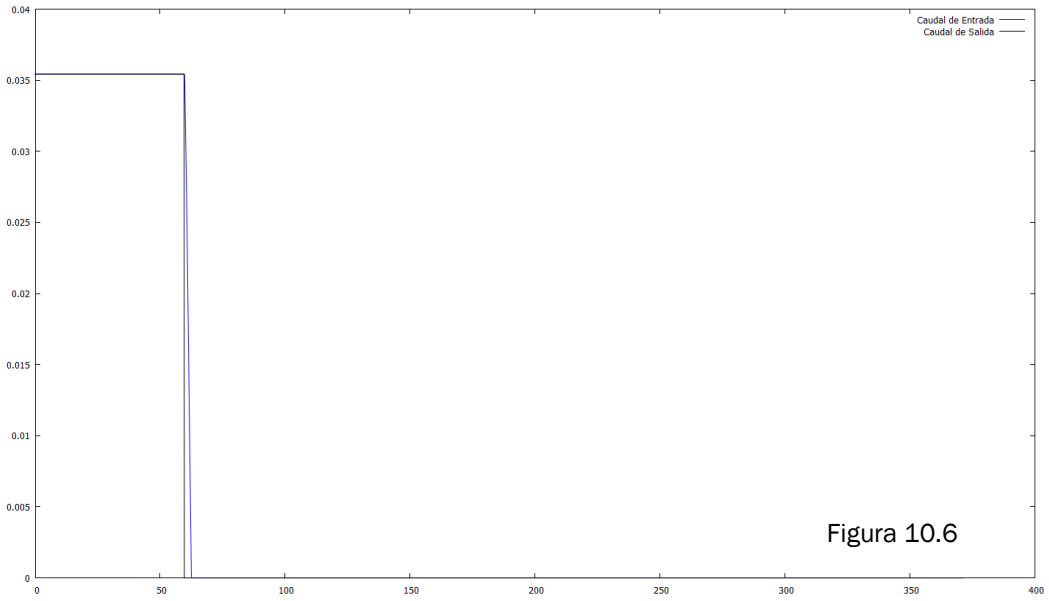


Figura 10.6

Y la altura crece algunos centímetros, en lo que tarda el caudal de entrada en ser cero.

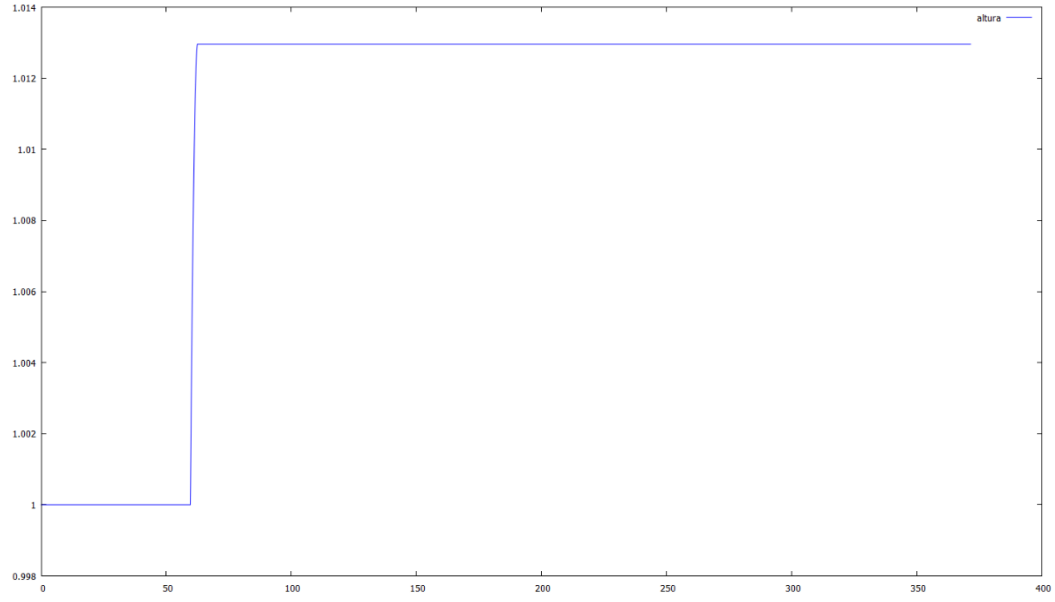


Figura 10.7

No se puede acercar a la altura de consigna porque no hay forma posible de vaciar el depósito desde el controlador (hemos cerrado nosotros el depósito con la válvula).

Con ello, el integrador del controlador está almacenando ese error y podría tener una respuesta inadecuada si se alarga en el tiempo el cierre, a no ser que se vuelva a abrir la válvula o se cambie el valor de consigna. Internamente, también se puede programar el controlador para que no integre cuando el caudal de salida es cero y la altura sea mayor que la consignada.

Si volvemos a abrir la válvula, de nuevo obtenemos un transitorio que tardará un tiempo en alcanzar el estacionario. En nuestro caso, aproximadamente 10 minutos.

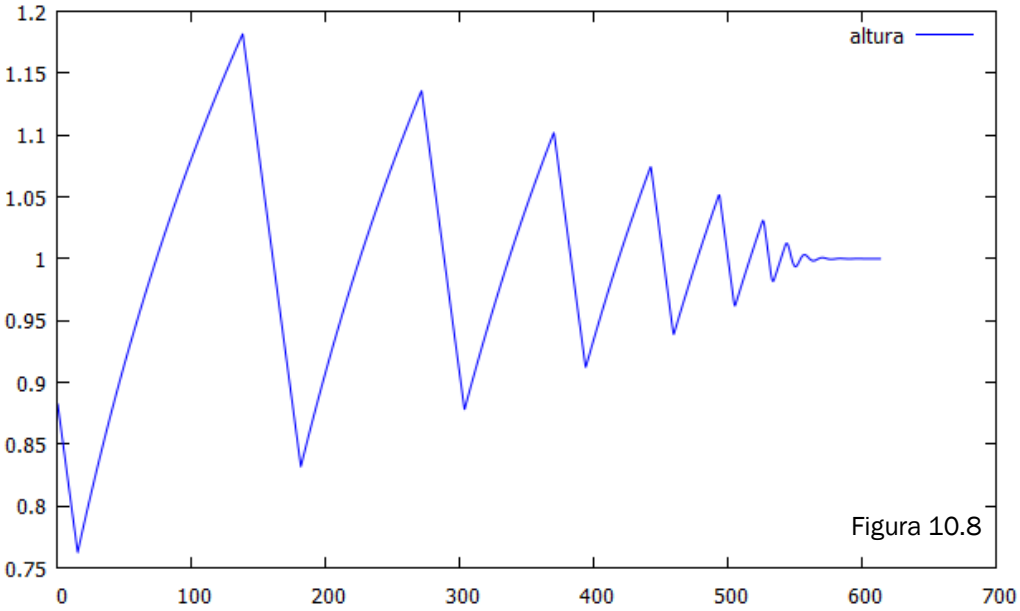


Figura 10.8

Se puede también ensayar el vaciado del depósito y comprobar qué ocurre en la zona de inestabilidad del modelo matemático. Como se ha calculado en 2.5.5, esa zona está, aproximadamente, por debajo de 10^{-7} metros, para un paso de integración de $p = 0.01$.

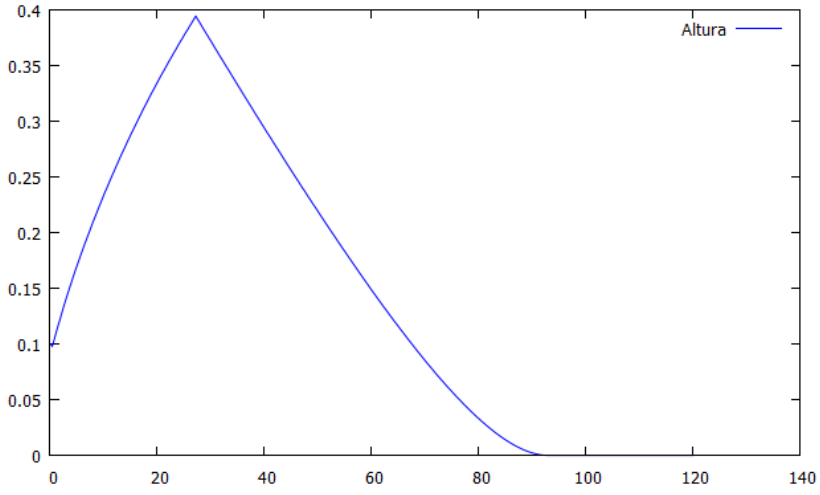


Figura 10.9

Para obtener la gráfica anterior, realizamos el ensayo con el ejecutable planta_deposito_solo.cpp, cortando a los 25 segundos el caudal que estaba llenando el depósito, y eliminando la parte estocástica en la válvula de descarga (anulamos el segundo coeficiente del sumador).

Es evidente que la altura del nivel de agua tiende a cero en el estacionario. A los 90 segundos, se alcanzan las cercanías del cero, pero nunca llega a cero, sino que oscila entre valores positivos muy pequeños.

```
# PLANTA_ID8601.dat
#Tiempo          Altura            Caudal salida          Alarma
104.399          8.65896e-08       1.04773e-05            0
104.499          1.63913e-07       1.43086e-05            0
104.599          1.19339e-09       1.22925e-06            0
104.699          2.53758e-09       1.7965e-06             0
104.799          2.15251e-08       5.18828e-06            0
104.899          6.22717e-08       8.80033e-06            0
104.999          1.37838e-07       1.32371e-05            0
105.099          3.46007e-07       2.06479e-05            0
105.199          9.11014e-10       1.06203e-06            0
105.299          8.0127e-10        1.00059e-06            0
105.399          7.85743e-10       9.91937e-07            0
105.499          4.38177e-09       2.36416e-06            0
105.599          2.47333e-08       5.63425e-06            0
```

Este comportamiento es debido a la inestabilidad del cálculo discreto, que tiene un paso pequeño, pero no infinitesimal, en dominios donde es más difícil realizar la integración. Además, en el tratamiento numérico hay errores de truncamiento. Por eso no se aconseja utilizar el modelo del depósito con alturas de agua muy pequeñas.

- Carro motorizado

Con este modelo se introduce el concepto de ligadura, con sistemas que tienen comportamientos diferentes, según el valor de alguna variable. En el carro el límite es físico, mecánico, puesto que cuando se alcanza un tope, actúa una fuerza de ligadura que anula toda velocidad (choque inelástico perfecto). Basándonos en esta ligadura, se modifican las ecuaciones de estado, con bloques **`if-else`** en el código. También se convierten los valores angulares (ángulo y velocidad angular) en valores lineales (posición y velocidad), pues el motor giratorio está unido con el carro mediante un tornillo sinfín.

Como entrada en el modelo físico tenemos el coeficiente de rozamiento a lo largo del raíl, que puede estar distribuido uniformemente entre dos valores, de acuerdo con la rugosidad del material. Se utiliza, por tanto, un generador de números aleatorios.

A partir del coeficiente de rozamiento, la masa del carro y la gravedad el motor tiene un nuevo par de oposición, aparte del rozamiento propio del eje, dependiente de la velocidad.

Como sistema de control, tenemos dos detectores de fin de carrera (DI y DD) que cortarán el suministro de voltaje al motor del carro, de tal forma que llegará al fin del carril (topes TI y TD) con una velocidad mínima, debido al frenado propio por rozamiento, tanto en el motor giratorio, como en el carril.

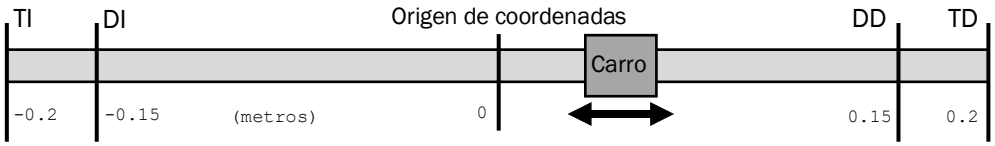


Figura 10.10

El control esperará cierto tiempo programado, e invertirá la velocidad cambiando de signo la tensión, para que el carro se dirija al otro extremo, repitiendo el ciclo. El ensayo muestra estas gráficas:

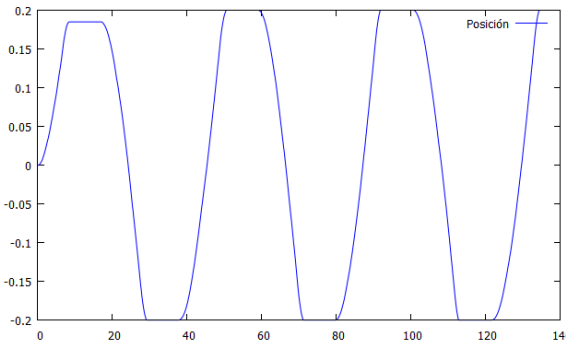


Figura 10.11

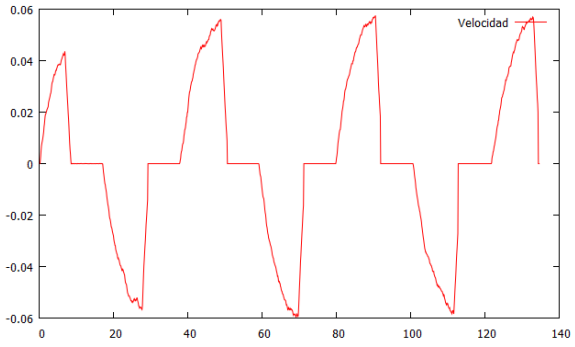


Figura 10.12

El primer recorrido, que parte de la posición cero, no llega al tope porque el carro no ha adquirido la suficiente inercia. Después, de extremo a extremo, se llega a uno u otro tope, con una velocidad lineal de 1 a 2 cm/s. Obsérvese cómo la "rugosidad" del raíl se manifiesta en la gráfica de velocidad cuando el carro llega a su velocidad máxima.

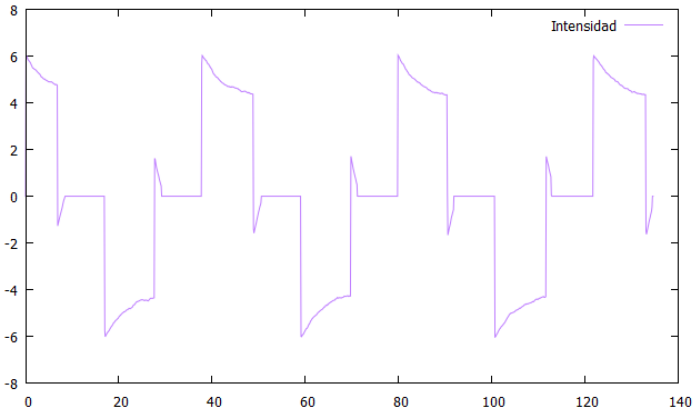


Figura 10.13

Se tiene que suministrar corriente en voltaje para compensar la pérdida de potencia de los rozamientos y sólo se hace nula cuando está el motor en reposo. Se pueden observar los picos al arrancar desde el reposo (±6 A), de forma similar a lo que ocurriría en un motor de corriente continua.

Los detectores solo se activan cuando el carro sobrepasa sus límites (a la derecha del detector derecho DD; o a la izquierda del detector izquierdo DI). La siguiente gráfica muestra el estado de los sensores que se envían al controlador.

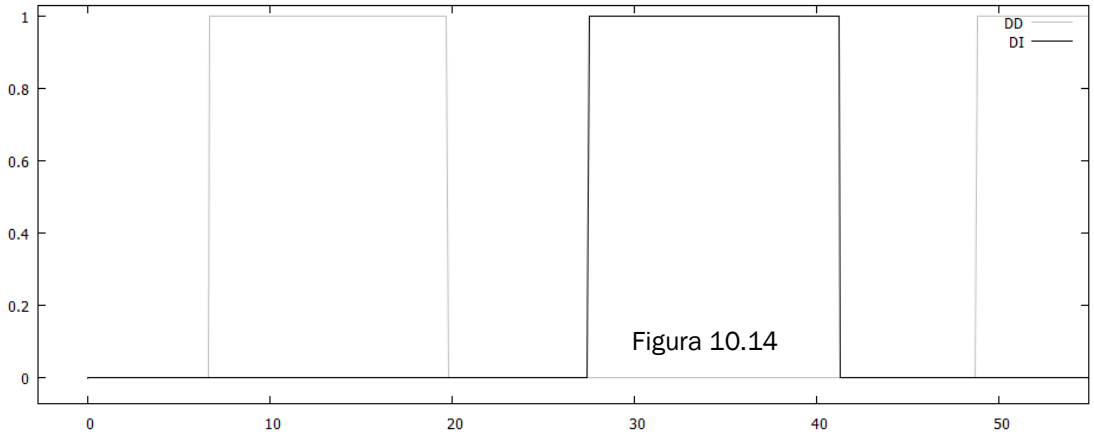


Figura 10.14

Y, por supuesto, la tensión de control del motor, que cortan el suministro al motor durante aproximadamente 10 segundos, cada vez que sobrepasa un detector.

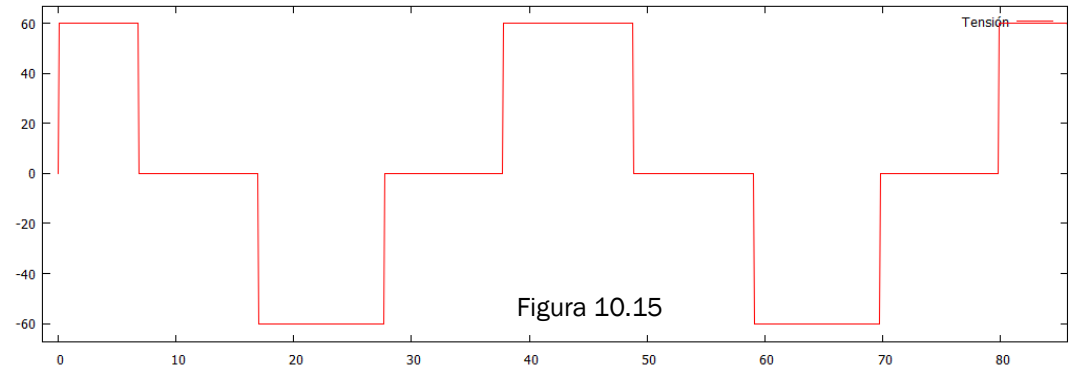


Figura 10.15

- <u>Motor de Corriente Continua</u>

Este sistema ya ha sido puesto a prueba, durante los conexionados en 3.6 y en el ejemplo del lazo de control en 4.3.

Volvemos a situarnos en planta_motorcc_solo.cpp y realizamos los cambios oportunos para poder guardar (comentamos conectores, descomentamos conectores de recogida, grabadoras, habilitamos grabadora en el menú, cambiamos generador de onda por generador de números constantes en el par de carga).

El ensayo simulará un motor sometido a una gran carga, unida por una correa de transmisión.

Antes de iniciar los hilos de la planta:

- Aplicamos 50 V al motor con el generador de números constantes.
- El par de carga será otro número constante, de valor 900 N · m.

Iniciamos todos los hilos, y el arranque llegará pronto al valor estacionario. Entonces, a los 47 segundos "cortamos" la correa, que equivale a hacer funcionar el motor en vacío, sin carga. Comprobamos hasta qué valor se sobrerrevoluciona.

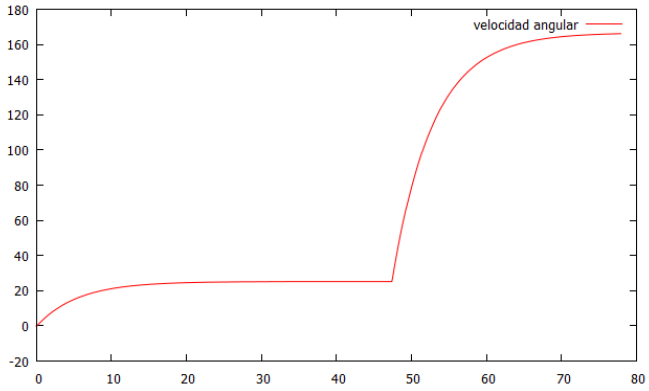


Figura 10.16

De aproximadamente 20 rads/s a plena carga se dispara a más de 160 rads/s debido al corte de la correa. Esto puede ser perjudicial para los cojinetes del motor y demás partes mecánicas.

El motor está sometido a un par en su giro en contra, es decir, está consumiendo una potencia que se reduce casi a cero al cortarse la correa. Lo vemos en la gráfica de intensidad:

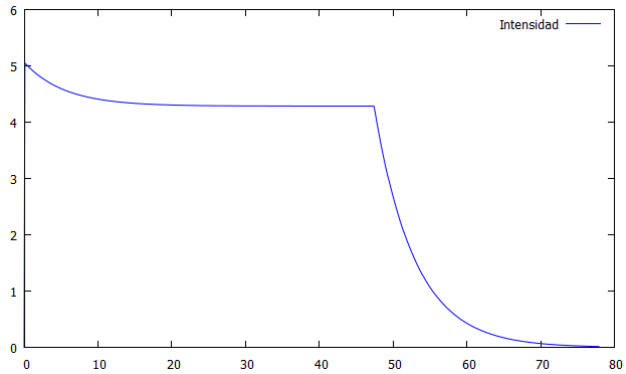


Figura 10.17

A plena carga, el motor está consumiendo una potencia eléctrica de $P = V \cdot I$

Sabiendo que V = 50 y que en el primer estacionario I = 4.286 A esto nos supone una potencia de $P = 50 \cdot 4.286 = 214.3\ W$. En el segundo tramo, la potencia consumida se hace mucho menor (casi nula).

Esta potencia no sólo se utiliza para contrarrestar el par, sino también para compensar las pérdidas en el eje. Dichas pérdidas pueden ser importantes si, por ejemplo, el eje está mal engrasado, llegando a provocar corrientes importantes que provocan una disipación de potencia en forma de calor inaceptable y mucho desgaste mecánico.

Realizamos un segundo ensayo en vacío y con un coeficiente de rozamiento que sube de $b = 1.9$ a $b = 1000.9$

A partir de los resultados de la grabación, vamos a calcular las pérdidas de potencia con el motor sin par de carga. La velocidad angular sin par de carga, que antes era superior a 160 rad/s, baja a 11 rad/s.

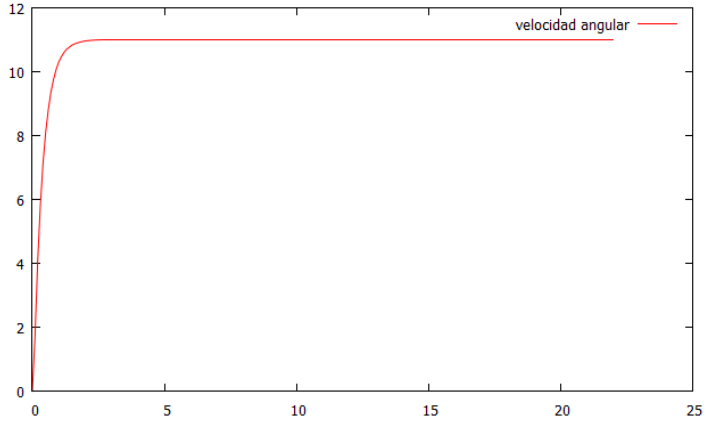


Figura 10.18

Y la intensidad para que el motor funcione en estacionario llega casi a 4.7 A.

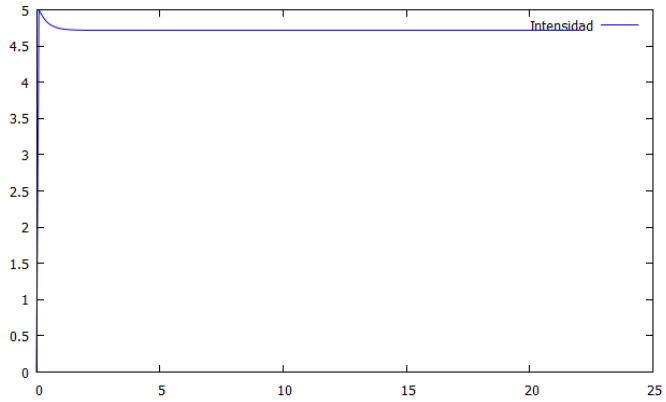


Figura 10.19

El motor en vacío está consumiendo un total de $P = 50 \cdot 4.7 = 235\ W$ que es superior a la potencia con carga aplicada del ensayo anterior.

Para finalizar...

Se pueden realizar más ensayos en cada uno de los modelos de planta y control que hay en el trabajo. Se ha preparado el código, y éste se ha comentado, para habilitar las sondas y grabadoras cuando sea necesario.